Владимир Дронов

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Django 2.1 Практика создания веб-сайтов на Python

Модели, контроллеры и шаблоны
Разграничение доступа
Аутентификация через социальные сети
Вывод миниатюр
Bootstrap
CAPTCHA
Angular
BBCode
REST

Материалы на www.bhv.ru

bhv®

Владимир Дронов

Django 2.1 Практика создания веб-сайтов на Python

Санкт-Петербург
«БХВ-Петербург»
2019

УДК 004.738.5+004.438Python
ББК 32.973.26-018.1
Д75

Дронов В. А.

Д75 Django 2.1. Практика создания веб-сайтов на Python. — СПб.: БХВ-Петербург, 2019. — 672 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-4058-2

Книга посвящена разработке веб-сайтов на Python с использованием веб-фреймворка Django 2.1. Рассмотрены основные функциональные возможности, необходимые для программирования сайтов общего назначения: модели, контроллеры, шаблоны, средства обработки пользовательского ввода, выгрузка файлов, разграничение доступа и др.

Рассказано о вспомогательных инструментах: посредниках, сигналах, средствах отправки электронной почты, подсистеме кэширования и пр. Описано форматирование текста посредством BBCode, обработка CAPTCHA, вывод графических миниатюр, аутентификация через социальные сети, интеграция с Bootstrap. Рассмотрено программирование веб-служб REST, использование административного веб-сайта Django, тестового сайта на Angular. Дан пример разработки полнофункционального веб-сайта — электронной доски объявлений. Исходный код доступен для загрузки с сайта издательства.

Для веб-программистов

УДК 004.738.5+004.438Python
ББК 32.973.26-018.1

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Марины Дамбиевой*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 28.02.19.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 54,18.
Тираж 1000 экз. Заказ № 8450.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-4058-2

© ООО "БХВ", 2019
© Оформление. ООО "БХВ-Петербург", 2019

Оглавление

Введение

Иногда случается так, что какой-либо многообещающий программный продукт или программная технология с шумом и треском появляются на рынке, напропалую грозят всех конкурентов если и не полностью уничтожить, то отодвинуть в глубокую тень, привлекают к себе внимание всех интересующихся информационными технологиями, после чего тихо уходят с рынка, и о них никто более не вспоминает.

Так вот — все это не о Django. Появившись в 2005 году — именно тогда вышла его первая версия, — он остается одним из популярнейших программных инструментов, предназначенных для разработки веб-сайтов.

Веб-фреймворк Django

Язык программирования Python исключительно развит сам по себе, но основную мощь ему придают всевозможные дополнительные библиотеки, которых существует превеликое множество. Есть библиотеки для научных расчетов, систем машинного зрения, программирования игр, обычных «настольных» приложений и, разумеется, веб-сайтов.

Среди последних особняком стоит ряд библиотек, реализующих бóльшую часть функциональности сайта. Эти библиотеки самостоятельно взаимодействуют с базами данных, обрабатывают клиентские запросы и формируют ответы, реализуют разграничение доступа, пуская к закрытым разделам сайта лишь тех посетителей, что перечислены в особом списке, и даже рассылают электронные письма. Разработчику остается только написать код, который генерирует веб-страницы сайта на основе данных, извлеченных из базы. И задача эта несравнимо менее трудоемкая, чем написание всей функциональности сайта, что называется, с нуля.

Такая всеобъемлющая библиотека напоминает готовый каркас (по-английски — framework), на который разработчик конкретного сайта «навешивает» свои узлы, механизмы и детали декора. Именно поэтому библиотеки подобного рода носят название *веб-фреймворков*, или просто *фреймворков*.

Один из фреймворков, написанных на языке Python, — Django. Среди всех такого рода разработок его стоит выделить особо. Хотя бы потому, что он, как было отме-

чено ранее, невероятно популярен. Более того, это наиболее часто применяемый на практике веб-фреймворк, разработанный на Python. И тому есть ряд причин.

- Django — это следование современным стандартам веб-разработки. В их числе: архитектура «модель-контроллер-шаблон», использование миграций для внесения изменений в базу данных и принцип «написанное однажды применяется везде» (или, другими словами, «не повторяйся»).
- Django — это полнофункциональный фреймворк. Для разработки среднестатистического сайта вам достаточно установить только его. Никаких дополнительных библиотек, необходимых, чтобы ваше веб-творение хотя бы заработало, ставить не придется.
- Django — это высокоуровневый фреймворк. Типовые задачи, наподобие соединения с базой данных, обработки данных, полученных от пользователя, сохранения выгруженных пользователем файлов, он выполняет самостоятельно. А еще он предоставляет полнофункциональную подсистему разграничения доступа и исключительно мощный и удобно настраиваемый административный веб-сайт, которые, в случае применения любого другого фреймворка, нам пришлось бы писать самостоятельно.
- Django — это удобство разработки. Легкий и быстрый отладочный веб-сервер, развитый механизм миграций, уже упомянутый административный веб-сайт — все это существенно упрощает программирование.
- Django — это дополнительные библиотеки. Нужен вывод графических миниатюр? Требуется обеспечить аутентификацию посредством социальных сетей? Необходима поддержка CAPTCHA? Для всего этого существуют библиотеки, которые нужно только установить.
- Django — это Python. Исключительно мощный и, вероятно, самый лаконичный язык из всех, что применяются в промышленном программировании.

Эта книга посвящена Django. Она описывает его наиболее важные и часто применяемые на практике функциональные возможности, ряд низкоуровневых инструментов, которые также могут пригодиться во многих случаях, и некоторые доступные для фреймворка дополнительные библиотеки. А в конце, в качестве практического упражнения, она описывает разработку полнофункционального сайта электронной доски объявлений.

> ***Внимание!***
>
> Автор предполагает, что читатели этой книги знакомы с языком разметки веб-страниц HTML, технологией каскадных таблиц стилей CSS, языком программирования веб-сценариев JavaScript и универсальным языком программирования Python. Описания всех этих технологий в книге приводиться не будут.

> ***Материалы полнофункционального сайта***
>
> Электронный архив с исходным кодом сайта электронной доски объявлений можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977540582.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение*).

Использованные программные продукты

Автор применял в работе следующие версии ПО:

- Microsoft Windows 10, русская 64-разрядная редакция со всеми установленными обновлениями;
- Python 3.6.5, 64-разрядная редакция;
- Django 2.1.3;
- Django Simple Captcha — 0.5.9;
- django-precise-bbcode — 1.2.9;
- django-bootstrap4 — 0.0.6 (разработка) и 0.0.7 (последующая проверка);
- Pillow — 5.2.0;
- django-cleanup — 2.1.0;
- easy-thumbnails — 2.5;
- Python Social Auth — 2.1.0;
- Django REST framework — 3.8.2;
- django-cors-headers — 2.4.0;
- mod-wsgi — 4.6.4;
- Angular — 6.1.7.

Типографские соглашения

В книге будут часто приводиться форматы написания различных языковых конструкций, применяемых в Python и Django. В них использованы особые типографские соглашения, которые мы сейчас изучим.

> ***Внимание!***
>
> Все эти типографские соглашения применяются автором только в форматах написания языковых конструкций Python и Django. В реальном программном коде они не имеют смысла.

- В угловые скобки (<>) заключаются наименования различных значений, которые дополнительно выделяются курсивом. В реальный код, разумеется, должны быть подставлены реальные значения. Например:

  ```
  django-admin startproject <имя проекта>
  ```

  Здесь вместо подстроки *имя проекта* должно быть подставлено реальное имя проекта.

- В квадратные скобки ([]) заключаются необязательные фрагменты кода. Например:

  ```
  django-admin startproject <имя проекта> [<путь к папке проекта>]
  ```

  Здесь *путь к папке проекта* может указываться, а может и не указываться.

- Слишком длинные, не помещающиеся на одной строке языковые конструкции автор разрывал на несколько строк и в местах разрывов ставил знаки ↳. Например:

```
background: url("bg.jpg") left / auto 100% no-repeat, ↳
url("bg.jpg") right / auto 100% no-repeat;
```

  Приведенный код разбит на две строки, но должен быть набран в одну. Символ ↳ при этом нужно удалить.

- Троеточием (. . .) помечены пропущенные ради сокращения объема текста фрагменты кода.

```
INSTALLED_APPS = [
    . . .
    'bboard.apps.BboardConfig',
]
```

  Здесь пропущены все элементы списка, присваиваемого переменной `INSTALLED_APPS`, кроме последнего.

  Обычно такое можно встретить в исправленных впоследствии фрагментах кода — приведены лишь собственно исправленные выражения, а оставшиеся неизмененными пропущены. Также троеточие используется, чтобы показать, в какое место должен быть вставлен вновь написанный код, — в начало исходного фрагмента, в его конец или в середину, между уже присутствующими в нем выражениями.

> ***Еще раз внимание!***
>
> Все приведенные здесь типографские соглашения имеют смысл лишь в форматах написания конструкций Python и Django. В коде примеров используются только знак ↳ и троеточие.

ЧАСТЬ I

Вводный курс

ГЛАВА 1

Основные понятия Django. Вывод данных

Давайте сразу же приступим к делу. Прямо сейчас, в этой главе, мы установим сам фреймворк Django и начнем разработку простенького веб-сайта — электронной доски объявлений.

> ***НА ЗАМЕТКУ***
>
> Эта книга не содержит описания языка программирования Python. Если вам, уважаемый читатель, необходима помощь в его освоении, обратитесь к другим учебным пособиям. Полное описание Python можно найти на его «домашнем» сайте **https://www.python.org/**, там же имеются дистрибутивы его исполняющей среды (интерпретатора) в различных редакциях для разных операционных систем.

1.1. Установка фреймворка

Начиная с версии Python 3.4, в составе исполняющей среды этого языка поставляется утилита pip, с помощью которой очень удобно выполнять установку любых дополнительных библиотек. Эта утилита самостоятельно ищет указанную при ее запуске библиотеку в штатном репозитории PyPI (Python Package Index, реестр пакетов Python) — интернет-хранилище самых разных библиотек для Python. Найдя запрошенную библиотеку, pip самостоятельно загружает и устанавливает наиболее подходящую ее редакцию, при этом загружая и устанавливая также и все библиотеки, которые она использует для работы.

Запустим командную строку и отдадим в ней команду на установку Django, которая вполне понята безо всяких комментариев:

```
pip install django
```

> ***ВНИМАНИЕ!***
>
> Если исполняющая среда Python установлена в папке *Program Files* или *Program Files (x86)*, для выполнения установки любых дополнительных библиотек командную строку следует запустить с повышенными правами. Для этого надо найти в меню **Пуск** пункт **Командная строка** (в зависимости от версии Windows он может находиться в группе **Стандартные** или **Служебные**), щелкнуть на нем правой кнопкой мыши и выбрать

в появившемся контекстном меню пункт **Запуск от имени администратора** (в Windows 10 этот пункт находится в подменю **Дополнительно**).

Помимо Django, эта команда установит также библиотеку pytz, выполняющую обработку значений даты и времени с учетом временных зон и используемую упомянутым ранее фреймворком в работе. Не удаляйте эту библиотеку!

Спустя некоторое время фреймворк будет установлен, о чем pip нам обязательно сообщит (приведены номера версий Django и pytz, актуальные на момент подготовки книги):

```
Successfully installed django-2.1.3 pytz-2018.7
```

Теперь мы можем начинать разработку нашего первого веб-сайта.

1.2. Проект Django

Следующее, что нам нужно сделать, — создать новый проект. *Проектом* называется совокупность всего программного кода, составляющего разрабатываемый сайт. Можно даже сказать, что проект — это и есть наш сайт. Физически он представляет собой папку, в которой находятся разнообразные файлы с исходным кодом и другие папки (назовем ее *папкой проекта*).

Давайте же создадим новый, пока еще пустой проект Django, которому дадим имя `samplesite`. Для этого в запущенной ранее командной строке перейдем в папку, в которой должна находиться папка проекта, и отдадим команду:

```
django-admin startproject samplesite
```

Утилита django-admin служит для выполнения разнообразных административных задач. В частности, команда `startproject` указывает ей создать новый проект с именем, записанным после этой команды.

В папке, в которую мы ранее перешли, будет создана следующая структура файлов и папок:

```
samplesite
  manage.py
  samplesite
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

«Внешняя» папка samplesite — это, как нетрудно догадаться, и есть папка проекта. Как видим, ее имя совпадает с именем проекта, записанным в вызове утилиты django-admin. А содержимое этой папки таково:

- manage.py — программный файл с кодом одноименной утилиты, с использованием которой производятся различные действия над самим проектом. Впрочем, единственное, чем она занимается, — вызывает утилиту django-admin, передавая ей все полученные команды и конфигурируя ее для обработки текущего проекта;

- «внутренняя» папка samplesite — формирует пакет языка Python, содержащий модули, которые относятся к проекту целиком и задают его конфигурацию (в частности, ключевые настройки). Название этого пакета совпадает с названием проекта и менять его не стоит — в противном случае придется вносить в код обширные правки.

  В документации по Django этот пакет не имеет какого-либо ясного и однозначного названия. Поэтому, чтобы избежать путаницы, давайте назовем его *пакетом конфигурации*.

  Пакет конфигурации включает в себя такие модули:

  - __init__.py — пустой файл, сообщающий Python, что папка, в которой он находится, является полноценным пакетом;
  - settings.py — модуль с настройками самого проекта. Включает описание конфигурации базы данных проекта, пути ключевых папок, важные параметры, связанные с безопасностью, и пр.;
  - urls.py — модуль с маршрутами уровня проекта (о них мы поговорим позже);
  - wsgi.py — модуль, связывающий проект с веб-сервером. Используется при публикации готового сайта в Интернете. Мы будет рассматривать этот модуль в *главе 29*.

Еще раз отметим, что пакет конфигурации хранит настройки, относящиеся к самому проекту и влияющие на все приложения, что входят в состав этого проекта (о приложениях мы поведем разговор очень скоро).

Проект Django мы можем поместить в любое место файловой системы компьютера. Мы также можем переименовать папку проекта. В результате всего этого проект не потеряет своей работоспособности.

1.3. Отладочный веб-сервер Django

В процессе разработки сайта нам придется неоднократно открывать его в веб-обозревателе для тестирования. Если бы мы использовали другую фундаментальную программную платформу, например PHP, и другой фреймворк, такой как Yii или Laravel, — нам пришлось бы устанавливать на свой компьютер программу веб-сервера. Но в случае с Django делать этого не нужно — в состав Django входит *отладочный веб-сервер*, написанный на самом языке Python, не требующий сложной настройки и всегда готовый к работе. Чтобы запустить его, следует в командной строке перейти непосредственно в папку проекта (именно в нее, а не в папку, в которой находится папка проекта!) и отдать команду:

```
manage.py runserver
```

Здесь мы пользуемся уже утилитой manage.py, сгенерированной программой django-admin при создании проекта. Команда `runserver`, которую мы записали после имени этой утилиты, как раз и запускает отладочный веб-сервер.

Последний сразу же выдаст нам небольшое сообщение, говорящее о том, что код сайта загружен, проверен на предмет ошибок и запущен в работу, и что сам сайт теперь доступен по интернет-адресу **http://127.0.0.1:8000/** (хотя намного удобнее набрать адрес **http://localhost:8000/** — он проще запоминается). Как видим, отладочный сервер по умолчанию работает через TCP-порт 8000 (впрочем, при необходимости можно использовать другой).

Запустим веб-обозреватель и наберем в нем один из интернет-адресов нашего сайта. Мы увидим информационную страничку, предоставленную самим Django и сообщающую, что сайт, хоть еще и «пуст», но, в целом, работает (рис. 1.1).

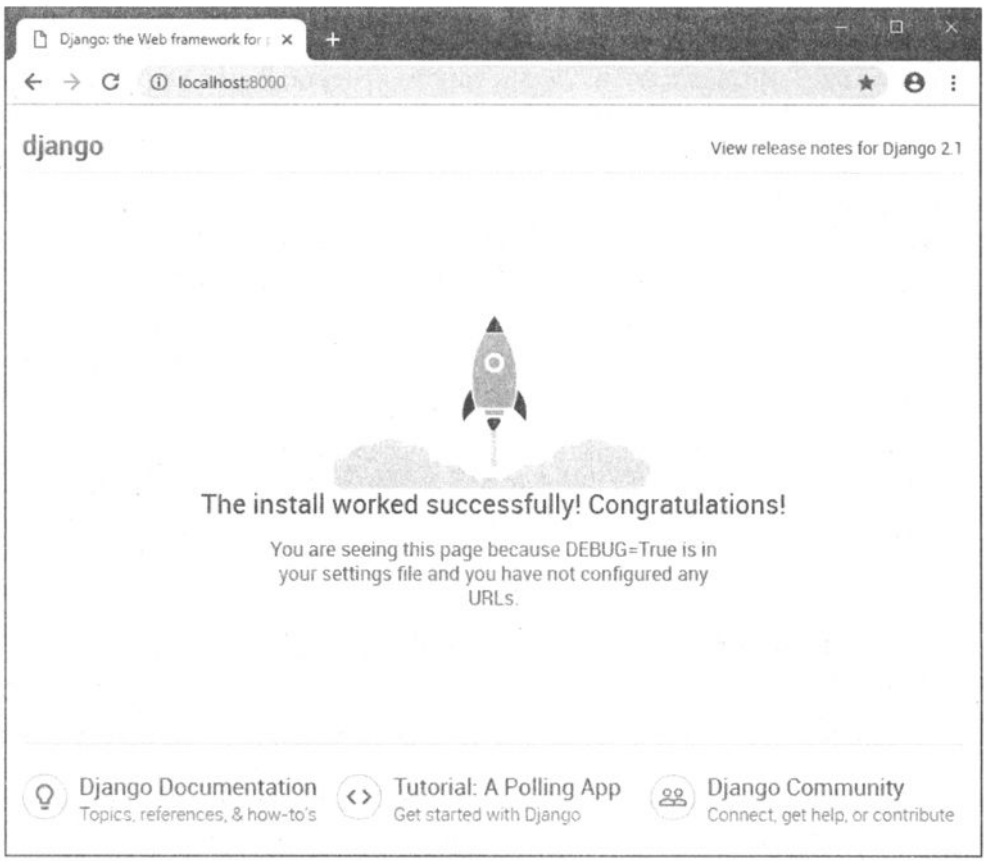


Рис. 1.1. Информационная веб-страница Django, сообщающая о работоспособности вновь созданного «пустого» сайта

Остановить отладочный веб-сервер можно, переключившись в экземпляр командной строки, в которой он был запущен, и нажав комбинацию клавиш <Ctrl>+<Break>.

1.4. Приложения

«Пустой» проект не содержит вообще никакой функциональности. (Вывод информационной страницы, которую мы только что наблюдали, не в счет.) Нам понадобится ее добавить. И реализуется эта функциональность в отдельных приложениях.

Приложение в терминологии Django — это отдельный фрагмент функциональности разрабатываемого сайта, более или менее независимый от других таких же фрагментов и входящий в состав проекта. Приложение может реализовывать работу целого сайта, его раздела или же какой-либо внутренней подсистемы сайта, используемой другими приложениями.

Любое приложение представляется обычным пакетом Python (*пакет приложения*), в котором находятся модули с программным кодом. Этот пакет находится в папке проекта — там же, где располагается пакет конфигурации. Имя пакета приложения станет именем самого приложения.

Нам нужно сделать так, чтобы наш сайт выводил перечень объявлений, оставленных посетителями. Для этого мы создадим новое приложение, которое незатейливо назовем `bboard`.

Новое приложение создается следующим образом. Сначала остановим отладочный веб-сервер. В командной строке проверим, находимся ли мы в папке проекта, и наберем команду:

```
manage.py startapp bboard
```

Команда `startapp` утилиты manage.py запускает создание нового «пустого» приложения, чье имя указано после этой команды.

Посмотрим, что же создала утилита manage.py. Прежде всего, это папка bboard, формирующая одноименный пакет приложения и расположенная в папке проекта. В ней находятся следующие папки и файлы:

- migrations — папка вложенного пакета, в котором будут сохраняться модули сгенерированных Django миграций (о них разговор обязательно пойдет, но позже). Пока что в папке находится лишь пустой файл __init__.py, помечающий ее как полноценный пакет Python;
- __init__.py — пустой файл, сигнализирующий языку Python, что эта папка — пакет;
- admin.py — модуль административных настроек и классов-редакторов;
- apps.py — модуль с настройками приложения;
- models.py — модуль с моделями;
- tests.py — модуль с тестирующими процедурами;
- views.py — модуль с контроллерами.

Пока что все это выглядит для нас как китайская грамота. Немного терпения — все это мы обязательно рассмотрим.

Внимание!

Подсистема тестирования кода, реализованная в Django, в этой книге не рассматривается, поскольку автор не считает ее сколь-нибудь полезной.

Теперь давайте зарегистрируем только что созданное приложение в проекте. Найдем в пакете конфигурации файл settings.py (о котором уже упоминалось ранее), откроем его в текстовом редакторе и отыщем следующий фрагмент кода:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Список, хранящийся в переменной `INSTALLED_APPS`, перечисляет все приложения, зарегистрированные в проекте и участвующие в его работе. Все эти приложения поставляются в составе Django и реализуют работу какой-либо из встроенных подсистем фреймворка. Так, приложение `django.contrib.auth` реализует работу подсистемы разграничения доступа, а приложение `django.contrib.sessions` — подсистемы, обслуживающих серверные сессии.

В этой теплой компании явно не хватает нашего приложения `bboard`. Добавим его, включив в список новый элемент:

```
INSTALLED_APPS = [
    . . .
    'bboard.apps.BboardConfig',
]
```

Обратим внимание на три важных момента. Во-первых, элемент списка приложений должен представлять собой строку с путем к классу `BboardConfig`, описывающему конфигурацию приложения и объявленному в упомянутом ранее модуле apps.py, что хранится в пакете приложения. Во-вторых, этот путь указывается в том формате, в котором записываются пути к модулям в стандарте языка Python (т. е. отдельные фрагменты пути разделяются точками, а не обратными слешами). В-третьих, этот путь указывается относительно папки проекта.

Сохраним и закроем файл settings.py. Но запускать отладочный веб-сервер пока не станем. Вместо этого сразу же напишем первый в нашей практике Django-программирования контроллер.

1.5. Контроллеры

Контроллер Django — это код, запускаемый в ответ на поступление клиентского запроса, который содержит интернет-адрес определенного формата. Именно в контроллерах выполняются все действия по подготовке данных для вывода, равно как и обработка данных, поступивших от посетителя.

> **Внимание!**
>
> В документации по Django используется термин «view» (вид, или представление). Автор книги считает его неудачным и предпочитает применять термин «контроллер», тем более что это устоявшееся название программных модулей такого типа.

Контроллер Django может представлять собой как функцию (*контроллер-функция*), так и класс (*контроллер-класс*). Первые более универсальны, но зачастую трудоемки в программировании, вторые позволяют выполнить типовые задачи, наподобие вывода списка каких-либо позиций, минимумом кода. И первые, и вторые мы обязательно рассмотрим в последующих главах.

Для хранения кода контроллеров изначально предназначается модуль views.py, создаваемый в каждом пакете приложения. Однако ничто не мешает нам поместить контроллеры в другие модули, благо Django не предъявляет к организации кода контроллеров никаких специальных требований.

Давайте напишем контроллер, который будет выводить... нет, не список объявлений — этого списка у нас пока нет (у нас и базы данных-то, можно сказать, нет), а пока только текст, сообщающий, что будущие посетители сайта со временем увидят на этой страничке список объявлений. Это будет контроллер-функция.

Откроем модуль views.py пакета приложения `bboard`, удалим имеющийся там небольшой код и заменим его кодом из листинга 1.1.

Листинг 1.1. Простейший контроллер-функция, выводящий текстовое сообщение

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Здесь будет выведен список объявлений.")
```

Наш контроллер — это, собственно, функция `index()`. Единственное, что она делает, — отправляет клиенту текстовое сообщение: **Здесь будет выведен список объявлений**. Но это только пока...

Любой контроллер-функция в качестве единственного обязательного параметра принимает экземпляр класса `HttpRequest`, хранящий различные сведения о полученном запросе: запрашиваемый интернет-адрес, данные, полученные от посетителя, служебную информацию от самого веб-обозревателя и пр. По традиции этот параметр называется `request`. В нашем случае мы его никак не используем.

В теле функции мы создаем экземпляр класса `HttpResponse` (он объявлен в модуле `django.http`), который будет представлять отправляемый клиенту ответ. Содержимое этого ответа — собственно текстовое сообщение — мы указываем единственным параметром конструктора этого класса. Готовый экземпляр класса мы возвращаем из функции в качестве результата.

Что ж, теперь мы с гордостью можем считать себя программистами — поскольку уже самостоятельно написали какой-никакой программный код. Осталось запус-

тить отладочный веб-сервер, набрать в любимом веб-обозревателе адрес вида **http://localhost:8000/bboard/** и посмотреть, что получится...

Минуточку! А с чего мы взяли, что при наборе такого интернет-адреса Django запустит на выполнение именно написанный нами контроллер-функцию `index()`? Ведь мы нигде явно не связали интернет-адрес с контроллером. Но как это сделать?..

1.6. Маршруты и маршрутизатор

Сделать это очень просто. Нужно всего лишь:

- объявить связь интернет-адреса определенного формата (*шаблонного интернет-адреса*) с определенным контроллером — иначе говоря, *маршрут*.

  Шаблонный интернет-адрес должен содержать только путь, без названия протокола, адреса хоста, номера порта, набора GET-параметров и имени якоря (поэтому его часто называют *шаблонным путем*). Он должен завершаться символом слеша (напротив, начальный слеш недопустим);
- оформить все объявленные нами маршруты в виде *списка маршрутов*;
- оформить маршруты в строго определенном формате, чтобы подсистема *маршрутизатора* смогла использовать готовый список в работе.

При поступлении любого запроса от клиента Django разбирает его на составные части (чем занимается целая группа программных модулей, называемых *посредниками* и описываемых в *главе 21*), извлекает запрошенный посетителем интернет-адрес, удаляет из него все составные части, за исключением пути, который передает маршрутизатору. Последний последовательно сравнивает его с шаблонными адресами, записанными в списке маршрутов. Как только будет найдено совпадение, маршрутизатор выясняет, какой контроллер связан с совпавшим шаблонным адресом, и передает этому контроллеру управление.

Давайте подумаем. Чтобы при наборе интернет-адреса **http://localhost:8000/bboard/** запускался только что написанный нами контроллер `index()`, нам нужно связать таковой с шаблонным адресом **bboard/**. Сделаем это.

В *разд. 1.2*, знакомясь с проектом, мы заметили хранящийся в пакете конфигурации модуль urls.py, в котором записываются маршруты уровня проекта. Давайте откроем этот модуль в текстовом редакторе и посмотрим, что он содержит (листинг 1.2).

Листинг 1.2. Изначальное содержимое модуля urls.py пакета конфигурации

```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Список маршрутов, оформленный в виде обычного списка Python, присваивается переменной `urlpatterns`. Каждый элемент списка маршрутов (т. е. каждый маршрут) должен представляться в виде результата, возвращаемого функцией `path()` из модуля `django.urls`. Последняя в качестве параметров принимает строку с шаблонным интернет-адресом и ссылку на контроллер-функцию.

В качестве второго параметра функцией `path()` также может быть принят список маршрутов уровня приложения. Кстати, этот вариант демонстрируется в выражении, задающем единственный маршрут в листинге 1.2. Мы рассмотрим его потом.

А сейчас давайте добавим в список новый маршрут, связывающий шаблонный адрес **bboard/** и контроллер-функцию `index()`. Для чего дополним имеющийся в модуле urls.py код согласно листингу 1.3.

Листинг 1.3. Новое содержимое модуля urls.py пакета конфигурации

```
from django.contrib import admin
from django.urls import path

from bboard.views import index

urlpatterns = [
    path('bboard/', index),
    path('admin/', admin.site.urls),
]
```

Сохраним исправленный файл, запустим отладочный веб-сервер и наберем в веб-обозревателе интернет-адрес **http://localhost:8000/bboard/**. Мы увидим текстовое сообщение, сгенерированное нашим контроллером (рис. 1.2).

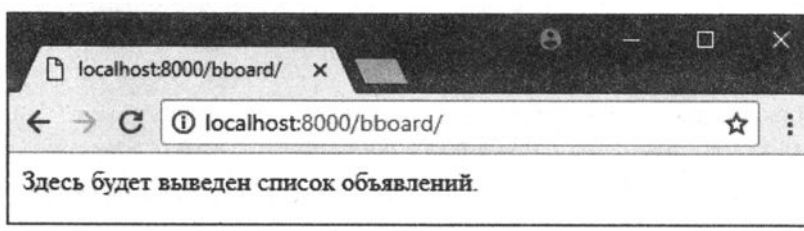


Рис. 1.2. Результат работы нашего первого контроллера — простое текстовое сообщение

Что ж, наши первые контроллер и маршрут работают, и по этому поводу можно порадоваться. Но лишь до поры до времени. Как только мы начнем создавать сложные сайты, состоящие из нескольких приложений, количество маршрутов в списке вырастет до таких размеров, что мы просто запутаемся в них. Поэтому создатели Django настоятельно рекомендуют применять для формирования списков маршрутов другой подход, о котором мы сейчас поговорим.

Маршрутизатор Django при просмотре списка маршрутов не требует, чтобы интернет-адрес, полученный из клиентского запроса, и шаблонный адрес, записанный

в очередном маршруте, совпадали полностью. Достаточно лишь того факта, что шаблонный адрес совпадает с началом реального. В таком случае шаблонизатор удаляет из реального адреса его начальную часть (префикс), совпавшую с шаблонным адресом, и запускает на исполнение указанный в маршруте контроллер.

Но, как было сказано ранее, функция `path()` позволяет указать во втором параметре вместо ссылки на контроллер-функцию другой список маршрутов. То есть мы можем указать для любого маршрута другой, *вложенный* в него список маршрутов. В таком случае маршрутизатор выполнит просмотр маршрутов, входящих в состав вложенного списка, используя для сравнения реальный интернет-адрес с уже удаленным из него префиксом.

Исходя из всего этого, мы можем создать иерархию списков маршрутов. В списке, созданном у самого проекта (*списке маршрутов уровня проекта*), мы укажем маршруты, которые указывают на вложенные списки маршрутов, записанные в отдельных приложениях проекта (*списки маршрутов уровня приложения*). А в последних мы уже запишем все контроллеры, что составляют программную логику нашего сайта.

Что ж, так и сделаем. И начнем со списка маршрутов уровня приложения `bboard`. Создадим в пакете этого приложения (т. е. в папке **bboard**) файл **urls.py** и занесем в него код из листинга 1.4.

Листинг 1.4. Код модуля urls.py пакета приложения bboard

```
from django.urls import path

from .views import index

urlpatterns = [
    path('', index),
]
```

Пустая строка, переданная первым параметром в функцию `path()`, обозначает корень пути из маршрута предыдущего уровня вложенности (*родительского*).

Наконец, исправим код модуля **urls.py** из пакета конфигурации, как показано в листинге 1.5.

Листинг 1.5. Окончательный код модуля urls.py пакета конфигурации

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('bboard/', include('bboard.urls')),
    path('admin/', admin.site.urls),
]
```

Вложенный список маршрутов, указываемый во втором параметре функции `path()`, должен представлять собой результат, возвращенный функцией `include()` из модуля `django.urls`. Единственным параметром эта функция принимает строку с путем к модулю, где записан список маршрутов.

Как только наш сайт получит запрос с интернет-адресом **http://localhost:8000/bboard/**, маршрутизатор обнаружит, что этот адрес совпадает с шаблонным адресом **bboard/**, записанным в первом маршруте из листинга 1.5. Он удалит из полученного в запросе адреса префикс, соответствующий шаблонному адресу, и получит пустую строку. Далее последует загрузка вложенного списка маршрутов из модуля urls.py пакета приложения `bboard`. Полученный интернет-адрес, представляющий собой пустую строку, совпадет с первым же маршрутом из вложенного списка, в результате чего запустится записанный в этом маршруте контроллер-функция `index()`, и на экране появится уже знакомое нам текстовое сообщение (см. рис. 1.2).

Поскольку зашла речь о вложенных списках маршрутов, давайте посмотрим на выражение, создающее второй маршрут из списка уровня проекта:

```
path('admin/', admin.site.urls),
```

Этот маршрут связывает шаблонный интернет-адрес **admin/** со списком маршрутов, возвращенным свойством `urls` экземпляра класса `AdminSite`, что хранится в переменной `site` и представляет текущий административный веб-сайт Django. Следовательно, набрав интернет-адрес **http://localhost:8000/admin/**, мы попадем на этот административный сайт (более подробно об административном сайте, встроенном во фреймворк, мы поговорим позже).

1.7. Модели

Настала пора сделать так, чтобы вместо намозолившего глаза текстового сообщения выводились реальные объявления, взятые из информационной базы. Если бы мы писали сайт на «чистом» Python, нам бы пришлось вручную создать в базе данных таблицу со всеми необходимыми полями и написать код, который будет открывать базу, считывать из нее данные и преобразовывать в нужный вид. Та еще работенка...

Однако мы работаем с Django — лучшим в мире веб-фреймворком. И для реализации хранения любых сущностей строго определенной структуры нам понадобится всего лишь объявить один-единственный класс, называемый *моделью*.

Модель — это однозначное и исчерпывающее описание сущности, хранящейся в базе данных в виде класса Python. Класс модели описывает таблицу базы данных, в которой будет храниться набор сущностей, и содержит атрибуты класса (в других языках программирования их называют свойствами класса, или статическими свойствами), каждый из которых описывает одно из полей этой таблицы. Можно сказать, что модель — это представление таблицы и ее полей средствами Python.

Отдельный экземпляр класса модели представляет отдельную конкретную сущность, извлеченную из базы, т. е. отдельную запись соответствующей таблицы. Пользуясь объявленными в модели атрибутами класса, мы можем получать значения, хранящиеся в полях записи, равно как и записывать в них новые значения.

Помимо этого, класс модели предоставляет инструменты для выборки сущностей из базы, их фильтрации и сортировки на основе заданных критериев. Полученный результат представляется в виде последовательности экземпляров класса модели.

Модели объявляются на уровне приложения. Объявляющий их код должен записываться в модуль models.py пакета приложения. Изначально этот модуль пуст.

Давайте объявим модель `Bb`, которая будет представлять объявление, со следующими полями:

- `title` — заголовок объявления, содержащий название продаваемого товара (тип — строковый, длина — 50 символов). Поле, обязательное к заполнению;
- `content` — сам текст объявления, описание товара (тип — memo);
- `price` — цена (тип — вещественное число);
- `published` — дата публикации (тип — дата и время, значение по умолчанию — текущие дата и время, индексированное).

Завершим работу отладочного веб-сервера. Откроем модуль models.py пакета приложения `bboard` и запишем в него код, объявляющий класс модели `Bb` (листинг 1.6).

Листинг 1.6. Код класса модели `Bb`

```
from django.db import models

class Bb(models.Model):
    title = models.CharField(max_length=50)
    content = models.TextField(null=True, blank=True)
    price = models.FloatField(null=True, blank=True)
    published = models.DateTimeField(auto_now_add=True, db_index=True)
```

Сама модель должна быть подклассом класса `Model` из модуля `django.db.models`. Отдельные поля модели, как говорилось ранее, оформляются как атрибуты класса, а в качестве значений им присваиваются экземпляры классов, представляющих поля разных типов и объявленных в том же модуле. Параметры полей указываются в конструкторах классов полей в виде значений именованных параметров.

Давайте рассмотрим использованные нами классы полей и их параметры:

- `CharField` — обычное строковое поле фиксированной длины. Допустимая длина значения указывается параметром `max_length` конструктора;
- `TextField` — текстовое поле неограниченной длины, или memo-поле. Присвоив параметрам `null` и `blank` конструктора значения `True`, мы укажем, что это поле можно не заполнять (по умолчанию любое поле обязательно к заполнению);

- `FloatField` — поле для хранения вещественных чисел. Оно также необязательно для заполнения (см. набор параметров его конструктора);
- `DateTimeField` — поле для хранения отметки даты и времени. Присвоив параметру `auto_now_add` конструктора значение `True`, мы предпишем Django при создании новой записи записывать в это поле текущие дату и время. А параметр `db_index` при присваивании ему значения `True` укажет создать для этого поля индекс (при выводе объявлений мы будем сортировать их по убыванию даты публикации, и индекс здесь очень пригодится).

Вот в чем основная прелесть Django и его механизма моделей: во-первых, мы описываем структуру таблицы базы данных в понятных нам терминах любимого Python, а во-вторых, описываем на очень высоком уровне, в результате чего нам не придется беспокоиться, скажем, о занесении нужного значения в поле `published` вручную. И это не может не радовать.

Еще один любопытный момент. Практически всегда таблицы баз данных имеют поле для хранения *ключей* — уникальных значений, которые будут однозначно идентифицировать соответствующие записи (*ключевое поле*). Как правило, это поле имеет целочисленный тип и помечено как автоинкрементное — тогда уникальные числовые значения в него будет заносить само программное ядро СУБД. В моделях Django такое поле явно объявлять не надо — фреймворк создаст его самостоятельно.

Сохраним исправленный файл. Сейчас мы сгенерируем на его основе миграцию, которая создаст в базе данных все необходимые структуры.

> ***На заметку***
>
> По умолчанию вновь созданный проект Django настроен на использование базы данных в формате SQLite, хранящейся в файле *db.sqlite3* в папке проекта. Эта база данных будет создана уже при первом запуске отладочного веб-сервера.

1.8. Миграции

Миграция — это модуль Python, созданный самим Django на основе определенной модели и предназначенный для формирования в базе данных всех требуемых этой моделью структур: таблиц, полей, индексов, правил и связей. Еще один замечательный инструмент фреймворка, заметно упрощающий жизнь программистам.

Для формирования миграции на основе модели `Bb` мы переключимся в командную строку, проверим, остановлен ли отладочный веб-сервер и находимся ли мы в папке проекта, и дадим команду:

```
manage.py makemigrations bboard
```

Команда `makemigrations` утилиты manage.py запускает генерирование миграций для всех моделей, объявленных в приложении, чье имя записано после самой команды, и не изменившихся с момента предыдущего генерирования миграций.

Сформированные таким образом модули с миграциями сохраняются в пакете `migrations`, находящемся в пакете приложения. Модуль с кодом нашей первой

миграции будет иметь имя 0001_initial.py. Откроем его в текстовом редакторе и посмотрим на хранящийся в нем код (листинг 1.7).

Листинг 1.7. Код миграции, создающей структуры для модели Bb (приводится с незначительными сокращениями)

```
from django.db import migrations, models

class Migration(migrations.Migration):
    initial = True
    dependencies = [ ]

    operations = [
        migrations.CreateModel(
            name='Bb',
            fields=[
                ('id', models.AutoField(auto_created=True,
                primary_key=True, serialize=False, verbose_name='ID')),
                ('title', models.CharField(max_length=50)),
                ('content', models.TextField(blank=True, null=True)),
                ('price', models.FloatField(blank=True, null=True)),
                ('published', models.DateTimeField(auto_now_add=True,
                db_index=True)),
            ],
        ),
    ]
```

Описание средств Django, применяемых для программирования миграций вручную, выходит за рамки этой книги. Однако приведенный здесь код вполне понятен и напоминает код написанной нами ранее модели. В частности, сразу можно видеть список полей, которые должны быть созданы в таблице базы данных для того, чтобы модель `Bb` смогла использовать ее для хранения сущностей. Также можно догадаться, что таблица базы данных будет иметь имя `Bb` — как и модель.

И наконец, первым же элементом списка создаваемых полей идет автоинкрементное ключевое поле `id`. Мы не объявили это поле в модели явно, и Django создал его самостоятельно для своих собственных нужд.

Миграция при выполнении порождает команды на языке SQL, которые будут отправлены СУБД и, собственно, выполнят все действия по созданию необходимых структур данных. Давайте посмотрим на результирующий SQL-код нашей миграции, отдав команду:

```
manage.py sqlmigrate bboard 0001
```

После команды `sqlmigrate`, выводящей на экран SQL-код, мы поставили имя приложения и числовую часть имени модуля с миграцией. Прямо в командной строке мы получим такой результат (код для удобства чтения был переформатирован):

```
BEGIN;
--
-- Create model Bb
--
CREATE TABLE "bboard_bb" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(50) NOT NULL,
    "content" text NULL,
    "price" real NULL,
    "published" datetime NOT NULL
);
CREATE INDEX "bboard_bb_published_58fde1b5" ON "bboard_bb" ("published");
COMMIT;
```

Этот код был сгенерирован для СУБД SQLite (вспомним — проект Django по умолчанию использует базу данных этого формата). Если используется другая СУБД, результирующий SQL-код будет соответственно отличаться.

Что ж, налюбовавшись на нашу первую миграцию, давайте выполним ее. В процессе *выполнения* миграция создает все описанные в ней структуры.

Первое выполнение миграций рекомендуется производить для всех приложений, входящих в проект. Дело в том, что многие приложения, входящие в состав Django, содержат свои миграции, создающие в базе данных структуры для моделей, которые используются в этих приложениях. Так что, если эти миграции не выполнить, приложения окажутся неработоспособными.

Наберем в командной строке команду выполнения всех миграций всех приложений проекта:

```
manage.py migrate
```

Судя по выводящимся в командной строке сообщениям, таковых миграций много — десятка два, и они заметно увеличивают объем базы данных. Поэтому при программировании реальных сайтов настоятельно рекомендуется исключать ненужные стандартные приложения из списка зарегистрированных в проекте сразу же (эти приложения будут рассмотрены в *главе 3*).

1.9. Консоль Django

Итак, у нас есть готовая модель для хранения объявлений. Но пока что нет ни одного объявления. Давайте создадим парочку для целей отладки.

Фреймворк включает в свой состав собственную редакцию консоли Python Shell, называемую *консолью Django*. От аналогичной командной среды Python она отличается тем, что в ней в состав путей поиска модулей добавляется путь к папке проекта, в которой запущена эта консоль.

В командной строке наберем команду для запуска консоли Django:

```
manage.py shell
```

И сразу увидим знакомое приглашение >>>, предлагающее нам ввести какое-либо выражение Python и получить результат его выполнения.

1.10. Работа с моделями

Не медля, создадим первое объявление — первую запись модели Bb:

```
>>> from bboard.models import Bb
>>> b1 = Bb(title='Дача', content='Общество "Двухэтажники". ' + \
'Два этажа, кирпич, свет, газ, канализация', price=500000)
```

Запись модели создается аналогично экземпляру любого другого класса — вызовом конструктора. Значения полей создаваемой записи можно указать в вызове конструктора посредством именованных параметров, чьи имена совпадают с именами соответствующих полей.

Созданная таким образом запись модели не сохраняется в базе данных, а существует только в оперативной памяти. Чтобы сохранить ее, достаточно вызвать у нее метод save() без параметров:

```
>>> b1.save()
```

Проверим, сохранилось ли наше первое объявление, получив значение ключевого поля:

```
>>> b1.pk
1
```

Отлично! Сохранилось.

Атрибут класса pk, поддерживаемый всеми моделями, хранит значение ключевого поля текущей записи. А это значение может быть получено только после того, как запись модели успешно сохранится в базе данных.

Мы можем обратиться к любому полю записи, воспользовавшись соответствующим ему атрибутом класса модели:

```
>>> b1.title
'Дача'
>>> b1.content
'Общество "Двухэтажники". Два этажа, кирпич, свет, газ, канализация'
>>> b1.price
500000
>>> b1.published
datetime.datetime(2018, 5, 30, 13, 19, 41, 904710, tzinfo=<UTC>)
>>> b1.id
1
```

В последнем случае мы обратились непосредственно к ключевому полю id.

Создадим еще одно объявление:

```
>>> b2 = Bb()
>>> b2.title = 'Автомобиль'
```

```
>>> b2.content = '"Жигули"'
>>> b2.save()
>>> b2.pk
2
```

Да, можно поступить и так: создать «пустую» запись модели, записав вызов конструктора ее класса без параметров и занеся нужные значения в поля позже.

Что-то во втором объявлении маловато информации о продаваемой машине... Давайте дополним ее:

```
>>> b2.content = '"Жигули", 1980 года, ржавая, некрашеная, сильно битая'
>>> b2.save()
>>> b2.content
'"Жигули", 1980 года, ржавая, некрашеная, сильно битая'
```

И добавим еще одно объявление:

```
>>> Bb.objects.create(title='Дом', content='Трехэтажный, кирпич',
price=50000000)
<Bb: Bb object (3)>
```

Все классы моделей поддерживают атрибут класса `objects`. Он хранит *диспетчер записей* — особую структуру, позволяющую манипулировать всей совокупностью имеющихся в модели записей. Диспетчер записей представляется экземпляром класса `Manager`.

Метод `create()` диспетчера записей создает новую запись модели, принимая в качестве набора именованных параметров значения ее полей. При этом он сразу же сохраняет созданную запись и возвращает ее в качестве результата.

Давайте выведем ключи и заголовки всех имеющихся в модели `Bb` объявлений:

```
>>> for b in Bb.objects.all():
...     print(b.pk, ': ', b.title)
...
1 :  Дача
2 :  Автомобиль
3 :  Дом
```

Метод `all()` диспетчера записей возвращает так называемый *набор записей* — последовательность, содержащую записи модели, в нашем случае — все, что есть в базе данных. Сам набор записей представляется экземпляром класса `QuerySet`, а отдельные записи — экземплярами соответствующего класса модели. Поскольку набор записей является последовательностью и поддерживает итерационный протокол, мы можем перебрать его в цикле.

Отсортируем записи модели по заголовку:

```
>>> for b in Bb.objects.order_by('title'):
...     print(b.pk, ': ', b.title)
...
```

```
2 : Автомобиль
1 : Дача
3 : Дом
```

Метод `order_by()` диспетчера записей сортирует записи по значению поля, чье имя указано в параметре, и сразу же возвращает получившийся в результате сортировки набор записей.

Извлечем объявления о продаже домов:

```
>>> for b in Bb.objects.filter(title='Дом'):
...     print(b.pk, ': ', b.title)
...
3 : Дом
```

Метод `filter()` диспетчера записей выполняет фильтрацию записей по заданным критериям. В частности, чтобы получить только записи, у которых определенное поле содержит заданное значение, следует указать в вызове этого метода именованный параметр, чье имя совпадает с именем поля, и присвоить ему значение, которое должно содержаться в указанном поле. Метод возвращает другой диспетчер записей, содержащий только отфильтрованные записи.

Объявление о продаже автомобиля имеет ключ 2. Отыщем его:

```
>>> b = Bb.objects.get(pk=2)
>>> b.title
'Автомобиль'
>>> b.content
'"Жигули", 1980 года, ржавая, некрашеная, сильно битая'
```

Метод `get()` диспетчера записей имеет то же назначение, что и метод `filter()`, и вызывается аналогичным образом. Однако он ищет не все записи, подходящие под заданные критерии, а лишь одну и возвращает ее в качестве результата. К тому же, он работает быстрее метода `filter()`.

Давайте удалим это ржавое позорище:

```
>>> b.delete()
(1, {'bboard.Bb': 1})
```

Метод `delete()` модели, как уже понятно, удаляет текущую запись и возвращает сведения о количестве удаленных записей, обычно малополезные.

Ладно, хватит пока! Выйдем из консоли Django привычным нам способом — набрав команду `exit()`.

И займемся контроллером `index()`. Сделаем так, чтобы он выводил список объявлений, отсортированный по убыванию даты их публикации.

Откроем модуль views.py пакета приложения `bboard` и исправим хранящийся в нем код согласно листингу 1.8.

Листинг 1.8. Код модуля views.py пакета приложения bboard

```
from django.http import HttpResponse

from .models import Bb

def index(request):
    s = 'Список объявлений\r\n\r\n\r\n'
    for bb in Bb.objects.order_by('-published'):
        s += bb.title + '\r\n' + bb.content + '\r\n\r\n'
    return HttpResponse(s, content_type='text/plain; charset=utf-8')
```

Чтобы отсортировать объявления по убыванию даты их публикации, мы в вызове метода `order_by()` диспетчера записей предварили имя поля `published` символом «минус». Список объявлений мы представили в виде обычного текста, разбитого на строки последовательностью специальных символов возврата каретки и перевода строки: `\r\n`.

При создании экземпляра класса `HttpResponse`, представляющего отсылаемый клиенту ответ, мы в именованном параметре `content_type` конструктора указали тип отправляемых данных: обычный текст, набранный в кодировке UTF-8 (если мы этого не сделаем, веб-обозреватель посчитает текст HTML-кодом и выведет его одной строкой, скорее всего, в нечитаемом виде).

Сохраним исправленный файл и запустим отладочный веб-сервер. На рис. 1.3 показан результат наших столь долгих трудов. Теперь наш сайт стал больше похож на доску объявлений.

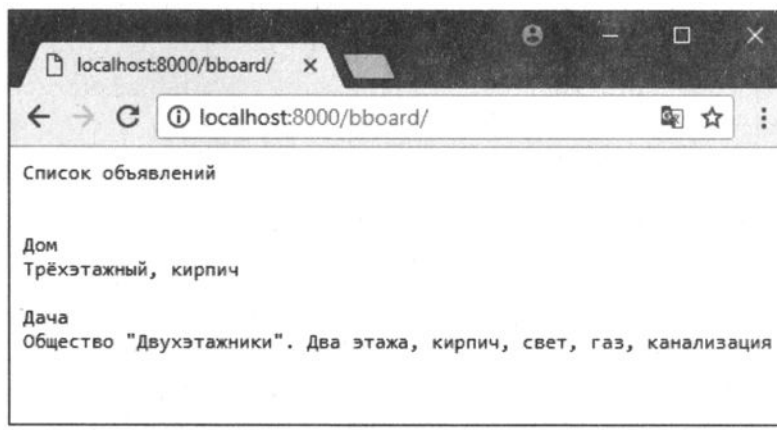


Рис. 1.3. Вывод списка объявлений в виде обычного текста

Аналогичным способом мы можем сформировать не текстовый документ, а полноценную веб-страницу. Вот только писать программу, формирующую строку с HTML-кодом, и, в особенности, отлаживать ее чрезвычайно трудоемко.

Есть ли другой способ? Безусловно, есть. Мы можем использовать шаблоны.

1.11. Шаблоны

Шаблон — это образец для формирования документа, который будет представлен клиенту: веб-страницы, файла в формате XML, PDF и пр. Подсистема Django, называемая *шаблонизатором*, загружает шаблон, объединяет его с данными, извлеченными из моделей, полученными от посетителя или сгенерированными в процессе работы, и формирует на основе всего этого полноценный документ, который и отправляется клиенту.

Применительно к веб-страницам, шаблон — это файл с HTML-кодом страницы, содержащий особые команды шаблонизатора: директивы, теги и фильтры. *Директивы* указывают поместить в заданное место HTML-кода какое-либо значение, *теги* управляют генерированием содержимого результирующего документа, а *фильтры* выполняют какие-либо преобразования указанного значения перед выводом.

По умолчанию шаблонизатор ищет все шаблоны в папках templates, вложенных в папки пакетов приложений (это поведение можно изменить, задав соответствующие настройки, о которых мы поговорим в *главе 11*). Сами файлы шаблонов веб-страниц должны иметь расширение html.

Создадим в папке пакета приложения `bboard` папку templates, а в ней — вложенную папку bboard.

Остановим отладочный сервер. Сейчас мы напишем наш первый Django-шаблон. Мы сохраним его в файле index.html в только что созданной папке templates\bboard. Код этого шаблона показан в листинге 1.9.

Листинг 1.9. Код шаблона bboard\index.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8">
        <title>Главная - Доска объявлений</title>
    </head>
    <body>
        <h1>Объявления</h1>
        {% for bb in bbs %}
        <div>
            <h2>{{ bb.title }}</h2>
            <p>{{ bb.content }}</p>
            <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
        </div>
        {% endfor %}
    </body>
</html>
```

В целом, здесь нам все знакомо. За исключением команд шаблонизатора. Давайте познакомимся с ними.

Начнем вот с этого тега шаблонизатора:

```
{% for bb in bbs %}
    . . .
{% endfor %}
```

По назначению он аналогичен циклу `for...in` языка Python. На каждом проходе он извлекает из последовательности, хранящейся в переменной `bbs` (эта переменная входит в состав контекста шаблона, который формируется самим программистом и о котором мы поговорим чуть позже), очередной элемент, заносит его в переменную `bb` и делает ее доступной в своем теле. В нашем случае в переменной `bb` на каждом проходе такого цикла будет сохраняться очередное объявление из списка `bbs`.

Теперь познакомимся с директивой шаблонизатора:

```
{{ bb.title }}
```

Она указывает извлечь значение из атрибута `title` объекта, хранящегося в созданной упомянутым ранее тегом переменной `bb`, и вставить это значение в то место кода, в котором находится она сама.

И, наконец, фильтр `date`:

```
<p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
```

Он преобразует значение из атрибута `published` объекта, хранящегося в переменной `bb`, т. е. дату и время публикации объявления, в формат, собственно, даты и времени. Сам формат, в который нужно преобразовать значение, указывается в параметре фильтра в виде строки. Строка `"d.m.Y H:i:s"` задает формат *<число>.<номер месяца>.<год из четырех цифр> <часы в 24-часовом формате>:<минуты>:<секунды>*.

1.12. Рендеринг шаблонов. Сокращения

Откроем модуль views.py пакета приложения `bboard` и внесем исправления в его код согласно листингу 1.10.

Листинг 1.10. Код модуля views.py пакета приложения bboard (используются низкоуровневые инструменты)

```
from django.http import HttpResponse
from django.template import loader

from .models import Bb

def index(request):
    template = loader.get_template('bboard/index.html')
```

```
    bbs = Bb.objects.order_by('-published')
    context = {'bbs': bbs}
    return HttpResponse(template.render(context, request))
```

Сначала мы загружаем шаблон, воспользовавшись функцией `get_template()` из модуля `django.template.loader`. В качестве параметра мы указали строку с путем к файлу шаблона, отсчитанному от папки templates. Результатом, возвращенным функцией, станет экземпляр класса `Template`, представляющий хранящийся в заданном файле шаблон.

Далее мы формируем контекст для нашего шаблона. *Контекстом шаблона* называется набор данных, которые должны быть доступны внутри шаблона в виде переменных и с которыми шаблонизатор объединит этот шаблон для получения выходного документа. Контекст шаблона должен представлять собой обычный словарь Python, ключи элементов которого станут переменными, доступными в шаблоне, а значения элементов — значениями этих переменных. В нашем случае контекст шаблона будет содержать только переменную `bbs`, где хранится список объявлений.

Далее мы выполняем обработку шаблона, в процессе которой шаблонизатор выполняет объединение его с данными из контекста (Django-программисты называют этот процесс *рендерингом*). Рендеринг запускается вызовом метода `render()` класса `Template`: первым параметром методу передается подготовленный контекст шаблона, а вторым — экземпляр класса `HttpRequest`, представляющий клиентский запрос и полученный контроллером-функцией через параметр `request`. Результат, возвращенный методом и представляющий собой строку с HTML-кодом готовой веб-страницы, мы передаем конструктору класса `HttpResponse` для формирования ответа.

Сохраним оба исправленных файла, запустим отладочный веб-сервер и посмотрим на результат. Вот теперь это действительно веб-страница (рис. 1.4)!

В коде контроллера `index()` (см. листинг 1.10) мы использовали, так сказать, сложный подход к рендерингу шаблона, применив низкоуровневые инструменты. Но Django предоставляет средства и более высокого уровня — функции-*сокращения* (shortcuts). В частности, функция-сокращение `render()` из модуля `django.shortcuts` позволяет выполнить рендеринг шаблона в одном выражении. Давайте попробуем ее в деле, исправив код модуля views.py, как показано в листинге 1.11.

Листинг 1.11. Код модуля views.py пакета приложения bboard (используется функция-сокращение `render()`)

```
from django.shortcuts import render

from .models import Bb

def index(request):
    bbs = Bb.objects.order_by('-published')
    return render(request, 'bboard/index.html', {'bbs': bbs})
```

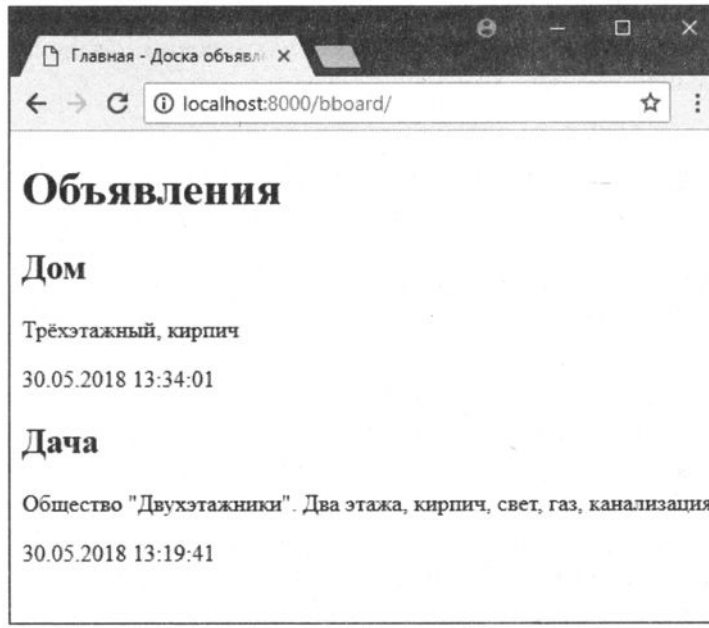


Рис. 1.4. Веб-страница, сформированная с применением шаблона

Обновим веб-страницу со списком объявлений — и убедимся, что этот код работает точно так же, как и написанный ранее, однако при этом имеет меньший объем.

1.13. Административный веб-сайт Django

Все-таки два объявления — это слишком мало... Давайте добавим еще несколько. Только сделаем это не вручную, посредством консоли Django, а через встроенный в этот фреймворк административный сайт.

Административный веб-сайт предоставляет доступ ко всем моделям, объявленным во всех приложениях, что составляют проект. Мы можем просматривать, добавлять, править и удалять записи, выполнять их фильтрацию и сортировку. Помимо этого, административный сайт не пускает к данным сайта посторонних, используя для этого встроенную во фреймворк подсистему разграничения доступа.

Эта подсистема реализована в стандартном приложении `django.contrib.auth`. А работу самого административного сайта обеспечивает стандартное приложение `django.contrib.admin`. Оба этих приложения заносятся в список зарегистрированных в проекте изначально.

Стандартное приложение `django.contrib.auth` использует для хранения списков зарегистрированных пользователей, групп и прав особые модели. Для них в базе данных должны быть созданы таблицы, и создание этих таблиц выполняют особые миграции. Следовательно, чтобы успешно задействовать встроенные средства разграничения доступа Django, нужно хотя бы один раз произвести операцию по выполнению миграций (мы это уже сделали).

После этого нужно создать зарегистрированного пользователя сайта с максимальными правами — *суперпользователя*. Для этого остановим отладочный веб-сервер и отдадим в командной строке команду:

```
manage.py createsuperuser
```

Получив команду создания суперпользователя `createsuperuser`, утилита manage.py запросит у нас его имя, его адрес электронной почты и пароль, который потребуется ввести дважды:

```
Username (leave blank to use '<имя учетной записи>'): admin
Email address: admin@somesite.ru
Password:
Password (again):
```

Отметим, что пароль должен содержать не менее 8-ми символов, буквенных и цифровых, набранных в разных регистрах, — в противном случае утилита откажется создавать пользователя.

Как только суперпользователь будет создан, настроим проект Django на использование русского языка. Откроем модуль settings.py пакета конфигурации и найдем в нем вот такое выражение:

```
LANGUAGE_CODE = 'en-us'
```

Переменная `LANGUAGE_CODE` задает код языка, используемого при выводе системных сообщений и страниц административного сайта. Изначально это американский английский язык (код `en-us`). Исправим это выражение, занеся в него код российского русского языка:

```
LANGUAGE_CODE = 'ru-ru'
```

Сохраним исправленный модуль и закроем это — более он нам не понадобится.

Теперь запустим отладочный веб-сервер и попробуем войти на административный сайт. Для этого в веб-обозревателе выполним переход по интернет-адресу **http://localhost:8000/admin/**. Сразу после этого будет выведена страница входа с формой (рис. 1.5), в которой нужно набрать введенные при создании суперпользователя имя и пароль и нажать кнопку **Войти**.

Если мы ввели имя и пароль пользователя без ошибок, то увидим страницу со списком приложений, зарегистрированных в проекте и объявляющих какие-либо модели (рис. 1.6). Под названием каждого приложения перечисляются объявленные в нем модели.

Но постойте! В списке присутствует только одно приложение — **Пользователи и группы** (так в списках административного сайта обозначается встроенное приложение `django.contrib.auth`) — с двумя моделями: **Группы** и **Пользователи**. Где же наше приложение `bboard` и его модель `Bb`?

Чтобы приложение появилось в списке административного сайта, его нужно явно зарегистрировать там. Сделать это очень просто. Откроем модуль административных настроек admin.py пакета приложения `bboard` и заменим имеющийся в нем небольшой код представленным в листинге 1.12.

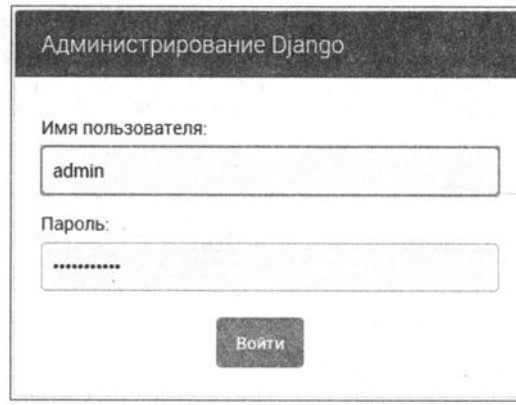


Рис. 1.5. Страница входа административного веб-сайта Django

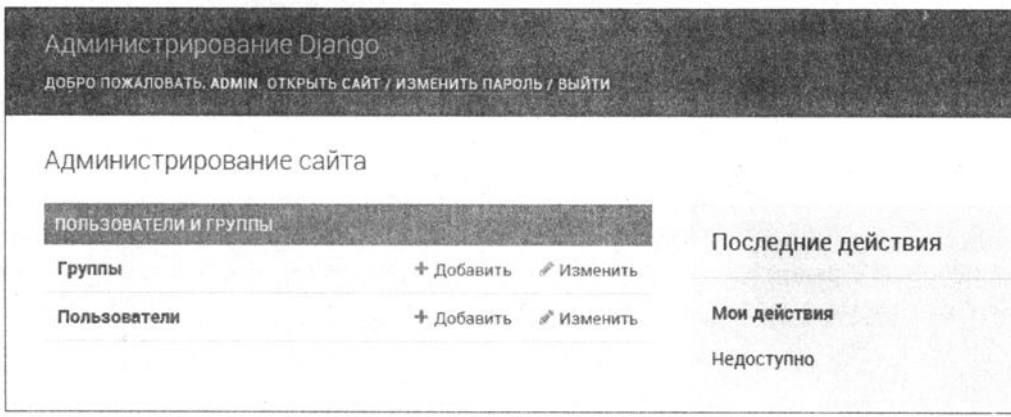


Рис. 1.6. Страница списка приложений административного сайта

Листинг 1.12. Код модуля admin.py пакета приложения bboard (выполнена регистрация модели Bb на административном сайте)

```
from django.contrib import admin

from .models import Bb

admin.site.register(Bb)
```

Мы вызвали метод `register()` у экземпляра класса `AdminSite`, представляющего сам административный сайт и хранящегося в переменной `site` модуля `django.contrib.admin`. Этому методу мы передали в качестве параметра ссылку на класс нашей модели `Bb`.

Как только мы сохраним модуль и обновим открытую в веб-обозревателе страницу списка приложений, то сразу увидим, что наше приложение также присутствует в списке (рис. 1.7). Совсем другое дело!

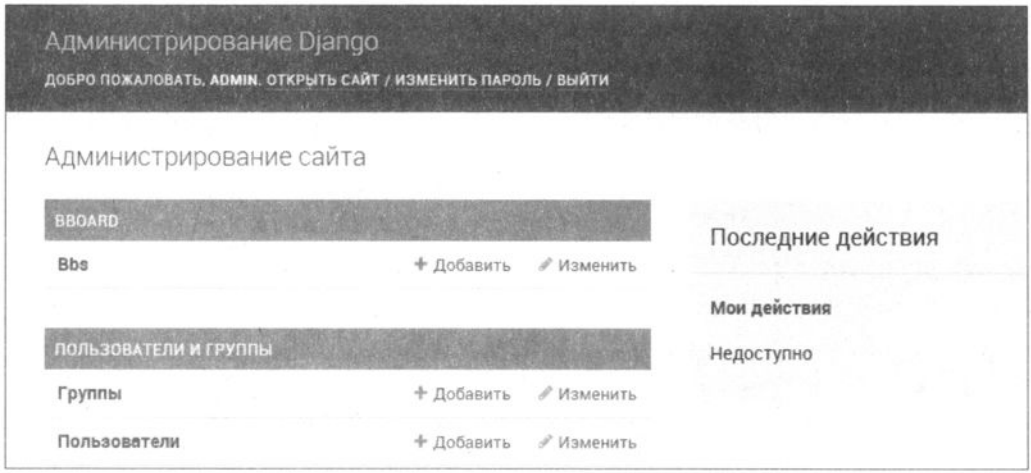


Рис. 1.7. Приложение `bboard` в списке приложений административного сайта

Каждое название модели в этом списке представляет собой гиперссылку, щелкнув на которой, мы попадем на страницу списка записей этой модели. Например, на рис. 1.8 показана страница списка записей, хранящихся в модели `Bb`.

Рис. 1.8. Страница списка записей, хранящихся в модели

Здесь мы можем выполнить следующие действия:

- щелкнуть на гиперссылке **Добавить *<имя класса модели>***, чтобы вывести страницу добавления новой записи (рис. 1.9). Занеся в элементы управления нужные данные, мы нажмем кнопку:
  - **Сохранить** — для сохранения записи и возврата на страницу списка записей;
  - **Сохранить и продолжить редактирование** — для сохранения записи;

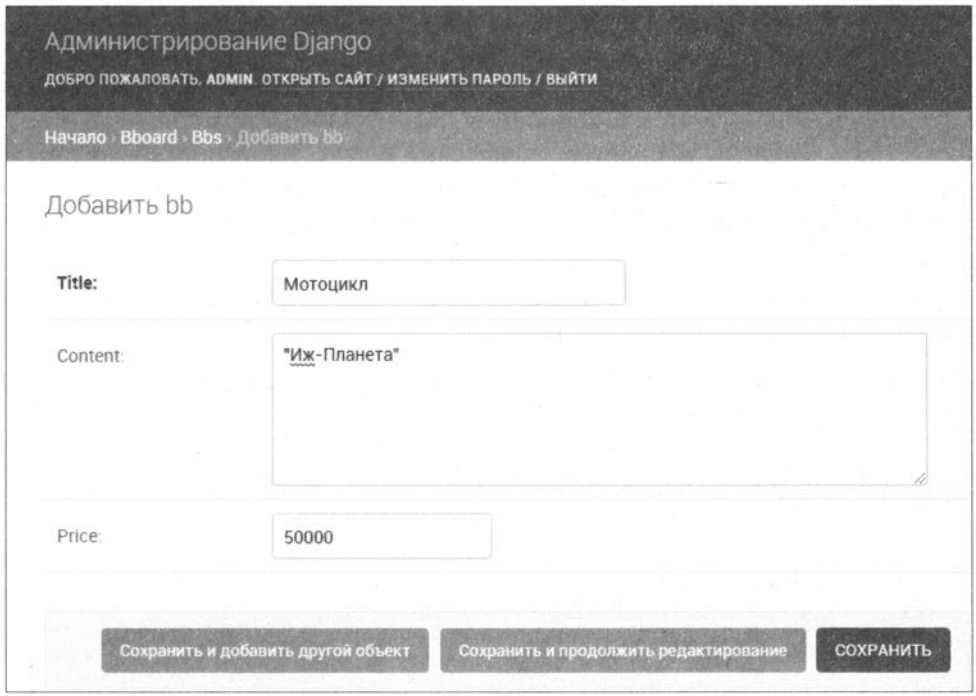


Рис. 1.9. Страница добавления записи в модель

- **Сохранить и добавить другой объект** — для сохранения записи и подготовки формы для ввода новой записи;

❒ щелкнуть на строке, обозначающей нужную запись, чтобы вывести страницу правки записи. Она похожа на страницу для добавления записи (см. рис. 1.9), за исключением того, что в наборе имеющихся на ней кнопок будет присутствовать еще одна — **Удалить**, выполняющая удаление текущей записи;

❒ выполнить над выбранными записями модели какое-либо из поддерживаемых действий.

Чтобы выбрать запись, следует найти в левой части соответствующей строки списка флажок и установить его. Выбрать таким образом можно произвольное количество записей.

Раскрывающийся список всех поддерживаемых моделью (точнее, заданным для нее классом-редактором, разговор о котором пойдет позже) действий находится над списком записей модели. Чтобы применить к выбранным записям действие, следует выбрать его из списка и нажать расположенную правее кнопку **Выполнить**. Веб-обозреватель покажет страницу подтверждения, на которой будут перечислены записи, над которыми выполнится действие, текст с вопросом, уверен ли пользователь, и кнопки с «говорящими» надписями **Да, я уверен** и **Нет, отменить и вернуться выбору**.

Мы можем попробовать ради эксперимента добавить несколько записей в модель `Bb`, исправить кое-какие записи и удалить ненужные. Но сначала давайте доведем нашу модель до ума.

1.14. Параметры полей и моделей

Во-первых, наша модель представляется какой-то непонятной аббревиатурой **Bbs**, а не простым и ясным текстом **Объявления**. Во-вторых, на страницах добавления и правки записи напротив элементов управления в качестве надписей проставлены имена полей модели (**title**, **content** и **published**), что может обескуражить пользователя. И в-третьих, объявления было бы неплохо отсортировать по убыванию даты публикации.

Одним словом, нам надо задать параметры как для полей модели, так и для самой модели.

Откроем модуль **models.py** пакета приложения `bboard`. Здесь нам нужно внести в код класса модели `Bb` ряд правок, сообразуясь с листингом 1.13.

Листинг 1.13. Код класса модели `Bb` (заданы параметры полей и самой модели)

```
class Bb(models.Model):
    title = models.CharField(max_length=50, verbose_name='Товар')
    content = models.TextField(null=True, blank=True,
    verbose_name='Описание')
    price = models.FloatField(null=True, blank=True, verbose_name='Цена')
    published = models.DateTimeField(auto_now_add=True, db_index=True,
    verbose_name='Опубликовано')

    class Meta:
        verbose_name_plural = 'Объявления'
        verbose_name = 'Объявление'
        ordering = ['-published']
```

Прежде всего, в вызов каждого конструктора класса поля мы добавили именованный параметр `verbose_name`. Он указывает «человеческое» название поля, которое будет выводиться на экран.

Далее, в классе модели мы объявили вложенный класс `Meta`, а в нем — атрибуты класса, которые зададут параметры уже самой модели:

- `verbose_name_plural` — название модели во множественном числе;
- `verbose_name` — название модели в единственном числе.

  Эти названия также будут выводиться на экран;
- `ordering` — последовательность полей, по которым по умолчанию будет выполняться сортировка записей.

Если мы теперь сохраним исправленный модуль и обновим открытую в веб-обозревателе страницу, мы увидим, что:

- в списке приложений наша модель обозначается надписью **Объявления** (значение атрибута `verbose_name_plural` вложенного класса `Meta`) вместо **Bbs**;

- на странице списка записей сама запись модели носит название **Объявление** (значение атрибута `verbose_name` вложенного класса `Meta`) вместо **Bb**;
- на страницах добавления и правки записи элементы управления имеют надписи **Товар**, **Описание** и **Цена** (значения параметра `verbose_name` конструкторов классов полей) вместо **title**, **content** и **price**.

Кстати, теперь мы можем исправить код контроллера `index()` (он, как мы помним, объявлен в модуле views.py пакета приложения), убрав из выражения, извлекающего список записей, указание сортировки:

```
def index(request):
    bbs = Bb.objects.all()
    . . .
```

Как видим, сортировка по умолчанию, заданная в параметрах модели, действует не только в административном сайте.

1.15. Редактор модели

Стало лучше, не так ли? Но до идеала все же далековато. Так, на странице списка записей все позиции представляются невразумительными строками вида "*<имя класса модели>* object (*<значение ключа>*)" (см. рис. 1.8), из которых невозможно понять, что же хранится в каждой из этих записей.

Таково представление модели на административном сайте по умолчанию. Если же оно нас не устраивает, мы можем задать свои параметры представления модели, объявив для нее класс-*редактор*.

Редактор объявляется в модуле административных настроек admin.py пакета приложения. Откроем его и заменим имеющийся в нем код тем, что представлен в листинге 1.14.

Листинг 1.14. Код модуля admin.py пакета приложения `bboard` (для модели `Bb` объявлен редактор `BbAdmin`)

```
from django.contrib import admin

from .models import Bb

class BbAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'price', 'published')
    list_display_links = ('title', 'content')
    search_fields = ('title', 'content', )

admin.site.register(Bb, BbAdmin)
```

Редактор объявляется как подкласс класса `ModelAdmin` из модуля `django.contrib.admin`. Он содержит набор атрибутов класса, которые и задают параметры представления модели. Мы использовали следующие атрибуты класса:

- `list_display` — последовательность имен полей, которые должны выводиться в списке записей;
- `list_display_links` — последовательность имен полей, которые должны быть преобразованы в гиперссылки, ведущие на страницу правки записи;
- `search_fields` — последовательность имен полей, по которым должна выполняться фильтрация.

У нас в списке записей будут присутствовать поля названия, описания продаваемого товара, его цены и даты публикации объявления. Значения, хранящиеся в первых двух полях, преобразуются в гиперссылки, посредством которых пользователь сможет попасть на страницы правки соответствующих записей. Фильтрация записей будет выполняться по тем же самым полям.

Обновим открытый в веб-обозревателе административный сайт и перейдем на страницу списка записей модели `Bb`. Она должна выглядеть так, как показано на рис. 1.10.

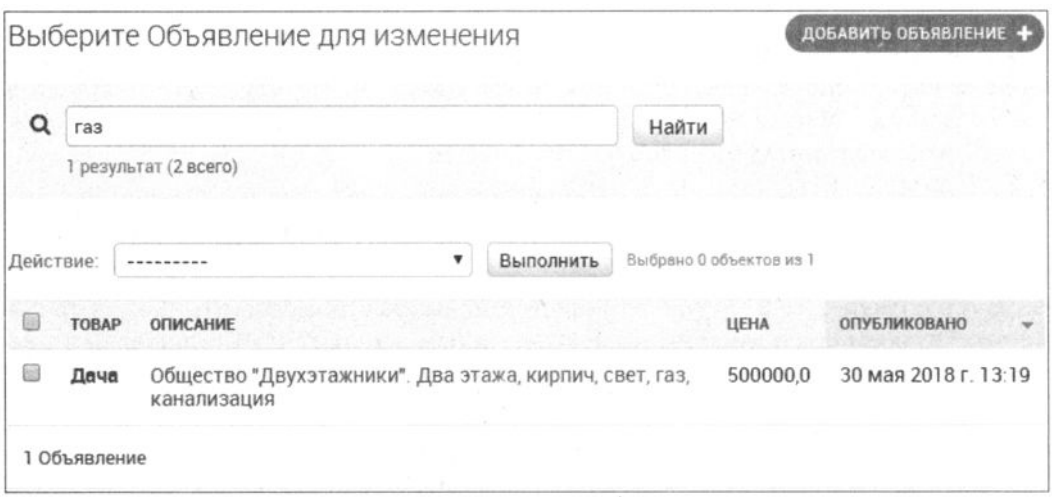


Рис. 1.10. Список записей модели `Bb` после указания для нее редактора (была выполнена фильтрация по слову «газ»)

Помимо всего прочего, мы можем выполнять фильтрацию записей по значениям полей, перечисленных в последовательности, которая была присвоена атрибуту `search_fields` класса редактора. Для этого достаточно занести в расположенное над списком поле ввода искомое слово и нажать расположенную правее кнопку **Найти**. Так, на рис. 1.10 показан список записей, отфильтрованных по слову «газ».

Теперь можно сделать небольшой перерыв. На досуге побродите по административному сайту, выясните, какие полезные возможности, помимо рассмотренных здесь, он предлагает. Можете также добавить в модель `Bb` еще пару объявлений.

ГЛАВА 2

Связи. Ввод данных. Статические файлы

Ладно, немного передохнули, повозились с административным сайтом — и достаточно. Хватит бездельничать! Пора заканчивать работу над электронной доской объявлений.

Предварительно выполним процедуру выхода с административного сайта. И остановим отладочный веб-сервер.

2.1. Связи между моделями

На всех приличных онлайновых досках объявлений все объявления разносятся по тематическим рубрикам: недвижимость, транспорт, бытовая техника и др. Давайте сделаем так и мы.

Сначала объявим класс модели `Rubric`, которая будет представлять рубрики объявлений. Допишем в модуль **models.py** пакета приложения `bboard` код из листинга 2.1.

Листинг 2.1. Код класса модели `Rubric`

```
class Rubric(models.Model):
    name = models.CharField(max_length=20, db_index=True,
    verbose_name='Название')

    class Meta:
        verbose_name_plural = 'Рубрики'
        verbose_name = 'Рубрика'
        ordering = ['name']
```

Модель очень проста — она содержит всего одно явно объявленное поле `name`, которое будет хранить название рубрики. Для этого поля мы сразу велели создать индекс, т. к. будем выводить перечень рубрик отсортированным по их названиям.

Теперь нам нужно добавить в модель `Bb` *поле внешнего ключа*, устанавливающее связь между текущей записью этой модели и записью модели `Rubric`, т. е. между

объявлением и рубрикой, к которой оно относится. Таким образом будет создана *связь «один-со-многими»*, при которой одна запись модели `Rubric` (рубрика) будет связана с произвольным количеством записей модели `Bb` (объявлений). Профессионалы в области баз данных скажут при этом, что модель `Rubric` станет первичной, а модель `Bb` — вторичной.

Создадим в последней такое поле, назвав его `rubric`. Для чего добавим непосредственно в код класса `Bb` следующее выражение (выделено полужирным шрифтом):

```
class Bb(models.Model):
    . . .
    rubric = models.ForeignKey('Rubric', null=True,
    on_delete=models.PROTECT, verbose_name='Рубрика')

    class Meta:
        . . .
```

Класс `ForeignKey` представляет поле внешнего ключа, в котором фактически будет храниться ключ записи из первичной модели. Первым параметром конструктору этого класса передается класс первичной модели в виде:

- ссылки на класс — если код, объявляющий класс первичной модели, располагается перед кодом класса вторичной модели;
- строки с именем класса — если вторичная модель объявлена раньше первичной (как у нас).

Все поля, создаваемые в моделях, по умолчанию обязательны к заполнению. Следовательно, добавить новое, обязательное к заполнению поле в модель, которая уже содержит записи, нельзя — сама СУБД откажется делать это и выведет сообщение об ошибке. Нам придется явно пометить поле `rubric` как необязательное, присвоив параметру `null` значение `True` — только после этого поле будет успешно добавлено в модель.

Именованный параметр `on_delete` управляет каскадными удалениями записей вторичной модели после удаления записи первичной модели, с которой они были связаны. Значение `PROTECT` этого параметра запрещает каскадные удаления (чтобы какой-нибудь несообразительный администратор не удалил разом сразу уйму объявлений).

Сохраним исправленный модуль и выполним генерирование миграций, которые внесут необходимые изменения в структуры базы данных:

```
manage.py makemigrations bboard
```

В результате в папке `migrations` будет создан модуль миграции с именем вида `0002_auto_`*<отметка текущих даты и времени>*`.py`. Если хотите, откройте его и посмотрите на его код (который слишком велик, чтобы приводить его здесь). Если вкратце, то новая миграция создаст таблицу для модели `Rubric` и добавит в таблицу модели `Bb` новое поле `rubric`.

На заметку

Помимо всего прочего, эта миграция задаст для полей модели `Bb` параметры `verbose_name`, а для самой модели — параметры `verbose_name_plural`, `verbose_name` и `ordering`, которые мы указали в *главе 1*. Впрочем, скорее всего, это делается, как говорится, для галочки — подобные изменения, произведенные в классе модели, в действительности никак не отражаются на базе данных.

Выполним созданную миграцию:

```
manage.py migrate
```

Сразу же зарегистрируем новую модель на административном сайте, добавив в модуль admin.py пакета приложения такие два выражения:

```
from .models import Rubric
admin.site.register(Rubric)
```

Запустим отладочный веб-сервер, войдем на административный сайт и добавим в модель `Rubric` пару рубрик.

2.2. Строковое представление модели

Все хорошо, вот только в списке записей модели `Rubric` все рубрики представляются строками вида "*<имя класса модели>* object (*<значение ключа записи>*)" (нечто подобное поначалу выводилось у нас в списке записей модели `Bb` — см. рис. 1.8). Работать с таким списком крайне неудобно, так что давайте что-то с ним сделаем.

Можно объявить для модели `Rubric` класс редактора и задать в нем перечень полей, которые должны выводиться в списке (см. *разд. 1.15*). Но этот способ лучше подходит только для моделей с несколькими значащими полями, а в нашей модели такое поле всего одно.

Еще можно переопределить в классе модели метод `__str__()`, возвращающий строковое представление класса. Давайте так и сделаем.

Откроем модуль models.py, если уже закрыли его, и добавим в объявление класса модели `Rubric` вот этот код (выделен полужирным шрифтом):

```
class Rubric(models.Model):
    . . .
    def __str__(self):
        return self.name

    class Meta:
        . . .
```

В качестве строкового представления мы выводим название рубрики (собственно, более там выводить нечего).

Сохраним файл, обновим страницу списка рубрик в административном сайте и посмотрим, что у нас получилось. Совсем другой вид (рис. 2.1)!

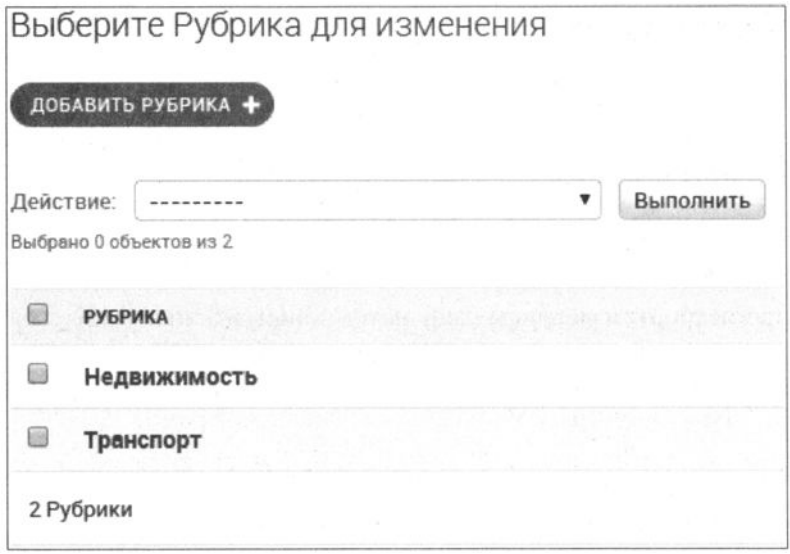


Рис. 2.1. Список рубрик (выводятся строковые представления записей модели)

Перейдем на страницу списка записей модели `Bb` и исправим каждое имеющееся в ней объявление, задав для него соответствующую рубрику. Обратим внимание, что на странице правки записи рубрика выбирается с помощью раскрывающегося списка, в котором в качестве пунктов перечисляются строковые представления рубрик (рис. 2.2).

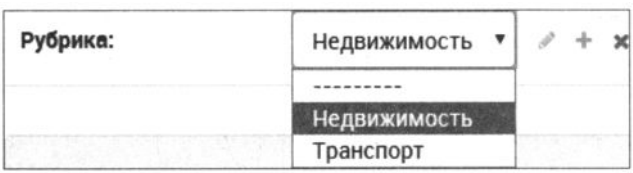


Рис. 2.2. Для указания значения поля внешнего ключа применяется раскрывающийся список

Осталось сделать так, чтобы в списке записей модели `Bb`, помимо всего прочего, выводились рубрики объявлений. Для чего достаточно добавить в последовательность имен полей, присвоенную атрибуту `list_display` класса `BbAdmin`, поле `rubric`:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'price', 'published', 'rubric')
    . . .
```

Обновим страницу списка объявления — и сразу увидим в нем новый столбец **Рубрика**. Отметим, что и здесь в качестве значения поля выводится строковое представление связанной записи модели.

2.3. URL-параметры и параметризованные запросы

Логичным шагом выглядит разбиение объявлений по рубрикам не только при хранении, но и при выводе на экран. Давайте создадим на пока что единственной странице нашего сайта панель навигации, в которой выведем список рубрик, и при щелчке на какой-либо рубрике будем выводить лишь относящиеся к ней объявления. Остановим отладочный сервер и подумаем.

Чтобы контроллер, который будет отфильтровывать из модели объявления, относящиеся к выбранной рубрике, смог, собственно, их отфильтровать, он должен получить ключ рубрики. Этот ключ удобнее всего передать в составе интернет-адреса, через GET-параметр: **/bboard/?rubric=*<ключ рубрики>***.

Однако Django предлагает нам другую возможность выполнить передачу параметров в контроллер — непосредственно в составе пути интернет-адреса: **bboard/rubric/*<ключ рубрики>*** или даже **bboard/*<ключ рубрики>***. То есть через *URL-параметры*.

Для этого нужно указать маршрутизатору, какую часть пути считать URL-параметром, каков тип значения этого параметра, и какое имя должно быть у параметра контроллера, которому будет присвоено значение этого URL-параметра. Давайте сделаем все это, открыв модуль urls.py пакета приложения `bboard` и внеся в него такие правки (добавленный код выделен полужирным шрифтом):

```
from .views import index, by_rubric

urlpatterns = [
    path('<int:rubric_id>/', by_rubric),
    path('', index),
]
```

Мы добавили в начало набора маршрутов еще один, с шаблонным интернет-адресом `<int:rubric_id>/`. В нем угловые скобки обозначают описание URL-параметра, языковая конструкция `int` — целочисленный тип этого параметра, а `rubric_id` — имя параметра контроллера, которому будет присвоено значение этого URL-параметра. Созданному маршруту мы сопоставили контроллер-функцию `by_rubric()`, который вскоре напишем.

Кстати, маршруты, содержащие URL-параметры, носят название *параметризованных*.

Откроем модуль views.py и добавим в него код контроллера-функции `by_rubric()` (листинг 2.2).

Листинг 2.2. Код контроллера-функции `by_rubric()`

```
from .models import Rubric

def by_rubric(request, rubric_id):
    bbs = Bb.objects.filter(rubric=rubric_id)
```

```
    rubrics = Rubric.objects.all()
    current_rubric = Rubric.objects.get(pk=rubric_id)
    context = {'bbs': bbs, 'rubrics': rubrics,
    'current_rubric': current_rubric}
    return render(request, 'bboard/by_rubric.html', context)
```

В объявлении функции мы добавили параметр `rubric_id` — именно ему будет присвоено значение URL-параметра, выбранное из интернет-адреса. В состав контекста шаблона мы поместили список объявлений, отфильтрованных по полю внешнего ключа `rubric`, список всех рубрик и текущую рубрику (она нужна нам, чтобы вывести на странице ее название). Остальное нам уже знакомо.

Создадим в папке templates\bboard пакета приложения шаблон by_rubric.html с кодом, показанным в листинге 2.3.

Листинг 2.3. Код шаблона bboard\by_rubric.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8">
        <title>{{ current_rubric.name }} - Доска объявлений</title>
    </head>
    <body>
        <h1>Объявления</h1>
        <h2>Рубрика: {{ current_rubric.name }}</h2>
        <div>
            <a href="/bboard/">Главная</a>
            {% for rubric in rubrics %}
            <a href="/bboard/{{ rubric.pk }}/">{{ rubric.name }}</a>
            {% endfor %}
        </div>
        {% for bb in bbs %}
        <div>
            <h2>{{ bb.title }}</h2>
            <p>{{ bb.content }}</p>
            <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
        </div>
        {% endfor %}
    </body>
</html>
```

Обратим внимание, как формируются интернет-адреса в гиперссылках, находящихся в панели навигации и представляющих отдельные рубрики.

Теперь исправим контроллер `index()` и шаблон bboard\index.html таким образом, чтобы на главной странице нашего сайта выводилась та же самая панель навигации.

Помимо этого, сделаем так, чтобы в составе каждого объявления выводилось название рубрики, к которой оно относится, выполненное в виде гиперссылки. Исправленный код показан в листингах 2.4 и 2.5 соответственно.

Листинг 2.4. Код контроллера-функции index() (реализован вывод панели навигации)

```
def index(request):
    bbs = Bb.objects.all()
    rubrics = Rubric.objects.all()
    context = {'bbs': bbs, 'rubrics': rubrics}
    return render(request, 'bboard/index.html', context)
```

Листинг 2.5. Код шаблона bboard\index.html (реализован вывод панели навигации и рубрики, к которой относится каждое объявление)

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8">
        <title>Главная - Доска объявлений</title>
    </head>
    <body>
        <h1>Объявления</h1>
        <div>
            <a href="/bboard/">Главная</a>
            {% for rubric in rubrics %}
            <a href="/bboard/{{ rubric.pk }}/">{{ rubric.name }}</a>
            {% endfor %}
        </div>
        {% for bb in bbs %}
        <div>
            <h2>{{ bb.title }}</h2>
            <p>{{ bb.content }}</p>
            <p><a href="/bboard/{{ bb.rubric.pk }}/">
            {{ bb.rubric.name }}</a></p>
            <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
        </div>
        {% endfor %}
    </body>
</html>
```

Сохраним исправленные файлы, запустим отладочный веб-сервер и перейдем на главную страницу. Мы сразу увидим панель навигации, располагающуюся непосредственно под заголовком, и гиперссылки на рубрики, к которым относятся объявления (рис. 2.3). Перейдем по какой-либо гиперссылке-рубрике и посмотрим на страницу со списком объявлений, относящихся к этой рубрике (рис. 2.4).

Рис. 2.3. Исправленная главная страница с панелью навигации и обозначениями рубрик

Рис. 2.4. Страница объявлений, относящихся к выбранной рубрике

2.4. Обратное разрешение интернет-адресов

Все просто замечательно! За исключением одного: если мы решим изменить шаблонный интернет-адрес в списке маршрутов, нам придется вносить уйму правок и в код шаблонов. Давайте посмотрим еще раз на листинги 2.3 и 2.5 — интернет-адреса гиперссылок в них формируются непосредственно в коде шаблона. Вдобавок в коде, формирующем интернет-адреса, нетрудно допустить ошибки...

Мы можем избежать всех этих неприятностей, если используем инструмент Django, называемый *обратным разрешением* интернет-адресов. Его суть: мы указываем маршрут, формирующий нужный нам интернет-адрес, и, если это параметризованный маршрут, значения URL-параметров, а Django сам генерирует на основе всего этого правильный интернет-адрес.

Чтобы реализовать обратное разрешение, нам следует выполнить два действия. Первое: дать нужным маршрутам имена, создав тем самым *именованные маршруты*. Откроем модуль urls.py пакета приложения и исправим код, создающий набор маршрутов, следующим образом (добавленный код выделен полужирным шрифтом):

```
urlpatterns = [
    path('<int:rubric_id>/', by_rubric, name='by_rubric'),
    path('', index, name='index'),
]
```

Как видим, имя маршрута указывается в именованном параметре `name` функции `path()`.

Второе действие: использование для создания интернет-адресов в гиперссылках тегов шаблонизатора `url`. Откроем шаблон bboard\index.html, найдем в нем фрагмент кода:

```
<a href="/bboard/{{ rubric.pk }}/">
```

и заменим его на:

```
<a href="{% url 'by_rubric' rubric.pk %}">
```

Имя маршрута указывается первым параметром тега `url`, а значение URL-параметра, которое нужно вставить в результирующий интернет-адрес, — вторым.

Теперь найдем код, создающий гиперссылку, которая указывает на главную страницу:

```
<a href="/bboard/">
```

Заменим его на:

```
<a href="{% url 'index' %}">
```

Маршрут `index` не является параметризованным, поэтому никакие URL-параметры здесь не указываются.

Внесем аналогичные правки во все остальные фрагменты кода обоих наших шаблонов. Обновим страницу, открытую в веб-обозревателе, и попробуем выполнить несколько переходов по гиперссылкам. Все должно работать.

2.5. Формы, связанные с моделями

Осталось создать еще одну страницу — для добавления в базу данных новых объявлений.

Для ввода данных язык HTML предлагает так называемые веб-формы. Их создание и, самое главное, обработка на стороне сервера — довольно сложный и кропотливый процесс. Но мы ведь пользуемся Django, лучшим в мире веб-фреймворком, и не должны беспокоиться о таких мелочах!

Сначала мы объявим класс *формы, связанной с моделью*. Такая форма «умеет» генерировать теги, что создадут входящие в состав формы элементы управления, проверять на корректность введенные данные и, наконец, сохранять их в модели, с которой она связана.

Давайте создадим такую форму, связанную с моделью `Bb`. Дадим ее классу имя `BbForm`.

Остановим отладочный веб-сервер. Создадим в пакете приложения `bboard` модуль forms.py, в который занесем код из листинга 2.6.

Листинг 2.6. Код класса `BbForm` — формы, связанной с моделью `Bb`

```
from django.forms import ModelForm

from .models import Bb

class BbForm(ModelForm):
    class Meta:
        model = Bb
        fields = ('title', 'content', 'price', 'rubric')
```

Класс формы, связанной с моделью, является производным от класса `ModelForm` из модуля `django.forms`. В классе формы мы объявили вложенный класс `Meta`, в котором указали параметры нашей формы: класс модели, с которой она связана (атрибут класса `model`), и последовательность из имен полей модели, которые должны присутствовать в форме (атрибут класса `fields`).

2.6. Контроллеры-классы

Обрабатывать формы, связанные с моделью, можно в знакомых нам контроллерах-функциях. Но давайте применим для этого высокоуровневый контроллер-класс, который возьмет бóльшую часть действий по выводу и обработке формы на себя.

Наш первый контроллер-класс будет носить имя `BbCreateView`. Его мы объявим в модуле views.py пакета приложения. Код класса можно увидеть в листинге 2.7.

Листинг 2.7. Код контроллера-класса BbCreateView

```
from django.views.generic.edit import CreateView

from .forms import BbForm

class BbCreateView(CreateView):
    template_name = 'bboard/create.html'
    form_class = BbForm
    success_url = '/bboard/'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Контроллер-класс мы сделали производным от класса `CreateView` из модуля `django.views.generic.edit`. Базовый класс реализует функциональность по созданию формы, выводу ее на экран с применением указанного шаблона, получению занесенных в форму данных, проверке их на корректность, сохранению их в новой записи модели и перенаправлению в случае успеха на интернет-адрес, который мы зададим. Вот какой полезный класс этот `CreateView`!

Все необходимые сведения мы указали в атрибутах объявленного класса:

- `template_name` — путь к файлу шаблона, что будет использован для вывода страницы с формой;
- `form_class` — сам класс формы, связанной с моделью;
- `success_url` — интернет-адрес, по которому будет выполнено перенаправление после успешного сохранения данных (в нашем случае это адрес главной страницы).

Собственно, этим можно было бы и ограничиться. Но у нас на каждой странице, не исключая и страницу добавления объявления, должна выводиться панель навигации, содержащая список рубрик. Следовательно, нам нужно добавить в контекст шаблона, формируемый классом `CreateView`, еще и этот список.

Метод `get_context_data()` этого класса формирует контекст шаблона. Мы переопределили метод, чтобы добавить в контекст дополнительные данные — список рубрик. В теле этого метода мы сначала получаем контекст шаблона от метода базового класса, затем добавляем в него список рубрик и, наконец, возвращаем в качестве результата.

Займемся шаблоном bboard\create.html. Его код представлен в листинге 2.8.

Листинг 2.8. Код шаблона bboard\create.html

```
<!DOCTYPE html>
<html>
```

```
    <head>
        <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8">
        <title>Добавление объявления - Доска объявлений</title>
    </head>
    <body>
        <h1>Добавление объявления</h1>
        <div>
            <a href="{% url 'index' %}">Главная</a>
            {% for rubric in rubrics %}
            <a href="{% url 'by_rubric' rubric.pk %}">
            {{ rubric.name }}</a>
            {% endfor %}
        </div>
        <form method="post">
            {% csrf_token %}
            {{ form.as_p }}
            <input type="submit" value="Добавить">
        </form>
    </body>
</html>
```

Здесь нас интересует только код, создающий веб-форму. Обратим внимание на четыре важных момента:

- форма в контексте шаблона хранится в переменной `form`. Эту переменную создает базовый класс `CreateView`;
- для вывода формы, чьи элементы управления находятся на отдельных абзацах, применяется метод `as_p()` класса `ModelForm`;
- метод `as_p()` генерирует только код, создающий элементы управления. Тег `<form>`, необходимый для создания самой формы, и тег `<input>`, формирующий кнопку отправки данных, нам придется вставить самостоятельно.

  В теге `<form>` мы указали метод отправки данных POST, но не записали интернет-адрес, по которому будут отправлены занесенные в форму данные. В этом случае данные будут отправлены по тому же интернет-адресу, с которого была загружена текущая страница, т. е., в нашем случае, тому же контроллеру-классу `BbCreateView`, который благополучно обработает и сохранит их;
- в теге `<form>` мы поместили тег шаблонизатора `csrf_token`. Он создает в форме скрытое поле, хранящее цифровой жетон, получив который, контроллер «поймет», что данные были отправлены с текущего сайта, и им можно доверять. Это часть подсистемы обеспечения безопасности Django.

Добавим в модуль urls.py пакета приложения маршрут, указывающий на контроллер `CreateView`:

```
. . .
from .views import BbCreateView

urlpatterns = [
    path('add/', BbCreateView.as_view(), name='add'),
    . . .
]
```

В вызов функции `path()` в таком случае подставляется не ссылка на сам контроллер-класс, а результат, возвращенный методом `as_view()` контроллера-класса.

Что-то мы забыли... Ах да, нужно же создать в панели навигации всех страниц гиперссылку, ведущую на страницу добавления объявления. Вот ее код:

```
<a href="{% url 'add' %}">Добавить</a>
```

Запустим отладочный веб-сервер, откроем сайт и щелкнем на гиперссылке **Добавить**. На странице добавления объявления (рис. 2.5) введем новое объявление и нажмем кнопку **Добавить**. Когда на экране появится главная страница, мы сразу же увидим новое объявление.

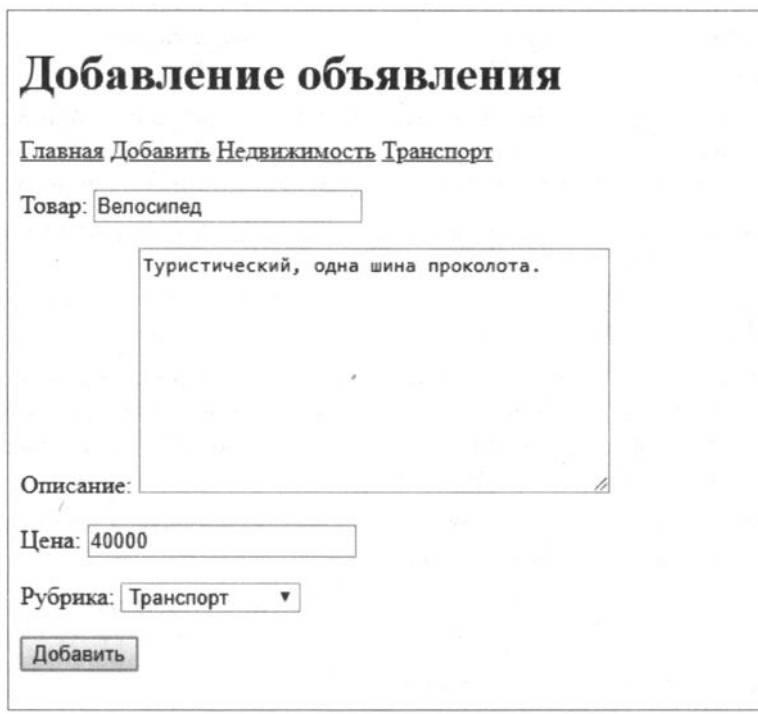


Рис. 2.5. Страница добавления нового объявления

Все просто замечательно, но, увы, не идеально. В объявлении класса `BbCreateView` мы опять указали интернет-адрес перенаправления (он записан в атрибуте класса `success_url`) непосредственно, что считается дурным тоном программирования. Давайте сгенерируем его путем обратного разрешения.

Откроем модель views.py и внесем следующие правки в код класса `BbCreateView`:

```
from django.urls import reverse_lazy

class BbCreateView(CreateView):
    . . .
    success_url = reverse_lazy('index')
    . . .
```

Функция `reverse_lazy()` из модуля `django.urls` в качестве параметров принимает имя маршрута и значения всех входящих в маршрут URL-параметров (если они там есть). Результатом станет готовый интернет-адрес.

2.7. Наследование шаблонов

Давайте еще раз посмотрим на листинги 2.3, 2.5 и 2.8. Что сразу бросается в глаза? Правильно, большой объем совершенно одинакового HTML-кода: секция заголовка, панель навигации, всевозможные служебные теги. Мало того, что это увеличивает совокупный объем шаблонов, так еще и усложняет сопровождение, — если мы решим изменить что-либо в этом коде, то будем вынуждены вносить правки в каждый из имеющихся у нас шаблонов.

Можно ли решить эту проблему? Разумеется, можно. Django предоставляет замечательный механизм, называемый *наследованием шаблонов*. Он аналогичен наследованию классов, применяемому в Python-программировании сплошь и рядом.

Шаблон, являющийся базовым, объявляет в составе своего содержимого так называемые *блоки*. Они определяют место в шаблоне, куда будет вставлено содержимое, извлеченное из шаблонов, которые станут производными по отношению к базовому. Каждый из блоков имеет уникальное в пределах шаблона имя.

Создадим в папке templates, что находится в папке пакета приложения `bboard` и хранит все шаблоны, вложенную папку layout. В нее мы создадим файл базового шаблона basic.html, в который вынесем весь одинаковый код из остальных шаблонов. Объявим в базовом шаблоне следующие блоки:

- `title` — будет помещаться в теге `<title>` и использоваться для создания уникального названия для каждой страницы;
- `content` — будет использоваться для размещения уникального содержимого страниц.

Код шаблона layout\basic.html можно увидеть в листинге 2.9.

Листинг 2.9. Код базового шаблона layout\basic.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
```

```
        content="text/html; charset=utf-8">
        <title>{% block title %}Главная{% endblock %} -
        Доска объявлений</title>
    </head>
    <body>
        <header>
            <h1>Объявления</h1>
        </header>
        <nav>
            <a href="{% url 'index' %}">Главная</a>
            <a href="{% url 'add' %}">Добавить</a>
            {% for rubric in rubrics %}
            <a href="{% url 'by_rubric' rubric.pk %}">
            {{ rubric.name }}</a>
            {% endfor %}
        </nav>
        <section>
        {% block content %}
        {% endblock %}
        </section>
    </body>
</html>
```

Начало объявляемого блока помечается тегом шаблонизатора `block`, за которым должно следовать имя блока. Завершается блок тегом `endblock`.

Объявленный в базовом шаблоне блок может быть как пустым:

```
{% block content %}
{% endblock %}
```

так и иметь какое-либо изначальное содержимое:

```
{% block title %}Главная{% endblock %}
```

Это содержимое будет присутствовать в блоке, если производный шаблон не задаст для него свое содержимое.

Теперь перепишем шаблон bboard\index.html таким образом, чтобы он стал производным от шаблона layout\basic.html. Новый код этого шаблона показан в листинге 2.10.

Листинг 2.10. Код производного шаблона bboard\index.html

```
{% extends "layout/basic.html" %}

{% block content %}
{% for bb in bbs %}
<div class="b">
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
```

```
    <p><a href="{% url 'by_rubric' bb.rubric.pk %}">
    {{ bb.rubric.name }}</a></p>
    <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
{% endblock %}
```

В самом начале кода любого производного шаблона должен стоять тег шаблонизатора `extends`, после которого записывается путь к базовому шаблону. Далее следуют объявления блоков, обозначаемые теми же тегами `block` и `endblock`, в которых записывается их содержимое.

Если мы теперь сохраним исправленные файлы и обновим открытую в веб-обозревателе главную страницу, то увидим, что она выводится точно так же, как ранее.

Аналогично исправим шаблоны bboard\by_rubric.html (листинг 2.11) и bboard\create.html (листинг 2.12).

Листинг 2.11. Код производного шаблона bboard\by_rubric.html

```
{% extends "layout/basic.html" %}

{% block title %}{{ current_rubric.name }}{% endblock %}

{% block content %}
<h2>Рубрика: {{ current_rubric.name }}</h2>
{% for bb in bbs %}
<div>
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
    <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
{% endblock %}
```

Листинг 2.12. Код производного шаблона bboard\create.html

```
{% extends "layout/basic.html" %}

{% block title %}Добавление объявления{% endblock %}

{% block content %}
<h2>Добавление объявления</h2>
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Добавить">
```

```
</form>
{% endblock %}
```

Очевидно, что применение наследования на практике позволит радикально уменьшить объем кода шаблонов.

2.8. Статические файлы

Наш первый сайт практически готов. Осталось лишь навести небольшой лоск. И наведем мы его, применив таблицу стилей style.css, чей код показан в листинге 2.13.

Листинг 2.13. Таблица стилей style.css

```
header h1 {
    font-size: 40pt;
    text-transform: uppercase;
    text-align: center;
    background: url("bg.jpg") left / auto 100% no-repeat;
}
nav {
    font-size: 16pt;
    width: 150px;
    float: left;
}
nav a {
    display: block;
    margin: 10px 0px;
}
section {
    margin-left: 170px;
}
```

Но где нам сохранить эту таблицу стилей и файл с фоновым изображением bg.jpg? И, вообще, как привязать таблицу стилей к базовому шаблону?

Файлы, содержимое которых не обрабатывается программно, а пересылается клиенту как есть, в терминологии Django носят название *статических*. К таким файлам относятся, например, таблицы стилей и графические изображения, помещаемые на страницы.

По умолчанию Django требует, чтобы статические файлы, относящиеся к определенному приложению, помещались в папке static пакета этого приложения (такое поведение можно изменить в настройках проекта — см. подробности в *главе 11*).

Остановим отладочный веб-сервер. Создадим в папке пакета приложения `bboard` папку static, а в ней — вложенную папку bboard. В последней сохраним файлы

style.css и bg.jpg (можно использовать любое подходящее изображение, загруженное из Интернета).

Откроем базовый шаблон layout\basic.html и вставим в него вот такой дополнительный код (выделен полужирным шрифтом):

```
{% load static %}

<!DOCTYPE html>
<html>
    <head>
        . . .
        <link rel="stylesheet" type="text/css"
        href="{% static 'bboard/style.css' %}">
        . . .
    </head>
    . . .
</html>
```

Шаблонизатор Django поддерживает много директив, тегов и фильтров. Часть их, применяемая наиболее часто, встроена в само программное ядро шаблонизатора и доступна для использования сразу, без каких-либо дополнительных действий. Другая часть, применяемая, по мнению разработчиков фреймворка, не столь часто, реализована в загружаемых *модулях расширения*, которые нужно предварительно загрузить.

Предварительная загрузка модуля расширения выполняется тегом шаблонизатора `load`, за которым указывается имя нужного модуля. Так, в нашем примере загружается модуль `static`, призванный обеспечивать поддержку статических файлов в шаблонах.

Чтобы сформировать интернет-адрес статического файла, нужно использовать тег шаблонизатора `static`. В качестве его параметра указывается путь к нужному статическому файлу относительно папки static. Например, в приведенном ранее примере мы помещаем в тег `<link>`, привязывающий таблицу стилей, интернет-адрес файла bboard\style.css.

Собственно, на этом все. Запустим отладочный сервер и откроем главную страницу сайта. Теперь она выглядит гораздо презентабельнее (рис. 2.6).

Занесем на сайт еще несколько объявлений (специально придумывать текст для них совершенно необязательно — для целей отладки можно записать любую тарабарщину, как это сделал автор). Попробуем при вводе очередного объявления не вводить какое-либо обязательное для занесения значение — скажем, название товара, и посмотрим, что случится. Попробуем добавить на страницы, например, поддон.

И закончив тем самым вводный курс Django-программирования, приступим к изучению базовых возможностей этого замечательного веб-фреймворка.

Рис. 2.6. Главная страница сайта после применения к ней таблицы стилей

ЧАСТЬ II

Базовые инструменты Django

ГЛАВА 3

Создание и настройка проекта

Прежде чем начать писать код сайта, следует создать проект этого сайта, указать его настройки и сформировать все необходимые приложения. Всему этому будет посвящена текущая глава.

3.1. Подготовительные действия

Перед началом разработки сайта на Django, даже перед созданием его проекта, следует выполнять действия, перечисленные далее.

1. Установить исполняющую среду языка Python. Django 2.1 требует для работы версию не старше 3.5 (Python 2 не поддерживается).

   Дистрибутив исполняющей среды Python можно загрузить со страницы **https://www.python.org/downloads/**.

2. Установить сам фреймворк Django. Сделать это можно отдачей в командной строке следующей команды:

   ```
   pip install django
   ```

   Описание других способов установки Django можно найти по интернет-адресу **https://docs.djangoproject.com/en/2.1/topics/install/**.

3. Установить СУБД для работы с базой данных выбранного формата:

   - SQLite — никаких дополнительных программ устанавливать не нужно, поскольку программное ядро этой СУБД поставляется непосредственно в составе исполняющей среды Python.

     База данных SQLite создается автоматически при первой попытке обращения к ней. Применительно к Django база данных будет создана при первом же запуске отладочного веб-сервера, встроенного во фреймворк;

   - MySQL — понадобится установить клиент и, если база данных будет находиться на том же компьютере, что и сайт, сервер этой СУБД. Поддерживаются MySQL 5.6 и более новые версии.

Дистрибутивный комплект СУБД MySQL, содержащий все необходимые программы, включая сервер и клиент, и доступный в разных редакциях, можно найти по интернет-адресу **https://dev.mysql.com/downloads/**.

Кроме того, следует установить Python-библиотеку mysqlclient, служащую коннектором (соединителем) между Python и MySQL. Поддерживаются версии mysqlclient 1.3.7 или более новые. Установить эту библиотеку можно отдачей команды:

```
pip install mysqlclient
```

Внимание!

В составе MySQL поставляется собственный коннектор такого рода, носящий название Connector/Python и не требующий наличия на компьютере ни клиента MySQL, ни каких-либо дополнительных Python-библиотек. Однако разработчики Django по какой-то причине не рекомендуют применять его.

База данных формата MySQL не будет создана автоматически, и ее понадобится создать вручную, равно как и пользователя, от имени которого Django будет подключаться к ней. Пользователь должен иметь полные права на доступ к базе данных, включая права на создание в ней таблиц, индексов и связей, их правку и удаление. Для создания базы данных и пользователя можно использовать поставляемую в составе СУБД программу MySQL Workbench.

- PostgreSQL — понадобится установить, прежде всего, Python-коннектор этой СУБД, называемый psycorg, который можно найти по интернет-адресу **http://www.psycopg.org/psycopg/**. Требуется версия psycorg 2.5.4 или более новая.

  Если база данных будет находиться на том же компьютере, что и сайт, нужно установить сервер, доступный на странице **https://www.openscg.com/bigsql/postgresql/installers.jsp/** (кстати, в составе имеющегося там дистрибутивного комплекта есть и Python-коннектор). Поддерживаются PostgreSQL 9.4 и более новые версии.

  База данных формата PostgreSQL не будет создана автоматически, и ее понадобится создать вручную, равно как и пользователя, от имени которого Django будет подключаться к ней. Пользователь должен иметь полные права на доступ к базе данных, включая права на создание в ней таблиц, индексов и связей, их правку и удаление. Для создания базы данных и пользователя можно использовать инструментальные средства, поставляемые в составе дистрибутива этой СУБД.

Также Django поддерживает работу с базами данных Oracle, Microsoft SQL Server, Firebird, IBM DB2, SAP SQL Anywhere и механизмом ODBC. Действия, которые необходимо выполнить для успешного подключения к таким базам, описаны на странице **https://docs.djangoproject.com/en/2.1/ref/databases/**.

3.2. Создание проекта Django

Создание нового проекта Django выполняется командой `startproject` утилиты django-admin, отдаваемой в следующем формате:

```
django-admin startproject <имя проекта> [<путь к папке проекта>]
```

Если в вызове команды `startproject` задано только *имя проекта*, в текущей папке будет создана папка проекта, имеющая то же имя, что и сам проект. Если же дополнительно указать *путь к папке проекта*, расположенная по этому пути папка станет папкой создаваемого проекта.

В любом случае в папку проекта будут записаны, во-первых, служебная утилита manage.py, предназначенная для выполнения различных действий над проектом, а во-вторых, пакет конфигурации, чье имя совпадает с именем проекта. Модули, помещаемые в пакет конфигурации, были рассмотрены в *разд. 1.2*.

Папка проекта может быть впоследствии перемещена в любое другое место файловой системы, а также переименована. Никакого влияния на работу проекта эти действия не оказывают.

3.3. Настройки проекта

На работу проекта Django значительное влияние оказывают многочисленные настройки, указываемые в модуле settings.py пакета конфигурации. Впрочем, бóльшая их часть имеет значения по умолчанию, которые являются оптимальными в большинстве случаев.

Все эти параметры хранятся в переменных. Имя переменной — и есть имя соответствующего ей параметра.

3.3.1. Основные настройки

Здесь приведены основные настройки, затрагивающие ключевую функциональность проекта:

- `BASE_DIR` — задает путь к папке проекта. По умолчанию этот путь вычисляется автоматически;
- `DEBUG` — указывает режим работы сайта: отладочный (значение `True`) или эксплуатационный (`False`). Значение по умолчанию — `False` (эксплуатационный режим), однако сразу при создании проекта для этого параметра указано значение `True` (т. е. сайт для облегчения разработки сразу же вводится в отладочный режим).

  Если сайт работает в *отладочном режиме*, при возникновении любой ошибки в коде сайта Django выводит веб-страницу с детальным описанием этой ошибки и сведениями, позволяющими понять ее причину. В *эксплуатационном режиме* в таких случаях выводятся стандартные сообщения веб-сервера наподобие «Страница не найдена» или «Внутренняя ошибка сервера». Помимо того, в эксплуатационном режиме действуют более строгие настройки безопасности;

- DEFAULT_CHARSET — кодировка веб-страниц по умолчанию. Значение этого параметра по умолчанию: utf-8;
- FILE_CHARSET — кодировка текстовых файлов, в частности, файлов шаблонов. Значение по умолчанию: utf-8;
- ROOT_URLCONF — путь к модулю, в котором записаны маршруты уровня проекта, в виде строки. Значение этого параметра указывается сразу при создании проекта;
- SECRET_KEY — секретный ключ, представляющий собой строку с произвольным набором символов. Активно используется программным ядром Django и подсистемой разграничения доступа для шифрования важных данных.

  Значение параметра по умолчанию — пустая строка. Однако непосредственно при создании проекта ему присваивается секретный ключ, сгенерированный утилитой django-admin.

  Менять этот секретный ключ без особой необходимости не стоит. Также его следует хранить в тайне, в противном случае он может попасть в руки злоумышленникам, которые используют его для атаки на сайт.

3.3.2. Параметры баз данных

Все базы данных, используемые проектом, записываются в параметре DATABASES. Его значением должен быть словарь Python. Ключи элементов этого словаря задают псевдонимы баз данных, зарегистрированных в проекте. Можно указать произвольное количество баз данных. База с псевдонимом default будет использоваться, если при выполнении операций с моделями база данных не указана явно.

В качестве значений элементов словаря также указываются словари, хранящие, собственно, параметры соответствующей базы данных. Каждый элемент вложенного словаря указывает отдельный параметр.

Значение параметра DATABASES по умолчанию — пустой словарь. Однако при создании проекта ему дается следующее значение:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Оно указывает единственную базу данных, применяемую по умолчанию. База записывается в формате SQLite и хранится в файле db.sqlite3, что находится в папке проекта.

Теперь поговорим о параметрах баз данных, поддерживаемых Django:

- ENGINE — формат используемой базы данных. Указывается как путь к модулю, реализующему работу с нужным форматом баз данных, в виде строки. Доступны следующие значения:

- `django.db.backends.sqlite3` — SQLite;
- `django.db.backends.mysql` — MySQL;
- `django.db.backends.postgresql` — PostgreSQL;
- `django.db.backends.oracle` — Oracle;

- `NAME` — путь к файлу базы данных, если используется SQLite, или имя базы данных в случае серверных СУБД;
- `TIME_ZONE` — временна́я зона для значений даты и времени, хранящихся в базе. Используется в том случае, если формат базы данных не поддерживает хранение значений даты и времени с указанием временно́й зоны. Значение по умолчанию — `None` (значение временно́й зоны берется из одноименного параметра проекта, описанного в *разд. 3.3.5*).

Следующие параметры используются только в случае серверных СУБД:

- `HOST` — интернет-адрес компьютера, на котором работает СУБД;
- `PORT` — номер TCP-порта, через который выполняется подключение к СУБД. Значение по умолчанию — пустая строка (используется порт по умолчанию);
- `USER` — имя пользователя, от имени которого Django подключается к базе данных;
- `PASSWORD` — пароль пользователя, от имени которого Django подключается к базе;
- `CONN_MAX_AGE` — время, в течение которого соединение с базой данных будет открыто, в виде целого числа в секундах. Если задано значение 0, соединение будет закрываться сразу после обработки запроса. Если задано значение `None`, соединение будет открыто всегда. Значение по умолчанию — 0;
- `OPTIONS` — дополнительные параметры подключения к базе данных, специфичные для используемой СУБД. Записываются в виде словаря, в котором каждый элемент указывает отдельный параметр. Значение по умолчанию — пустой словарь.

Вот пример кода, задающего параметры для подключения к базе данных `site` формата MySQL, обслуживаемой СУБД, которая работает на том же компьютере, пользователя под именем `siteuser` с паролем `sitepassword`:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'HOST': 'localhost',
        'USER': 'siteuser',
        'PASSWORD': 'sitepassword',
        'NAME': 'site'
    }
}
```

Внимание!

В абсолютном большинстве случаев веб-сайты хранят данные в одной базе, поэтому в книге будет описана работа только с одной базой данных. Конфигурирование Django для использования нескольких баз данных и последующая работа с ними описана на странице https://docs.djangoproject.com/en/2.1/topics/db/multi-db/.

3.3.3. Список зарегистрированных приложений

Список приложений, зарегистрированных в проекте, задается параметром `INSTALLED_APPS`. Все приложения, составляющие проект — как написанные самим разработчиком сайта, так и входящие в состав Django, — должны быть приведены в этом списке, в противном случае они не будут работать.

Значение этого параметра по умолчанию — пустой список. Однако сразу при создании проекта для него указывается следующий список:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Он содержит такие стандартные приложения:

- `django.contrib.admin` — реализует функциональность административного веб-сайта Django;
- `django.contrib.auth` — реализует работу встроенной во фреймворк подсистемы разграничения доступа. Используется административным сайтом (приложением `django.contrib.admin`);
- `django.contrib.contenttypes` — хранит список всех моделей, объявленных во всех приложениях сайта. Используется при создании полиморфных связей между моделями (см. *главу 16*), административным сайтом, подсистемой разграничения доступа (приложениями `django.contrib.admin` и `django.contrib.auth`);
- `django.contrib.sessions` — обеспечивает хранение данных клиента на стороне сервера в сессиях (см. *главу 22*). Требуется при задействовании сессий и используется административным сайтом (приложением `django.contrib.admin`);
- `django.contrib.messages` — выводит всплывающие сообщения (о них будет рассказано в *главе 22*). Требуется для обработки всплывающих сообщений и используется административным сайтом (приложением `django.contrib.admin`);
- `django.contrib.staticfiles` — реализует обработку статических файлов (см. *главу 11*). Необходимо, если в составе сайта имеются статические файлы.

Если какое-либо из указанных приложений не нужно для работы сайта, его можно удалить из этого списка. Также следует убрать используемые удаленным приложением посредники (о них мы скоро поговорим).

3.3.4. Список зарегистрированных посредников

Посредник (middleware) Django — это программный модуль, выполняющий предварительную обработку клиентского запроса перед передачей его контроллеру, а также окончательную обработку ответа, сгенерированного контроллером. Список посредников, зарегистрированных в проекте, указывается в параметре `MIDDLEWARE`.

Все посредники, используемые приложениями проекта — как написанные самим разработчиком сайта, так и входящие в состав Django, — должны быть приведены в этом списке. В противном случае они не будут работать.

Значение параметра `MIDDLEWARE` по умолчанию — пустой список. Однако сразу при создании проекта для него указывается следующий список:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Все эти посредники входят в состав фреймворка: Давайте их рассмотрим:

- `django.middleware.security.SecurityMiddleware` — реализует дополнительную защиту сайта от сетевых атак;
- `django.contrib.sessions.middleware.SessionMiddleware` — обеспечивает работу сессий на низком уровне. Используется подсистемами разграничения доступа, сессий и всплывающих сообщений (приложениями `django.contrib.auth`, `django.contrib.sessions` и `django.contrib.messages`);
- `django.middleware.common.CommonMiddleware` — участвует в предварительной обработке запросов;
- `django.middleware.csrf.CsrfViewMiddleware` — осуществляет защиту от межсайтовых запросов при обработке данных, переданных сайту HTTP-методом POST;
- `django.contrib.auth.middleware.AuthenticationMiddleware` — добавляет в объект запроса атрибут, хранящий текущего пользователя. Через этот атрибут в контроллерах и шаблонах можно выяснить, какой пользователь выполнил вход на сайт и выполнил ли вообще. Используется административным сайтом и подсистемой разграничения доступа (приложениями `django.contrib.admin` и `django.contrib.auth`);
- `django.contrib.messages.middleware.MessageMiddleware` — обеспечивает обработку всплывающих сообщений на низком уровне. Используется административным сайтом и подсистемой всплывающих сообщений (приложениями `django.contrib.admin` и `django.contrib.messages`);

- `django.middleware.clickjacking.XFrameOptionsMiddleware` — реализует дополнительную защиту сайта от сетевых атак.

Если какой-либо из этих посредников не используется ни одним из приложений сайта, его можно удалить из списка. Исключение составляют посредники `django.middleware.security.SecurityMiddleware`, `django.middleware.common.CommonMiddleware`, `django.middleware.clickjacking.XFrameOptionsMiddleware` и, если сайт получает данные от посетителей, `django.middleware.csrf.CsrfViewMiddleware` — их удалять не стоит.

3.3.5. Языковые настройки

Теперь рассмотрим настройки, касающиеся языка, на котором будут выводиться всевозможные системные сообщения и страницы административного сайта.

- `LANGUAGE_CODE` — код языка, на котором будут выводиться системные сообщения и страницы административного сайта. Значение по умолчанию: `"en-us"` (американский английский). Для указания русского языка следует записать этот параметр следующим образом:

```
LANGUAGE_CODE = 'ru-ru'
```

- `USE_I18N` — если `True`, будет активизирована встроенная в Django система автоматического перевода на язык, записанный в параметре `LANGUAGE_CODE`, после чего все системные сообщения и страницы административного сайта будут выводиться на этом языке. Если `False`, автоматический перевод выполняться не будет, и сообщения и страницы станут выводиться на английском языке. Значение по умолчанию — `True`.

- `USE_L18N` — если `True`, числа, значения даты и времени при выводе будут форматироваться по правилам языка, записанного в параметре `LANGUAGE_CODE`. Если `False`, все эти значения будут форматироваться согласно настройкам, заданным в проекте.

  Значение этого параметра по умолчанию — `False`. Однако при создании проекта ему дается значение `True`.

- `TIME_ZONE` — обозначение временной зоны в виде строки.

  Значение этого параметра по умолчанию — `"America/Chicago"`. Однако сразу же при создании проекта ему присваивается значение `"UTC"` (всемирное координированное время). Список всех доступных временных зон можно найти по интернет-адресу **https://en.wikipedia.org/wiki/List_of_tz_database_time_zones**.

- `USE_TZ` — если `True`, Django будут хранить значения даты и времени с указанием временной зоны, — в этом случае параметр `TIME_ZONE` указывает временную зону по умолчанию. Если `False`, значения даты и времени будут храниться без отметки временной зоны, и временную зону для них укажет параметр `TIME_ZONE`.

  Значение этого параметра по умолчанию — `False`. Однако при создании проекта ему дается значение `True`.

Следующие параметры будут приниматься Django в расчет только в том случае, если отключено автоматическое форматирование чисел, значений даты и времени (параметру `USE_L18N` дано значение `False`):

- `DECIMAL_SEPARATOR` — символ, используемый для разделения целой и дробной частей вещественных чисел при выводе на экран. Значение по умолчанию: `"."` (точка);
- `NUMBER_GROUPING` — количество цифр в числе, составляющих группу, в виде целого числа. Используется при выводе чисел. Значение по умолчанию — 0 (группировка цифр не используется);
- `THOUSAND_SEPARATOR` — символ, используемый для разделения групп цифр при выводе на экран. Значение по умолчанию: `","` (запятая);
- `USE_THOUSANDS_SEPARATOR` — если `True`, числа при выводе будут разбиваться на отделенные друг от друга группы, если `False` — не будут. Значение по умолчанию — `False`;
- `SHORT_DATE_FORMAT` — формат по умолчанию для вывода значений даты в короткой записи.

  Значение по умолчанию: `"m/d/Y"` (*<месяц>/<число>/<год из четырех цифр>*). Чтобы указать привычный нам формат *<число>.<месяц>.<год из четырех цифр>*, следует добавить в код модуля settings.py выражение:

  ```
  SHORT_DATE_FORMAT = 'j.m.Y'
  ```

- `SHORT_DATETIME_FORMAT` — формат по умолчанию для вывода значений даты и времени в короткой записи.

  Значение по умолчанию: `"m/d/Y P"` (*<месяц>/<число>/<год из четырех цифр> <часы в 12-часовом формате>*). Чтобы указать привычный нам формат *<число>.<месяц>.<год из четырех цифр> <часы в 24-часовом формате>*: *<минуты>*:*<секунды>*, следует добавить в код модуля settings.py выражение:

  ```
  SHORT_DATETIME_FORMAT = 'j.m.Y H:i:s '
  ```

  То же самое, только без секунд:

  ```
  SHORT_DATETIME_FORMAT = 'j.m.Y H:i'
  ```

- `DATE_FORMAT` — формат по умолчанию для вывода значений даты в полной записи.

  Значение этого параметра по умолчанию: `"N j, Y"` (*<название месяца по-английски> <число>, <год из четырех цифр>*). Чтобы указать формат *<число> <название месяца> <год из четырех цифр>*, следует добавить в код модуля settings.py выражение:

  ```
  DATE_FORMAT = 'j E Y'
  ```

- `DATETIME_FORMAT` — формат по умолчанию для вывода значений даты и времени.

  Значение этого параметра по умолчанию: `"N j, Y, P"` (*<название месяца по-английски> <число>, <год из четырех цифр> <часы в 12-часовом формате>*).

Чтобы указать привычный нам формат *<число> <название месяца> <год из четырех цифр> <часы в 24-часовом формате>:<минуты>:<секунды>*, следует добавить в код модуля settings.py выражение:

```
DATETIME_FORMAT = 'j E Y H:i:s'
```

То же самое, только без секунд:

```
DATETIME_FORMAT = 'j E Y H:i'
```

- `TIME_FORMAT` — формат по умолчанию для вывода значений времени.

  Значение по умолчанию: `"P"` (*только часы в 12-часовом формате*). Для задания формата *<часы в 24-часовом формате>:<минуты>:<секунды>* достаточно добавить в код модуля settings.py выражение:

  ```
  TIME_FORMAT = 'H:i:s'
  ```

  То же самое, только без секунд:

  ```
  TIME_FORMAT = 'H:i'
  ```

- `MONTH_DAY_FORMAT` — формат по умолчанию для вывода месяца и числа. Значение по умолчанию: `"F, j"` (*<название месяца по-английски>*, *<число>*);

- `YEAR_MONTH_FORMAT` — формат по умолчанию для вывода месяца и года. Значение по умолчанию: `"F Y"` (*<название месяца по-английски> <год из четырех цифр>*).

  При записи значений всех этих параметров применяются те же специальные символы, что используются для записи форматов в фильтре шаблонизатора `date` (см. *главу 11*);

- `DATE_INPUT_FORMATS` — список форматов, в которых посетителям допускается заносить значения даты в поля ввода. Получив из веб-формы значение даты, представленное в виде строки, Django будет последовательно сравнивать его со всеми форматами, имеющимися в этом списке, пока не найдет подходящий для преобразования строки в дату.

  Значение по умолчанию этого параметра — довольно длинный список доступных форматов, среди которого, к сожалению, нет привычного нам формата *<число>.<месяц>.<год из четырех цифр>*. Чтобы указать его, следует записать в модуле settings.py следующее выражение:

  ```
  DATE_INPUT_FORMATS = ['%d.%m.%Y']
  ```

- `DATETIME_INPUT_FORMATS` — список форматов, в которых посетителям допускается заносить значения даты и времени в поля ввода. Получив из веб-формы значение даты и времени, представленное в виде строки, Django будет последовательно сравнивать его со всеми форматами, имеющимися в этом списке, пока не найдет подходящий для преобразования строки в значение даты и времени.

  Значение по умолчанию этого параметра — довольно длинный список доступных форматов, среди которого, к сожалению, нет привычного нам формата *<число>.<месяц>.<год из четырех цифр> <часы в 24-часовом формате>*:

<минуты>:*<секунды>*. Чтобы указать его, следует записать в модуле settings.py следующее выражение:

```
DATETIME_INPUT_FORMATS = ['%d.%m.%Y %H:%M:%S']
```

То же самое, только без секунд:

```
DATETIME_INPUT_FORMATS = ['%d.%m.%Y %H:%M']
```

- `TIME_INPUT_FORMATS` — список форматов, в которых посетителям допускается заносить значения времени в поля ввода. Получив из веб-формы значение времени, представленное в виде строки, Django будет последовательно сравнивать его со всеми форматами, имеющимися в этом списке, пока не найдет подходящий для преобразования строки в значение времени.

  Значение по умолчанию этого параметра: `['%H:%M:%S', '%H:%M:%S.%f', '%H:%M']`. Как видим, здесь уже присутствует форматы *<часы в 24-часовом формате>*:*<минуты>*:*<секунды>* и *<часы в 24-часовом формате>*:*<минуты>*, так что нам менять значения этого параметра, скорее всего, не придется.

  При записи значений всех этих параметров применяются специальные символы, используемые в строках формата, указываемых в вызовах функций `strftime()` и `strptime()` Python. Перечень таких специальных символов можно найти в документации по Python или на странице **https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior**.

- `FIRST_DAY_OF_WEEK` — номер дня, с которого начинается неделя, в виде целого числа от `0` (воскресенье) до `6` (суббота). Значение по умолчанию: `0` (воскресенье).

Здесь описаны не все параметры проекта, что мы можем указать. Остальные параметры, затрагивающие работу отдельных подсистем и касающиеся функционирования сайта в эксплуатационном режиме, мы рассмотрим в последующих главах.

3.4. Создание, настройка и регистрация приложений

Приложения реализуют отдельные части функциональности сайта Django. Любой проект должен содержать, по крайней мере, одно приложение.

3.4.1. Создание приложений

Создание приложение выполняется командой `startapp` утилиты manage.py, записываемой в формате:

```
manage.py startapp <имя приложения> [<путь к папке пакета приложения>]
```

Если в вызове команды `startapp` задано только *имя приложения*, в текущей папке будет создана папка пакета приложения с указанным *именем*. Если же дополнительно указать *путь к папке пакета приложения*, расположенная по этому пути папка будет преобразована в пакет Python, который и станет пакетом приложения.

В любом случае в пакете приложения будут созданы пакет `migrations`, модули admin.py, apps.py, models.py, tests.py и views.py.

3.4.2. Настройка приложений

Модуль настроек приложения apps.py содержит объявление *конфигурационного класса*, описывающего настройки приложения. Код такого класса, задающего параметры приложения `bboard`, что мы создали в *главах 1* и *2*, можно увидеть в листинге 3.1.

Листинг 3.1. Пример конфигурационного класса

```
from django.apps import AppConfig

class BboardConfig(AppConfig):
    name = 'bboard'
```

Он является подклассом класса `AppConfig` из модуля `django.apps` и содержит набор атрибутов класса, задающих различные параметры приложения, впрочем, немногочисленные. Давайте рассмотрим их:

- `name` — полный путь к пакету приложения, записанный относительно папки проекта, в виде строки. Единственный параметр приложения, обязательный для указания. Задается утилитой manage.py непосредственно при создании приложения, и менять его не нужно;
- `label` — псевдоним приложения в виде строки. Используется, в том числе, для указания приложения в вызовах утилиты manage.py. Должен быть уникальным в пределах проекта. Если не указан, в качестве псевдонима принимается последний компонент пути, заданного в атрибуте `name`;
- `verbose_name` — название приложения, выводимое на страницах административного сайта Django. Если не указан, на экран будет выводиться псевдоним приложения;
- `path` — файловый путь к папке пакета приложения. Если не указан, Django определит его самостоятельно.

3.4.3. Регистрация приложения в проекте

Как уже говорилось, чтобы приложение успешно работало, оно должно быть зарегистрировано в списке приложений `INSTALLED_APPS`, что находится в параметрах проекта (см. *разд. 3.3.3*). А зарегистрировать его можно двумя способами.

1. Добавить в список приложений новый элемент — путь к конфигурационному классу приложения, заданному в виде строки:

```
INSTALLED_APPS = [
    . . .
    'bboard.apps.BboardConfig',
]
```

2. В модуле __init__.py пакета приложения присвоить путь к конфигурационному классу переменной `default_app_config`:

```
default_app_config = 'bboard.apps.BboardConfig'
```

После чего добавить в список приложений новый элемент — строку с путем к пакету приложения:

```
INSTALLED_APPS = [
    . . .
    'bboard',
]
```

3.5. Отладочный веб-сервер Django

Отладочный веб-сервер Django предназначен для обслуживания сайта в процессе его разработки. Запуск сервера выполняется утилитой manage.py при получении ей команды `runserver`. Эта команда записывается в таком формате:

```
manage.py runserver [[<интернет-адрес>][:][<порт>]] [--noreload]
[--nothreading] [--ipv6] [-6]
```

По умолчанию отладочный сервер доступен по интернет-адресу **http://localhost:8000/** и **http://127.0.0.1:8000/** (т. е. использует интернет-адрес локального хоста и TCP-порт № 8000).

Сайт, написанный на языке Python, имеет одну особенность: составляющие его программные модули загружаются при запуске отладочного сервера в оперативную память и, пока сайт работает, постоянно находятся там. Это значительно увеличивает быстродействие сайта, поскольку не приходится при каждом запросе загружать весь необходимый программный код (что происходит в случае использования некоторых других программных платформ — таких, например, как PHP). Однако при любом изменении кода требуется перезапускать сервер, чтобы в память загрузился исправленный код.

Отладочный веб-сервер Django самостоятельно отслеживает изменения в программных модулях при их сохранении. И, в случае необходимости, перезапускается самостоятельно. Нам об этом беспокоиться не следует.

> ***Внимание!***
>
> Тем не менее, в некоторых случаях (в частности, при создании новых модулей или шаблонов) отладочный сервер не перезапускается вообще или перезапускается некорректно, вследствие чего совершенно правильный код перестает работать. Поэтому перед внесением значительных правок в программный код рекомендуется остановить сервер, нажав комбинацию клавиш <Ctrl>+<Break>, а после правок вновь запустить его.
>
> Вообще, если сайт стал вести себя странно, притом, что код не содержит никаких критических ошибок, прежде всего попробуйте перезапустить отладочный сервер — возможно, после этого все заработает. Автор не раз сталкивался с подобной ситуацией.

Мы можем указать для отладочного веб-сервера другие *интернет-адрес* и (или) *порт*. Например, так можно задать использование интернет-адреса **1.2.3.4** (при этом будет задействован все тот же порт № 8000):

```
manage.py runserver 1.2.3.4
```

Задаем использование порта № 4000 со стандартным интернет-адресом **localhost**:

```
manage.py runserver 4000
```

Задаем использование интернет-адреса **1.2.3.4** и порта № 4000:

```
manage.py runserver 1.2.3.4:4000
```

Команда `runserver` поддерживает следующие дополнительные ключи:

- `--noreload` — отключить автоматический перезапуск при изменении программного кода;
- `--nothreading` — принудительный запуск отладочного сервера в однопоточном режиме (если ключ не указан, сервер запускается в многопоточном режиме);
- `--ipv6` или `-6` — использовать протокол IPv6 вместо IPv4. В этом случае по умолчанию будет использоваться интернет-адрес **::1**.

ГЛАВА 4

Модели: базовые инструменты

В настоящее время весьма трудно найти более или менее крупный сайт, не хранящий свои внутренние данные в информационной базе. Стало быть, нашим следующим шагом после создания и настройки проекта и входящих в него приложений будет написание моделей.

4.1. Введение в модели

Еще в *главе 1* говорилось, что модель — это класс, описывающий хранящуюся в базе данных сущность и являющийся представлениями таблицы и отдельной ее записи средствами Python.

Механизм моделей Django основывается на следующих принципах:

- сам класс модели представляет всю совокупность сущностей, принадлежащих этому классу и хранящихся в базе данных;
- отдельный экземпляр класса модели представляет отдельную сущность, принадлежащую этому классу и извлеченную из базы данных;
- класс модели описывает набор отдельных значений (*полей*), входящих в состав сущности этого класса;
- класс модели предоставляет инструменты для работы со всеми сущностями этого класса: их извлечения, фильтрации, сортировки и пр.;
- экземпляр класса модели предоставляет инструменты для работы с отдельными значениями, входящими в состав сущности: ее сохранения, удаления, а также создания новых сущностей;
- набор извлеченных из базы сущностей представляется последовательностью экземпляров соответствующего класса модели.

Для представления отдельного поля, входящего в состав сущности, в классе модели объявляется отдельный атрибут класса, которому присваивается экземпляр класса, представляющего поле нужного типа: строковое, целочисленное, вещественное

число, дату и время и др. Через эти атрибуты впоследствии можно получить доступ к значениям этих полей.

Модели объявляются на уровне отдельного приложения.

По поводу низкоуровневых структур, создаваемых в базе данных и используемых моделью для хранения сущностей, можно отметить следующее:

- каждая модель представляет отдельную таблицу в базе данных;
- по умолчанию таблицы, представляемые моделями, получают имена вида *<псевдоним приложения>_<имя класса модели>* (о псевдонимах приложений рассказывалось в *разд. 3.4.2*);
- каждое поле модели представляет отдельное поле в соответствующей таблице базы данных;
- по умолчанию поля в таблице получают имена, совпадающие с именами полей модели, которые их представляют;
- если в модели не было явно объявлено ключевое поле для хранения значения, однозначно идентифицирующего запись модели (ключа), оно будет создано самим фреймворком, получит имя `id`, целочисленный тип, будет помечено как автоинкрементное, и для него будет создан ключевой индекс;
- по умолчанию никакие дополнительные индексы в таблице не создаются.

Например, для модели `Bb` приложения `bboard`, объявленной нами в *главе 1* (см. листинг 1.6), в базе данных будет создана таблица `bboard_bb` с полями `id` (ключевое поле будет неявно создано самим фреймворком), `title`, `content`, `price` и `published`.

Модель может быть создана для представления как уже существующей в базе таблицы (в этом случае при объявлении модели нам придется дополнительно указать имя таблицы и имена всех входящих в нее полей), так и еще не существующей (тогда для создания таблицы нам нужно сгенерировать миграцию, о чем пойдет разговор в *главе 5*).

4.2. Объявление моделей

Условия, которым должны удовлетворять классы моделей, очень просты:

- класс модели должен быть производным от класса `Model` из модуля `django.db.models`.

  Также имеется возможность сделать класс модели производным от другого класса модели. Однако такой вариант наследования имеет ряд особенностей, о которых мы поговорим в *главе 16*;
- код классов моделей записывается в модуле models.py пакета приложения, в котором они объявляются;
- чтобы модели были успешно обработаны программным ядром Django, содержащее их приложение должно быть зарегистрировано в списке приложений проекта (см. *разд. 3.4.3*).

4.3. Объявление полей модели

Как уже говорилось, каждое поле модели представляется отдельным атрибутом класса модели. Ему присваивается экземпляр класса, представляющего поле определенного типа. Дополнительные параметры создаваемого поля можно указать в соответствующих им именованных параметрах конструктора класса поля.

4.3.1. Параметры, поддерживаемые полями всех типов

Рассказ об объявлении полей в модели имеет смысл начать с описания параметров, поддерживаемых всеми классами полей. Таких параметров довольно много:

- `verbose_name` — название поля, которое будет выводиться на страницах административного сайта и напротив элементов управления в веб-формах. Если оно не указано, в таковом качестве будет использовано имя поля;
- `help_text` — дополнительный поясняющий текст, выводимый на экран. Значение по умолчанию — пустая строка.

  Следует иметь в виду, что при выводе этого текста содержащиеся в нем специальные символы HTML не преобразуются в литералы и выводятся как есть. Это позволяет нам отформатировать поясняющий текст HTML-тегами;
- `default` — значение по умолчанию, записываемое в поле, если в него явно не было занесено никакого значения. Может быть указано двумя способами:
  - как обычное значение любого неизменяемого типа:

    ```
    is_active = models.BooleanField(default=True)
    ```

    Если в качестве значения по умолчанию должно выступать значение изменяемого типа (список или словарь Python), для его указания следует использовать второй способ;
  - как ссылка на функцию, вызываемую при создании каждой новой записи и возвращающую в качестве результата нужное значение:

    ```
    def is_active_default():
        return not is_all_posts_passive
    . . .
    is_active = models.BooleanField(default=is_active_default)
    ```
- `unique` — если `True`, в текущее поле может быть занесено только уникальное в пределах таблицы значение (*уникальное поле*). При попытке занести значение, уже имеющееся в том же поле другой записи, будет возбуждено исключение `IntegrityError` из модуля `django.db`.

  Если поле помечено как уникальное, по нему автоматически будет создан индекс. Поэтому явно задавать для него индекс не нужно.

  Если `False`, текущее поле может хранить любые значения. Значение по умолчанию — `False`;

- `unique_for_date` — если в этом параметре указать представленное в виде строки имя поля даты (`DateField`) или даты и времени (`DateTimeField`), текущее поле может хранить только значения, уникальные в пределах даты, что хранится в указанном поле. Пример:

  ```
  title = models.CharField(max_length=50, unique_for_date='published')
  published = models.DateTimeField()
  ```

  В этом случае Django позволит сохранить в поле `title` только значения, уникальные в пределах даты, хранящейся в поле `published`;

- `unique_for_month` — то же самое, что и `unique_for_date`, но в расчет принимается не все значение даты, а лишь месяц;

- `unique_for_year` — то же самое, что и `unique_for_date`, но в расчет принимается не все значение даты, а лишь год;

- `null` — если `True`, поле в таблице базы данных может хранить значение `null` и, таким образом, являться необязательным к заполнению. Если `False`, поле в таблице должно иметь какое-либо значение, хотя бы пустую строку. Значение по умолчанию — `False`.

  Отметим, что для строковых и текстовых полей, даже являющихся обязательными к заполнению (т. е. при их объявлении параметру `null` было присвоено `False`), вполне допустимым значением является пустая строка. Если сделать поля необязательными к заполнению, задав `True` для параметра `null`, они вдобавок к этому могут хранить значение `null`. Оба значения фактически представляют отсутствие каких-либо данных в поле, и эту ситуацию придется как-то обрабатывать. Поэтому, чтобы упростить обработку отсутствия значения в таком поле, его не стоит делать необязательным к заполнению.

  Параметр `null` затрагивает только поле таблицы, но не поведение Django. Даже если какое-то поле присвоением параметру значения `True` было помечено как необязательное, фреймворк по умолчанию все равно не позволит занести в него пустое значение;

- `blank` — если `True`, Django позволит занести в поле пустое значение, тем самым сделав поле необязательным к заполнению, если `False` — не будет. Значение по умолчанию — `False`.

  Параметр `blank` задает поведение самого фреймворка при выводе на экран веб-форм и проверке введенных в них данных. Если для этого параметра было указано значение `True`, Django позволит занести в поле пустое значение (например, для строкового поля — пустую строку), даже если это поле было помечено как обязательное к заполнению (параметру `null` было дано значение `False`);

- `choices` — последовательность значений, доступных для занесения в текущее поле. Может быть использован для создания полей, способных хранить ограниченный набор значений, которые посетитель выбирает из списка (*поле со списком*).

Каждый элемент последовательности, задаваемой в качестве значения этого параметра, представляет одно значение, доступное для записи в поле, и должен записываться в виде последовательности из двух элементов:

- значения, которое будет непосредственно записано в поле. Оно должно принадлежать к типу, поддерживаемому полем (так, если поле имеет строковый тип, значение должно представлять собой строку);
- значения, которое будет выводиться на экран в качестве соответствующего пункта списка, должно представлять собой строку.

Пример задания для поля `kind` списка из трех возможных значений:

```
class Bb(models.Model):
    KINDS = (
        ('b', 'Куплю'),
        ('s', 'Продам'),
        ('c', 'Обменяю'),
    )
    kind = models.CharField(max_length=1, choices=KINDS)
    . . .
```

Если поле помечено как обязательное к заполнению на уровне фреймворка (параметру `blank` конструктора присвоено значение `False`), в списке, перечисляющем доступные для ввода в поле значения, появится пункт -------- (9 знаков «минус»), обозначающий отсутствие в поле какого-либо значения. Мы можем задать для этого пункта другой текст, добавив в последовательность значений элемент вида (None, "*<Новый текст "пустого" пункта>*") (если поле имеет текстовый тип, вместо `None` можно поставить пустую строку). Пример:

```
KINDS = (
    (None, 'Выберите разряд публикуемого объявления'),
    ('b', 'Куплю'),
    ('s', 'Продам'),
    ('c', 'Обменяю'),
)
```

- `db_index` — если `True`, по текущему полю в таблице будет создан индекс, если `False` — не будет. Значение по умолчанию — `False`;
- `primary_key` — если `True`, текущее поле станет ключевым. При этом ключевое поле будет помечено как обязательное к заполнению и уникальное (параметру `null` неявно будет присвоено значение `False`, а параметру `unique` — `True`).

> ***Внимание!***
> В модели может присутствовать только одно ключевое поле.

Если `False`, поле не будет преобразовано в ключевое. Значение по умолчанию — `False`.

Если ключевое поле в модели не задано явно, сам фреймворк создаст в ней ключевое поле с именем `id`;

- `editable` — если `True`, поле будет выводиться на экран в составе формы, если `False`, не будет (даже если явно создать его в форме). Значение по умолчанию — `True`;
- `db_column` — имя поля таблицы в виде строки. Если оно не указано, имя поля будет сформировано по описанным в *разд. 4.1* соглашениям.

Здесь приведены не все параметры, поддерживаемые конструкторами классов полей. Некоторые параметры мы изучим позже.

4.3.2. Классы полей моделей

Теперь можно приступить к рассмотрению классов полей, предоставляемых Django и объявленных в модуле `django.db.models`. Каждое такое поле позволяет хранить значения определенного типа. Помимо этого, мы познакомимся с параметрами, специфическими для различных классов полей.

- `CharField` — *строковое поле*, хранящее строку ограниченной длины. Такое поле занимает в базе данных объем, необходимый для хранения количества символов, указанного в качестве размера этого поля. Поэтому, по возможности, лучше исключить создание в моделях строковых полей большой длины.

  Строковое поле поддерживает обязательный параметр `max_length`, указывающий максимальную длину заносимого в поле значения в виде целого числа в символах.

- `TextField` — *текстовое поле*, хранящее строку неограниченной длины. Такие поля рекомендуется применять для сохранения больших текстов, длина которых заранее не известна и может варьироваться.

  Поддерживается дополнительный необязательный параметр `max_length`, указывающий максимальную длину заносимого в поле текста. Если параметр не указан, в поле можно записать значение любой длины.

- `EmailField` — корректно сформированный адрес электронной почты в строковом виде.

  Поддерживает необязательный параметр `max_length`, указывающий максимальную длину заносимого в поле адреса в виде целого числа в символах. Значение этого параметра по умолчанию: `254`.

- `URLField` — корректно сформированный интернет-адрес.

  Поддерживает необязательный параметр `max_length`, указывающий максимальную длину заносимого в поле интернет-адреса в виде целого числа в символах. Значение этого параметра по умолчанию: `200`.

- `SlugField` — *слаг*, т. е. строка, однозначно идентифицирующая запись и применяемая в качестве части интернет-адреса. Поддерживает два необязательных дополнительных параметра:

  - `max_length` — максимальная длина заносимой в поле строки в символах. Значение по умолчанию: `50`;

- `allow_unicode` — если `True`, хранящийся в поле слаг может содержать любые символы Unicode, если `False` — только символы из кодировки ASCII. Значение по умолчанию — `False`.

Для каждого поля такого типа автоматически создается индекс, поэтому указывать параметр `db_index` со значением `True` нет нужды.

- `BooleanField` — логическое поле, хранящее значение `True` или `False`.

> ***Внимание!***
> Значение поля `BooleanField` по умолчанию — `None`, а не `False`, как можно было бы предположить.

Для поля этого типа можно указать параметр `null` со значением `True`, в результате чего оно получит возможность хранить еще и значение `null`.

- `NullBooleanField` — то же самое, что `BooleanField`, но дополнительно позволяет хранить значение `null`. Этот тип поля оставлен для совместимости с более старыми версиями Django, и использовать его во вновь разрабатываемых проектах не рекомендуется.
- `IntegerField` — знаковое целочисленное поле обычной длины (32-разрядное).
- `SmallIntegerField` — знаковое целочисленное поле половинной длины (16-разрядное).
- `BigIntegerField` — знаковое целочисленное значение двойной длины (64-разрядное).
- `PositiveIntegerField` — беззнаковое целочисленное поле обычной длины (32-разрядное).
- `PositiveSmallIntegerField` — беззнаковое целочисленное поле половинной длины (16-разрядное).
- `FloatField` — вещественное число.
- `DecimalField` — вещественное число фиксированной точности, представленное объектом типа `Decimal` из модуля `decimal` Python. Поддерживает два дополнительных обязательных параметра:
  - `max_digits` — максимальное количество цифр в числе;
  - `decimal_places` — количество цифр в дробной части числа.

  Вот пример объявления поля для хранения чисел с шестью цифрами в целой части и двумя — в дробной:

```
price = models.DecimalField(max_digits=8, decimal_places=2)
```

- `DateField` — значение даты, представленное в виде объекта типа `date` из модуля `datetime` Python.

  Класс `DateField` поддерживает два дополнительных необязательных параметра:
  - `auto_now` — если `True`, при каждом сохранении записи в поле будет заноситься текущее значение даты. Это может использоваться для хранения даты последнего изменения записи.

Если `False`, хранящееся в поле значение при сохранении записи никак не затрагивается. Значение по умолчанию — `False`;

- `auto_now_add` — то же самое, что `auto_now`, но текущая дата заносится в поле только при создании записи и при последующих сохранениях не изменяется. Может пригодиться для хранения даты создания записи.

Указание значения `True` для любого из этих параметров приводит к тому, что поле становится невидимым и необязательным для занесения на уровне Django (т. е. параметру `editable` присваивается `False`, а параметру `blank` — `True`).

- `DateTimeField` — то же самое, что и `DateField`, но хранит значение даты и времени в виде объекта типа `datetime` из модуля `datetime`.
- `TimeField` — значение времени, представленное в виде объекта типа `time` из модуля `datetime` Python. Поддерживаются дополнительные необязательные параметры `auto_now` и `auto_now_add` (см. описание класса `DateField`).
- `DurationField` — промежуток времени, представленный объектом типа `timedelta` из модуля `datetime` Python.
- `BinaryField` — двоичные данные произвольной длины. Значение этого поля представляется объектом типа `bytes`.
- `GenericIPAddressField` — IP-адрес, записанный для протокола IPv4 или IPv6, в виде строки. Поддерживает два дополнительных необязательных параметра:
  - `protocol` — обозначение допустимого протокола для записи IP-адресов, представленный в виде строки. Поддерживаются значения `"IPv4"`, `"IPv6"` и `"both"` (поддерживаются оба протокола). Значение по умолчанию — `"both"`;
  - `inpack_ipv4` — если `True`, IP-адреса протокола IPv4, записанные в формате IPv6, будут преобразованы к виду, применяемому в IPv4. Если `False`, такое преобразование не выполняется. Значение по умолчанию — `False`. Этот параметр принимается во внимание только в том случае, если для параметра `protocol` указано значение `"both"`.
- `AutoField` — *автоинкрементное поле*. Хранит уникальные, постоянно увеличивающиеся целочисленные значения обычной длины (32-разрядные). Практически всегда используется в качестве ключевого поля.

  Как правило, нет необходимости объявлять такое поле явно. Если модель не содержит явно объявленного ключевого поля любого типа, Django самостоятельно создаст ключевое поле типа `AutoField`.
- `BigAutoField` — то же самое, что `AutoField`, но хранит целочисленное значение двойной длины (64-разрядное).
- `UUIDField` — уникальный универсальный идентификатор, представленный объектом типа `UUID` из модуля `uuid` Python, в виде строки.

  Поле такого типа может использоваться в качестве ключевого вместо поля `AutoField` или `BigAutoField`. Единственный недостаток: нам придется генерировать идентификаторы для создаваемых записей самостоятельно.

Вот пример объявления поля `UUIDField`:

```
import uuid
from django.db import models

class Bb(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
    editable=False)
    . . .
```

Мы не рассмотрели здесь типы полей, предназначенных для хранения файлов и изображений. Они будут описаны в *главе 19*.

4.4. Создание связей между моделями

Связи между моделями создаются объявлением в них полей, формируемых особыми классами из того же модуля `django.db.models`. Сейчас мы с ними познакомимся.

4.4.1. Связь «один-со-многими»

Связь «один-со-многими» позволяет связать одну запись первичной модели с произвольным количеством записей вторичной модели. Это наиболее часто применяемый на практике вид связей.

Для создания связи такого типа в классе *вторичной модели* следует объявить поле типа `ForeignKey`. Вот формат конструктора этого класса:

```
ForeingKey(<связываемая первичная модель>,
           on_delete=<поведение при удалении записи>[,
           <остальные параметры>])
```

Первым, позиционным, параметром указывается связываемая первичная модель (не забываем, что поле описываемого типа создается во вторичной модели!). Ее можно записать в виде:

- непосредственно ссылки на класс модели, если код объявления первичной модели находится перед кодом класса вторичной модели (в которой и создается поле внешнего ключа):

```
class Rubric(models.Model):
    . . .
class Bb(models.Model):
    rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT)
    . . .
```

- строки с именем класса, если вторичная модель объявлена перед первичной:

```
class Bb(models.Model):
    rubric = models.ForeignKey('Rubric', on_delete=models.PROTECT)
    . . .
class Rubric(models.Model):
    . . .
```

Если необходимо передать ссылку на модель, объявленную в другом приложении проекта, необходимо использовать строку формата *<имя приложения>.<имя класса модели>*:

```
rubric = models.ForeingKey('rubrics.Rubric', on_delete=models.PROTECT)
```

Если нужно создать модель, ссылающуюся на себя (создать *рекурсивную связь*), первым параметром конструктору следует передать строку `self`:

```
parent_rubric = models.ForeingKey('self', on_delete=models.PROTECT)
```

Вторым параметром `on_delete` указывается поведение фреймворка в случае, если будет выполнена попытка удалить запись первичной модели, на которой ссылаются какие-либо записи вторичной модели. Параметру присваивается значение одной из переменных, объявленных в модуле `django.db.models`:

- `CASCADE` — удаляет все связанные записи вторичной модели (*каскадное удаление*);
- `PROTECT` — возбуждает исключение `ProtectedError` из модуля `django.db.models`, тем самым предотвращая удаление записи первичной модели;
- `SET_NULL` — заносит в поле внешнего ключа всех связанных записей вторичной модели значение `null`. Сработает только в том случае, если поле внешнего ключа объявлено необязательным к заполнению на уровне базы данных (параметр `null` конструктора поля имеет значение `True`);
- `SET_DEFAULT` — заносит в поле внешнего ключа всех связанных записей вторичной модели заданное для него значение по умолчанию. Сработает только в том случае, если у поля внешнего ключа было указано значение по умолчанию (оно задается параметром `default` конструктора поля);
- `SET(<значение>)` — заносит в поле внешнего ключа указанное *значение*:

```
rubric = models.ForeingKey(Rubric, on_delete=models.SET(1))
```

Также можно указать ссылку на функцию, которая не должна принимать параметров и должна возвращать значение, которое будет записано в поле:

```
def get_first_rubric():
    return Rubric.objects.first()
. . .
rubric = models.ForeingKey(Rubric,
                           on_delete=models.SET(get_first_rubric))
```

- `DO_NOTHING` — ничего не делает.

> ***Внимание!***
>
> Если СУБД поддерживает межтабличные связи с сохранением ссылочной целостности, попытка удаления записи первичной модели, с которой связаны записи вторичной модели, все равно не увенчается успехом. В этом случае будет возбуждено исключение `IntegrityError` из модуля `django.db.models`.

Полю внешнего ключа рекомендуется давать имя, обозначающее связываемую сущность и записанное в единственном числе. Например, для представления рубрики в модели Bb мы объявили поле rubric.

На уровне базы данных поле внешнего ключа модели представляется полем таблицы, имеющем имя вида *<имя поля внешнего ключа>*_id. В веб-форме такое поле будет представляться раскрывающимся списком, в котором будут содержаться строковые представления записей первичной модели.

Теперь рассмотрим несколько дополнительных необязательных параметров, поддерживаемых конструктором класса ForeignKey:

- limit_choices_to — позволяет вывести в списке связываемых записей первичной модели только те, что удовлетворяют заданным критериям фильтрации.

  Критерии фильтрации записываются в виде обычного словаря Python, имена элементов которого должны совпадать с именами полей первичной модели, по которым должна выполняться фильтрация, а значения элементов укажут значения для этих полей. Выведены будут записи, удовлетворяющие всем критериям, заданным в таком словаре (т. е. критерии объединяются по правилу логического И).

  Для примера укажем Django выводить только рубрики, поле show которых содержит значение True:

  ```
  rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
                             limit_choices_to={'show': True})
  ```

  В качестве значения параметра также может быть использован экземпляр класса Q, способный задать более сложные критерии фильтрации (класс Q мы рассмотрим в *главе 7*).

  Если параметр не указан, список связываемых записей будет включать все записи первичной модели;

- related_name — имя атрибута записи первичной модели, предназначенного для доступа к связанным записям вторичной модели, в виде строки:

  ```
  class Bb(models.Model):
      rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
                                 related_name='entries')
  . . .
  # Получаем первую рубрику
  first_rubric = Rubric.objects.first()
  # Получаем доступ к связанным объявлениям через атрибут entries,
  # указанный в параметре related_name
  bbs = first_rubric.entries.all()
  ```

  Если доступ из записи первичной модели к связанным записям вторичной модели не требуется, можно указать Django не создавать такой атрибут и тем самым немного сэкономить системные ресурсы. Для этого достаточно присвоить параметру related_name символ «плюс».

Если параметр не указан, атрибут такого рода получит стандартное имя вида *<имя связанной вторичной модели>*_set;

- related_query_name — имя фильтра, что будет применяться во вторичной модели для фильтрации по значениям из записи первичной модели:

```
class Bb(models.Model):
    rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
                               related_query_name='entry')
. . .
# Получаем все рубрики, содержащие объявления о продаже домов,
# воспользовавшись фильтром, заданным в параметре related_query_name
rubrics = Rubric.objects.filter(entry__title='Дом')
```

Если параметр не указан, фильтр такого рода получит стандартное имя, совпадающее с именем класса вторичной модели.

Более подробно работа с записями связанных моделей, применяемые для этого атрибуты и фильтры будут рассмотрены в *главе 7*;

- to_field — имя поля первичной модели, по которому будет выполнена связь, в виде строки. Такое поле должно быть помечено как уникальное (параметр unique конструктора должен иметь значение True).

  Если параметр не указан, связывание выполняется по ключевому полю первичной модели — неважно, созданному явно или неявно;

- db_constraint — если True, в таблице базы данных будет создана связь, позволяющая сохранять ссылочную целостность, если False, ссылочная целостность будет поддерживаться только на уровне Django.

  Значение по умолчанию — True. Менять его на False имеет смысл только, если модель создается на основе уже существующей базы с некорректными данными.

4.4.2. Связь «один-с-одним»

Связь «один-с-одним» соединяет одну запись первичной модели с одной записью вторичной модели. Такие связи служат для объединения моделей, одна из которых хранит данные, дополняющие данные из другой модели.

Такая связь создается в классе *вторичной модели* объявлением поля типа OneToOneField. Вот формат конструктора этого класса:

```
OneToOneField(<связываемая первичная модель>,
              on_delete=<поведение при удалении записи>[,
              <остальные параметры>])
```

Первые два параметра точно такие же, как и у конструктора класса ForeignField (см. *разд. 4.4.1*).

Для хранения списка зарегистрированных на Django-сайте пользователей стандартная подсистема разграничения доступа использует особую модель. Ссылка на класс заменяемой модели пользователя хранится в параметре AUTH_USER_MODEL настроек проекта.

Давайте создадим модель AdvUser, хранящую дополнительные сведения о зарегистрированном пользователе. Мы свяжем ее со стандартной моделью пользователя User из модуля django.contrib.auth.models. Готовый код класса этой модели показан в листинге 4.1.

Листинг 4.1. Пример создания связи «один-с-одним»

```
from django.db import models
from django.contrib.auth.models import User

class AdvUser(models.Model):
    is_activated = models.BooleanField(default=True)
    user = models.OneToOneField(User, on_delete=models.CASCADE)
```

На уровне базы данных связь такого рода представляется точно таким же полем, что и связь типа «один-со-многими» (см. *разд. 4.4.1*).

Конструктор класса поддерживает те же дополнительные необязательные параметры, что и конструктор класса ForeignField, плюс параметр parent_link, применяемый при наследовании моделей (разговор о котором пойдет в *главе 16*).

4.4.3. Связь «многие-со-многими»

И, наконец, самая сложная в реализации (для Django — но не для нас) *связь «многие-со-многими»* — она позволяет связать произвольное количество записей одной модели с произвольным количеством записей другой (обе модели здесь выступают как равноправные, и определить, какая из них первичная, а какая вторичная, не представляется возможным).

Для создания такой связи нужно объявить *в одной из моделей* (но не в обеих сразу!) поле внешнего ключа типа ManyToManyField. Вот формат его конструктора:

```
ManyToManyField(<вторая связываемая модель>[, <остальные параметры>])
```

Первый параметр задается в таком же формате, что и в конструкторах классов ForeignField и OneToOneField (см. *разд. 4.4.1* и *4.4.2*).

Модель, в которой было объявлено поле внешнего ключа, носит название *ведущей*, а вторая модель в таком случае станет *ведомой*.

Для примера создадим модели Machine и Spare, из которых первая, ведущая, будет хранить готовые машины, а вторая, ведомая, — отдельные детали для них. Код обеих моделей приведен в листинге 4.2.

Листинг 4.2. Пример создания связи «многие-со-многими»

```
class Spare(models.Model):
    name = models.CharField(max_length=30)
```

```
class Machine(models.Model):
    name = models.CharField(max_length=30)
    spares = models.ManyToManyField(Spare)
```

В отличие от связей описанных ранее типов, имя поля, образующего связь «многие-со-многими», рекомендуется записывать во множественном числе. Что и логично — ведь такая связь позволяет связать произвольное количество записей, что называется, с обеих сторон.

На уровне базы данных для представления связи такого типа создается таблица, по умолчанию имеющая имя вида *<псевдоним приложения>_<имя класса ведущей модели>_<имя класса ведомой модели>* (*связующая таблица*). Она будет иметь ключевое поле `id` и по одному полю с именем вида *<имя класса связываемой модели>*`_id` на каждую из связываемых моделей. Так, в нашем случае будет создана связующая таблица с именем `samplesite_machine_spare`, имеющая поля `id`, `machine_id` и `spare_id`.

Если создается связь с той же самой моделью, связующая таблица будет иметь поля `id`, `from_`*<имя класса модели>*`_id` и `to_`*<имя класса модели>*`_id`.

Конструктор класса `ManyToManyField` поддерживает дополнительные необязательные параметры `limit_choices_to`, `related_name`, `related_query_name` и `db_constraint`, описанные в *разд. 4.4.1*, а также следующие:

- `symmetrical` — используется только в тех случаях, когда модель связывается сама с собой. Если `True`, Django создаст *симметричную связь*, действующую в обоих направлениях (применительно к нашему случаю: если какая-то деталь А входит в машину Б, то машина Б содержит деталь А). Если `False`, связь будет *асимметричной* (чисто гипотетически: какая-то деталь А входит в машину Б, однако машина Б не содержит деталь А). Значение по умолчанию — `True`.

  Для асимметричной связи Django создаст в классе модели атрибут для доступа к записям связанной модели в обратном направлении (подробно об этом — в *главе 7*);

- `through` — класс модели, которая представляет связующую таблицу (*связующая модель*) либо в виде ссылки на него, либо в виде имени, представленном строкой. Если он не указан, связующая таблица будет создана самим Django.

  При использовании связующей модели нужно иметь в виду следующее:

  - поле внешнего ключа для связи объявляется и в ведущей, и в ведомой моделях. При создании этих полей следует указать как саму связующую модель (параметр `through`), так и поля внешних ключей, по которым будет установлена связь (параметр `through_fields`, описанный далее);
  - в связующей модели следует явно объявить поля внешних ключей для установления связи с обеими связываемыми моделями: и ведущей, и ведомой;

- `through_fields` — используется только в том случае, если для установления связи применяется связующая модель, записанная в параметре `through` конструктора. Указывает поля внешних ключей, по которым будет создаваться связь.

Значение параметра должно представлять собой кортеж из двух элементов: имени поля ведущей модели и имени поля ведомой модели — оба имени должны быть представлены в виде строк. Если параметр не указан, поля будут созданы самим фреймворком.

Пример использования связующей модели для установления связи «многие-со-многими» и правильного заполнения параметров `through` и `through_fields` будет приведен в *главе 16*;

- `db_table` — имя связующей таблицы. Обычно применяется, если связующая модель не используется. Если оно не указано, связующая таблица получит имя по умолчанию.

4.5. Параметры самой модели

Параметры самой модели описываются различными атрибутами класса `Meta`, вложенного в класс модели и не являющегося производным ни от никакого класса. Далее приведен список этих атрибутов:

- `verbose_name` — название сущности, хранящейся в модели, которое будет выводиться на экран;
- `verbose_name_plural` — название набора сущностей, хранящихся в модели, которая будет выводиться на экран;
- `unique_together` — последовательность имен полей, представленных в виде строк, которые должны хранить уникальные в пределах таблицы комбинации значений. При попытке занести в них уже имеющуюся в таблице комбинацию значений будет возбуждено исключение `ValidationError` из модуля `django.core.exceptions`. Пример:

```
class Bb(models.Model):
    . . .

    class Meta:
        unique_together = ('title', 'published')
```

Теперь комбинация названия товара и даты публикации объявления должна быть уникальной в пределах модели. Занести в модель еще одну запись с такими же товаром и датой публикации будет нельзя.

Можно указать несколько подобных групп полей, объединив их в последовательность:

```
class Bb(models.Model):
    . . .

    class Meta:
        unique_together = (
          ('title', 'published'),
          ('title', 'price', 'rubric'),
        )
```

Теперь уникальными должны быть и комбинация названия товара и даты публикации объявления, и комбинация названия товара, цены и рубрики;

- `ordering` — параметры сортировки записей модели по умолчанию. Всегда задаются в виде последовательности имен полей, по которым должна выполняться сортировка, представленных строками. Если перед именем поля поставить символ «минус», порядок сортировки будет обратным. Пример:

```
class Bb(models.Model):
    . . .

    class Meta:
        ordering = ['-published', 'title']
```

Здесь мы сортируем записи модели сначала по убыванию значения поля `published`, а потом по возрастанию значения поля `title`;

- `get_latest_by` — имя поля типа `DateField` или `DateTimeField`, которое будет взято в расчет при получении наиболее поздней или наиболее ранней записи с помощью метода `latest()` или `earliest()` соответственно, вызванного без параметров. Можно задать:

  - имя поля в виде строки — тогда в расчет будет взято только это поле:

```
class Bb(models.Model):
    . . .
    published = models.DateTimeField()

    class Meta:
        get_latest_by = 'published'
```

Теперь метод `latest()` вернет запись с наиболее поздним значением даты и времени, хранящемся в поле `published`.

Если имя поля предварить символом «минус», порядок сортировки окажется обратным, и при вызове `latest()` мы получим, напротив, самую раннюю запись, а при вызове метода `earliest()` — самую позднюю:

```
    class Meta:
        get_latest_by = '-published'
```

  - последовательность имен полей — тогда в расчет будут взяты значения всех этих полей, и, если у каких-то записей первое поле хранит одинаковое значение, будет проверяться значение второго поля и т. д.:

```
class Bb(models.Model):
    . . .
    added = models.DateTimeField()
    published = models.DateTimeField()

    class Meta:
        get_latest_by = ['added', 'published']
```

- `order_with_respect_to` — позволяет сделать набор записей произвольно упорядочиваемым. В качестве значения параметра задается строка с именем поля текущей модели, и в дальнейшем записи, в которых это поле хранит одно и то же значение, могут быть упорядочены произвольным образом. Пример:

```
class Bb(models.Model):
    . . .
    rubric = models.ForeignKey('Rubric')

    class Meta:
        order_with_respect_to = 'rubric'
```

  Теперь объявления, относящиеся к одной и той же рубрике, могут быть переупорядочены произвольным образом.

  При указании в модели этого параметра в таблице будет дополнительно создано поле с именем вида *<имя поля, заданного в качестве значения параметра>*`_order`. Оно будет хранить целочисленное значение, указывающее порядковый номер текущей записи в последовательности.

  Одновременно вновь созданное поле с порядковым номером будет задано в качестве значения параметра `ordering` (см. ранее). Следовательно, записи, которые мы извлечем из модели, по умолчанию будут отсортированы по значению этого поля. Указать другие параметры сортировки в таком случае будет невозможно;

- `indexes` — последовательность индексов, включающих в себя несколько полей. Каждый элемент такой последовательности должен представлять собой экземпляр класса `Index` из модуля `django.db.models`. Формат конструктора класса:

```
Index(fields=[][, name=None])
```

  В параметре `fields` указывается список строк с именами полей, которые должны быть включены в индекс. По умолчанию сортировка значений поля выполняется по их возрастанию, а чтобы отсортировать по убыванию, нужно предварить имя поля знаком «минус». Параметр `name` задает имя индекса — если он не указан, имя будет создано самим фреймворком. Пример:

```
class Bb(models.Model):
    . . .

    class Meta:
        indexes = [
            models.Index(fields=['-published', 'title'], name='main'),
            models.Index(fields=['title', 'price', 'rubric']),
        ]
```

- `index_together` — предлагает другой способ создания индексов, содержащих несколько полей. Строки с именами полей указываются в виде последовательности, а набор таких последовательностей также объединяется в последовательность. Пример:

```
class Bb(models.Model):
    . . .

    class Meta:
        index_together = [
            ['-published', 'title'],
            ['title', 'price', 'rubric'],
        ]
```

Если нужно создать всего один такой индекс, последовательность с именами его полей можно просто присвоить этому параметру:

```
class Bb(models.Model):
    . . .

    class Meta:
        index_together = ['-published', 'title']
```

- `default_related_name` — имя атрибута записи первичной модели, предназначенного для доступа к связанным записям вторичной модели, в виде строки. Соответствует параметру `related_name` конструкторов классов полей, предназначенных для установления связей между моделями (см. *разд. 4.4.1*). Неявно задает значения параметрам `related_name` и `related_query_name` конструкторов;
- `db_table` — имя таблицы, в которой хранятся данные модели. Если оно не указано, таблица получит имя по умолчанию.

4.6. Интернет-адрес модели и его формирование

Django позволяет сформировать интернет-адрес, указывающий на конкретную запись модели, — *интернет-адрес модели*. По этому адресу может находиться страница с содержимым этой записи, списком связанных записей и др.

Сформировать интернет-адрес модели можно двумя способами: декларативным и императивным.

Декларативный способ заключается в том, что мы описываем формат интернет-адреса в настройках проекта. Набор таких адресов оформляется в виде словаря Python и записывается в параметре `ABSOLUTE_URL_OVERRIDES` модуля settings.py пакета конфигурации.

Ключи элементов этого словаря должны иметь вид *<псевдоним приложения>.<имя класса модели>*. Значениями элементов станут функции, в качестве единственного параметра принимающие объект записи модели, а в качестве результата возвращающие строку с готовым интернет-адресом. Здесь удобно использовать лямбда-функции Python.

Вот пример объявления словаря, который на основе рубрики (экземпляра класса модели `Rubric`) сформирует интернет-адрес вида **/bboard/*<ключ рубрики>*/**, ведущий на страницу со списком относящихся к этой рубрике объявлений:

```
ABSOLUTE_URL_OVERRIDES = {
    'bboard.rubric': lambda rec: "/bboard/%s/" % rec.pk,
}
```

Теперь, чтобы поместить в код шаблона интернет-адрес модели, нам достаточно вставить туда вызов метода `get_absolute_url()`, унаследованного всеми моделями от базового класса `Model`:

```
<a href="{{ rubric.get_absolute_url }}">{{ rubric.name }}</a>
```

Точно таким же образом можно получить интернет-адрес модели где-либо еще — например, в коде контроллера.

Императивный способ заключается в непосредственном переопределении метода `get_absolute_url()` в классе модели. Вот пример:

```
class Rubric(models.Model):
    . . .
    def get_absolute_url(self):
        return "/bboard/%s/" % self.pk
```

Разумеется, в параметре `ABSOLUTE_URL_OVERRIDES` настроек проекта в таком случае нет нужды.

4.7. Методы модели

Помимо атрибутов класса, представляющих поля модели, и вложенного класса `Meta`, где объявляются параметры модели, мы можем объявить в классе модели дополнительные методы:

- `__str__()` — возвращает строковое представление записи модели. Строковое представление будет выводиться, если в коде шаблона указать вывод непосредственно объекта записи, а не значения его поля или результата, возвращенного его методом:

  ```
  {{ rubric }}
  ```

  Пример переопределения этого метода можно найти в *разд. 2.2*;

- `save()` — выполняет сохранение записи. При определении этого метода обязательно следует вставить в нужное место кода вызов метода, унаследованного от базового класса. Вот пример:

  ```
  def save(self, *args, **kwargs):
      # Выполняем какие-либо действия перед сохранением
      super().save(*args, **kwargs)  # Сохраняем запись, вызвав
                                     # унаследованный метод
      # Выполняем какие-либо действия после сохранения
  ```

  Мы можем, в зависимости от выполнения или невыполнения какого-то условия, отменить сохранение записи, для чего достаточно просто не вызывать унаследованный метод `save()`. Пример:

```
def save(self, *args, **kwargs):
    # Выполняем сохранение записи, только если метод
    # is_model_correct() вернет True
    if self.is_model_correct():
        super().save(*args, **kwargs)
```

- `delete()` — выполняет удаление записи. Этот метод также переопределяется для добавления какой-либо логики, что должна выполняться перед и (или) после удаления. Пример:

```
def delete(self, *args, **kwargs):
    # Выполняем какие-либо действия перед удалением
    super().delete(*args, **kwargs)  # Удаляем запись, вызвав
                                     # унаследованный метод
    # Выполняем какие-либо действия после удаления
```

И точно таким же образом в случае необходимости мы можем предотвратить удаление записи:

```
def delete(self, *args, **kwargs):
    # Удаляем запись, только если метод need_to_delete() вернет True
    if self.need_to_delete():
        super().delete(*args, **kwargs)
```

Мы можем создать в модели дополнительное поле, значение которого вычисляется на основе каких-то других данных и которое доступно только для чтения (*функциональное поле*). Для этого достаточно объявить метод, не принимающий параметров и возвращающий нужное значение. Имя этого метода станет именем функционального поля.

В качестве примера создадим в модели `Bb` функциональное поле `title_and_price`, объявив одноименный метод:

```
class Bb(models.Model):
    . . .
    def title_and_price(self):
        if self.price:
            return '%s (%.2f)' % (self.title, self.price)
        else:
            return self.title
```

После этого мы можем вывести значение функционального поля в шаблоне:

```
<h2>{{ bb.title_and_price }}</h2>
```

Для функционального поля допускается указать название, которое будет выводиться на страницах сайта, в качестве заголовка столбца или надписи для элемента управления. Строку с названием просто нужно присвоить атрибуту `short_description` объекта метода, который реализует это поле:

```
class Bb(models.Model):
    . . .
```

```
    def title_and_price(self):
        . . .
    title_and_price.short_description = 'Название и цена'
```

4.8. Валидация модели. Валидаторы

Валидацией называется проверка на корректность данных, занесенных в поля модели. Мы можем реализовать валидацию непосредственно в модели или же в форме, которая используется для занесения в нее данных (об этом мы поговорим в *главе 13*).

4.8.1. Стандартные валидаторы Django

Валидацию значений, заносимых в отдельные поля модели, выполняют *валидаторы*, которые могут быть реализованы в виде функций или классов. Некоторые типы полей уже используют определенные валидаторы — так, строковое поле `CharField` задействует валидатор `MaxLengthValidator`, проверяющий, не превышает ли длина заносимого строкового значения указанную максимальную длину.

Помимо этого, мы можем указать для любого поля другие валидаторы, предоставляемые Django. Реализующие их классы объявлены в модуле `django.core.validators`. А указываются они в параметре `validators` конструктора класса поля. Пример:

```
from django.core import validators

class Bb(models.Model):
    title = models.CharField(max_length=50,
            validators=[validators.RegexValidator(regex='^.{4,}$')])
```

Здесь мы использовали валидатор, представляемый классом `RegexValidator`. Он проверяет заносимое в поле значение на соответствие заданному регулярному выражению.

Если значение не проходит проверку валидатором, он возбуждает исключение `ValidationError` из модуля `django.core.exceptions`.

Давайте познакомимся с классами валидаторов:

- `MinLengthValidator` — проверяет, не меньше ли длина заносимой строки заданного в первом параметре минимума. Формат конструктора:

  `MinLengthValidator(`*<минимальная длина>*`[, message=None])`

  Параметр `message` задает сообщение об ошибке — если он не указан, используется стандартное сообщение. Код ошибки: `"min_length"`;

- `MaxLengthValidator` — проверяет, не превышает ли длина заносимой строки заданный в первом параметре максимум. Используется полем типа `CharField`. Формат конструктора:

  `MaxLengthValidator(`*<максимальная длина>*`[, message=None])`

Параметр `message` задает сообщение об ошибке — если он не указан, используется стандартное. Код ошибки: `"max_length"`;

- `RegexValidator` — проверяет значение на соответствие заданному регулярному выражению. Конструктор класса:

```
RegexValidator(regex=None[, message=None][, code=None][,
                          inverse_match=None][, flags=0])
```

Он принимает следующие параметры:

- `regex` — само регулярное выражение. Может быть указано в виде строки или объекта типа `regex`, встроенного в Python;
- `message` — строка с сообщением об ошибке. Если не указан, используется стандартное сообщение;
- `code` — код ошибки. Если не указан, используется код по умолчанию: `"invalid"`;
- `inverse_match` — если `False`, значение должно соответствовать регулярному выражению (поведение по умолчанию). Если `True`, значение, напротив, не должно соответствовать регулярному выражению;
- `flag` — флаги регулярного выражения. Используется только, если таковое задано в виде строки;

- `EmailValidator` — проверяет на корректность заносимый в поле адрес электронной почты. Используется полем типа `EmailField`. Конструктор класса:

```
EmailValidator([message=None][,][code=None][,][, whitelist=None])
```

Параметры:

- `message` — строка с сообщением об ошибке. Если не указан, используется стандартное сообщение;
- `code` — код ошибки. Если не указан, используется код по умолчанию: `"invalid"`;
- `whitelist` — последовательность доменов, которые не будут проверяться валидатором, представленных в виде строк. Если не указан, в эту последовательность входит только адрес локального хоста **localhost**;

- `URLValidator` — проверяет на корректность заносимый в поле интернет-адрес. Используется полем типа `URLField`. Конструктор класса:

```
URLValidator([schemes=None][,][regex=None][,][message=None][,]
             [code=None])
```

Параметры:

- `schemes` — последовательность обозначений протоколов, в отношении которых будет выполняться валидация, в виде строк. Если не указан, используется последовательность `['http', 'https', 'ftp', 'ftps']`;

- `regex` — регулярное выражение, с которым должен совпадать интернет-адрес. Может быть указано в виде строки или объекта типа `regex`, встроенного в Python. Если параметр отсутствует, такого рода проверка не проводится;
- `message` — строка с сообщением об ошибке. Если не указан, используется стандартное сообщение;
- `code` — код ошибки. Если не указан, используется код по умолчанию: `"invalid"`;

❐ `ProhibitNullCharactersValidator` — проверяет, не содержит ли заносимая строка нулевой символ: `\x00`. Формат конструктора:

```
ProhibitNullCharactersValidator([message=None][,][code=None])
```

Он принимает следующие параметры:

- `message` — строка с сообщением об ошибке. Если не указан, используется стандартное сообщение;
- `code` — код ошибки. Если не указан, используется код по умолчанию: `"null_characters_not_allowed"`;

❐ `MinValueValidator` — проверяет, не меньше ли заносимое число заданного в первом параметре минимума. Формат конструктора:

```
MinValueValidator(<минимальное значение>[, message=None])
```

Параметр `message` задает сообщение об ошибке; если он не указан, используется стандартное. Код ошибки: `"min_value"`;

❐ `MaxValueValidator` — проверяет, не превышает ли заносимое число заданный в первом параметре максимум. Формат конструктора:

```
MaxValueValidator(<максимальное значение>[, message=None])
```

Параметр `message` задает сообщение об ошибке — если он не указан, используется стандартное. Код ошибки: `"max_value"`;

❐ `DecimalValidator` — проверяет заносимое вещественное число фиксированной точности, представленное объектом типа `Decimal` из модуля `decimal` Python. Формат конструктора:

```
DecimalValidator(<максимальное количество цифр в числе>,
                 <количество цифр в дробной части>)
```

Коды ошибок:

- `"max_digits"` — если общее количество цифр в числе больше заданного;
- `"max_decimal_places"` — если количество цифр в дробной части больше заданного;
- `"max_whole_digits"` — если количество цифр в целой части числа больше разности между общим количеством и количеством цифр в дробной части.

Часть валидаторов реализована в виде функций.

- `validate_ipv46_address()` — проверяет на корректность интернет-адреса протоколов IPv4 и IPv6;
- `validate_ipv4_address()` — проверяет на корректность интернет-адреса только протокола IPv4;
- `validate_ipv6_address()` — проверяет на корректность интернет-адреса только протокола IPv6.

  Эти три валидатора используются полем типа `GenericIPAddressField`;
- `int_list_validator()` — возвращает экземпляр класса `RegexValidator`, настроенный на проверку последовательностей целых чисел, которые разделяются указанным символом-разделителем. Формат вызова:

```
int_list_validator([sep=','][,][message=None][,][code='invalid'][,]
                                [allow_negative=False])
```

  Параметры:

  - `sep` — строка с символом-разделителем. Если не указан, в качестве разделителя используется запятая;
  - `message` — строка с сообщением об ошибке. Если не указан, используется стандартное сообщение;
  - `code` — код ошибки. Если не указан, используется код по умолчанию: `"invalid"`;
  - `allow_negative` — если `True`, допускаются отрицательные числа, если `False` — не допускаются (поведение по умолчанию).

Помимо классов и функций, модуль `django.core.validators` объявляет ряд переменных. Каждая из них хранит готовый объект валидатора, настроенный под определенное применение:

- `validate_email` — экземпляр класса `EmailValidator` с настройками по умолчанию;
- `validate_slug` — экземпляр класса `RegexValidator`, настроенный на проверку слагов. Допускает наличие в слагах только латинских букв, цифр, символов «минус» и подчеркивания;
- `validate_unicode_slug` — экземпляр класса `RegexValidator`, настроенный на проверку слагов. Допускает наличие в слагах только букв в кодировке Unicode, цифр, символов «минус» и подчеркивания;

  Эти два валидатора используются полем типа `SlugField`;
- `validate_comma_separated_integer_list` — экземпляр класса `RegexValidator`, настроенный на проверку последовательностей целых чисел, которые разделены запятыми.

4.8.2. Вывод собственных сообщений об ошибках

Во многих случаях стандартные сообщения об ошибках, выводимые валидаторами, вполне понятны. Но временами возникает необходимость указать для какого-либо поля свои сообщения об ошибках, чтобы дать посетителю больше сведений о заносимых в поле значениях.

Такие сообщения об ошибках указываются в параметре `error_messages` конструктора класса поля. Значением этого параметра должен быть словарь Python, у которого ключи элементов должны совпадать с кодами ошибок, а значения — задавать сами тексты сообщений.

Вот пример указания для поля `title` модели `Bb` собственного сообщения об ошибке:

```
from django.core import validators

class Bb(models.Model):
    title = models.CharField(max_length=50, verbose_name='Товар',
            validators=[validators.RegexValidator(regex='^.{4,}$')],
            error_messages={'invalid': 'Неправильное название товара'})
    . . .
```

Доступные для указания коды ошибок:

- `"null"` — поле таблицы не может хранить значение `null`, т. е. его следует заполнить;
- `"blank"` — в элемент управления должно быть занесено значение;
- `"invalid"` — неверный формат значения;
- `"invalid_choice"` — в поле со списком заносится значение, не указанное в списке;
- `"unique"` — в поле заносится неуникальное значение, что недопустимо;
- `"unique_for_date"` — в поле заносится значение, неуникальное в пределах даты, что недопустимо;
- `"invalid_date"` — значение даты хоть и введено правильно, но является некорректным (например, 30.14.2018);
- `"invalid_time"` — значение времени хоть и введено правильно, но является некорректным (например, 25:73:80);
- `"invalid_datetime"` — значение даты и времени хоть и введено правильно, но является некорректным (например, 30.14.2018 25:73:80);
- `"min_length"` — длина сохраняемой в поле строки меньше указанного минимума;
- `"max_length"` — длина сохраняемой в поле строки больше указанного максимума;
- `"null_characters_not_allowed"` — сохраняемая строка содержит нулевые символы `\x00`;

- "min_value" — сохраняемое в поле число меньше указанного минимума;
- "max_value" — сохраняемое в поле число больше указанного максимума;
- "max_digits" — общее количество цифр в сохраняемом числе типа Decimal больше заданного;
- "max_decimal_places" — количество цифр в дробной части сохраняемого числа типа Decimal больше заданного;
- "max_whole_digits" — количество цифр в целой части сохраняемого числа типа Decimal больше разности между максимальным общим количеством цифр и количеством цифр в дробной части.

4.8.3. Написание своих валидаторов

Если нужный нам валидатор отсутствует в стандартном наборе, мы можем написать его самостоятельно. У нас даже есть выбор — мы можем реализовать валидатор в виде функции или класса.

Валидатор, выполненный в виде функции, должен принимать один параметр — значение, которое следует проверить. Если значение некорректно, функция должна возбудить исключение ValidationError из модуля django.core.exceptions. Возвращать результат она не должна.

Для вызова конструктора класса исключения ValidationError мы используем следующий формат:

```
ValidationError(<описание ошибки>[, code=None][, params=None])
```

Первым, позиционным, параметром передается строка с текстовым описанием ошибки, допущенной посетителем при вводе значения. Если в этот текст нужно вставить какое-либо значение, следует использовать заменитель вида %(<*ключ элемента словаря, переданного параметром params*>s).

В параметре code указывается код ошибки. Можно указать подходящий код из числа приведенных в *разд. 4.8.2* или же придумать свой собственный.

Разработчики Django настоятельно рекомендуют задавать код ошибки. Однако в этом случае нужно помнить, что текст сообщения об ошибке, для которой был указан код, может быть изменен формой, привязанной к этой модели, или самим разработчиком посредством параметра error_messages конструктора поля. Поэтому, если вы хотите, чтобы заданный вами в валидаторе текст сообщения об ошибке всегда выводился как есть, не указывайте для ошибки код.

И наконец, в параметре params задается словарь со значениями, которые нужно поместить в текст сообщения об ошибке (он передается первым параметром) вместо заменителей.

Листинг 4.3 показывает код валидатора, реализованного в виде функции validate_even(). Он проверяет, является ли число четным.

Листинг 4.3. Пример валидатора-функции

```
from django.core.exceptions import ValidationError

def validate_even(val):
    if val % 2 != 0:
        raise ValidationError('Число %(value)s нечетное', code='odd',
                              params={'value': val})
```

Мы можем указать этот валидатор для поля точно таким же образом, как указывали стандартный:

```
class Bb(models.Model):
    . . .
    price = models.FloatField(validators=[validate_even])
```

Если валидатору при создании следует передавать какие-либо параметры, задающие режим его работы, этот валидатор нужно реализовать в виде класса. Параметры валидатору будут передаваться через конструктор класса, а сама валидация станет выполняться в переопределенном методе `__call__()`. Последний должен принимать с параметром проверяемое значение и возбуждать исключение `ValidationError`, если оно окажется некорректным.

Листинг 4.4 показывает код класса `MinMaxValueValidator`, проверяющего, находится ли заносимое в поле числовое значение в заданном диапазоне. Нижняя и верхняя границы этого диапазона передаются через параметры конструктора класса.

Листинг 4.4. Пример валидатора-класса

```
from django.core.exceptions import ValidationError

class MinMaxValueValidator:
    def __init__(self, min_value, max_value):
        self.min_value = min_value
        self.max_value = max_value

    def __call__(self, val):
        if val < self.min_value or val > self.max_value:
            raise ValidationError('Введенное число должно ' +\
                  'находиться в диапазоне от %(min)s до %(max)s',
                  code='out_of_range',
                  params={'min': self.min_value, 'max': self.max_value})
```

4.8.4. Валидация модели

Может возникнуть необходимость проверить на корректность не значение одного поля, а всю модель (выполнить *валидацию модели*). Мы можем с легкостью сделать это, переопределив в классе модели метод `clean()`.

Метод не должен принимать параметры и возвращать результат. Единственное, что он обязан сделать, — в случае необходимости возбудить исключение `ValidationError`.

Поскольку некорректные значения могут быть занесены сразу в несколько полей, нам придется как-то формировать целый список ошибок. В этом нам на помощь придет второй формат конструктора класса `ValidationError`:

```
ValidationError(<список ошибок>)
```

Список ошибок удобнее всего представлять в виде словаря. В качестве ключей элементов указываются имена полей модели, в которые были занесены некорректные значения. В качестве значений этих элементов должны выступать последовательности из экземпляров класса `ValidationError`, каждый из которых представляет одну из ошибок.

Для примера давайте сделаем так, чтобы занесение описания продаваемого товара было обязательным, и предотвратим ввод отрицательного значения цены (разумеется, это проще сделать, задав соответствующие значения параметров у конструкторов полей, но давайте попрактикуемся в валидации модели). Вот код метода `clean()`, который реализует все это:

```
class Bb(models.Model):
    . . .
    def clean(self):
        errors = {}
        if not self.content:
            errors['content'] = ValidationError('Укажите описание ' +\
                                                'продаваемого товара')
        if self.price and self.price < 0:
            errors['price'] = ValidationError('Укажите ' +\
                              'неотрицательное значение цены')
        if errors:
            raise ValidationError(errors)
```

Если нам нужно вывести какое-либо сообщение об ошибке, относящейся не к определенному полю модели, а ко всей модели, следует использовать в качестве ключа словаря, хранящего список ошибок, значение переменной `NON_FIELD_ERRORS` из модуля `django.core.exceptions`. Пример:

```
from django.core.exceptions import NON_FIELDS_ERRORS
. . .
errors[NON_FIELDS_ERRORS] = ValidationError('Ошибка в модели!')
```

Конструктор класса `ValidationError` поддерживает и прием списка ошибок в виде собственно списка или кортежа Python, однако в этом случае мы не сможем указать Django, к какому полю модели относятся те или иные ошибки, и форма, связанная с моделью, не сможет вывести их напротив нужного элемента управления.

ГЛАВА 5

Миграции

Миграция — это программный модуль, создающий в базе данных все необходимые для модели структуры: таблицы, поля, индексы, правила и связи. При выполнении миграция генерирует SQL-код, выполняющий создание этих структур и рассчитанный на выполнение той СУБД, что записана в настройках баз данных проекта (см. *разд. 3.3.2*).

Практически всегда миграции формируются самим Django по запросу разработчика. Писать свои собственные миграции приходится крайне редко. На случай, если это все же понадобится, — вот интернет-адрес страницы с описанием всех необходимых программных инструментов: **https://docs.djangoproject.com/en/2.1/ref/migration-operations/**.

5.1. Формирование миграций

Для формирования миграций служит команда `makemigrations` утилиты manage.py:

```
manage.py makemigrations [<список псевдонимов приложений, разделенных
пробелами>] [--name|-n <имя миграции>] [--noinput] [--no-input]
[--dry-run] [--check] [--merge] [--empty]
```

Если при отдаче команды не указать *псевдонимов приложений*, будут обработаны модели, объявленные во всех приложениях проекта. Если же указать *псевдоним приложения* или несколько их *псевдонимов*, разделив их пробелами, будут обработаны только миграции из указанных приложений.

Команда `makemigrations` поддерживает следующие ключи:

- `--name` или `-n` — указывает имя формируемой миграции, которое будет добавлено к порядковому номеру для получения полного имени файла с кодом миграции. Если отсутствует, миграция получит имя по умолчанию;
- `--noinput` или `--no-input` — отключает вывод на экран сведений о формируемой миграции;
- `--dry-run` — выводит на экран сведения о формируемой миграции, но не формирует ее;

- `--check` — выводит сведения о том, изменились ли модели после последнего формирования миграций, но не формирует саму миграцию;
- `--merge` — используется для устранения конфликтов между миграциями;
- `--empty` — создает «пустую» миграцию для программирования ее вручную (если такое понадобится).

Если после последнего формирования миграции в приложении появилась новая модель, Django вставит в новую миграцию код, создающий соответствующие структуры в базе данных. То же самое происходит, если до этого момента формирование миграции еще не проводилось.

Если после последнего формирования миграции какая-либо модель изменилась, Django вставит в новую миграцию код, вносящий нужные изменения в соответствующие структуры базы данных: добавит, исправит и удалит поля, индексы, правила и связи.

Если же какая-либо модель за это время не изменилась, Django не будет помещать в миграцию никакого кода, который бы ее затрагивал.

При каждом формировании создается только один файл миграции, который и выполняет все необходимые действия.

> ***Внимание!***
>
> Django отслеживает любые изменения в коде моделей, даже те, которые не затрагивают структуры базы данных напрямую. Так, если мы укажем для поля модели название (параметр `verbose_name` конструктора поля), фреймворк все равно создаст в миграции код, который изменит параметры нижележащего поля таблицы в базе данных. Поэтому крайне желательно продумывать структуру моделей заранее и впоследствии, по возможности, не менять ее.
>
> Для отслеживания, какие миграции уже были выполнены, а какие — еще нет, Django создает в базе данных по умолчанию таблицу `django_migrations`. Править вручную как ее структуру, так и хранящиеся в ней записи настоятельно не рекомендуется.

5.2. Файлы миграций

Все программные модули миграций сохраняются в пакете `migrations`, расположенном в пакете приложения. По умолчанию они получают имена формата *<последовательно увеличивающиеся порядковые номера>_<имя миграции>*.py. *Порядковые номера* состоят из четырех цифр и просто помечают очередность, в которой формировались миграции. *Имя миграции* задается в ключе `--name` (`-n`) команды `makemigretions` — если этот ключ не указан, сам фреймворк сгенерирует имя следующего вида:

- initial — если это *начальная миграция*, т. е. та, что создает самые первые версии всех необходимых структур в базе данных;
- auto_*<отметка даты и времени формирования миграции>* — если это миграции, сформированные после начальной и дополняющие и изменяющие созданные ранее структуры.

Так, при выполнении упражнений, приведенных в *главах 1* и *2*, у автора книги были сформированы миграции 0001_initial.py и 0002_auto_20180601_1608.py.

Впоследствии мы можем переименовать миграцию, но только в том случае, если она до этого еще ни разу не выполнялась. Дело в том, что имена модулей миграций сохраняются в таблице `django_migrations`, и, если мы переименуем уже выполненную миграцию, Django не сможет проверить, была ли она выполнена, и выполнит ее снова.

5.3. Выполнение миграций

В процессе выполнения миграций в базе данных создаются или изменяются нижележащие структуры. Выполнение запускается командой `migrate` утилиты manage.py:

```
manage.py migrate [<псевдоним приложения> [<имя миграции>]]
[--fake-initial] [--noinput] [--no-input] [--fake]
```

Если не указывать ни *псевдоним приложения*, ни *имени миграции*, будут выполнены все не выполненные к настоящему моменту миграции во всех приложениях проекта. Если указать только *псевдоним приложения*, будут выполнены все миграции в этом приложении, а если дополнительно задать *имя миграции*, будет выполнена только эта миграция.

Задавать имя модуля миграции полностью нет необходимости — достаточно записать только находящийся в начале ее имени порядковый номер:

```
manage.py migrate bboard 0001
```

В команде мы можем применить такие дополнительные ключи:

- `--fake-initial` — пропускает выполнение начальной миграции. Применяется, если в базе данных на момент первого выполнения миграций уже присутствуют все необходимые структуры, и их нужно просто модифицировать;
- `--noinput` или `--no-input` — отключает вывод на экран сведений о применении миграций;
- `--fake` — помечает миграции как выполненные, но не вносит никаких изменений в базу данных. Может пригодиться, если все необходимые изменения в базу были внесены вручную.

На каждую выполненную миграцию в таблицу `django_migrations` базы данных добавляется отдельная запись, хранящая имя модуля миграции, имя приложения, в котором она была создана, дату и время выполнения миграции.

5.4. Слияние миграций

Если в модели неоднократно вносились изменения, после чего выполнялось генерирование миграций на основе исправленных моделей, таких миграций может накопиться довольно много. Чтобы уменьшить их количество и заодно ускорить процесс их применения к «свежей» базе данных, рекомендуется выполнить *слияние миграций* — объединение их в одну.

Для слияния миграций достаточно отдать команду `squashmigrations` утилиты manage.py:

```
manage.py squashmigrations <псевдоним приложения> [<имя первой миграции>]
<имя последней миграции> [--squashed_name <имя результирующей миграции>]
[--no-optimize] [--noinput] [--no-input]
```

Обязательными для указания являются только *псевдоним приложения* и *имя последней миграции* из числа подвергаемых слиянию. В этом случае будут обработаны все миграции, начиная с самой первой из сформированных (обычно это начальная миграция) и заканчивая указанной в команде. Пример:

```
manage.py squashmigrations bboard 0004
```

Если задать *имя первой миграции* из подвергаемых слиянию, будут обработаны миграции, начиная с нее. Более ранние миграции будут пропущены. Пример:

```
manage.py squashmigrations testapp 0002 0004
```

Давайте рассмотрим поддерживаемые командой ключи:

- `--squashed_name` — задает имя миграции, которая будет получена в результате слияния. Если оно отсутствует, модуль результирующей миграции получит имя вида *<имя первой миграции>*_squashed_*<имя последней миграции>*.py;
- `--no-optimize` — отменяет оптимизацию кода миграции, что применяется для уменьшения его объема и повышения быстродействия. Рекомендуется указывать этот ключ, если слияние миграций выполнить не удалось, или если результирующая миграция оказалась неработоспособной;
- `--noinput` или `--no-input` — отключает вывод на экран сведений о слиянии миграций.

5.5. Вывод списка миграций

Чтобы просмотреть список всех миграций, имеющихся в проекте, следует отдать команду `showmigrations` утилиты manage.py:

```
manage.py showmigrations [<список псевдонимов приложений, разделенных
пробелами>] [--plan] [-p]
```

Если не указывать *псевдоним приложения*, будут выведены все имеющиеся в проекте миграции с разбиением по приложениям. Если указать *псевдоним приложения*, будут выведены только миграции из этого приложения. При задании *списка псевдонимов приложений, разделенных пробелами*, выводятся только миграции из этих приложений, опять же, с разбиением по отдельным приложениям.

Список миграций при выводе сортируется в алфавитном порядке. Левее имени каждой миграции выводится значок `[X]`, если миграция была выполнена, и `[ ]` в противном случае.

Команда `showmigrations` поддерживает ключи `--plan` и `-p`, указывающие команде вместо списка вывести план миграций. План также представляет собой список, но

отсортированный в последовательности, в которой Django будет выполнять миграции.

5.6. Отмена всех миграций

Наконец, Django позволяет нам отменить все миграции в приложении, тем самым удалив все созданные ими в базе данных структуры. Для этого достаточно выполнить команду `migrate` утилиты manage.py, указав в ней имя нужного приложения и `zero` в качестве имени миграции:

```
manage.py migrate testapp zero
```

К сожалению, отменить отдельную, произвольно выбранную миграцию невозможно.

ГЛАВА 6

Запись данных

Главное предназначение моделей — упрощение работы с данными, хранящимися в информационной базе. В том числе упрощение записи данных в базу.

6.1. Правка записей

Проще всего исправить уже имеющуюся в модели запись. Для этого предварительно нужно извлечь ее каким-либо образом (начала чтения данных из моделей мы постигли в *разд. 1.10*):

```
>>> from bboard.models import Bb
>>> b = Bb.objects.get(pk=17)
>>> b
<Bb: Bb object (17)>
```

Здесь мы в консоли Django извлекаем объявление с ключом `17` (это объявление автор создал в процессе отладки сайта, написанного в *главах 1* и *2*).

Занести в поля извлеченной записи новые значения можно, просто присвоив их атрибутам класса модели, представляющим эти поля:

```
>>> b.title = 'Земельный участок'
>>> b.content = 'Большой'
>>> b.price = 100000
```

После чего останется выполнить сохранение записи, вызвав метод `save()` модели:

```
>>> b.save()
```

Поскольку эта запись имеет ключ (ее ключевое поле заполнено), Django сразу узнает, что она уже была ранее сохранена в базе, и выполнит обновление, отправив СУБД SQL-команду `UPDATE`.

В поле со списком следует заносить значение, предназначенное для записи в поле (оно задается первым элементом последовательности, подробнее об этом см. в *разд. 4.3.1*):

```
>>> # Это будет объявление о продаже
>>> b.kind = 's'
```

6.2. Создание записей

Создать новую запись в модели можно тремя способами:

- создать новый экземпляр класса модели, не передавая конструктору никаких параметров, занести в поля нужные значения и выполнить сохранение, вызвав метод `save()` модели:

```
>>> from bboard.models import Rubric
>>> r = Rubric()
>>> r.name = 'Бытовая техника'
>>> r.save()
```

- создать новый экземпляр класса модели, указав значения полей непосредственно в вызове конструктора — через одноименные параметры. После этого, опять же, необходимо сохранить запись вызовом метода `save()` модели:

```
>>> r = Rubric(name='Сельхозинвентарь')
>>> r.save()
```

- все классы моделей поддерживают атрибут `objects`, в котором хранится *диспетчер записей* — объект, предоставляющий инструменты для манипулирования всей совокупностью хранящихся в модели записей. Диспетчер записей — это экземпляр класса `Manager`, объявленного в модуле `django.db.models`.

  Класс `Manager` поддерживает метод `create()`, который принимает с именованными параметрами значения полей создаваемой записи, создает эту запись, сразу же сохраняет и возвращает в качестве результата. Вот пример использования этого метода:

```
>>> r = Rubric.objects.create(name='Мебель')
>>> r.pk
5
```

  Мы можем удостовериться в том, сохранена ли запись, запросив значение ее ключевого поля (оно всегда доступно через универсальный атрибут класса `pk`). Если оно хранит значение, значит, запись была сохранена, в чем мы и убедились.

При создании новой записи любым из описанных ранее способов Django проверяет значение ее ключевого поля. Если таковое хранит пустую строку или `None` (т. е. ключ отсутствует), фреймворк вполне резонно предполагает, что запись нужно добавить в базу, и выполняет ее добавление посылкой СУБД SQL-команды `INSERT`.

Если уж зашла речь о диспетчере записей `Manager`, то нужно рассказать еще о паре методов, которые он поддерживает. Эти методы помогут несколько сократить код:

- `get_or_create(<набор фильтров>[, defaults=None])` — выполняет поиск записи модели на основе заданного *набора фильтров* (о написании фильтров и вообще

о поиске записей будет рассказано в *главе 7*). Если подходящая запись не будет найдена, метод создаст и сохранит ее, использовав *набор фильтров* для указания значений полей новой записи.

Необязательному параметру `defaults` можно присвоить словарь, указывающий значения для остальных полей создаваемой записи (подразумевается, что модель не содержит поля с именем `defaults`. Если же такое поле есть, и по нему нужно выполнить поиск, следует использовать фильтр вида `defaults__exact`).

В качестве результата метод возвращает кортеж из двух значений:

- записи модели, найденной в базе или созданной только что;
- `True`, если эта запись была создана, или `False`, если она была найдена в базе.

Пример:

```
>>> r = Rubric.objects.get_or_create(name='Мебель')
>>> r
(<Rubric: Мебель>, False)
>>> r = Rubric.objects.get_or_create(name='Сантехника')
>>> r
(<Rubric: Сантехника>, True)
```

> ***Внимание!***
>
> **Метод** `get_or_create()` **способен вернуть только одну запись, удовлетворяющую заданным критериям поиска. Если таких записей в модели окажется более одной, будет возбуждено исключение** `MultipleObjectsReturned` **из модуля** `django.core.exceptions`.

- `update_or_create(<набор фильтров>[, defaults=None])` — аналогичен методу `get_or_create()`, но в случае, если запись найдена, заносит в ее поля новые значения, заданные в словаре, который присвоен параметру `defaults`. Пример:

```
>>> Rubric.objects.update_or_create(name='Цветы')
(<Rubric: Цветы>, True)
>>> Rubric.objects.update_or_create(name='Цветы',
                                    defaults={'name': 'Растения'})
(<Rubric: Растения>, False)
```

6.3. Некоторые замечания о методе *save()*

Метод `save()` модели, которым мы неоднократно пользовались, имеет следующий формат вызова:

```
save([update_fields=None][,][force_insert=False][,][force_update=False])
```

Необязательный параметр `update_fields` указывает последовательность имен полей модели, которые нужно обновить. Этот параметр имеет смысл указывать только при обновлении записи, если были изменены значения не всех, а одного или двух полей, и если поля, не подвергшиеся обновлению, хранят объемистые данные (например, большой текст в поле текстового типа). Пример:

```
>>> b = Bb.objects.get(pk=17)
>>> b.title = 'Земельный участок'
>>> b.save(update_fields=['title'])
```

Если параметр `update_fields` не задан, будут обновлены все поля модели.

Если параметрам `force_insert` и `force_update` дать значение `False`, Django сам будет принимать решение, создать новую запись или обновить имеющуюся (на основе чего он принимает такое решение, мы уже знаем). Однако мы можем явно указать ему создать или изменить запись, дав значение `True` параметру `force_insert` или `force_update` соответственно.

Такое может пригодиться, например, в случае, если какая-то таблица содержит ключевое поле не целочисленного автоинкрементного, а какого-то иного типа — например, строкового. Значение в такое поле при создании записи придется заносить вручную, но при сохранении записи, выяснив, что ключевое поле содержит значение, Django решит, что эта запись уже была сохранена ранее, и выполнит ее обновление, что приведет к ошибке. Чтобы исключить такую ситуацию, при вызове метода `save()` следует задать параметр `force_insert` со значением `True`.

Здесь нужно иметь в виду две особенности. Во-первых, задание списка обновляемых полей в параметре `update_fields` автоматически дает параметру `force_update` значение `True` (т. е. явно указывает обновить запись). Во-вторых, указание `True` для обоих описанных ранее параметров вызовет ошибку.

6.4. Удаление записей

Удалить не нужную более запись также несложно. Достаточно найти ее и вызвать у нее поддерживаемый всеми классами моделей метод `delete()`:

```
>>> b = Bb.objects.get(pk=5)
>>> b
<Bb: Bb object (5)>
>>> b.delete()
(1, {'bboard.Bb': 1})
```

Метод `delete()` возвращает в качестве результата кортеж. Его первым элементом станет количество удаленных записей во всех моделях, что имеются в проекте. Вторым элементом является словарь, в котором ключи элементов представляют отдельные модели, а их значения — количество удаленных из них записей. Особой практической ценности этот результат не представляет.

6.5. Особенности обработки связанных записей

Сама природа реляционных баз данных приводит к необходимости создавать между таблицами многочисленные связи. Django предоставляет ряд инструментов для удобной работы со связанными записями: создания, установления и удаления связи.

6.5.1. Особенности обработки связи «один-со-многими»

При создании записи вторичной модели мы можем присвоить полю внешнего ключа непосредственно объект, представляющий связываемую запись первичной модели. Вот пример создания нового объявления, использующий такой подход:

```
>>> r = Rubric.objects.get(name='Мебель')
>>> r
<Rubric: Мебель>
>>> b = Bb()
>>> b.title = 'Диван'
>>> b.contents = 'Продавленный'
>>> b.price = 100
>>> b.rubric = r
>>> b.save()
```

Тот же самый подход можно применить при создании записей вызовом метода `create()` диспетчера записей (см. *разд. 6.2*).

Впоследствии мы можем связать запись вторичной модели с другой записью первичной таблицы, просто занеся последнее в поле внешнего ключа первой:

```
>>> r2 = Rubric.objects.get(name='Сантехника')
>>> b.rubric = r2
>>> b.save()
>>> b.rubric
<Rubric: Сантехника>
```

Модель, представляющая запись первичной таблицы, получает атрибут с именем вида *<имя связанной вторичной модели>*`_set`. Он хранит экземпляр класса `RelatedManager` из модуля `django.db.models.fields.related`, представляющий набор связанных записей вторичной таблицы и называемый *диспетчером обратной связи*.

> ***Внимание!***
>
> Описанный ранее атрибут класса получает имя *<имя связанной вторичной модели>*`_set` по умолчанию. Однако это имя можно изменить, при объявлении поля внешнего ключа указав его в параметре `related_name` конструктора класса поля (см. *разд. 4.4.1*).

Класс `RelatedManager` поддерживает два очень полезных метода:

- `add(`*<связываемая запись 1>*`, `*<связываемая запись 2>*` . . . `*<связываемая запись n>*`[, bulk=True])` — связывает с текущей записью первичной модели записи вторичной модели, переданные в качестве параметров.

  Если значение параметра `bulk` равно `True`, связывание записей будет осуществляться непосредственно отдачей СУБД SQL-команды, без манипуляций с объектами моделей, представляющих связываемые записи. Это поведение по умолчанию, и оно позволяет увеличить производительность.

  Если значение параметра `bulk` равно `False`, связывание записей будет выполняться посредством манипуляций объектами модели, представляющих связы-

ваемые записи. Это может пригодиться, если класс модели содержит переопределенные методы `save()` и `delete()`.

К моменту вызова метода `add()` текущая запись первичной модели должна быть сохранена. Не забываем, что в поле внешнего ключа записи вторичной модели сохраняется ключ записи первичной модели, а он может быть получен только после сохранения записи (если в модели используется стандартное ключевое поле целочисленного автоинкрементного типа).

Пример:

```
>>> r = Rubric.objects.get(name='Сельхозинвентарь')
>>> b = Bb.objects.get(pk=24)
>>> r.bb_set.add(b)
>>> b.rubric
<Rubric: Сельхозинвентарь>
```

- `create()` — унаследован от класса `Manager` и, помимо создания записи вторичной модели, также выполняет ее связывание с текущей записью первичной модели:

```
>>> b2 = r.bb_set.create(title='Лопата', price=10)
>>> b2.rubric
<Rubric: Сельхозинвентарь>
```

6.5.2. Особенности обработки связи «один-с-одним»

Связь такого рода очень проста, соответственно, программных инструментов для ее установления Django предоставляет немного.

В записи вторичной модели мы можем установить связь с записью первичной модели, присвоив последнюю полю внешнего ключа первой. Вот пример создания записи вторичной модели `AdvUser` (см. листинг 4.1) и связывания ее с записью первичной модели `User`, представляющей пользователя `admin`:

```
>>> from django.contrib.auth.models import User
>>> from testapp.models import AdvUser
>>> u = User.objects.get(username='admin')
>>> au = AdvUser.objects.create(user=u)
>>> au.user
<User: admin>
>>> u.advuser
<AdvUser: AdvUser object (1)>
```

Первичная модель при этом получит атрибут, хранящий связанную запись вторичной модели. Имя этого атрибута совпадет с именем вторичной модели. Следовательно, мы можем связать запись первичной модели с записью вторичной модели, также присвоив последнюю описанному ранее атрибуту. Вот пример связывания записи модели `User` с другой записью модели `AdvUser`:

```
>>> au2 = AdvUser.objects.get(pk=2)
>>> u.advuser = au2
```

```
>>> u.save()
>>> u.advuser
<AdvUser: AdvUser object (2)>
```

6.5.3. Особенности обработки связи «многие-со-многими»

Если между двумя моделями была установлена связь такого рода, перед собственно связыванием записей нам обязательно нужно их сохранить.

В случае связей «один-со-многими» и «один-с-одним» поле внешнего ключа, объявленное во вторичной модели, всегда хранит непосредственно объект первичной модели, представляющий связанную запись. Но в случае связи «многие-со-многими» это не так — атрибут, представляющий поле, хранит экземпляр класса `RelatedManager` — диспетчер обратной связи.

Следовательно, для установления связей между записями мы можем пользоваться следующими методами этого класса, первые два из которых уже знакомы нам по *разд. 6.5.1*:

- `add()` — для добавления указанных записей в число связанных с текущей записью:

```
>>> from testapp.models import Spare, Machine
>>> s1 = Spare.objects.create(name='Болт')
>>> s2 = Spare.objects.create(name='Гайка')
>>> s3 = Spare.objects.create(name='Шайба')
>>> s4 = Spare.objects.create(name='Шпилька')
>>> m1 = Machine.objects.create(name='Самосвал')
>>> m2 = Machine.objects.create(name='Тепловоз')
>>> m1.spares.add(s1, s2)
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (1)>, <Spare: Spare object (2)>]>
>>> s1.machine_set.all()
<QuerySet [<Machine: Machine object (1)>]>
>>> m1.spares.add(s4)
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (1)>, <Spare: Spare object (2)>],
<Spare: Spare object (4)>>
```

- `create()` — для создания новых записей связанной модели и одновременного связывания их с текущей записью:

```
>>> m1.spares.create(name='Винт')
<Spare: Spare object (5)>
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (1)>, <Spare: Spare object (2)>,
<Spare: Spare object (4)>], <Spare: Spare object (5)>]>
```

- `set(`*<последовательность связываемых записей>*`[, bulk=True][, clear=False])` — то же самое, что `add()`, но не добавляет указанные записи в число связанных с текущей записью, а заменяет ими те, что были связаны с ней ранее.

Если значение параметра `bulk` равно `True`, связывание записей будет осуществляться непосредственно отдачей СУБД SQL-команды, без манипуляций с объектами моделей, представляющих связываемые записи. Это поведение по умолчанию, и оно позволяет увеличить производительность.

Если значение параметра `bulk` равно `False`, связывание записей будет выполняться посредством манипуляций объектами модели, представляющих связываемые записи. Это может пригодиться, если класс модели содержит переопределенные методы `save()` и `delete()`.

Если значение параметра `clear` равно `True`, сначала будет выполнена очистка списка связанных записей, а потом заданные в методе записи будут связаны с текущей записью. Если же его значение равно `False`, указанные записи, отсутствующие в списке связанных, будут добавлены в него, а связанные записи, отсутствующие в последовательности указанных в вызове метода, будут удалены из списка связанных (поведение по умолчанию).

Пример:

```
>>> s5 = Spare.objects.get(pk=5)
>>> m1.spares.set([s2, s4, s5])
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (2)>, <Spare: Spare object (4)>,
<Spare: Spare object (5)>]>
```

- `remove(`*<удаляемая запись 1>, <удаляемая запись 2> . . . <удаляемая запись n>*`)` — удаляет указанные записи из списка связанных с текущей записью:

```
>>> m1.spares.remove(s4)
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (2)>, <Spare: Spare object (5)>]>
```

- `clear()` — полностью очищает список записей, связанных с текущей:

```
>>> m2.spares.set([s1, s2, s3, s4, s5])
>>> m2.spares.all()
<QuerySet [<Spare: Spare object (1)>, <Spare: Spare object (2)>,
<Spare: Spare object (3)>, <Spare: Spare object (4)>,
<Spare: Spare object (5)>]>
>>> m2.spares.clear()
>>> m2.spares.all()
<QuerySet []>
```

6.6. Произвольное переупорядочивание записей

Если у вторичной модели был указан параметр `order_with_respect_to`, ее записи, связанные с какой-либо записью первичной модели, могут быть произвольно переупорядочены (подробности — в *разд. 4.5*). Для этого следует получить запись первичной модели и вызвать у нее нужный метод:

- `get_<имя вторичной модели>_order()` — возвращает список ключей записей вторичной модели, связанных с текущей записью первичной модели:

```
class Bb(models.Model):
    . . .
    rubric = models.ForeignKey('Rubric')

    class Meta:
        order_with_respect_to = 'rubric'
. . .
>>> r = Rubric.objects.get(name='Мебель')
>>> r.get_bb_order()
[33, 34, 37]
```

- `set_<имя вторичной модели>_order(<список ключей записей вторичной модели>)` — задает новый порядок следования записей вторичной модели. В качестве параметра указывается список ключей записей, в котором ключи должны быть выстроены в нужном порядке. Пример:

```
>>> r.set_bb_order([37, 34, 33])
```

6.7. Массовая запись данных

Если возникает необходимость создать, исправить или удалить сразу большое количество записей, удобнее использовать средства Django для *массовой записи данных*. Это следующие методы класса `Manager` (они также поддерживаются производным от него классом `RelatedManager`):

- `bulk_create(<последовательность добавляемых записей>[, batch_size=None])` — добавляет в модель записи, указанные в *последовательности*.

  Параметр `batch_size` задает количество записей, которые будут добавлены в одной SQL-команде. Если он не указан, все заданные записи будут добавлены в одной команде (поведение по умолчанию).

  В качестве результата возвращается набор добавленных в модель записей, представленный экземпляром класса `QuerySet`.

  > ***Внимание!***
  >
  > Метод `bulk_create()` не создает объекты модели, а непосредственно отправляет СУБД создающую записи SQL-команду. Поэтому в записях, возвращенных им, ключевое поле не заполнено.
  >
  > Исходя из этого, нужно помнить, что метод `save()` при создании записей таким образом не вызывается. Если он переопределен в модели с целью произвести при сохранении какие-либо дополнительные действия, эти действия выполнены не будут.

  Пример:

```
>>> r = Rubric.objects.get(name='Бытовая техника')
>>> Bb.objects.bulk_create([
        Bb(title='Пылесос', content='Хороший, мощный', price=1000,
           rubric=r),
```

```
        Bb(title='Стиральная машина', content='Автоматическая',
           price=3000, rubric=r)
    ])
[<Bb: Bb object (None)>, <Bb: Bb object (None)>]
```

- `update(<новые значения полей>)` — исправляет все записи в наборе, задавая для них *новые значения полей*. Эти значения задаются для параметров метода, чьи имена совпадают с именами нужных полей модели.

  В качестве результата возвращается количество исправленных записей.

  Метод `update()` также не вызывает метод `save()` модели, что в случае его переопределения станет критичным.

  Для примера запишем в объявления, в которых не была указана цена, какую-либо цену, так сказать, по умолчанию:

```
>>> Bb.objects.filter(price=None).update(price=10)
4
```

- `delete()` — удаляет все записи в наборе. В качестве результата возвращает словарь, аналогичный таковому, возвращаемому методом `delete()` модели (см. *разд. 6.4*).

  Метод `delete()` не вызывает метод `delete()` модели, что в случае его переопределения станет критичным.

  Для примера удалим все объявления, в которых не было указано описание товара:

```
>>> Bb.objects.filter(content=None).delete()
(2, {'bboard.Bb': 2})
```

Методы для массовой записи данных работают быстрее, чем программные инструменты моделей, поскольку напрямую «общаются» с базой данных. Однако при их использовании программные инструменты, определенные в моделях (в частности, автоматическое получение ключей записей и выполнение методов `save()` и `delete()`), не работают.

6.8. Выполнение валидации модели

Как правило, валидация непосредственно модели выполняется редко — обычно это делается на уровне формы, связанной с ней. Но на всякий случай давайте выясним, как проверить модель на корректность заносимых в нее данных.

Валидация модели запускает метод `full_clean()`:

```
full_clean([exclude=None][,][validate_unique=True])
```

Параметр `exclude` задает последовательность имен полей, значения которых проверяться не будут. Если он опущен, будут проверяться все поля.

Если параметру `validate_unique` присвоить значение `True`, то, если модель содержит уникальные поля, будет также проверяться уникальность заносимых в них зна-

чений (поведение по умолчанию). Если значение этого параметра — `False`, такая проверка проводиться не будет.

Метод не возвращает никакого результата. Если в модель занесены некорректные данные, он возбуждает исключение `ValidationError` из модуля `django.core.exceptions`.

В последнем случае в атрибуте `message_dict` модели будет храниться словарь с сообщениями об ошибках. Ключи элементов будут соответствовать полям модели, а значениями элементов станут списки с сообщениями об ошибках.

Примеры:

```
>>> # Извлекаем из модели запись с заведомо правильными данными
>>> # и проверяем ее на корректность. Запись корректна
>>> b = Bb.objects.get(pk=1)
>>> b.full_clean()

>>> # Создаем новую "пустую" запись и выполняем ее проверку. Запись
>>> # некорректна, т. к. в обязательные поля не занесены значения
>>> b = Bb()
>>> b.full_clean()
Traceback (most recent call last):
    raise ValidationError(errors)
django.core.exceptions.ValidationError: {
  'title': ['This field cannot be blank.'],
  'rubric': ['This field cannot be blank.'],
  'content': ['Укажите описание продаваемого товара']
}
```

ГЛАВА 7

Выборка данных

Выборка данных из базы — весьма широкое понятие. Оно включает в себя не только извлечение всех записей, что имеются в модели, но и поиск одной-единственной записи по ее ключу или значению какого-либо поля, фильтрацию записей по заданным критериям, их сортировку, проверку, есть ли записи в наборе, вычисление количества записей, расчет среднего арифметического значений, хранящихся в определенном поле, и многое другое. Наконец, извлечение значений из полей записи — это тоже выборка.

Механизм моделей Django поддерживает развитые средства для выборки данных. Бóльшую их часть мы рассмотрим в этой главе.

7.1. Извлечение значений из полей записи

Получить значения полей записи можно из атрибутов класса модели, представляющих эти поля:

```
>>> from bboard.models import Bb
>>> b = Bb.objects.get(pk=1)
>>> b.title
'Дача'
>>> b.content
'Общество "Двухэтажники". Два этажа, кирпич, свет, газ, канализация'
>>> b.price
500000.0
```

Атрибут класса `pk` хранит значение ключа для текущей записи:

```
>>> b.pk
1
```

Им удобно пользоваться, когда в модели есть явно созданное ключевое поле с именем, отличным от стандартного `id`, — нам не придется вспоминать, как называется это поле.

7.2. Доступ к связанным записям

Средства, предназначенные для доступа к связанным записям и создаваемые самим фреймворком, различаются для разных типов связей.

Для связи *«один-со-многими»* из вторичной модели можно получить связанную запись первичной модели посредством атрибута класса, представляющего поле внешнего ключа:

```
>>> b.rubric
<Rubric: Недвижимость>
```

Мы можем получить значение любого поля связанной записи:

```
>>> b.rubric.name
'Недвижимость'
>>> b.rubric.pk
1
```

В классе первичной модели будет создан атрибут с именем вида *<имя связанной вторичной модели>*_set. Он хранит диспетчер обратной связи, представленный экземпляром класса RelatedManager, который является производным от класса диспетчера записей Manager и, таким образом, поддерживает все его методы.

Диспетчер обратной связи, в отличие от диспетчера записей, манипулирует только записями, связанными с текущей записью первичной модели.

Давайте посмотрим, чем торгуют в рубрике «Недвижимость»:

```
>>> from bboard.models import Rubric
>>> r = Rubric.objects.get(name='Недвижимость')
>>> for bb in r.bb_set.all(): print(bb.title)
...
Земельный участок
Дом
Дача
```

Посмотрим, есть ли там что-нибудь дешевле 10 000 руб.:

```
>>> for bb in r.bb_set.filter(price__lte=10000): print(bb.title)
...
```

Похоже, что ничего...

> ***На заметку***
>
> Имеется возможность задать другое имя для атрибута класса первичной модели, хранящего диспетчер обратной связи. Имя указывается в параметре related_name конструктора класса поля:
>
> ```
> class Bb(models.Model):
>     rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
>                                related_name='entries')
> . . .
> ```
>
> Теперь мы можем получить доступ к диспетчеру обратной связи по заданному имени:
>
> ```
> >>> for bb in r.entries.all(): print(bb.title)
> ```

Если мы установили связь *«один-с-одним»*, все гораздо проще. Из вторичной модели можно получить доступ к связанной записи первичной модели через атрибут класса, представляющий поле внешнего ключа:

```
>>> from testapp.models import AdvUser
>>> au = AdvUser.objects.first()
>>> au.user
<User: admin>
>>> au.user.username
'admin'
```

Из первичной модели можно получить доступ к связанной записи вторичной модели через атрибут класса, чье имя совпадает с именем вторичной модели:

```
>>> from django.contrib.auth import User
>>> u = User.objects.first()
>>> u.advuser
<AdvUser: AdvUser object (1)>
```

В случае связи *«многие-со-многими»* через атрибут класса ведущей модели, представляющий поле внешнего ключа, доступен диспетчер обратной связи, представляющий набор связанных записей ведомой модели:

```
>>> from testapp.models import Machine
>>> m = Machine.objects.get(pk=1)
>>> m.name
'Самосвал'
>>> for s in m.spares.all(): print(s.name)
...
Гайка
Винт
```

В ведомой модели будет присутствовать атрибут класса *<имя связанной ведущей модели>*`_set`. Его можно использовать для доступа к записям связанной ведущей модели. Пример:

```
>>> from testapp.models import Spare
>>> s = Spare.objects.get(name='Гайка')
>>> for m in s.machine_set.all(): print(m.name)
...
Самосвал
```

7.3. Выборка записей

Теперь выясним, как выполнить выборку из модели записей — как всех, так и лишь тех, что удовлетворяют определенным условиям.

7.3.1. Выборка всех записей

Все модели поддерживают атрибут класса `objects`. Он хранит диспетчер записей (представленный экземпляром класса `Manager`), который позволяет манипулировать всеми записями, что хранятся в модели.

Метод `all()`, поддерживаемый классом `Manager`, возвращает набор из всех записей модели в виде экземпляра класса `QuerySet`. Последний обладает функциональностью последовательности и поддерживает итерационный протокол. Так что мы можем просто перебрать записи набора и выполнить над ними какие-либо действия в обычном цикле `for...in`. Пример:

```
>>> for r in Rubric.objects.all(): print(r.name, end=' ')
...
Бытовая техника Мебель Недвижимость Растения Сантехника Сельхозинвентарь
Транспорт
```

Класс `RelatedManager` является производным от класса `Manager`, следовательно, тоже поддерживает метод `all()`. Только в этом случае возвращаемый им набор будет содержать лишь связанные записи. Пример:

```
>>> r = Rubric.objects.get(name='Недвижимость')
>>> for bb in r.bb_set.all(): print(bb.title)
...
Земельный участок
Дом
Дача
```

7.3.2. Извлечение одной записи

Ряд методов позволяют извлечь из модели всего одну запись:

- `first()` — возвращает первую запись набора или `None`, если набор пуст:

  ```
  >>> b = Bb.objects.first()
  >>> b.title
  'Стиральная машина'
  ```

- `last()` — возвращает последнюю запись набора или `None`, если набор пуст:

  ```
  >>> b = Bb.objects.last()
  >>> b.title
  'Дача'
  ```

  Оба эти метода при поиске записи учитывают сортировку набора записей, заданную либо вызовом метода `order_by()`, либо в параметре `ordering` модели (см. *разд. 4.5*);

- `earliest([`*<имя поля 1>, <имя поля 2> . . . <имя поля n>*`])` — возвращает запись, у которой значение даты и времени, записанное в полях с указанными именами, является наиболее ранним.

  Предварительно выполняется временная сортировка записей по указанным полям. По умолчанию записи сортируются по возрастанию значений этих полей. Чтобы задать сортировку по убыванию, имя поля нужно предварить символом «минус».

  Сначала проверяется значение, записанное в поле, чье имя указано первым. Если это значение одинаково у нескольких записей, проверяется значение следующего поля и т. д.

Если в модели указан параметр `get_latest_by`, задающий поля для просмотра (см. *разд. 4.5*), метод можно вызвать без параметров.

Если ни одной подходящей записи не нашлось, возбуждается исключение `DoesNotExist`.

Ищем самое раннее из оставленных на сайте объявлений:

```
>>> b = Bb.objects.earliest('published')
>>> b.title
'Дача'
```

А теперь найдем самое позднее, для чего укажем сортировку по убыванию:

```
>>> b = Bb.objects.earliest('-published')
>>> b.title
'Стиральная машина'
```

- `latest([`*<имя поля 1>, <имя поля 2> . . . <имя поля n>*`])` — то же самое, что `earliest()`, но ищет запись с наиболее поздним значением даты и времени:

```
>>> b = Bb.objects.latest('published')
>>> b.title
'Стиральная машина'
```

Все эти методы поддерживаются классами `Manager`, `RelatedManager` и `QuerySet`. Следовательно, мы можем вызывать их также у набора связанных записей:

```
>>> # Найдем самое раннее объявление о продаже транспорта
>>> r = Rubric.objects.get(name='Транспорт')
>>> b = r.bb_set.earliest('published')
>>> b.title
'Мотоцикл'

>>> # Извлекаем самое первое объявление с ценой не менее 10 000 руб.
>>> b = Bb.objects.filter(price__gte=10000).first()
>>> b.title
'Земельный участок'
```

> ***Внимание!***
>
> Все методы, которые мы рассмотрим далее, также поддерживаются классами `Manager`, `RelatedManager` и `QuerySet`, вследствие чего могут быть вызваны не только у диспетчера записей, но и у диспетчера обратной связи и набора записей.

7.3.3. Получение количества записей в наборе

Два следующих метода позволят нам получить количество записей, имеющихся в наборе, равно как и быстро проверить, есть ли там записи.

- `exists()` — возвращает `True`, если в наборе есть записи, и `False`, если набор записей пуст:

```
>>> # Проверяем, продают ли у нас сантехнику
>>> r = Rubric.objects.get(name='Сантехника')
>>> Bb.objects.filter(rubric=r).exists()
False
```

- `count()` — возвращает количество записей, имеющихся в наборе:

```
>>> # А сколько у нас всего объявлений?..
>>> Bb.objects.count()
12
```

Эти методы выполняются очень быстро, поэтому для проведения простых проверок рекомендуется применять именно их.

7.3.4. Поиск записи

Вероятно, наиболее часто приходится выполнять поиск записи, удовлетворяющей определенным условиям, обычно по значению ее ключа.

Для поиска записи служит метод `get(<условия поиска>)`. Сами *условия поиска* записываются в виде именованных параметров, каждый из которых представляет одноименное поле. Значение, присвоенное такому параметру, задает искомое значение для поля.

Если совпадающая с заданными условиями запись нашлась, она будет возвращена в качестве результата. Если ни одной подходящей записи не было найдено, будет возбуждено исключение `DoesNotExist`, класс которого является вложенным в класс модели, чья запись не была найдена. Если же подходящих записей оказалось несколько, возбуждается исключение `MultipleObjectsReturned` из модуля `django.core.exceptions`.

Рассмотрим пару наиболее типичных примеров:

- найдем запись, представляющую рубрику «Растения»:

```
>>> r = Rubric.objects.get(name='Растения')
>>> r.pk
7
```

- найдем рубрику с ключом 5:

```
>>> r = Rubric.objects.get(pk=5)
>>> r
<Rubric: Мебель>
```

Если в методе `get()` указать сразу несколько условий поиска, они будут объединяться по правилам логического И. Для примера давайте найдем запись с ключом 5 И названием «Сантехника»:

```
>>> r = Rubric.objects.get(pk=5, name='Сантехника')
. . .
bboard.models.DoesNotExist: Rubric matching query does not exist.
```

Такой записи нет, вследствие чего было возбуждено исключение `DoesNotExist`.

Если в модели есть хотя бы одно поле типа `DateField` или `DateTimeField`, модель получает поддержку методов с именами вида `get_next_by_`*<имя поля>*`()` и `get_previous_by_`*<имя поля>*`()`. Формат вызова у обоих методов одинаковый:

```
get_next_by_<имя поля> | get_previous_by_<имя поля>([<условия поиска>])
```

Первый метод возвращает запись, чье поле с указанным *именем* хранит следующее в порядке увеличения значение даты, второй метод — запись с предыдущим значением. Если указаны *условия поиска*, они также принимаются во внимание. Примеры:

```
>>> b = Bb.objects.get(pk=1)
>>> b.title
'Дача'
>>> # Следующее в хронологическом порядке объявление сообщает
>>> # о продаже дома
>>> b2 = b.get_next_by_published()
>>> b2.title
'Дом'
>>> # Теперь найдем следующее в хронологическом порядке объявление,
>>> # в котором заявленная цена меньше 1 000 руб. Оно гласит о продаже
>>> # мотоцикла
>>> b3 = b.get_next_by_published(price__lt=1000)
>>> b3.title
'Мотоцикл'
```

Если текущая модель является вторичной, и у нее было задано произвольное переупорядочивание записей, связанных с одной и той же записью первичной модели (т. е. был указан параметр `order_with_respect_to`, описанный в *разд. 4.5*), эта вторичная модель получает поддержку методов `get_next_in_order()` и `get_previous_in_order()`. Оба эти метода вызываются у какой-либо записи вторичной модели: первый метод возвращает следующую в установленном порядке запись, а второй — предыдущую. Пример:

```
>>> r = Rubric.objects.get(name='Мебель')
>>> bb2 = r.bb_set.get(pk=34)
>>> bb2.pk
34
>>> bb1 = bb2.get_previous_in_order()
>>> bb1.pk
37
>>> bb3 = bb2.get_next_in_order()
>>> bb3.pk
33
```

7.3.5. Фильтрация записей

Под *фильтрацией* здесь понимается отбор только тех записей, которые удовлетворяют заданным критериям, причем, в отличие от поиска, таких записей может быть

произвольное количество (в том числе и ноль, что не воспринимается фреймворком как нештатная ситуация и не приводит к возбуждению исключения).

Для выполнения фильтрации Django предусматривает два следующих метода — диаметральные противоположности друг друга:

- `filter(<условия фильтрации>)` — отбирает из текущего набора только те записи, которые удовлетворяют заданным *условиям фильтрации*. Сами *условия фильтрации* задаются точно в таком же формате, что и условия поиска в вызове метода `get()` (см. *разд. 7.3.4*). Пример:

```
>>> # Отбираем только объявления с ценой не менее 10 000 руб.
>>> for b in Bb.objects.filter(price__gte=10000):
        print(b.title, end=' ')
...
Земельный участок Велосипед Дом Дача
```

- `exclude(<условия фильтрации>)` — то же самое, что `filter()`, но, наоборот, отбирает записи, не удовлетворяющие заданным условиям фильтрации:

```
>>> # Отбираем все объявления, кроме тех, в которых указана цена
>>> # не менее 10 000 руб.
>>> for b in Bb.objects.exclude(price__gte=10000):
        print(b.title, end=' ')
...
Стиральная машина Пылесос Мотоцикл
```

Поскольку оба эти метода поддерживаются классом `QuerySet`, мы можем «сцеплять» их вызовы друг с другом. Для примера найдем все объявления о продаже недвижимости ценой менее 1 000 000 руб.:

```
>>> r = Rubric.objects.get(name='Недвижимость')
>>> for b in Bb.objects.filter(rubric=r).filter(price__lt=1000000):
        print(b.title, end=' ')
...
Земельный участок Дача
```

Впрочем, такой запрос на фильтрацию можно записать и в одном вызове метода `filter()`:

```
>>> for b in Bb.objects.filter(rubric=r, price__lt=1000000):
        print(b.title, end=' ')
...
Земельный участок Дача
```

7.3.6. Написание условий фильтрации

Если в вызове метода `filter()` или `exclude()` мы запишем условие фильтрации в формате *<имя поля>=<значение поля>*, Django будет отбирать записи, у которых значение заданного *поля* точно совпадает с указанной величиной *значения*, причем в случае строковых и текстовых полей сравнение выполняется с учетом регистра.

Но что делать, если нам нужно выполнить сравнение без учета регистра или отобрать записи, у которых значение поля больше или меньше заданной величины? Использовать *модификаторы*. Такой модификатор добавляется к имени поля и отделяется от него двойным символом подчеркивания: *<имя поля>__<модификатор>*. Все поддерживаемые Django модификаторы приведены в табл. 7.1.

Таблица 7.1. Модификаторы

Модификатор	Описание
exact	Точное совпадение значения с учетом регистра символов. В плане получаемого результата аналогичен записи *<имя поля>=<значение поля>*. Применяется в случаях, если имя какого-либо поля совпадает с ключевым словом Python: `class__exact='superclass'`
iexact	Точное совпадение значения без учета регистра символов
contains	Заданное значение должно присутствовать в значении, хранящемся в поле. Регистр символов учитывается
icontains	Заданное значение должно присутствовать в значении, хранящемся в поле. Регистр символов не учитывается
startswith	Заданное значение должно присутствовать в начале значения, хранящегося в поле. Регистр символов учитывается
istartswith	Заданное значение должно присутствовать в начале значения, хранящегося в поле. Регистр символов не учитывается
endswith	Заданное значение должно присутствовать в конце значения, хранящегося в поле. Регистр символов учитывается
iendswith	Заданное значение должно присутствовать в конце значения, хранящегося в поле. Регистр символов не учитывается
lt	Значение, хранящееся в поле, должно быть меньше заданного
lte	Значение, хранящееся в поле, должно быть меньше заданного или равно ему
gt	Значение, хранящееся в поле, должно быть больше заданного
gte	Значение, хранящееся в поле, должно быть больше заданного или равно ему
range	Значение, хранящееся в поле, должно находиться внутри заданного диапазона, включая его начальное и конечное значения. Диапазон значений задается в виде кортежа, первым элементом которого записывается начальное значение, вторым — конечное
date	Значение, хранящееся в поле, рассматривается как дата. Пример: `published__date=datetime.date(2018, 6, 1)`

year	Из значения даты, хранящегося в поле, извлекается год, и дальнейшее сравнение выполняется с ним. Примеры:
	`published__year=2018`
	`published__year__lte=2017`

month	Из значения даты, хранящегося в поле, извлекается номер месяца, и дальнейшее сравнение выполняется с ним
day	Из значения даты, хранящегося в поле, извлекается число, и дальнейшее сравнение выполняется с ним

Таблица 7.1 (окончание)

Модификатор	Описание
week	Из значения даты, хранящегося в поле, извлекается номер недели, и дальнейшее сравнение выполняется с ним
week_day	Из значения даты, хранящегося в поле, извлекается номер дня недели, и дальнейшее сравнение выполняется с ним
quarter	Из значения даты, хранящегося в поле, извлекается номер квартала года (от 1 до 4), и дальнейшее сравнение выполняется с ним
time	Значение, хранящееся в поле, рассматривается как время. Пример: `published__time=datetime.time(12, 0)`
hour	Из значения времени, хранящегося в поле, извлекаются часы, и дальнейшее сравнение выполняется с ними. Примеры: `published__hour=12` `published__hour__gte=13`
minute	Из значения времени, хранящегося в поле, извлекаются минуты, и дальнейшее сравнение выполняется с ними
second	Из значения времени, хранящегося в поле, извлекаются секунды, и дальнейшее сравнение выполняется с ними
isnull	Если `True`, указанное поле должно хранить значение `null` (быть пустым). Если `False`, поле должно хранить значение, отличное от `null` (быть заполненным). Пример: `content__isnull=False`
in	Значение, хранящееся в поле, должно присутствовать в указанном списке, кортеже или наборе записей `QuerySet`. Пример: `pk__in=(1, 2, 3, 4)`
regex	Значение, хранящееся в поле, должно совпадать с заданным регулярным выражением. Регистр символов учитывается. Пример: `content__regex='газ|вода'`
iregex	Значение, хранящееся в поле, должно совпадать с заданным регулярным выражением. Регистр символов не учитывается

Мы уже использовали некоторые из этих модификаторов ранее, когда программировали наш первый сайт и рассматривали различные функциональные возможности Django.

7.3.7. Фильтрация по значениям полей связанных записей

А что делать, если нам нужно отфильтровать записи по значениям полей записей из связанной модели? И для таких случаев у этого замечательного фреймворка есть удобные средства.

Чтобы выполнить фильтрацию записей вторичной модели по значениям полей из первичной модели, мы запишем условие фильтрации в формате *<имя поля внешнего*

*ключа*__*<имя поля первичной модели>*. Для примера давайте выберем все объявления о продаже транспорта:

```
>>> for b in Bb.objects.filter(rubric__name='Транспорт'):
        print(b.title, end=' ')
...
Велосипед Мотоцикл
```

Для выполнения фильтрации записей первичной модели по значениям из полей вторичной модели следует записать условие вида *<имя вторичной модели>*__*<имя поля вторичной модели>*. В качестве примера выберем все рубрики, в которых есть объявления о продаже с заявленной ценой более 10 000 руб.:

```
>>> for r in Rubric.objects.filter(bb__price__gt=10000):
        print(r.name, end=' ')
...
Недвижимость Недвижимость Недвижимость Транспорт
```

Здесь у нас получились три одинаковые записи «Недвижимость», поскольку в этой рубрике хранятся три объявления, удовлетворяющие заявленным условиям фильтрации. О способе выводить только уникальные записи, мы узнаем позже.

Как видим, в подобного рода условиях фильтрации мы можем применять и модификаторы!

> ***НА ЗАМЕТКУ***
>
> Имеется возможность указать другой фильтр, который будет применяться вместо имени вторичной модели в условиях такого рода. Он указывается в параметре `related_query_name` конструктора класса поля:
>
> ```
> class Bb(models.Model):
>     rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
>                                related_query_name='entry')
> ```
>
> После этого мы можем записать рассмотренное ранее выражение в таком виде:
>
> ```
> >>> for r in Rubric.objects.filter(entry__price__gt=10000):
>         print(r.name, end=' ')
> ```

Все это касалось связей «один-со-многими» и «один-с-одним». А что же связи «многие-со-многими»? В них действуют те же самые правила. Убедимся сами:

```
>>> # Получаем все машины, в состав которых входят гайки
>>> for m in Machine.objects.filter(spares__name='Гайка'):
        print(m.name, end=' ')
...
Самосвал
>>> # Получаем все детали, входящие в состав самосвала
>>> for s in Spare.objects.filter(machine__name='Самосвал'):
        print(s.name, end=' ')
...
Гайка Винт
```

7.3.8. Сравнение со значениями других полей

До этого момента при написании условий фильтрации мы сравнивали значения, хранящиеся в полях, с константами. Но иногда бывает необходимо сравнить значения из одного поля записи со значением другого ее поля. Может ли Django и в этом нам помочь?

Безусловно. Для этого предназначен класс с именем `F`, объявленный в модуле `django.db.models`. Вот формат его конструктора:

`F(`*<имя поля модели, с которым должно выполняться сравнение>*`)`

Имя поля модели записывается в виде строки.

Получив экземпляр этого класса, мы можем использовать его в правой части любого условия.

Вот пример извлечения объявлений, в которых название товара встречается в тексте его описания:

```
>>> from django.db.models import F
>>> f = F('title')
>>> for b in Bb.objects.filter(content__icontains=f):
        print(b.title, end=' ')
```

Экземпляры класса `F` можно использовать не только при фильтрации, но и для занесения нового значения в поля модели. Например, так можно уменьшить цены во всех объявлениях вдвое:

```
>>> f = F('price')
>>> for b in Bb.objects.all():
        b.price = f / 2
        b.save()
```

7.3.9. Сложные условия фильтрации

По большей части, возможности методов `filter()` и `exclude()` не так уж и велики — они позволяют нам написать произвольное количество условий, объединив их по правилам логического И. Если же мы хотим объединять условия по правилам логического ИЛИ, этими методами не обойтись.

Нам понадобится класс `Q` из модуля `django.db.models`. Его конструктор записывается в следующем формате:

`Q(`*<условие фильтрации>*`)`

Условие фильтрации записывается в том же виде, что применяется в вызовах методов `filter()` и `exclude()`. Отметим только, что условие должно быть всего одно.

Полученный экземпляр класса `Q` можно использовать в вызовах упомянутых ранее методов вместо обычных условий фильтрации.

Также можно объединять два экземпляра класса `Q` посредством операторов `&` и `|`, которые обозначают, соответственно, логическое И и ИЛИ. Для выполнения логи-

ческого НЕ применяется оператор ~. Все эти три оператора в качестве результата возвращают новый экземпляр класса Q.

Пример выборки объявлений о продаже ИЛИ недвижимости, ИЛИ бытовой техники:

```
>>> from django.db.models import Q
>>> q = Q(rubric__name='Недвижимость') | \
        Q(rubric__name='Бытовая техника')
>>> for b in Bb.objects.filter(q): print(b.title, end=' ')
...
Пылесос Стиральная машина Земельный участок Дом Дача
```

А вот пример выборки объявлений о продаже транспорта, в которых цена НЕ больше 5 000 руб.:

```
>>> q = Q(rubric__name='Транспорт') & ~Q(price__gt=5000)
>>> for b in Bb.objects.filter(q): print(b.title, end=' ')
...
Мотоцикл
```

7.3.10. Выборка уникальных записей

В *разд. 7.3.7* мы рассматривали пример фильтрации записей первичной модели по значениям полей из вторичной модели и получили в результате набор из четырех записей, три из которого были одинаковыми. Такая коллизия не всегда приемлема.

Для вывода только уникальных записей служит метод `distinct()`:

```
distinct([<имя поля 1>, <имя поля 2> . . . <имя поля n>])
```

Если мы пользуемся СУБД PostgreSQL, то можем указать в параметрах *имена полей*, значения которых определят уникальность записей. Если же не задавать параметров, уникальность каждой записи будет определяться значениями всех ее полей.

Перепишем пример из *разд. 7.3.7*, чтобы он выводил только уникальные записи:

```
>>> for r in Rubric.objects.filter(bb__price__gt=10000).distinct():
        print(r.name, end=' ')
...
Недвижимость Транспорт
```

7.3.11. Выборка указанного количества записей

И наконец, мы можем получить набор, содержащий не все записи, удовлетворяющие заданным условиям фильтрации, а определенное их количество. Также мы можем указать номер записи, с которой должна начаться выборка.

Для этого мы применим хорошо знакомый нам оператор взятия среза `[]`. Записывается он точно так же, как и в случае использования обычных последовательностей Python. Единственное исключение — не поддерживаются отрицательные индексы.

Вот три примера:

```
>>> # Извлекаем первые три рубрики
>>> Rubric.objects.all()[:3]
<QuerySet [<Rubric: Бытовая техника>, <Rubric: Мебель>,
<Rubric: Недвижимость>]>

>>> # Извлекаем все рубрики, начиная с шестой
>>> Rubric.objects.all()[5:]
<QuerySet [<Rubric: Сельхозинвентарь>, <Rubric: Транспорт>]>

>>> # Извлекаем третью и четвертую рубрики
>>> Rubric.objects.all()[2:4]
<QuerySet [<Rubric: Недвижимость>, <Rubric: Растения>]>
```

7.4. Сортировка записей

Для сортировки записей в наборе применяется метод `order_by()`:

```
order_by([<имя поля 1>, <имя поля 2> . . . <имя поля n>])
```

В качестве параметров указываются *имена полей*, представленные в виде строк. Сначала сортировка выполняется по значению первого поля. А если у каких-то записей оно хранит одно и то же значение, производится сортировка по второму полю и т. д.

По умолчанию сортировка выполняется по возрастанию значения поля. Чтобы отсортировать по убыванию значения, следует предварить имя поля знаком «минус».

Вот пара примеров:

```
>>> # Сортируем рубрики по названиям
>>> for r in Rubric.objects.order_by('name'): print(r.name, end=' ')
...
Бытовая техника Мебель Недвижимость Растения Сантехника Сельхозинвентарь
Транспорт

>>> # Сортируем объявления сначала по названиям рубрик, а потом по
>>> # убыванию цены
>>> for b in Bb.objects.order_by('rubric__name', '-price'):
        print(b.title, end=' ')
...
Стиральная машина Пылесос Дом Дача Земельный участок Велосипед Мотоцикл
```

Каждый вызов метода `order_by()` отменяет параметры сортировки, заданные предыдущим вызовом или в параметрах модели (параметр `ordering`). Поэтому, если записать нечто, наподобие следующему:

```
Bb.objects.order_by('rubric__name').order_by('-price')
```

объявления будут отсортированы только по убыванию цены.

Если передать методу `order_by()` в качестве единственного параметра строку `'?'`, записи будут выстроены в случайном порядке. Однако это может отнять много времени.

Вызов метода `reverse()` меняет порядок сортировки записей на противоположный:

```
>>> for r in Rubric.objects.order_by('name').reverse():
        print(r.name, end=' ')
...
Транспорт Сельхозинвентарь Сантехника Растения Недвижимость Мебель
Бытовая техника
```

Чтобы отменить сортировку, заданную предыдущим вызовом метода `order_by()` или в параметре `ordering` модели, следует вызвать метод `order_by()` без параметров.

7.5. Агрегатные вычисления

Агрегатные вычисления затрагивают значения определенного поля всех записей, что есть в модели. К такого рода действиям относится вычисление количества объявлений — всех или удовлетворяющих определенным условиям, среднего арифметического цены, наименьшего и наибольшего значения цены и т. п.

Каждое из возможных действий, выполняемых при агрегатных вычислениях, представляется определенной *агрегатной функцией*. Так, существуют агрегатные функции для подсчета количества записей, среднего арифметического, минимума, максимума и др.

7.5.1. Вычисления по всем записям модели

Если нужно провести агрегатное вычисление по всем записям модели, нет ничего лучше метода `aggregate()`:

```
aggregate(<агрегатная функция 1>, <агрегатная функция 2> . . .
<агрегатная функция n>)
```

Сразу отметим два момента. Во-первых, *агрегатные функции* представляются экземплярами особых классов, которые объявлены в модуле `django.db.models` и которые мы рассмотрим чуть позже. Во-вторых, возвращаемый методом результат — словарь Python, в котором отдельные элементы представляют результаты выполнения соответствующих им агрегатных функций.

Агрегатные функции можно указать в виде как позиционных, так и именованных параметров:

- если агрегатная функция указана в виде *позиционного* параметра, в результирующем словаре будет создан элемент с ключом вида *<имя поля, по которому выполняется вычисление>__<имя класса агрегатной функции>*, хранящий результат выполнения агрегатной функции. В качестве примера определим наименьшее значение цены, указанное в объявлениях:

```
>>> from django.db.models import Min
>>> Bb.objects.aggregate(Min('price'))
{'price__min': 10.0}
```

- если агрегатная функция указана в виде *именованного* параметра, ключ элемента, создаваемого в словаре, будет совпадать с именем этого параметра. Выясним наибольшее значение цены в объявлениях:

```
>>> from django.db.models import Max
>>> Bb.objects.aggregate(max_price=Max('price'))
{'max_price': 50000000.0}
```

Мы можем указывать в вызове метода `aggregate()` произвольное количество агрегатных функций:

```
>>> result = Bb.objects.aggregate(Min('price'), Max('price'))
>>> result['price__min'], result['price__max']
(10.0, 50000000.0)
```

А использовав для указания именованный параметр (с позиционным такой номер не пройдет) — выполнять вычисления над результатами агрегатных функций:

```
>>> result = Bb.objects.aggregate(diff=Max('price')-Min('price'))
>>> result['diff']
49999990.0
```

7.5.2. Вычисления по группам записей

В случае, если нужно провести отдельное агрегатное вычисление по группам записей, сформированным по определенному критерию (например, узнать, сколько объявлений находится в каждой рубрике), следует применить метод `annotate()`:

```
annotate(<агрегатная функция 1>, <агрегатная функция 2> . . .
<агрегатная функция n>)
```

Как видим, этот метод вызывается так же, как и `aggregate()` (см. *разд. 7.5.1*). Есть только два существенных отличия:

- в качестве результата он возвращает новый набор записей;
- каждая запись из возвращенного им набора будет содержать атрибут, чье имя генерируется по тем же правилам, что и ключ элемента в словаре, возвращенном методом `aggregate()`. Этот атрибут будет содержать результат выполнения агрегатной функции.

Пример подсчета количества объявлений, оставленных в каждой из рубрик (агрегатная функция указана в позиционном параметре):

```
>>> from django.db.models import Count
>>> for r in Rubric.objects.annotate(Count('bb')):
        print(r.name, ': ', r.bb__count, sep='')
...
Бытовая техника: 2
Мебель: 0
```

```
Недвижимость: 5
Растения: 0
Сантехника: 0
Сельхозинвентарь: 0
Транспорт: 5
```

То же самое, но теперь агрегатная функция указана в именованном параметре:

```
>>> for r in Rubric.objects.annotate(cnt=Count('bb')):
...     print(r.name, ': ', r.cnt, sep='')
```

Посчитаем для каждой из рубрик минимальную цену, указанную в объявлении:

```
>>> for r in Rubric.objects.annotate(min=Min('bb__price')):
...     print(r.name, ': ', r.min, sep='')
...
Бытовая техника: 1000.0
Мебель: None
Недвижимость: 10.0
Растения: None
Сантехника: None
Сельхозинвентарь: None
Транспорт: 10.0
```

У рубрик, не содержащих объявлений, значение минимальной цены равно `None`.

Использовав именованный параметр, мы фактически создаем в наборе записей новое поле (более подробно об этом приеме разговор пойдет позже). Следовательно, мы можем выполнить фильтрацию по значению этого поля. Давайте же уберем из полученного ранее результата рубрики, в которых нет объявлений:

```
>>> for r in Rubric.objects.annotate(cnt=Count('bb'),
                                     min=Min('bb__price')).filter(cnt__gt=0):
...     print(r.name, ': ', r.min, sep='')
...
Бытовая техника: 1000.0
Недвижимость: 10.0
Транспорт: 10.0
```

7.5.3. Агрегатные функции

Пора бы рассмотреть агрегатные функции, поддерживаемые Django. Все они представляются классами, объявленными в модуле `django.db.models`.

Конструкторы этих классов принимают ряд необязательных параметров. Указывать их можно лишь в том случае, если сама агрегатная функция задана в вызове метода `aggregate()` или `annotate()` с помощью именованного параметра — в противном случае мы получим сообщение об ошибке.

- `Count` — вычисляет количество записей. Вот формат его конструктора:

  ```
  Count(<имя поля>[, distinct=False][, filter=None])
  ```

Первым параметром указывается *имя поля*, имеющегося в записях, чье количество должно быть подсчитано. Если нужно узнать количество записей вторичной модели, связанных с записью первичной модели, следует указать имя вторичной модели.

Если значение параметра `distinct` равно `True`, будут подсчитываться только уникальные записи, если `False` — все записи (поведение по умолчанию). Параметр `filter` указывает условие для фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*). По умолчанию фильтрация записей не выполняется.

Пример подсчета объявлений, в которых указана цена более 100 000 руб., по рубрикам:

```
>>> for r in Rubric.objects.annotate(cnt=Count('bb',
                                     filter=Q(bb__price__gt=100000))):
...     print(r.name, ': ', r.cnt, sep='')
...
Бытовая техника: 0
Мебель: 0
Недвижимость: 2
Растения: 0
Сантехника: 0
Сельхозинвентарь: 0
Транспорт: 0
```

- `Sum` — вычисляет сумму значений, хранящихся в поле. Формат конструктора:

```
Sum(<имя поля или выражение>[, output_field=None][, filter=None])
```

Первым параметром указывается *имя поля*, сумма чьих значений будет вычисляться, или *выражение*, представленное экземпляром класса `F` (см. *разд. 7.3.8*). Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа. По умолчанию тип результата совпадает с типом поля. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*). По умолчанию фильтрация записей не выполняется.

- `Min` — вычисляет наименьшее значение из хранящихся в заданном поле. Формат конструктора:

```
Min(<имя поля или выражение>[, output_field=None][, filter=None])
```

Первым параметром указывается *имя поля* или *выражение*, представленное экземпляром класса `F`. Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа. По умолчанию тип результата совпадает с типом поля. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`, если он не указан, фильтрация записей не выполняется.

- `Max` — вычисляет наибольшее значение из хранящихся в заданном поле. Формат конструктора:

```
Max(<имя поля или выражение>[, output_field=None][, filter=None])
```

Первым параметром указывается *имя поля* или *выражение* в виде экземпляра класса `F`. Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа. По умолчанию тип результата совпадает с типом поля. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`. По умолчанию фильтрация не выполняется.

- `Avg` — вычисляет среднее арифметическое. Формат конструктора:

```
Avg(<имя поля или выражение>[, output_field=FloatField()][,
                              filter=None])
```

Первым параметром указывается *имя поля*, по содержимому которого будет вычисляться среднее арифметическое, или *выражение*, представленное экземпляром класса `F`. Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа. По умолчанию это величина вещественного типа. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`. По умолчанию фильтрация не выполняется.

- `StdDev` — вычисляет стандартное отклонение:

```
StdDev(<имя поля или выражение>[, sample=False][, filter=None])
```

Первым параметром указывается *имя поля*, по содержимому которого будет вычисляться стандартное отклонение, или *выражение*, представленное экземпляром класса `F`. Если значение параметра `sample` равно `True`, вычисляется стандартное отклонение выборки, если `False` — собственно стандартное отклонение (поведение по умолчанию). Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`. По умолчанию фильтрация записей не выполняется.

- `Variance` — вычисляет дисперсию:

```
Variance(<имя поля или выражение>[, sample=False][, filter=None])
```

Первым параметром указывается *имя поля*, по содержимому которого будет вычисляться дисперсия, или *выражение*, представленное экземпляром класса `F`. Если значение параметра `sample` равно `True`, вычисляется стандартное дисперсия образца, если `False` — собственно дисперсия (поведение по умолчанию). Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`. По умолчанию фильтрация записей не выполняется.

7.6. Вычисляемые поля

В *разд. 4.7* мы научились объявлять в моделях функциональные поля, значения которых не берутся из базы, а вычисляются на основе каких-либо других данных. В ряде случаев от них может быть польза, однако не помешало бы перенести вычисление значений таких полей на СУБД, что позволит несколько уменьшить нагрузку на компьютер, где развернут Django-сайт. То есть создать полноценные вычисляемые поля.

Значение *вычисляемого поля*, в отличие от поля функционального, вычисляется средствами самой СУБД. Обычно в виде вычисляемых выполняются поля, получение значений которых не требует слишком сложных вычислений, т. к. набор операторов и функций, поддерживаемых языком SQL, существенно меньше такового у Python.

7.6.1. Простейшие вычисляемые поля

В самом простом виде вычисляемое поле создается с помощью метода `annotate()`, который нам уже знаком. В нем, с применением именованного параметра, указывается выражение, вычисляющее значение поля.

- Для указания значения какого-либо поля применяется экземпляр класса `F` (см. *разд. 7.3.8*).
- Константы целочисленного, вещественного и логического типов мы можем указывать как есть.
- Для указания константы строкового типа следует использовать экземпляр класса `Value` из модуля `django.db.models`, чей конструктор вызывается в формате:

  `Value(<значение константы>[, output_field=None])`

  Само *значение константы* записывается в первом параметре конструктора этого класса.

  В необязательном параметре `output_field` можно задать тип константы в виде экземпляра класса, представляющего поле нужного типа. Если параметр отсутствует, тип значения будет определен автоматически.
- Для вычислений применяются операторы `+`, `-`, `*`, `/` и `//`. Они записываются как есть.

  Любой из этих операторов выполняет соответствующие действия над переданными ему экземплярами классов `F` и `Value` и возвращает результат этих действий также в виде экземпляра класса `F`.

  Оператор `-` может быть использован для изменения знака значения, представленного экземпляром класса `F` или `Value`.

В качестве примера вычислим для каждого объявления половину указанной в нем цены:

```
>>> from django.db.models import F
>>> for b in Bb.objects.annotate(half_price=F('price')/2):
        print(b.title, b.half_price)
...
Стиральная машина 1500.0
Пылесос 500.0
# Остальной вывод пропущен
```

Здесь мы указали константу 2 как есть, поскольку она принадлежит целочисленному типу, и Django благополучно обработает ее.

Теперь выведем записанные в объявлениях названия товаров и, в скобках, названия рубрик:

```
>>> from django.db.models.functions import Concat
>>> for b in Bb.objects.annotate(full_name=Concat(F('title'),
        Value(' ('), F('rubric__name'), Value(')'))): print(b.full_name)
...
Стиральная машина (Бытовая техника)
Пылесос (Бытовая техника)
Земельный участок (Недвижимость)
# Остальной вывод пропущен
```

В некоторых случаях может понадобиться указать для значения выражения, записанного в виде экземпляра класса `F`, тип возвращаемого им результата. Непосредственно в конструкторе класса `F` это сделать не получится. Положение спасет класс `ExpressionWrapper`, объявленный в модуле `django.db.models`. Вот формат его конструктора:

```
ExpressionWrapper(<выражение>, <тип результата>)
```

Выражение представляется экземпляром класса `F`, а тип данных — экземпляром класса поля нужного типа. Пример:

```
>>> from django.db.models import ExpressionWrapper, IntegerField
>>> for b in Bb.objects.annotate(
            half_price=ExpressionWrapper(F('price')/2,IntegerField())):
        print(b.title, b.half_price)
...
Стиральная машина 1500
Пылесос 500
# Остальной вывод пропущен
```

7.6.2. Функции СУБД

Функция СУБД реализуется и выполняется не Django и не исполняющей средой Python, а СУБД. Django просто предоставляет для этих функций удобный объектный интерфейс.

Функции СУБД представляются набором классов, объявленных в модуле `django.db.models.functions`. Вот все эти классы:

- `Coalesce` — возвращает первое переданное ему значение, отличное от `null` (даже если это пустая строка или 0). Конструктор класса вызывается в формате:

  ```
  Coalesce(<значение 1>, <значение 2> . . . <значение n>)
  ```

  Значения представляются строковыми именами полей, экземплярами классов `F` и `Value`. Все они должны иметь одинаковый тип (например, только строковый или только числовой), в противном случае мы получим ошибку.

  Пример:

  ```
  Coalesce('content', 'addendum', Value('--пусто--'))
  ```

Если значение поля `content` отлично от `null`, будет возвращено оно. В противном случае будет проверено значение поля `addendum` и, если оно не равно `null`, функция вернет его. Если же и значение поля `addendum` равно `null`, будет возвращена константа `'--пусто--'`;

- `Greatest` — возвращает наибольшее значение из переданных ему:

  `Greatest(<значение 1>, <значение 2> . . . <значение n>)`

  Значения представляются строковыми именами полей, экземплярами классов `F` и `Value`. Все они должны иметь одинаковый тип (например, только строковый или только числовой), в противном случае мы получим ошибку;

- `Least` — возвращает наименьшее значение из переданных ему:

  `Least(<значение 1>, <значение 2> . . . <значение n>)`

  Значения представляются строковыми именами полей, экземплярами классов `F` и `Value`. Все они должны иметь одинаковый тип (например, только строковый или только числовой), в противном случае мы получим ошибку;

- `Cast` — принудительно преобразует заданное *значение* к указанному *типу* и возвращает результат преобразования:

  `Cast(<значение>, <тип>)`

  Значение представляется строкой с именем поля или экземпляром класса `F`. *Тип* должен указываться в виде экземпляра класса, представляющего поле соответствующего типа;

- `Concat` — объединяет переданные ему *значения* в одну строку, которая и возвращается в качестве результата:

  `Concat(<значение 1>, <значение 2> . . . <значение n>)`

  Значения представляются строковыми именами полей, экземплярами классов `F` и `Value`. Все они должны иметь строковый или текстовый тип;

- `Lower` — преобразует символы строки к нижнему регистру и возвращает преобразованную строку в качестве результата:

  `Lower(<значение>)`

  Значение представляется строкой с именем поля или экземпляром класса `F`. Оно должно иметь строковый или текстовый тип;

- `Upper` — преобразует символы строки к верхнему регистру и возвращает преобразованную строку в качестве результата:

  `Upper(<значение>)`

  Значение представляется строкой с именем поля или экземпляром класса `F`. Оно должно иметь строковый или текстовый тип;

- `Length` — возвращает длину полученного *значения* в символах:

  `Length(<значение>)`

Значение представляется строкой с именем поля или экземпляром класса `F`. Оно должно иметь строковый или текстовый тип.

Если значение равно `null`, в качестве результата будет возвращено `None`;

- `StrIndex` — возвращает номер вхождения указанной *подстроки* в строковое *значение*. Нумерация символов в строке начинается с 1. Если *подстрока* отсутствует в *значении*, возвращается 0. Формат конструктора:

  `StrIndex(<значение>, <подстрока>)`

  Значение и *подстрока* представляются строками с именами поля или экземплярами классов `F` или `Value`. Они должны иметь строковый или текстовый тип;

- `Substr` — извлекает из *значения подстроку* с указанными *позицией* и *длиной* и возвращает в качестве результата:

  `Substr(<значение>, <позиция>[, <длина>])`

  Значение представляется строкой с именем поля или экземпляром класса `F`. Оно должно иметь строковый или текстовый тип.

  При указании *позиции* и *длины* извлекаемой подстроки нужно иметь в виду, что нумерация символов в строке начинается с 1. Если *длина* не задана, извлекается вся оставшаяся часть *значения*;

- `Left` — возвращает подстроку, начинающуюся с начала заданного *значения* и имеющую заданную *длину*:

  `Left(<значение>, <длина>)`

- `Right` — возвращает подстроку, заканчивающуюся в конце заданного *значения* и имеющую заданную *длину*:

  `Right(<значение>, <длина>)`

- `Replace` — возвращает значение, в котором все вхождения заменяемой подстроки заменены на заменяющую подстроку:

  `Replace(<значение>, <заменяемая подстрока>[, <заменяющая подстрока>])`

  Если *заменяющая подстрока* не указана, используется пустая строка (т. е. функция фактически будет удалять все вхождения *заменяемой подстроки*). Поиск *заменяемой подстроки* выполняется с учетом регистра символов;

- `Repeat` — возвращает заданное *значение*, повторенное заданное *количество* раз:

  `Repeat(<значение>, <количество повторов>)`

- `LPad` — возвращает заданное *значение*, дополненное слева *символами-заполнителями* таким образом, чтобы достичь указанной *длины*:

  `LPad(<значение>, <длина>[, <символ-заполнитель>])`

  Если *символ-заполнитель* не указан, используется пробел;

- `RPad` — возвращает заданное *значение*, дополненное справа *символами-заполнителями* таким образом, чтобы достичь указанной *длины*:

  `RPad(<значение>, <длина>[, <символ-заполнитель>])`

  Если *символ-заполнитель* не указан, используется пробел;

- `Trim` — возвращает указанное *значение* с удаленными начальными и конечными пробелами:

  ```
  Trim(<значение>)
  ```

- `LTrim` — возвращает указанное *значение* с удаленными начальными пробелами:

  ```
  LTrim(<значение>)
  ```

- `RTrim` — возвращает указанное *значение* с удаленными конечными пробелами:

  ```
  RTrim(<значение>)
  ```

- `Chr` — подобно функции `chr()` языка Python, возвращает символ с указанным в качестве параметра целочисленным *кодом*:

  ```
  Chr(<код символа>)
  ```

- `Ord` — подобно функции `ord()` языка Python, возвращает целочисленный код первого символа заданного *значения*:

  ```
  Ord(<значение>)
  ```

- `Now()` — возвращает текущие дату и время;

- `Extract` — извлекает часть значения даты и времени и возвращает его в виде числа:

  ```
  Extract(<значение даты и (или) времени>,
          <извлекаемая часть значения>[, tzinfo=None])
  ```

  Значение представляется экземпляром класса `F` и должно принадлежать типу даты, времени или даты и времени.

  Извлекаемая часть даты и времени представляется в виде строки `"year"` (год), `"quarter"` (номер квартала), `"month"` (номер месяца), `"day"` (число), `"week"` (порядковый номер недели), `"week_day"` (номер дня недели), `"hour"` (часы), `"minute"` (минуты) или `"second"` (секунды).

  Параметр `tzinfo` задает временную зону. На практике его приходится задавать крайне редко.

  Пример:

  ```
  Extract('published', 'year')
  ```

  Вместо класса `Extract` можно использовать более простые в применении классы: `ExtractYear` (извлекает год), `ExtractQuarter` (номер квартала), `ExtractMonth` (номер месяца), `ExtractDay` (число), `ExtractWeek` (порядковый номер недели), `ExtractWeekDay` (номер дня недели), `ExtractHour` (часы), `ExtractMinute` (минуты) и `ExtractSecond` (секунды). Конструкторы этих классов имеют такой формат вызова:

  ```
  <класс>(<значение даты и (или) времени>[, tzinfo=None])
  ```

  Пример:

  ```
  ExtractYear('published')
  ```

- `Trunc` — сбрасывает в ноль *значение даты и (или) времени* до указанной *конечной части*, если считать справа:

```
Trunc(<значение даты и (или) времени>, <конечная часть>[,
      output_field=None][, tzinfo=None])
```

Значение представляется экземпляром класса `F` и должно принадлежать типу даты, времени или даты и времени.

Конечная часть представляется в виде строки `"year"` (год), `"quarter"` (номер квартала), `"month"` (номер месяца), `"week"` (неделя), `"day"` (число), `"hour"` (часы), `"minute"` (минуты) или `"second"` (секунды).

Параметр `output_field` указывает тип возвращаемого значения даты и (или) времени. Его значение должно представлять собой экземпляр класса `DateField`, `TimeField` или `DateTimeField` (они описаны в *разд. 4.3.2*). Если параметр не отсутствует, возвращаемый результат будет иметь тот же тип, что и изначальное значение.

Параметр `tzinfo` задает временную зону. На практике его приходится задавать крайне редко.

Примеры:

```
# Предположим, в поле published хранится значение даты и времени
# 05.06.2018 08:02:28.366947
Trunc('published', 'year')      #  01.01.2018 00:00:00
Trunc('published', 'quarter')   #  01.04.2018 00:00:00
Trunc('published', 'month')     #  01.06.2018 00:00:00
Trunc('published', 'week')      #  04.06.2018 00:00:00
Trunc('published', 'day')       #  05.06.2018 00:00:00
Trunc('published', 'hour')      #  05.06.2018 08:00:00
Trunc('published', 'minute')    #  05.06.2018 08:02:00
Trunc('published', 'second')    #  05.06.2018 08:02:28
```

Вместо класса `Trunc` можно использовать более простые в применении классы: `TruncYear` (сбрасывает до года), `TruncQuarter` (до квартала), `TruncMonth` (до месяца), `TruncWeek` (до полуночи понедельника текущей недели), `TruncDay` (до числа), `TruncHour` (до часов), `TruncMinute` (до минут), `TruncSecond` (до секунд), `TruncDate` (извлекает значение даты) и `TruncTime` (извлекает значение времени). Конструкторы этих классов имеют такой формат вызова:

```
<класс>(<значение даты и (или) времени>[, output_field=None][,
                                        tzinfo=None])
```

Пример:

```
TruncYear('published')
```

7.6.3. Условные выражения СУБД

Возможности СУБД по написанию команд для работы с базами данных, безусловно, не сравнятся с аналогичными средствами Python. Однако условные выражения (или нечто похожее на них) мы в этих командах использовать сможем.

Такие выражения выполняют ряд проверок, сообразуясь с записанными в них условиями. Когда очередное условие выполняется, в качестве результата возвращается записанное в нем значение. Дальнейшие условия в таком случае не просматриваются.

Для записи условного выражения применяется класс `Case` из модуля `django.db.models`. Его конструктор вызывается в формате:

```
Cast(<условие 1>, <условие 2> . . . <условие n>[, default=None][,
                                                output_field=None])
```

Каждое *условие* записывается в виде экземпляра класса `When` из модуля `django.db.models`, чей конструктор имеет следующий формат:

```
When(<условие>, then=None)
```

Здесь *условие* можно записать в формате, рассмотренном в *разд. 7.3.4*, или же в виде экземпляра класса `Q` (см. *разд. 7.3.9*). Параметр `then` указывает значение, которое будет возвращено при выполнении *условия*.

Вернемся к классу `Case`. Указанные в нем *условия* проверяются в порядке, в котором они записаны. Если какое-либо из *условий* выполняется, возвращается результат, заданный в параметре `then` этого *условия*, при этом остальные *условия* из присутствующих в конструкторе класса `Case` не проверяются. Если ни одно из *условий* не выполнилось, возвращается значение, заданное параметром `default`, или `None`, если таковой отсутствует.

Параметр `output_field` задает тип возвращаемого результата в виде экземпляра класса поля подходящего типа. Хоть в документации по Django он и помечен как необязательный, по опыту автора, лучше его указывать.

Давайте выведем список рубрик и, против каждой из них, пометку, говорящую о том, сколько объявлений оставлено в этой рубрике:

```
>>> from django.db.models import Case, When, Value, Count, CharField
>>> for r in Rubric.objects.annotate(cnt=Count('bb'),
            cnt_s=Case(When(cnt__gte=5, then=Value('Много')),
                       When(cnt__gte=3, then=Value('Средне')),
                       When(cnt__gte=1, then=Value('Мало')),
                       default=Value('Вообще нет'),
                       output_field=CharField()),):
        print('%s: %s' % (r.name, r.cnt_s))
...
Бытовая техника: Мало
Мебель: Вообще нет
Недвижимость: Много
Растения: Вообще нет
Сантехника: Вообще нет
Сельхозинвентарь: Вообще нет
Транспорт: Много
```

7.6.4. Вложенные запросы

Вложенные запросы могут использоваться в условиях фильтрации или для расчета результатов у вычисляемых полей. Django позволяет создавать вложенные запросы двух видов.

Запросы первого вида — полнофункциональные — возвращают какой-либо результат. Они создаются с применением класса `Subquery` из модуля `django.db.models`. Формат конструктора этого класса таков:

```
Subquery(<вложенный набор записей>[, output_field=None])
```

Вложенный набор записей формируется с применением тех же инструментов, что мы использовали ранее. Необязательный параметр `output_field` задает тип возвращаемого вложенным запросом результата, если этим результатом является единичное значение (как извлечь из набора записей значение единственного поля, мы узнаем очень скоро).

Если во *вложенном наборе записей* необходимо ссылаться на поле «внешнего» набора записей, ссылка на это поле, записываемая в условиях фильтрации *вложенного набора записей*, оформляется как экземпляр класса `OuterRef` из того же модуля `django.db.models`:

```
OuterRef(<поле "внешнего" набора записей>)
```

Давайте в качестве примера извлечем список рубрик и для каждой выведем дату и время публикации самого «свежего» объявления:

```
>>> from django.db.models import Subquery, OuterRef, DateTimeField
>>> sq = Subquery(Bb.objects.filter(
    rubric=OuterRef('pk')).order_by('-published').values(
    'published')[:1])
>>> for r in Rubric.objects.annotate(last_bb_date=sq):
        print(r.name, r.last_bb_date)
...
Бытовая техника 2018-06-13 11:05:59.421877+00:00
Мебель None
Недвижимость 2018-06-12 07:41:14.050001+00:00
Растения None
Сантехника None
Сельхозинвентарь None
Транспорт 2018-06-13 10:43:47.354819+00:00
```

Во вложенном запросе нам необходимо сравнить значение поля `rubric` объявления со значением ключевого поля `pk` рубрики, которое присутствует во «внешнем» запросе. Ссылку на это поле мы оформили как экземпляр класса `OuterRef`.

Вложенные запросы второго вида лишь позволяют проверить, присутствуют ли в таком запросе записи. Они создаются с применением класса `Exists` из модуля `django.db.models`:

```
Exists(<вложенный набор записей>)
```

Для примера давайте выведем список только тех рубрик, в которых присутствуют объявления с заявленной ценой более 100 000 руб.:

```
>>> from django.db.models import Exists
>>> subquery = Exists(Bb.objects.filter(rubric=OuterRef('pk'),
                                        price__gt=100000))
>>> for r in Rubric.objects.annotate(is_expensive=subquery).filter(
    is_expensive=True): print(r.name)
...
Недвижимость
```

7.7. Объединение наборов записей

Если нам понадобится объединить два или более набора записей в один набор, мы воспользуемся методом `union()`:

```
union(<набор записей 1>, <набор записей 2> . . . <набор записей n>[,
                                                  all=False])
```

Все заданные в методе *наборы записей* будут объединены с текущим. Результатом, возвращаемым методом, станет набор, содержащий все записи из объединенных *наборов*.

Если значение необязательного параметра `all` равно `True`, в результирующем наборе будут присутствовать все записи, в том числе и одинаковые. Если же для параметра указать значение `False`, результирующий набор будет содержать только уникальные записи (поведение по умолчанию).

> ***Внимание!***
>
> У наборов записей, предназначенных к объединению, не следует задавать сортировку. Если же таковая все же была указана, нужно убрать ее, вызвав метод `order_by()` без параметров.

Для примера давайте сформируем набор из объявлений с заявленной ценой более 100 000 руб. и объявлений по продаже бытовой техники:

```
>>> bbs1 = Bb.objects.filter(price__gte=100000).order_by()
>>> bbs2 = Bb.objects.filter(rubric__name='Бытовая техника').order_by()
>>> for b in bbs1.union(bbs2): print(b.title, sep=' ')
...
Дача
Дом
Земельный участок
Пылесос
Стиральная машина
```

Django поддерживает два более специализированных метода для объединения наборов записей:

- `intersection(<набор записей 1>, <набор записей 2> . . . <набор записей n>)` — возвращает набор, содержащий только записи, которые имеются во всех объединяемых наборах;

- `difference(<набор записей 1>, <набор записей 2> . . . <набор записей n>)` — возвращает набор, содержащий только записи, которые имеются только в каком-либо одном из объединяемых наборов, но не в двух или более сразу.

7.8. Извлечение значений только из заданных полей

Каждый из ранее описанных методов возвращает в качестве результата набор записей — последовательность объектов, представляющих записи. Такие структуры данных удобны в работе над целыми записями, однако отнимают много системных ресурсов.

Если необходимо всего лишь извлечь из модели только значения определенного поля (полей) хранящихся там записей, удобнее применить следующие методы:

- `values([<поле 1>, <поле 2> . . . <поле n>])` — извлекает из модели значения только указанных *полей*. В качестве результата возвращает набор записей (экземпляр класса `QuerySet`), элементами которого являются словари. Ключи элементов таких словарей совпадают с именами заданных полей, а значения элементов — это и есть значения полей.

  Поле может быть задано:

  - позиционным параметром — в виде строки со своим именем;
  - именованным параметром — в виде экземпляра класса `F`. Имя параметра станет именем поля.

  Если в числе полей, указанных в вызове метода, присутствует поле внешнего ключа, элемент результирующего словаря, соответствующий этому полю, будет хранить значение ключа связанной записи, а не саму эту запись.

  Примеры:

```
>>> Bb.objects.values('title', 'price', 'rubric')
<QuerySet [
    {'title': 'Стиральная машина', 'price': 3000.0, 'rubric': 3},
    {'title': 'Пылесос', 'price': 1000.0, 'rubric': 3},
    # Часть вывода пропущена
]>

>>> Bb.objects.values('title', 'price', rubric_id=F('rubric'))
<QuerySet [
    {'title': 'Стиральная машина', 'price': 3000.0, 'rubric_id': 3},
    {'title': 'Пылесос', 'price': 1000.0, 'rubric_id': 3},
    # Часть вывода пропущена
]>
```

  Если метод `values()` вызван без параметров, он вернет набор словарей со всеми полями модели. При этом вместо полей модели он будет содержать поля таб-

лицы из базы данных, обрабатываемой этой моделью. Вот пример (обратим внимание на имена полей — это поля таблицы, а не модели):

```
>>> Bb.objects.values()
<QuerySet [
    {'id': 23, 'title': 'Стиральная машина',
    'content': 'Автоматическая', 'price': 3000.0,
    'published': datetime.datetime(2018, 6, 13, 11, 5, 59, 421877,
    tzinfo=<UTC>), 'rubric_id': 3},
    # Часть вывода пропущена
]>
```

- `values_list([`*<поле 1>*`, `*<поле 2>*` . . . `*<поле n>*`][,][flat=False][,][named=False])` — то же самое, что `values()`, но возвращенный им набор записей будет содержать кортежи:

```
>>> Bb.objects.values_list('title', 'price', 'rubric')
<QuerySet [
    ('Стиральная машина', 3000.0, 3),
    ('Пылесос', 1000.0, 3),
    # Часть вывода пропущена
]>
```

Параметр `flat` имеет смысл указывать только, если возвращается значение одного поля. Если его значение — `False`, значения этого поля будут оформлены как кортежи из одного элемента (поведение по умолчанию). Если же задать для него значение `True`, возвращенный набор записей будет содержать значения поля непосредственно. Пример:

```
>>> Bb.objects.values_list('id')
<QuerySet [(23,), (22,), (8,), (17,), (16,), (13,), (12,), (10,),
(11,), (4,), (3,), (1,)]>
>>> Bb.objects.values_list('id', flat=True)
<QuerySet [23, 22, 8, 17, 16, 13, 12, 10, 11, 4, 3, 1]>
```

Если параметру `named` дать значение `True`, набор записей будет содержать не обычные кортежи, а именованные;

- `dates(`*<имя поля>*`, `*<часть даты>*`[, order='ASC'])` — возвращает набор записей (экземпляр класса `QuerySet`) с уникальными значениями даты, которые присутствуют в поле с заданным *именем* и урезаны до заданной *части*. В качестве *части даты* можно указать `"year"` (год), `"month"` (месяц) или `"day"` (число, т. е. дата не будет урезаться). Если параметру `order` дать значение `"ASC"`, значения в наборе записей будут отсортированы по возрастанию (поведение по умолчанию), если `"DESC"` — по убыванию. Примеры:

```
>>> Bb.objects.dates('published', 'day')
<QuerySet [datetime.date(2018, 5, 30), datetime.date(2018, 6, 4),
datetime.date(2018, 6, 5), datetime.date(2018, 6, 12),
datetime.date(2018, 6, 13)]>
```

```
>>> Bb.objects.dates('published', 'month')
<QuerySet [datetime.date(2018, 5, 1), datetime.date(2018, 6, 1)]>
```

- datetimes(*<имя поля>*, *<часть даты и времени>*[, order='ASC'][, tzinfo=None]) — то же самое, что dates(), но манипулирует значениями даты и времени. В качестве *части даты и времени* можно указать "year" (год), "month" (месяц), "day" (число), "hour" (часы), "minute" (минуты) или "second" (секунды, т. е. значение не будет урезаться). Параметр tzinfo указывает временну́ю зону. Пример:

```
>>> Bb.objects.datetimes('published', 'day')
<QuerySet [datetime.datetime(2018, 5, 30, 0, 0, tzinfo=<UTC>),
datetime.datetime(2018, 6, 4, 0, 0, tzinfo=<UTC>),
# Часть вывода пропущена
]>
```

- in_bulk(*<последовательность значений>*[, field_name='pk']) — ищет в модели записи, у которых поле, чье имя задано параметром field_name, хранит значения из указанной *последовательности*. Возвращает словарь, ключами элементов которого станут значения из *последовательности*, а значениями элементов — объекты модели, представляющие найденные записи. Заданное поле должно хранить уникальные значения (параметру unique конструктора поля модели нужно дать значение True). Примеры:

```
>>> Rubric.objects.in_bulk([1, 2, 3])
{1: <Rubric: Недвижимость>, 2: <Rubric: Транспорт>,
3: <Rubric: Бытовая техника>}

>>> Rubric.objects.in_bulk(['Транспорт', 'Мебель'], field_name='name')
{'Мебель': <Rubric: Мебель>, 'Транспорт': <Rubric: Транспорт>}
```

7.9. Получение значения из полей со списком

Если просто обратиться к какому-либо полю со списком, мы получим значение, которое непосредственно хранится в поле, а не то, которое должно выводиться на экран:

```
>>> b = Bb.objects.get(pk=1)
>>> b.kind
's'
```

Чтобы получить «экранное» значение, следует вызвать у модели метод с именем вида get_*<имя поля>*_display(), который это значение и вернет:

```
>>> b.get_kind_display()
'Продам'
```

ГЛАВА 8

Маршрутизация

Маршрутизация в терминах Django — это процесс выяснения, какой контроллер следует выполнить при получении в составе клиентского запроса интернет-адреса определенного формата. Подсистема фреймворка, выполняющая маршрутизацию, носит название *маршрутизатора*.

8.1. Как работает маршрутизатор

Маршрутизатор Django работает, основываясь на написанном разработчиком списке маршрутов. Каждый элемент такого списка — *маршрут* — устанавливает связь между интернет-адресом определенного формата (шаблонным интернет-адресом или шаблонным путем) и контроллером. Вместо контроллера в маршруте может быть указан вложенный список маршрутов.

Алгоритм работы маршрутизатора следующий:

1. Из полученного запроса извлекается запрашиваемый клиентом интернет-адрес.
2. Из интернет-адреса удаляются обозначение протокола, доменное имя (или IP-адрес) хоста, номер TCP-порта, набор GET-параметров и имя якоря, если они там присутствуют. В результате остается один только путь.
3. Этот путь последовательно сравнивается с шаблонными интернет-адресами всех записанных в списке маршрутов.
4. Как только шаблонный адрес из очередного проверяемого маршрута совпадет с *началом* полученного пути:
   - если в маршруте указан контроллер, он выполняется;
   - если в маршруте указан вложенный список маршрутов, из полученного пути удаляется начальный фрагмент, совпадающий с шаблонным адресом, и начинается просмотр маршрутов из вложенного списка.
5. Если ни один из шаблонных адресов, записанных в маршрутах, не совпал с полученным путем, клиенту отправляется стандартная страница сообщения об ошибке 404 (запрошенная страница не найдена).

Обратим особое внимание на то, что маршрутизатор считает полученный путь совпавшим с шаблонным, если последний присутствует в начале первого. Это может привести к неприятной коллизии. Например, если мы имеем список маршрутов с шаблонными путями **create/** и **create/comment/** (расположенными именно в таком порядке), то при получении запроса с интернет-адресом **/create/comment/** маршрутизатор посчитает, что произошло совпадение с шаблонным адресом **create/**, т. к. он присутствует в начале запрашиваемого адреса. Поэтому в рассматриваемом случае нужно расположить маршруты в порядке **create/comment/** и **create/** — тогда описанная здесь коллизия не возникнет.

Через запрашиваемый интернет-адрес во входящем в его состав пути может быть передано какое-либо значение. Чтобы получить его, достаточно поместить в нужное место шаблонного пути в соответствующем маршруте обозначение URL-параметра. Маршрутизатор извлечет значение и передаст его либо в контроллер, либо вложенному списку маршрутов (и тогда полученное значение станет доступно всем объявленным во вложенном списке контроллерам).

По возможности, маршруты должны содержать уникальные шаблонные интернет-адреса. Создавать маршруты с одинаковыми шаблонными интернет-адресами допускается, но не имеет смысла, т. к. в любом случае будет срабатывать самый первый из группы, а последующие окажутся бесполезны.

8.2. Списки маршрутов уровня проекта и уровня приложения

Поскольку маршрутизатор Django поддерживает объявление вложенных списков маршрутов, все маршруты проекта описываются в виде своего рода иерархии:

- маршруты, ведущие на отдельные разделы сайта (фактически — на разные приложения проекта), объявляются в списке, входящем в состав модуля конфигурации, — в списке маршрутов уровня проекта.

  Маршруты уровня проекта в качестве шаблонных адресов должны содержать префиксы, к которых начинаются интернет-адреса, относящиеся к разным приложениям;

- маршруты, ведущие на отдельные контроллеры одного и того же приложения, объявляются в списке, входящем в состав этого приложения, — в списке маршрутов уровня приложения.

  Как правило, маршруты уровня приложения не содержат вложенных списков маршрутов. Они непосредственно указывают на контроллеры, связанные с этими маршрутами.

По умолчанию список маршрутов уровня приложения записывается в модуле urls.py пакета конфигурации. Впрочем, можно использовать любой другой модуль, указав его путь, отсчитанный от папки проекта, в параметре `ROOT_URLCONF` модуля settings.py, также находящегося в пакете конфигурации (см. *главу 3*).

Маршруты уровня приложения могут быть объявлены в модуле, который должен находиться в пакете приложения и которому можно дать любое имя. Единственное

условие — этот модуль нам придется создать самостоятельно, поскольку команда `startapp` утилиты manage.py это за нас не сделает.

8.3. Объявление маршрутов

Любые списки маршрутов — неважно, уровня проекта или приложения, — объявляются согласно схожим правилам.

Список маршрутов оформляется как обычный список Python. Он присваивается переменной с именем `urlpatterns` — именно там маршрутизатор Django будет искать его (примеры объявления списков маршрутов можно увидеть в листингах из *глав 1* и *2*).

Каждый элемент списка маршрутов должен представлять собой результат, возвращаемый функцией `path()` из модуля `django.urls`. Формат вызова этой функции таков:

```
path(<шаблонный путь>, <контроллер>|<вложенный список маршрутов>[,
    <дополнительные параметры>][, name=<имя маршрута>])
```

Шаблонный путь записывается в виде строки. Прямой слеш в его начале не ставится, но обязательно ставится в конце.

Для объявления в *шаблонном пути* URL-параметров (параметризованный маршрут) применяется следующий формат:

```
< [<обозначение формата>:]<имя URL-параметра> >
```

Поддерживаются следующие *обозначения форматов* для значений URL-параметров:

- `str` — любая непустая строка, не включающая слеши (формат по умолчанию);
- `int` — положительное целое число, включая 0;
- `slug` — строковый слаг, содержащий латинские буквы, цифры, знаки дефиса и подчеркивания;
- `uuid` — уникальный универсальный идентификатор. Он должен быть правильно отформатирован: должны присутствовать дефисы, а буквы следует указать в нижнем регистре;
- `path` — любая непустая строка, включающая слеши. Обычно применяется для установления совпадения с целым фрагментом полученного в запросе интернет-адреса.

Имя URL-параметра задает имя для параметра контроллера, через который последний сможет получить переданное в составе интернет-адреса значение. Это имя должно удовлетворять правилам именования переменных Python.

Вернемся к функции `path()`. *Контроллер* указывается в виде ссылки на функцию (если это контроллер-функция, речь о которых пойдет в *главе 9*), или результата, возвращенного методом `as_view()` контроллера-класса (контроллерам-классам посвящена *глава 10*).

Пример написания маршрутов, указывающих на контроллеры, можно увидеть в листинге 8.1. Первый маршрут указывает на контроллер-класс `BbCreateView`, а второй и третий — на контроллеры-функции `by_rubric()` и `index()`.

Листинг 8.1. Маршруты, указывающие на контроллеры

```
from django.urls import path
from .views import index, by_rubric, BbCreateView

urlpatterns = [
    path('add/', BbCreateView.as_view()),
    path('<int:rubric_id>/', by_rubric),
    path('', index),
]
```

Вложенный список маршрутов указывается в виде результата, возвращенного функцией `include()` из того же модуля `django.urls`. Вот ее формат вызова:

```
include(<путь к модулю>|<список маршрутов>[,
        namespace=<префикс интернет-адреса>])
```

В качестве первого параметра можно указать либо строку с *путем к модулю*, содержащему правильно оформленный список маршрутов, либо непосредственно *список маршрутов* (о *префиксе интернет-адреса* мы поговорим позже).

Пример указания маршрутов, ведущих на вложенные списки маршрутов, показан в листинге 8.2. В первом маршруте вложенный список задан в виде пути к модулю, а во втором — в виде непосредственно списка маршрутов (он хранится в свойстве `urls` экземпляра класса `AdminSite`, хранящегося в переменной `site` модуля `django.contrib.admin`).

Листинг 8.2. Маршруты, указывающие на вложенные списки маршрутов

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('bboard/', include('bboard.urls')),
    path('admin/', admin.site.urls),
]
```

8.4. Передача данных в контроллеры

Еще в *главе 2* мы узнали, как передавать данные от одного контроллера к другому через URL-параметры, являющиеся частью интернет-адреса. Эти данные контроллер-функция сможет получить через параметры, чьи имена совпадают с именами URL-параметров.

Так, в приведенном далее примере контроллер-функция `by_rubric()` получает значение URL-параметра `rubric_id` через параметр `rubric_id`:

```
urlpatterns = [
    path('<int:rubric_id>/', by_rubric),
]

def by_rubric(request, rubric_id):
    . . .
```

Есть еще один способ передать какие-либо данные в контроллер. Для этого нужно объявить словарь Python, создать в нем столько элементов, сколько нужно передать значений, присвоить передаваемые значения этим элементам и передать полученный словарь функции `path()` в третьем параметре. Эти значения контроллер сможет получить также через одноименные параметры.

Вот пример передачи контроллеру-функции значения `mode`:

```
vals = {'mode': 'index'}
urlpatterns = [
    path('<int:rubric_id>/', by_rubric, vals),
]

def by_rubric(request, rubric_id, mode):
    . . .
```

> ***Внимание!***
>
> Если в маршруте присутствует URL-параметр с тем же именем, что и у дополнительного значения, передаваемого через словарь, контроллеру будет передано значение из словаря. К сожалению, извлечь значение URL-параметра в таком случае не получится.

8.5. Именованные маршруты

Мы можем дать любому маршруту имя, указав его в необязательном параметре `name` функции `path()` (см. *разд. 8.2*), создав тем самым именованный маршрут. После чего сможем воспользоваться преимуществами обратного разрешения интернет-адресов, т. е. автоматического формирования готового интернет-адреса по имени маршрута и набору параметров.

Пример указания имени для маршрута:

```
urlpatterns = [
    path('<int:rubric_id>/', by_rubric, name='by_rubric'),
]
```

Теперь мы можем использовать обратное разрешение, просто указав имя маршрута и значения входящих в него URL-параметров (если это параметризованный маршрут), как в коде контроллера:

```
from django.urls import reverse
. . .
url = reverse('by_rubric', kwargs={'rubric_id': 2})
```

так и в коде шаблона:

```
<a href="{% url 'by_rubric' rubric_id=2 %}">. . .</a>
```

Более подробно использование обратного разрешения будет описано в *главах 9* и *11*.

8.6. Пространства имен. Корневое приложение

Сложные сайты могут состоять из нескольких приложений. В каждом из таких приложений могут быть маршруты с совпадающими шаблонными интернет-адресами (например, и в приложении `bboard`, и в приложении `testapp` могут быть объявлены маршруты с одинаковым шаблонным адресом **index/**).

Как в таком случае дать понять Django, какой интернет-адрес мы хотим сформировать с применением обратного разрешения? Задать для каждого приложения *пространство имен* — своего рода область, к которой относится тот или иной список маршрутов.

Пространство имен указывается в модуле, где объявляется список маршрутов уровня приложения. Строку с его названием нужно присвоить переменной `app_name`. Пример:

```
app_name = 'bboard'
urlpatterns = [
    . . .
]
```

Пространства имен необходимо задать для каждого приложения, что имеются в проекте.

Чтобы сослаться на маршрут, объявленный в нужном пространстве имен, достаточно предварить имя маршрута именем пространства имен, разделив их двоеточием:

```
url = reverse('bboard:index')
. . .
<a href="{% url 'bboard:index' %}">. . .</a>
<a href="{% url 'testapp:index' %}">. . .</a>
```

Пространство имен задает префикс для интернет-адреса, формируемого средствами обратного разрешения. Так, выражение:

```
url = reverse('bboard:by_rubric', kwargs={'rubric_id': 2})
```

сформирует интернет-адрес **/bboard/2/**.

Но что, если нам нужно, чтобы какое-либо приложение было доступно по интернет-адресу без префикса (не **/bboard/2/**, а **/2**)? Или, иными словами, было связано с «корнем» сайта (*корневое приложение*).

Во-первых, нам нужно указать пустую строку в качестве шаблонного интернет-адреса в соответствующем маршруте уровня проекта. Во-вторых, понадобится задать для того же маршрута новый префикс, записав его в параметре `namespace` функции `include()` (см. *разд. 8.3*). Вот так:

```
urlpatterns = [
    path('', include('bboard.urls', namespace='')),
    . . .
]
```

Здесь мы указали «пустой» префикс (в виде пустой строки). В результате все интернет-адреса, относящиеся к пространству имен приложения `bboard`, будут начинаться с «корня» сайта. И мы сможем попасть на главную страницу этого приложения, набрав интернет-адрес вида **http://localhost:8000/**.

8.7. Указание шаблонных путей в виде регулярных выражений

Наконец, поддерживается указание шаблонных путей в виде регулярных выражений. Это может пригодиться, если шаблонный путь очень сложен, или при переносе кода, написанного под Django 1.11.

Для записи шаблонного пути в виде регулярного выражения применяется функция `re_path()` из модуля `django.urls`:

```
re_path(<регулярное выражение>,
        <контроллер>|<вложенный список маршрутов>[,
        <дополнительные параметры>][, name=<имя маршрута>])
```

Регулярное выражение должно быть представлено в виде строки и записано в формате, «понимаемом» модулем `re` языка Python. Остальные параметры точно такие же, как и у функции `path()`.

Пример указания шаблонных интернет-адресов в виде регулярных выражений можно увидеть в листинге 8.3.

Листинг 8.3. Указание шаблонных путей в виде регулярных выражений

```
from django.urls import re_path
from .views import index, by_rubric, BbCreateView

urlpatterns = [
    re_path(r'^add/$', BbCreateView.as_view(), name='add'),
    re_path(r'^(?P<rubric_id>[0-9]*)/$', by_rubric, name='by_rubric'),
    re_path(r'^$', index, name='index'),
]
```

Запись шаблонных интернет-адресов в виде регулярных выражений имеет одно преимущество: мы сами указываем, как будет выполняться сравнение. Записав в начале регулярного выражения обозначение начала строки, мы дадим понять Django, что шаблонный путь должен присутствовать в начале полученного интернет-адреса (обычный режим сравнения). А дополнительно записав в конец регулярного выражения обозначение конца строки, мы укажем, что полученный путь должен полностью совпадать с шаблонным. Иногда это может пригодиться.

ГЛАВА 9

Контроллеры-функции

Контроллеры — это программные модули, выполняющие всю работу по выборке данных из базы, подготовке их для вывода, обработке данных, полученных от посетителя, сохранению их в базе и, наконец, по формированию веб-страниц. Контроллеры реализуют основную часть логики сайта.

9.1. Введение в контроллеры-функции

Контроллеры-функции реализуются в виде обычных функций Python. Такая функция обязана принимать следующие параметры (указаны в порядке очередности их объявления):

- экземпляр класса `HttpRequest` (объявлен в модуле `django.http`), хранящий сведения о полученном клиентском запросе. Традиционно этот параметр носит имя `request`;
- набор именованных параметров, чьи имена совпадают с именами URL-параметров, объявленных в связанном с контроллером маршруте.

Контроллер-функция должен возвращать в качестве результата экземпляр класса `HttpResponse`, также объявленного в модуле `django.http`, или какого-либо из его подклассов. Этот экземпляр класса представляет ответ, отсылаемый клиенту (веб-страница, обычный текстовый документ, файл, данные в формате JSON, перенаправление или сообщение об ошибке).

При создании нового приложения утилита manage.py записывает в пакет приложения модуль views.py, в котором, как предполагают разработчики фреймворка, и будет находиться код контроллеров. Однако ничто не мешает нам создать под написание контроллеров другие модули, назвав их, как нам заблагорассудится.

9.2. Как пишутся контроллеры-функции

Принципы написания контроллеров-функций достаточно просты. Однако здесь имеется один подводный камень, обусловленный самой природой Django.

9.2.1. Контроллеры, выполняющие одну задачу

Если контроллер-функция должен выполнять всего одну задачу — например, вывод веб-страницы, все очень просто. Для примера давайте рассмотрим код из листинга 2.2 — он объявляет контроллер-функцию `by_rubric()`, которая выводит страницу с объявлениями, относящимися к выбранной посетителем рубрике.

Если нам нужно выводить на экран страницу для занесения нового объявления и потом сохранять это объявление в базе данных, нам понадобятся два контроллера такого рода.

Давайте напишем контроллер-функцию, который создаст форму и выведет на экран страницу добавления объявления (листинг 9.1).

Листинг 9.1. Контроллер-функция add()

```
def add(request):
    bbf = BbForm()
    context = {'form': bbf}
    return render(request, 'bboard/create.html', context)
```

Далее напишем второй контроллер-функцию `add_save()`, который сохранит введенное объявление (листинг 9.2).

Листинг 9.2. Контроллер-функция add_save()

```
def add_save(request):
    bbf = BbForm(request.POST)
    if bbf.is_valid():
        bbf.save()
        return HttpResponseRedirect(reverse('by_rubric',
               kwargs={'rubric_id': bbf.cleaned_data['rubric'].pk}))
    else:
        context = {'form': bbf}
        return render(request, 'bboard/create.html', context)
```

Объявим маршруты, ведущие на эти контроллеры:

```
from .views import add, add_save

urlpatterns = [
    path('add/save/', add_save, name='add'),
    path('add/', add, name='add'),
    . . .
]
```

После чего нам останется открыть шаблон bboard\create.html, написанный в *главе 2*, и исправить имеющийся в нем тег `<form>`, создающий форму, добавив в него целевой интернет-адрес:

```
<form action="{% url 'add_save' %}" method="post">
```

9.2.2. Контроллеры, выполняющие несколько задач

Но на практике для обработки данных, вводимых посетителями в формы, обычно применяют не два контроллера, а один. Он выполнит сразу две задачи: выведет страницу с формой и выполнит сохранение занесенных в нее данных.

Код контроллера-функции `add_and_save()`, который выполнит и вывод страницы с формой, и сохранение введенных данных, показан в листинге 9.3.

Листинг 9.3. Контроллер-функция add_and_save ()

```
def add_and_save (request):
    if request.method == 'POST':
        bbf = BbForm(request.POST)
        if bbf.is_valid():
            bbf.save()
            return HttpResponseRedirect(reverse('bboard:by_rubric',
                    kwargs={'rubric_id': bbf.cleaned_data['rubric'].pk}))
        else:
            context = {'form': bbf}
            return render(request, 'bboard/create.html', context)
    else:
        bbf = BbForm()
        context = {'form': bbf}
        return render(request, 'bboard/create.html', context)
```

Здесь мы сначала проверяем, с применением какого метода был отправлен запрос. Если применялся метод POST, значит, осуществляется отсылка введенных в форму данных, и их надо сохранить в базе. Предварительно выполняется проверка занесенных в форму данных на корректность и, если она завершилась неудачей, повторный вывод страницы с формой на экран. После успешного сохранения производится перенаправление на страницу со списком объявлений, относящихся к категории, заданной при вводе текущего объявления. Если же запрос был отправлен методом GET, на экран выводится страница с пустой формой.

Поскольку мы обходимся только одним контроллером, нам понадобится лишь один маршрут:

```
from .views import add_and_save

urlpatterns = [
    path('add/', add_and_save, name='add'),
    . . .
]
```

В коде шаблона bboard\create.html — в теге `<form>`, создающем форму, — уже не нужно указывать интернет-адрес, по которому будут отправлены занесенные в форму данные:

```
<form method="post">
```

В этом случае данные будут отправлены по тому же интернет-адресу, с которого была загружена текущая страница.

9.3. Формирование ответа

Основная задача контроллера — сформировать ответ, который будет отправлен посетителю. Обычно такой ответ содержит веб-страницу.

9.3.1. Низкоуровневые средства для формирования ответа

Формированием ответа на самом низком уровне занимается класс `HttpResponse` из модуля `django.http`. Его конструктор вызывается в следующем формате:

```
HttpResponse([<содержимое>][,][content_type=None][,][status=200][,]
                                                    [reason=None])
```

Содержимое должно быть указано в виде строки.

Необязательный параметр `content_type` задает MIME-тип ответа и его кодировку. Если он отсутствует, ответ получит MIME-тип `text/html` и кодировку из параметра `DEFAULT_CHARSET`, указанного в настройках проекта (см. *разд. 3.3.1*).

Параметр `status` указывает целочисленный код статуса ответа (по умолчанию — `200`, то есть файл успешно отправлен), а параметр `reason` — строковый статус (по умолчанию — `"OK"`).

Класс ответа поддерживает атрибуты:

- `content` — содержимое ответа в виде объекта типа `bytes`;
- `charset` — обозначение кодировки;
- `status_code` — целочисленный код статуса;
- `reason_phrase` — строковое обозначение статуса;
- `streaming` — если `True`, это потоковый ответ, если `False` — обычный. У класса `HttpResponse` всегда возвращает `False`;
- `closed` — если `True`, ответ закрыт, и `False`, если еще нет.

Также поддерживаются следующие методы:

- `write(<строка>)` — добавляет *строку* в ответ;
- `writelines(<последовательность строк>)` — добавляет к ответу строки из указанной *последовательности*. Разделители строк при этом не вставляются;
- `flush()` — принудительно переносит содержимое буфера записи в ответ;
- `has_header(<заголовок>)` — возвращает `True`, если указанный заголовок существует в ответе, и `False` — в противном случае;
- `setdefault(<заголовок>, <значение>)` — создает в ответе указанный *заголовок* с указанным *значением*, если таковой там отсутствует.

Класс `HttpResponse` поддерживает функциональность словарей, которой мы можем пользоваться, чтобы указывать и получать значения заголовков:

```
response['pragma'] = 'no-cache'
age = response['Age']
del response['Age']
```

В листинге 9.4 приведен код простого контроллера, использующего низкоуровневые средства для создания ответа и выводящего строку: **Здесь будет главная страница сайта**. Кроме того, мы задаем заголовок `keywords` со значением `"Python, Django"`, пользуясь поддерживаемой классом ответа функциональностью словаря.

Листинг 9.4. Использование низкоуровневых средств формирования ответа

```
from django.http import HttpResponse

def index(request):
    resp = HttpResponse("Здесь будет",
                        content_type='text/plain; charset=utf-8')
    resp.write(' главная')
    resp.writelines((' страница', ' сайта'))
    resp['keywords'] = 'Python, Django'
    return resp
```

9.3.2. Формирование ответа на основе шаблона

Применять низкоуровневые средства для создания ответа на практике приходится крайне редко. Гораздо чаще мы имеем дело с высокоуровневыми средствами, а именно с шаблонизатором.

Для загрузки нужного шаблона Django предоставляет две функции, объявленные в модуле `django.template.loader`:

- `get_template(<путь к шаблону>)` — загружает шаблон с указанным *путем* и возвращает представляющий его экземпляр класса `Template` из модуля `django.template`.
- `select_template(<последовательность путей шаблонов>)` — перебирает указанную *последовательность путей шаблонов*, пытается загрузить шаблон, расположенный по очередному пути, и возвращает первый шаблон, который удалось загрузить. Возвращаемый результат также представляет собой экземпляр класса `Template`.

Если шаблон с указанным путем загрузить не получилось, будет возбуждено исключение `TemplateDoesNotExist`. Если в коде шаблона встретилась ошибка, возбуждается исключение `TemplateSyntaxError`. Оба класса исключений объявлены в модуле `django.template`.

Пути шаблонов указываются относительно папки, в которой хранятся шаблоны (как указать эту папку, будет рассказано в *главе 11*).

Для получения обычной веб-страницы нужно объединить шаблон с данными — выполнить *рендеринг* шаблона. Для этого мы можем использовать следующие методы:

- `render([context=<контекст данных>][,][request=<запрос>])` — метод класса `Template`. Выполняет рендеринг, объединяя текущий шаблон с *контекстом данных* (словарем со значениями, которые должны быть доступны в шаблоне). Если указан *запрос* (экземпляр класса `Request`), он также будет добавлен к контекст данных. В качестве результата возвращает строку с HTML-кодом сформированной страницы. Пример использования этого метода можно увидеть в листинге 9.5.

Листинг 9.5. Применение метода `render()` для рендеринга шаблона

```
from django.http import HttpResponse
from django.template.loader import get_template

def index(request):
    bbs = Bb.objects.all()
    rubrics = Rubric.objects.all()
    context = {'bbs': bbs, 'rubrics': rubrics}
    template = get_template('bboard/index.html')
    return HttpResponse(template.render(context=context,
                                        request=request))
```

Параметры у метода `render()` можно указать не только как именованные, но и как позиционные:

```
return HttpResponse(template.render(context, request))
```

- `render_to_string(<путь к шаблону>[, context=<контекст данных>][, request=<запрос>])` — функция из модуля `django.template.loader`. Загружает шаблон с указанным *путем* и выполняет его рендеринг с применением заданных *контекста данных* и *запроса*. В качестве результата возвращает строку с HTML-кодом сформированной страницы. Пример использования функции `render_to_string()` можно увидеть в листинге 9.6.

Листинг 9.6. Применение функции `render_to_string()` для рендеринга шаблона

```
from django.http import HttpResponse
from django.template.loader import render_to_string

def index(request):
    bbs = Bb.objects.all()
    rubrics = Rubric.objects.all()
    context = {'bbs': bbs, 'rubrics': rubrics}
    return HttpResponse(render_to_string('bboard/index.html',
                                         context, request))
```

9.3.3. Класс *TemplateResponse*: отложенный рендеринг шаблона

Помимо класса `HttpResponse`, для формирования ответа клиенту мы можем использовать класс `TemplateResponse`, объявленный в модуле `django.template.response`. По функциональным возможностям эти классы примерно схожи.

Основное преимущество класса `TemplateResponse` проявляется при использовании посредников, добавляющих в контекст шаблона дополнительные данные (о посредниках разговор пойдет в *главе 21*). Этот класс поддерживает отложенный рендеринг шаблона, выполняющийся только после «прохождения» всей цепочки зарегистрированных в проекте посредников, непосредственно перед отправкой ответа клиенту. Благодаря этому посредники, собственно, и могут добавить в контекст шаблона дополнительные данные.

Кроме того, в виде экземпляра класса `TemplateResponse` генерируют ответы все высокоуровневые контроллеры-классы, о которых будет рассказано в *главе 10*.

Конструктор класса `TemplateResponse` вызывается в следующем формате:

```
TemplateResponse(<запрос>, <путь к шаблону>[, context=None][,
                                content_type=None][, status=200])
```

Запрос должен быть представлен в виде экземпляра класса `HttpRequest`. Он, как мы уже знаем, доступен внутри контроллера-функции через его первый параметр. Параметр `context` указывает контекст шаблона.

Необязательный параметр `content_type` задает MIME-тип ответа и его кодировку. Если он отсутствует, ответ получит MIME-тип `text/html` и кодировку из параметра `DEFAULT_CHARSET`, указанного в настройках проекта (см. *разд. 3.3.1*). Параметр `status` указывает целочисленный код статуса ответа (по умолчанию — `200`, то есть файл успешно отправлен).

Из всех атрибутов, поддерживаемых описываемым классом, для нас интересны только `template_name` и `context_data`. Первый хранит путь к шаблону, а второй — контекст шаблона.

Листинг 9.7. Использование класса `TemplateResponse`

```
from django.template.response import TemplateResponse

def index(request):
    bbs = Bb.objects.all()
    rubrics = Rubric.objects.all()
    context = {'bbs': bbs, 'rubrics': rubrics}
    return TemplateResponse(request, 'bboard/index.html',
                                     context=context)
```

9.4. Получение сведений о запросе

Полученный от посетителя запрос представляется экземпляром класса `HttpRequest`. Он хранит разнообразные сведения о запросе, которые могут оказаться очень полезными.

Прежде всего, это ряд атрибутов, хранящих различные величины:

- `GET` — словарь, хранящий все GET-параметры, полученные в составе запроса. Ключи элементов этого словаря совпадают с именами GET-параметров, а значения элементов — суть значения этих параметров;
- `POST` — словарь, хранящий все POST-параметры, полученные в составе запроса. Ключи элементов этого словаря совпадают с именами POST-параметров, а значения элементов — суть значения этих параметров;
- `FILES` — словарь, хранящий все выгруженные файлы. Ключи элементов этого словаря совпадают с именами POST-параметров, посредством которых передается содержимое файлов, а значения элементов — сами файлы, представленные экземплярами класса `UploadedFile`;
- `method` — обозначение метода отправки данных в виде строки, набранной прописными буквами (`"GET"`, `"POST"` и т. д.);
- `scheme` — обозначение протокола (`"http"` или `"https"`);
- `path` и `path_info` — путь;
- `encoding` — обозначение кодировки, в которой был получен запрос. Если `None`, запрос закодирован в кодировке по умолчанию (она задается в параметре `DEFAULT_CHARSET` настроек проекта);
- `content_type` — обозначение MIME-типа полученного запроса, извлеченное из HTTP-заголовка `Content-Type`;
- `content_params` — словарь, содержащий дополнительные параметры MIME-типа полученного запроса, которые извлекаются из HTTP-заголовка `Content-Type`. Ключи элементов соответствуют самим параметрам, а значения элементов — значениям параметров;
- `META` — словарь, содержащий дополнительные параметры в виде следующих элементов:
  - `CONTENT_LENGTH` — длина тела запроса в символах, заданная в виде строки;
  - `CONTENT_TYPE` — MIME-тип тела запроса (может быть `"application/x-www-form-urlencoded"`, `"multipart/form-data"` или `"text/plain"`);
  - `HTTP_ACCEPT` — строка с перечнем поддерживаемых веб-обозревателем MIME-типов данных, разделенных запятыми;
  - `HTTP_ACCEPT_ENCODINGS` — строка с перечнем поддерживаемых веб-обозревателей кодировок, разделенных запятыми;
  - `HTTP_ACCEPT_LANGUAGES` — строка с перечнем поддерживаемых веб-обозревателем языков, разделенных запятыми;

- HTTP_HOST — доменное имя (или IP-адрес) и номер порта, если он отличается от используемого по умолчанию, веб-сервера, с которого была загружена страница;
- HTTP_REFERER — интернет-адрес страницы, с которой был выполнен переход на текущую страницу (может отсутствовать, если это первая страница, открытая в веб-обозревателе);
- HTTP_USER_AGENT — строка с обозначением веб-обозревателя, запрашивающего страницу;
- QUERY_STRING — строка с необработанными GET-параметрами;
- REMOTE_ADDR — IP-адрес клиентского компьютера, запрашивающего страницу;
- REMOTE_HOST — доменное имя клиентского компьютера, запрашивающего страницу. Если доменное имя не удается определить, элемент хранит пустую строку;
- REMOTE_USER — имя пользователя, выполнившего вход на веб-сервер. Если вход на веб-сервер не был выполнен, или если используется другой способ аутентификации, этот элемент будет отсутствовать;
- REQUEST_METHOD — обозначение метода отправки данных ("GET", "POST" и т. д.);
- SERVER_NAME — доменное имя сервера;
- SERVER_PORT — номер TCP-порта, через который работает веб-сервер, в виде строки.

В словаре, хранящемся в атрибуте META, могут присутствовать и другие элементы. Они формируются на основе любых нестандартных заголовков, что имеются в клиентском запросе. Имя такого элемента создается на основе имени соответствующего заголовка, преобразованного к верхнему регистру, с подчеркиваниями вместо дефисов и с префиксом HTTP_, добавленным в начало. Так, на основе заголовка UPGRADE_INSECURE_REQUESTS в словаре META будет создан элемент с ключом HTTP_UPGRADE_INSECURE_REQUESTS;

- body — «сырое» содержимое запроса в виде объекта типа bytes.

Теперь рассмотрим методы, поддерживаемые классом HttpRequest:

- get_host() — возвращает строку с комбинацией IP-адреса (или доменного имени, если его удастся определить) и номера TCP-порта, через который работает веб-сервер;
- get_port() — возвращает строку с номером TCP-порта, через который работает веб-сервер;
- get_full_path() — возвращает полный путь к текущей странице;
- build_absolute_uri(*<путь>*) — строит полный интернет-адрес на основе доменного имени (IP-адреса) сервера и указанного *пути*:

```
print(request.build_absolute_uri('/test/url/'))
# Будет выведено: http://localhost:8000/test/uri/
```

- `is_secure()` — возвращает `True`, если обращение выполнялось по протоколу HTTPS, и `False` — если по протоколу HTTP;
- `is_ajax()` — возвращает `True`, если это AJAX-запрос, и `False` — если обычный.

  AJAX-запросы выявляются фреймворком по наличию в запросе заголовка `HTTP_X_REQUESTED_WITH` со значением `"XMLHttpRequest"`.

9.5. Перенаправление

Очень часто приходится выполнять *перенаправление* — отсылку веб-обозревателю своего рода предписания перейти на страницу с другим интернет-адресом, указанным в этом предписании. Например, после добавления объявления мы выполняем перенаправление на страницу списка объявлений, относящихся к рубрике, в которую было помещено добавленное объявление.

Перенаправление такого рода называется *временным*. Получив уведомление о нем, веб-обозреватель просто переходит на страницу с указанным интернет-адресом.

Для выполнения временного перенаправления нужно создать экземпляр класса `HttpResponseRedirect`, являющегося подклассом класса `HttpResponse` и также объявленного в модуле `django.http`. Вот формат конструктора этого класса:

```
HttpResponseRedirect(<целевой интернет-адрес>[, status=302][,
                                                reason=None])
```

Целевой интернет-адрес указывается в виде строки. Он может быть как полным, так и сокращенным.

Созданный таким образом экземпляр класса `HttpResponseRedirect` следует вернуть из контроллера-функции в качестве результата.

Пример выполнения временного перенаправления:

```
return HttpResponseRedirect(reverse('bboard:index'))
```

Постоянное перенаправление применяется в тех случаях, когда сайт «переезжает» на новый интернет-адрес, и при заходе по старому адресу посетителя нужно перенаправить на новое местонахождение. Получив уведомление о постоянном перенаправлении, веб-обозреватель заменяет новым старый интернет-адрес везде, где он присутствует: в списке истории, в избранном и др.

Постоянное перенаправление реализуется другим классом из модуля `django.http` — `HttpResponsePermanentRedirect`, также производным от `HttpResponse`:

```
HttpResponsePermanentRedirect(<целевой интернет-адрес>[, status=301][,
                                                        reason=None])
```

Пример:

```
return HttpResponsePermanentRedirect('http://www.new_address.ru/')
```

9.6. Формирование интернет-адресов путем обратного разрешения

Как мы уже знаем, Django поддерживает механизм обратного разрешения интернет-адресов, который формирует их на основе объявленных в списках маршрутов (см. *главу 8*). Единственное, что требуется для этого — сделать все маршруты, на основе которых будут формироваться интернет-адреса, именованными.

Для формирования интернет-адресов в коде контроллеров применяется функция `reverse()` из модуля `django.urls`:

```
reverse(<имя маршрута>[, args=None][, kwargs=None][, urlconf=None])
```

Имя маршрута указывается в виде строки. Если функциональность сайта реализована несколькими приложениями, для которых указаны пространства имен, первым параметром функции указывается строка вида *<пространство имен>:<имя маршрута>*.

Если маршрут, на основе которого нужно сформировать интернет-адрес, является параметризованным, следует указать значения URL-параметров для вставки в этот интернет-адрес. Их можно задать:

- в параметре `args` — в виде последовательности. Первый элемент такой последовательности задаст значение первого по счету URL-параметра в маршруте, второй элемент задаст значение второго URL-параметра и т. д.;
- в параметре `kwargs` — в виде словаря. Ключи его элементов соответствуют именам URL-параметров, а значения элементов зададут значения параметров.

Допускается указывать только один из этих параметров: `args` или `kwargs`. Указание обоих параметров вызовет ошибку.

Параметр `urlconf` задает путь к модулю со списком маршрутов, который будет использоваться для обратного разрешения. Если он не указан, задействуется модуль со списком маршрутов уровня проекта, заданный в его настройках (параметр `ROOT_URLCONF`. Более подробно о нем см. в *разд. 3.3.1*).

Если в параметре `urlconf` указан модуль с маршрутами уровня приложения, записывать пространство имен в первом параметре функции `reverse()` не нужно.

Примеры:

```
url1 = reverse('index', urlconf='bboard.urls')
url2 = reverse('bboard:by_rubric', args=(current_rubric.pk,))
url3 = reverse('bboard:by_rubric',
               kwargs={'rubric_id': current_rubric.pk})
```

Функция `reverse()` имеет большой недостаток — она работает только после того, как список маршрутов был загружен и обработан. Из-за этого ее можно использовать только непосредственно в коде контроллеров.

Если же нужно записать интернет-адрес где-либо еще — например, в атрибуте контроллера-класса (о них речь пойдет в *главе 10*), следует применить функцию

`reverse_lazy()` из того же модуля `django.urls`. Ее формат вызова точно такой же, как и у функции `reverse()`. Пример:

```
class BbCreateView(CreateView):
    . . .
    success_url = reverse_lazy('index')
```

Для формирования интернет-адресов в шаблонах применяется тег шаблонизатора `url`. Мы рассмотрим его в *главе 11*.

9.7. Выдача сообщений об ошибках и обработка особых ситуаций

Временами в работе сайта случаются нештатные ситуации, которые нужно как-то обрабатывать. Наиболее часто при возникновении такой ситуации посетителю отправляется веб-страница с сообщением об ошибке. Помимо этого, приходится уведомлять веб-обозреватель о возникновении особых ситуаций — например, если запрашиваемая страница не изменилась с момента предыдущего запроса и может быть извлечена из локального кэша.

Для выдачи сообщений об ошибке Django предоставляет ряд классов. Все они являются производными от класса `HttpResponse` и объявлены в модуле `django.http`. Далее приведены форматы вызова конструкторов этих классов и их описания:

- `HttpResponseNotFound([<содержимое>][,][content_type=None][,][status=404][,][reason=None])` — запрашиваемая страница не найдена:

```
def detail(request, bb_id):
    try:
        bb = Bb.objects.get(pk=bb_id)
    except Bb.DoesNotExist:
        return HttpResponseNotFound('Такое объявление не существует')
    return HttpResponse( . . . )
```

  Вместо создания экземпляра класса `HttpResponseNotFound` можно возбудить исключение `Http404` из модуля `django.http`:

```
def detail(request, bb_id):
    try:
        bb = Bb.objects.get(pk=bb_id)
    except Bb.DoesNotExist:
        raise Http404('Такое объявление не существует')
    return HttpResponse( . . . )
```

- `HttpResponseBadRequest([<содержимое>][,][content_type=None][,][status=400][,][reason=None])` — клиентский запрос некорректно сформирован;
- `HttpResponseForbidden([<содержимое>][,][content_type=None][,][status=403][,][reason=None])` — доступ к запрошенной странице запрещен (скорее всего, вследствие того, что посетитель не выполнил процедуру входа).

Вместо создания экземпляра класса `HttpResponseForbidden` мы можем возбудить исключение `PermissionDenied` из модуля `django.core.exceptions`;

- `HttpResponseNotAllowed(<последовательность обозначений разрешенных методов>[,][content_type=None][,][status=405][,][reason=None])` — клиентский запрос был выполнен с применением недопустимого метода. Первым параметром конструктора указывается последовательность методов, которые допустимы для запрошенной страницы. Пример:

```
return HttpResponseNotAllowed(['GET'])
```

- `HttpResponseGone([<содержимое>][,][content_type=None][,][status=410][,][reason=None])` — запрошенная страница более не существует;
- `HttpResponseServerError([<содержимое>][,][content_type=None][,][status=500][,][reason=None])` — ошибка в программном коде сайта;
- `HttpResponseNotModified([<содержимое>][,][content_type=None][,][status=304][,][reason=None])` — запрашиваемая страница не изменилась с момента последнего запроса и может быть извлечена веб-обозревателем из локального кэша.

Эти классы помогут нам, если мы собираемся самостоятельно обработать какую-либо нештатную или специфическую ситуацию и отправить клиенту соответствующее уведомление. Но Django в большинстве случаев справляется с обработкой таких ситуаций сам. По умолчанию он отправляет клиенту краткое текстовое сообщение о возникшей ошибке (мы можем настроить его под свои нужды, воспользовавшись инструкциями из *главы 29*).

9.8. Специальные ответы

Класс `HttpResponse` применяется для формирования ответа в большинстве случаев (если не рассматривать перенаправления и выдачу сообщений об ошибках). Однако иногда нужно отправить посетителю данные формата, отличного от веб-страницы или обычного текста. Поэтому Django предлагает нам три класса специальных ответов, объявленные в модуле `django.http`.

9.8.1. Потоковый ответ

Обычный ответ `HttpResponse` сначала полностью формируется в оперативной памяти компьютера, а потом отправляется клиенту. Если объем ответа невелик, это вполне допустимо.

Но для отправки таким образом веб-страниц большого объема класс `HttpResponse` не годится — ответ при формировании займет очень много памяти. Здесь нужен *потоковый ответ*, который формируется и отсылается по частям, имеющим небольшие размеры.

Потоковый ответ представляется классом `StreamingHttpResponse`. Формат его конструктора:

```
StreamingHttpResponse(<содержимое>[, content_type=None][, status=200][,
                                                          reason=None])
```

Здесь *содержимое* задается в виде последовательности строк или итератора, на каждом проходе возвращающего строку. Остальные параметры такие же, как и у класса `HttpResponse`.

Класс поддерживает следующие атрибуты:

- `streaming_content` — итератор, на каждом проходе возвращающий фрагмент содержимого ответа в виде объекта типа `bytes`;
- `status_code` — целочисленный код статуса;
- `reason_phrase` — строковый статус;
- `streaming` — если `True`, это потоковый ответ, если `False` — обычный. У класса `StreamingHttpResponse` всегда возвращает `True`.

Пример кода, выполняющего отправку потокового ответа, показан в листинге 9.8.

Листинг 9.8. Отправка потокового ответа

```
from django.http import StreamingHttpResponse

def index(request):
    resp_content = ('Здесь будет',' главная', ' страница', ' сайта')
    resp = StreamingHttpResponse(resp_content,
           content_type='text/plain; charset=utf-8')
    resp['keywords'] = 'Python, Django'
    return resp
```

9.8.2. Отправка файлов

Для отправки клиентам файлов применяется класс `FileResponse` — производный от класса `StreamingHttpResponse`:

```
FileResponse(<файловый объект>[, as_attachment=False][, filename=''][,
             content_type=None][, status=200][, reason=None])
```

Пример:

```
from django.http import FileResponse
filename = r'c:/images/image.png'
return FileResponse(open(filename, 'rb'))
```

Если отправить файл таким образом, он будет открыт непосредственно в веб-обозревателе. Чтобы дать веб-обозревателю указание сохранить файл на локальном диске, достаточно указать в вызове конструктора класса `FileResponse` параметры:

- `as_attachment` со значением `True`;
- `filename`, в котором задать имя сохраняемого файла, — если заданный первым параметром *файловый объект* не содержит имени файла (например, если он был сформирован программно в оперативной памяти).

Пример:

```
filename = r'c:/archives/archive.zip'
return FileResponse(open(filename, 'rb'), as_attachment=True)
```

9.8.3. Отправка данных в формате JSON

Для отправки данных в формате JSON применяется класс `JsonResponse` — производный от класса `HttpResponse`. Формат его конструктора:

```
JsonResponse(<данные>[, safe=True][, encoder=DjangoJSONEncoder])
```

Кодируемые в JSON *данные* должны быть представлены в виде словаря Python. Если нужно закодировать и отправить что-либо отличное от словаря, надо указать для параметра `safe` значение `False`. Параметр `encoder` задает кодировщик в формат JSON, если он не указан, используется стандартный кодировщик.

Пример:

```
data = {'title': 'Мотоцикл', 'content': 'Старый', 'price': 10000.0}
return JsonResponse(data)
```

9.9. Сокращения Django

Сокращение — это функция, производящая сразу несколько действий и предназначенная для выполнения типичных задач. Применение сокращений, в частности для рендеринга шаблонов, позволит несколько уменьшить код и упростить программирование.

Далее приведены все сокращения, что доступны в Django. Они объявлены в модуле `django.shortcuts`:

- `render(<запрос>, <путь к шаблону>[, context=None][, content_type=None][, status=200])` — выполняет рендеринг шаблона и отправку получившейся в результате страницы клиенту. *Запрос* должен быть представлен в виде экземпляра класса `Request`, *путь к шаблону* — в виде строки.

  Необязательный параметр `context` указывает контекст данных, `content_type` — MIME-тип и кодировку отправляемого ответа (по умолчанию `text/html` с кодировкой, взятой из параметра `DEFAULT_CHARSET` настроек проекта), а `status` — числовой код статуса (по умолчанию — `200`).

  В качестве результата возвращается готовый ответ в виде экземпляра класса `HttpResponse`.

  Пример:

  ```
  return render(request, 'bboard/index.html', context)
  ```

- `redirect(<цель>[, permanent=False][, <значения URL-параметров>])` — автоматически, сообразно заданным параметрам, формирует интернет-адрес для перенаправления и выполняет перенаправление.

В качестве цели могут быть заданы:

- объект модели — тогда интернет-адрес для перенаправления будет получен вызовом метода `get_absolute_url()` модели (см. *разд. 4.6*);
- имени маршрута (возможно, с указанием пространства имен) и набора *значений URL-параметров* — тогда интернет-адрес для перенаправления будет сформирован с применением обратного разрешения.

  Значения URL-параметров могут быть указаны в вызове функции в виде как позиционных, так и именованных параметров;
- полного интернет-адреса.

Необязательный параметр `permanent` указывает тип перенаправления. Если его значение равно `False`, или если он опущен, выполняется временное перенаправление. Если задать для этого параметра значение `True`, будет выполняться постоянное перенаправление.

В качестве результата возвращается полностью сформированный экземпляр класса `HttpResponseRedirect`.

Пример:

```
return redirect('bboard:by_rubric',
                rubric_id=bbf.cleaned_data['rubric'].pk)
```

- `get_object_or_404(<источник>, <условия поиска>)` — ищет запись согласно заданным *условиям поиска* и возвращает ее в качестве результата. Если запись найти не удается, возбуждает исключение `Http404`. В качестве *источника* можно указать класс модели, диспетчер записей (экземпляр класса `Manager`) или набор записей (экземпляр класса `QuerySet`).

  Если заданным условиям поиска удовлетворяют несколько записей, возбуждается исключение `MultipleObjectsReturned`.

  Пример:

```
def detail(request, bb_id):
    bb = get_object_or_404(Bb, pk=bb_id)
    return HttpResponse( . . . )
```

- `get_list_or_404(<источник>, <условия фильтрации>)` — применяет к записям заданные *условия фильтрации* и возвращает в качестве результата полученный набор записей (экземпляр класса `QuerySet`). Если ни одной записи, удовлетворяющей условиям, не существует, возбуждает исключение `Http404`. В качестве *источника* можно указать класс модели, диспетчер записей или набор записей.

  Пример:

```
def by_rubric(request, rubric_id):
    bbs = get_list_or_404(Bb, rubric=rubric_id)
    . . .
```

9.10. Дополнительные настройки контроллеров

Django предоставляет ряд декораторов, позволяющий задать дополнительные настройки контроллеров-функций. Рассмотрим их.

Прежде всего, это несколько декораторов, устанавливающих набор HTTP-методов, с применением которых разрешено обращаться к контроллеру. Они объявлены в модуле `django.views.decorators.http`:

- `require_http_methods(`*<последовательность обозначений методов>*`)` — разрешает для контроллера только те HTTP-методы, чьи обозначения указаны в заданной *последовательности*:

```
from Django.views.decorators.http import require_http_methods
@require_http_methods(['GET', 'POST'])
def add(request):
    . . .
```

- `require_get()` — разрешает для контроллера только метод GET;
- `require_post()` — разрешает для контроллера только метод POST;
- `require_safe()` — разрешает для контроллера только методы GET и HEAD (они считаются безопасными, так как не изменяют внутренние данные сайта).

Если к контроллеру, помеченному одним из этих декораторов, выполняется попытка доступа с применением недопустимого HTTP-метода, декоратор возвращает экземпляр класса `HttpResponseNotAllowed` (см. *разд. 9.7*), тем самым отправляя клиенту сообщение о недопустимом методе.

Декоратор `gzip_page()` из модуля `django.views.decorators.gzip` сжимает ответ, сгенерированный помеченным контроллером, с применением алгоритма GZIP (конечно, если веб-обозреватель поддерживает такое сжатие).

ГЛАВА 10

Контроллеры-классы

Контроллер-класс, как уже понятно из названия, реализован в виде класса. В отличие от функции, это сущность более высокого уровня — ряд типичных действий он выполняет самостоятельно.

Основная часть функциональности различных контроллеров-классов реализована в *примесях* (классах, предназначенных лишь для расширения функциональности других классов), от которых наследуют контроллеры-классы. Мы рассмотрим как примеси, так и полноценные классы.

10.1. Введение в контроллеры-классы

Контроллер-класс записывается в маршруте не в виде ссылки, как контроллер-функция, а в виде результата, возвращенного методом `as_view()`, который поддерживается всеми контроллерами-классами. Вот пример указания в маршруте контроллера-класса `CreateView`:

```
path('add/', CreateView.as_view()),
```

В вызове метода `as_view()` можно указать параметры контроллера-класса. Вот пример указания модели и пути к шаблону (параметры `model` и `template_name` соответственно):

```
path('add/', CreateView.as_view(model=Bb,
                                template_name='bboard/create.html')),
```

Задать значения параметров для контроллера-класса можно и по-другому: создав производный от него класс и указав в нем значения параметров в соответствующих атрибутах класса. Вот пример:

```
class BbCreateView(CreateView):
    template_name = 'bboard/create.html'
    model = Bb
. . .
path('add/', BbCreateView.as_view()),
```

Второй подход позволяет более радикально изменить поведение контроллера-класса, переопределив его методы. Поэтому и применяется он чаще.

10.2. Базовые контроллеры-классы

Знакомство с контроллерами этого типа мы начнем с самых простых и низкоуровневых классов, называемых *базовыми*. Все они объявлены в модуле `django.views.generic.base`.

10.2.1. Контроллер *View*: диспетчеризация по HTTP-методу

Контроллер-класс `View` является наиболее простым. Он всего лишь определяет HTTP-метод, посредством которого был выполнен запрос, и исполняет код, соответствующий этому методу.

Класс поддерживает атрибут `http_method_names`, хранящий список имен допустимых HTTP-методов. По умолчанию он хранит список `['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']`, включающий все методы, поддерживаемые протоколом HTTP.

Также поддерживаются три метода (помимо уже знакомого нам `as_view()`):

- `dispatch(<запрос>[, <значения URL-параметров>])` — должен выполнять обработку полученного *запроса* (экземпляра класса `HttpRequest`) и *значений URL-параметров*, извлеченных из интернет-адреса (*значения URL-параметров* передаются, как и в случае контроллеров-функций, через именованные параметры этого метода). Должен возвращать ответ, представленный экземпляром класса `HttpResponse` или его подклассом.

  В изначальной реализации извлекает обозначение HTTP-метода, вызывает метод класса, чье имя совпадает с этим обозначением, передает этому методу полученные *запрос* и *значения URL-параметров* и возвращает результат, возвращенный от этого метода. Например, если запрос был получен с применением HTTP-метода GET, будет вызван метод `get()`, а если был получен POST-запрос, вызывается метод `post()`.

  Если метод класса, одноименный с HTTP-методом, отсутствует, ничего не делает. Единственное исключение — HTTP-метод HEAD: при отсутствии метода `head()` вызывается метод `get()`;

- `http_method_not_allowed(<запрос>[, <значения URL-параметров>])` — вызывается, если запрос был выполнен с применением неподдерживаемого HTTP-метода.

  В изначальной реализации возвращает ответ типа `HttpResponseNotAllowed` со списком допустимых методов;

- `options(<запрос>[, <значения URL-параметров>)` — должен обрабатывать запрос с применением HTTP-метода OPTIONS.

В изначальной реализации возвращает ответ с заголовком `Allow`, в котором записаны все поддерживаемые HTTP-методы.

Помимо этого, в процессе работы класс создает несколько атрибутов экземпляра и сохраняет в них полезные данные. Вот эти атрибуты экземпляра:

- `request` — запрос, представленный экземпляром класса `Request`;
- `kwargs` — словарь с полученными из интернет-адреса URL-параметрами.

Класс `View` крайне редко используется как есть — обычно применяются производные от него контроллеры-классы, более сложные и выполняющие больше действий самостоятельно.

10.2.2. Примесь *ContextMixin*: создание контекста шаблона

Класс-примесь `ContextMixin` добавляет контроллеру-классу средства для формирования контекста шаблона. Их немного:

- `extra_context` — атрибут, задающий содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `get_context_data([<элементы контекста шаблона>])` — метод, должен создавать и возвращать контекст данных.

  В изначальной реализации создает пустой контекст данных, добавляет в него элемент `view`, хранящий ссылку на текущий экземпляр контроллера-класса, *элементы контекста шаблона*, полученные через именованные параметры, и все элементы из словаря `extra_context`.

  Внимание!

  Словарь, заданный в качестве значения атрибута `extra_content`, будет создан при первом обращении к контроллеру-классу и в дальнейшем останется неизменным. Если значением какого-либо элемента этого словаря является набор записей, он также останется неизменным, даже если впоследствии какие-то записи будут добавлены, исправлены или удалены.

  Поэтому добавлять в контекст шаблона элементы, хранящие наборы записей, следует только в переопределенном методе `get_context_data()`. В этом случае набор записей будет создаваться каждый раз при выполнении этого метода и отразит самые последние изменения в модели.

10.2.3. Примесь *TemplateResponseMixin*: рендеринг шаблона

Другой класс-примесь — `TemplateResponseMixin` — добавляет наследующему классу средства для рендеринга шаблона.

На заметку

Эта примесь и все наследующие от нее классы формируют ответ в виде экземпляра класса `TemplateResponse`, описанного в *разд. 9.3.3*.

Поддерживаются следующие атрибуты и методы:

- `template_name` — атрибут, задающий путь к шаблону в виде строки;
- `get_template_names()` — метод, должен возвращать список путей к шаблонам, заданных в виде строк.

  В изначальной реализации возвращает список из одного элемента — пути к шаблону, извлеченного из атрибута `template_name`;
- `content_type` — атрибут, задающий MIME-тип ответа и его кодировку. Если его значение равно `None`, будет использован MIME-тип и кодировка по умолчанию;
- `render_to_response(<контекст шаблона>)` — выполняет рендеринг шаблона и возвращает в качестве результата экземпляр класса `TemplateResponse`, представляющий ответ.

  Этот метод необходимо вызывать явно в нужном месте кода.

10.2.4. Контроллер *TemplateView*: все вместе

Контроллер-класс `TemplateView` наследует классы `View`, `ContextMixin` и `TemplateResponseMixin`. Он автоматически выполняет рендеринг шаблона и отправку ответа при получении запроса по методу GET.

При формировании контекста данных в него добавляются все URL-параметры, что присутствовали в маршруте.

Класс `TemplateView` уже можно применять в практической работе. Например, в листинге 10.1 показан код производного от него контроллера-класса `BbIndexView`, который выполняет вывод главной страницы сайта с объявлениями.

Листинг 10.1. Использование контроллера-класса `TemplateView`

```
from django.views.generic.base import TemplateView
from .models import Bb, Rubric

class BbIndexView(TemplateView):
    template_name = 'bboard/index.html'

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['bbs'] = Bb.objects.all()
        context['rubrics'] = Rubric.objects.all()
        return context
```

10.3. Классы, выводящие сведения о выбранной записи

Когда мы в *главах 1* и *2* писали наш первый Django-сайт — электронную доску объявлений — мы совсем забыли дать посетителю возможность просмотра сведений о выбранном им объявлении, в частности, заявленной в нем цены товара.

Разумеется, мы сейчас исправим и в будущем более не повторим эту ошибку. Нам понадобятся соответствующие классы, которые объявлены в модуле `django.views.generic.detail` и которые мы сейчас рассмотрим.

Это классы более высокоуровневые, чем рассмотренные нами в *разд. 10.2* базовые. Они носят название *обобщенных*, поскольку выполняют типовые задачи и могут быть использованы в различных ситуациях.

10.3.1. Примесь *SingleObjectMixin*: извлечение записи из модели

Класс-примесь `SingleObjectMixin`, наследующий от `ContextMixin`, — это настоящая находка для программиста! Он самостоятельно извлекает из модели запись по значению ее ключа или слага, которые, в свою очередь, самостоятельно получает из URL-параметров. Найденную запись он помещает в контекст шаблона.

Примесь поддерживает довольно много атрибутов и методов:

- `model` — атрибут, задает модель для извлечения записи;
- `queryset` — атрибут, указывает либо диспетчер записей, либо набор записей (`QuerySet`), в котором будет выполняться поиск записи;
- `get_queryset()` — метод, должен возвращать набор записей (`QuerySet`), в котором будет выполняться поиск записи.

  В изначальной реализации метод возвращает значение атрибута `queryset`, если оно задано, или набор записей, извлеченный из модели, заданной в атрибуте `model`;
- `pk_url_kwarg` — атрибут, задает имя URL-параметра, через который контроллер-класс получит значение ключа записи (по умолчанию: `"pk"`);
- `slug_field` — атрибут, задает имя поля модели, в котором хранится слаг (по умолчанию: `"slug"`);
- `get_slug_field()` — метод, должен возвращать строку с именем поля модели, в котором хранится слаг. В реализации по умолчанию возвращает значение из атрибута `slug_field`;
- `slug_url_kwarg` — атрибут, задает имя URL-параметра, через который контроллер-класс получит значение слага (по умолчанию: `"slug"`);
- `query_pk_and_slug` — атрибут. Если его значение равно `False`, поиск записи будет выполняться только по ключу (поведение по умолчанию), если `True` — по ключу и слагу;
- `context_object_name` — атрибут, задает имя переменной контекста шаблона, в которой будет сохранена найденная запись;
- `get_context_object_name(<запись>)` — метод, должен возвращать имя переменной контекста шаблона, в которой будет сохранена найденная *запись*, в виде строки.

В изначальной реализации возвращает имя, заданное в атрибуте `context_object_name` или, если этот атрибут хранит значение `None`, приведенное к нижнему регистру имя модели (так, если задана модель `Rubric`, переменная контекста шаблона получит имя `rubric`);

- `get_object([queryset=None])` — метод, выполняющий поиск записи по указанным критериям и возвращающий найденную запись в качестве результата. В необязательном параметре `queryset` можно указать набор записей, где будет выполняться поиск. Если он не задан, поиск будет производиться в наборе записей, возвращенном методом `get_queryset()`.

  Значения ключа и слага можно получить из словаря, содержащего все полученные контроллером URL-параметры и сохраненного в атрибуте экземпляра `kwargs`. Имена нужных URL-параметров можно найти в атрибутах `pk_url_kwarg` и `slug_url_kwarg`.

  Если запись найдена не будет, метод возбуждает исключение `Http404` из модуля `django.http`;

- `get_context_data([<дополнительные элементы контекста шаблона>])` — переопределенный метод, создающий и возвращающий контекст данных.

  В изначальной реализации требует, чтобы в экземпляре текущего контроллера-класса присутствовал атрибут `object`, хранящий найденную запись или `None`, если таковая не была найдена, или если контроллер используется для создания новой записи. В контексте шаблона создаются переменная `object` и переменная с именем, возвращенным методом `get_context_object_name()`, хранящие найденную запись.

10.3.2. Примесь *SingleObjectTemplateResponseMixin*: рендеринг шаблона на основе найденной записи

Класс-примесь `SingleObjectTemplateResponseMixin`, наследующий от `TemplateResponseMixin`, выполняет рендеринг шаблона на основе найденной в модели записи. Он требует, чтобы в контроллере-классе присутствовал атрибут `object`, в котором хранится либо найденная запись в виде объекта модели, либо `None`, если запись не была найдена, или если контроллер используется для создания новой записи.

Вот атрибуты и методы, поддерживаемые этим классом:

- `template_name_field` — атрибут, содержащий имя поля модели, в котором хранится путь к шаблону. Если `None`, путь к шаблону не будет извлекаться из записи модели (поведение по умолчанию);
- `template_name_suffix` — атрибут, хранящий строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_detail"`);
- `get_template_names()` — переопределенный метод, возвращающий список путей к шаблонам, заданных в виде строк.

В изначальной реализации возвращает список:

- либо из пути, извлеченного из унаследованного атрибута `template_name`, если этот путь указан;
- либо из:
  - пути, извлеченного из поля модели, чье имя хранится в атрибуте `template_name_field`, если все необходимые данные (имя поля, запись модели и сам путь в поле этой записи) указаны;
  - пути вида *<псевдоним приложения>\<имя модели><суффикс>*.html (так, для модели `Bb` из приложения `bboard` будет сформирован путь bboard\bb_detail.html).

10.3.3. Контроллер *DetailView*: все вместе

Контроллер-класс `DetailView` наследует классы `View`, `TemplateResponseMixin`, `SingleObjectMixin` и `SingleObjectTemplateResponseMixin`. Он самостоятельно ищет запись по значению ключа или слага, заносит ее в атрибут `object` (чтобы успешно работали наследуемые им примеси) и выводит на экран страницу со сведениями об этой записи.

В листинге 10.2 показан код контроллера-класса `BbDetailView`, производного от `DetailView` и выводящего страницу со сведениями о выбранном посетителем объявлении. Обратим внимание, насколько он компактен.

Листинг 10.2. Использование контроллера-класса `DetailView`

```
from django.views.generic.detail import DetailView
from .models import Bb, Rubric

class BbDetailView(DetailView):
    model = Bb

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Компактность кода класса обусловлена в том числе и тем, что мы следуем соглашениям. Так, мы не указали путь к шаблону — следовательно, класс будет искать шаблон со сформированным по умолчанию путем bboard\bb_detail.html.

Добавим в список маршрутов вот такой маршрут:

```
from .views import BbDetailView
urlpatterns = [
    . . .
    path('detail/<int:pk>/', BbDetailView.as_view(), name='detail'),
    . . .
]
```

Опять же, мы следуем соглашениям, указав для URL-параметра ключа записи имя pk, используемое классом DetailView по умолчанию.

Теперь напишем шаблон bboard\bb_detail.html. Его код можно увидеть в листинге 10.3.

Листинг 10.3. Код шаблона, выводящего объявление

```
{% extends "layout/basic.html" %}

{% block title %}{{ bb.title }}{% endblock %}

{% block content %}
<p>Рубрика: {{ bb.rubric.name }}</p>
<h2>{{ bb.title }}</h2>
<p>{{ bb.content }}</p>
<p>Цена: {{ bb.price }}</p>
{% endblock %}
```

По умолчанию класс BbDetailView создаст в контексте шаблона переменную bb, хранящую найденную запись (эту функциональность он унаследовал от базового класса DetailView). В коде шаблона мы используем эту переменную.

Осталось добавить в шаблоны bboard\index.html и bboard\by_rubric.html код, создающий гиперссылки, которые будут вести на страницу выбранного объявления. Вот этот код:

```
<h2><a href="{% url 'detail' pk=bb.pk %}">{{ bb.title }}</a></h2>
```

Как видим, применяя контроллеры-классы достаточно высокого уровня и, главное, следуя заложенным в них соглашениям, можно выполнять весьма сложные действия минимумом программного кода.

10.4. Классы, выводящие наборы записей

Другая разновидность обобщенных классов выводит на экран целый набор записей. Они объявлены в модуле django.views.generic.list.

10.4.1. Примесь *MultipleObjectMixin*: извлечение набора записей из модели

Класс-примесь MultipleObjectMixin, наследующий от ContextMixin, извлекает из модели набор записей, возможно, с применением фильтрации, сортировки и разбиения на части посредством пагинатора (о пагинаторе разговор пойдет в *главе 12*). Полученный набор записей он помещает в контекст шаблона.

Номер части, которую нужно извлечь, передается в составе интернет-адреса, через URL- или GET-параметр `page`. Номер должен быть целочисленным и начинаться с 1. Если это правило нарушено, будет возбуждено исключение `Http404`. Допустимо также указывать значение `"last"`, обозначающее последнюю часть.

Перечень поддерживаемых примесью атрибутов и методов весьма велик:

- `model` — атрибут, задает модель для извлечения записей;
- `queryset` — атрибут, указывает либо диспетчер записей, либо исходный набор записей (`QuerySet`), из которого будут извлекаться записи;
- `get_queryset()` — метод, должен возвращать исходный набор записей (`QuerySet`), из которого будут извлекаться записи.

  В изначальной реализации метод возвращает значение атрибута `queryset`, если оно задано, или набор записей, извлеченный из модели, что задана в атрибуте `model`;
- `ordering` — атрибут, задающий параметры сортировки записей. Значение может быть указано в виде:
  - строки с именем поля — для сортировки только по этому полю. По умолчанию будет выполняться сортировка по возрастанию значения поля. Чтобы указать сортировку по убыванию, нужно предварить имя поля символом «минус»;
  - последовательности строк с именами полей — для сортировки сразу по нескольким полям.

  Если значение атрибута не указано, станет выполняться сортировка, заданная в параметрах модели, или, когда таковое не указано, записи сортироваться не будут;
- `get_ordering()` — метод, должен возвращать параметры сортировки записей. В изначальной реализации возвращает значение атрибута `ordering`;
- `paginate_by` — атрибут, задающий целочисленное количество записей в одной части пагинатора. Если не указан, или его значение равно `None`, набор записей не будет разбиваться на части;
- `get_paginate_by(`*<набор записей>*`)` — метод, должен возвращать количество записей полученного *набора*, помещающихся в одной части пагинатора. В изначальной реализации просто возвращает значение из атрибута `paginate_by`;
- `page_kwarg` — атрибут, указывающий имя URL- или GET-параметра, через который будет передаваться номер выводимой части пагинатора, в виде строки (по умолчанию: `"page"`);
- `paginate_orphans` — атрибут, задающий целочисленное минимальное количество записей, что могут присутствовать в последней части пагинатора. Если последняя часть пагинатора содержит меньше записей, оставшиеся записи будут выведены в предыдущей части. Если задать значение 0, в последней части может присутствовать сколько угодно записей (поведение по умолчанию);

- `get_paginate_orphans()` — метод, должен возвращать минимальное количество записей, помещающихся в последней части пагинатора. В изначальной реализации возвращает значение атрибута `paginate_orphans`;
- `allow_empty` — атрибут. Значение `True` разрешает извлечение «пустой», т. е. не содержащей ни одной записи, части пагинатора (поведение по умолчанию). Значение `False`, напротив, предписывает при попытке извлечения «пустой» части возбудить исключение `Http404`;
- `get_allow_empty()` — метод, должен возвращать `True`, если разрешено извлечение «пустой» части пагинатора, или `False`, если такое недопустимо. В изначальной реализации возвращает значение атрибута `allow_empty`;
- `paginator_class` — атрибут, указывающий класс используемого пагинатора (по умолчанию: `Paginator` из модуля `django.core.paginator`);
- `get_paginator(`*`<набор записей>`*`, `*`<количество записей в части>`*`[, orphans=0][, allow_empty_first_page=True])` — метод, должен создавать объект пагинатора и возвращать его в качестве результата. Первый параметр задает набор записей, который должен разбиваться на части, второй — количество записей в части. Необязательный параметр `orphans` указывает минимальное количество записей в последней части пагинатора, а параметр `allow_empty_first_page` — разрешено извлечение «пустой» части (`True`) или нет (`False`).

  В изначальной реализации создает экземпляр класса пагинатора, указанного в атрибуте `paginator_class`, передавая его конструктору все полученные им параметры;
- `paginate_queryset(`*`<набор записей>`*`, `*`<количество записей в части>`*`)` — метод, должен разбивать полученный *набор записей* на части с указанным *количеством записей* в каждой. Должен возвращать кортеж из четырех элементов:
  - объекта самого пагинатора;
  - объекта его текущей части, номер которой был получен с URL- или GET-параметром;
  - набора записей из текущей части;
  - `True`, если извлеченный набор записей действительно был разбит на части с применением пагинатора, и `False` в противном случае;
- `context_object_name` — атрибут, задает имя переменной контекста шаблона, в которой будет сохранен извлеченный набор записей;
- `get_context_object_name(`*`<набор записей>`*`)` — метод, должен возвращать имя переменной контекста шаблона, в которой будет сохранен извлеченный *набор записей*, в виде строки.

  В изначальной реализации возвращает имя, заданное в атрибуте `context_object_name` или, если оно не указано, приведенное к нижнему регистру имя модели, полученное из *набора записей*, с добавленным суффиксом `_list`;

- `get_context_data([object_list=None][,][<дополнительные элементы контекста шаблона>])` — переопределенный метод, создающий и возвращающий контекст данных.

  В изначальной реализации получает набор записей из необязательного параметра `object_list` или, если этот параметр не указан, из атрибута `object_list`. После чего возвращает контекст с пятью переменными:

  - `object_list` — набор всех записей или же только набор записей из текущей части пагинатора;
  - переменная с именем, возвращенным методом `get_context_object_name()`, — то же самое;
  - `is_paginated` — `True`, если применялся пагинатор, и `False` в противном случае;
  - `paginator` — объект пагинатора или `None`, если пагинатор не применялся;
  - `page_obj` — объект текущей страницы пагинатора или `None`, если пагинатор не применялся.

10.4.2. Примесь *MultipleObjectTemplateResponseMixin*: рендеринг шаблона на основе набора записей

Класс-примесь `MultipleObjectTemplateResponseMixin`, наследующий от `TemplateResponseMixin`, выполняет рендеринг шаблона на основе извлеченного из модели набора записей. Он требует, чтобы в контроллере-классе присутствовал атрибут `object_list`, в котором хранится набор записей.

Вот список атрибутов и методов, поддерживаемых им:

- `template_name_suffix` — атрибут, хранящий строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_list"`);
- `get_template_names()` — переопределенный метод, возвращающий список путей к шаблонам, заданных в виде строк.

  В изначальной реализации возвращает список из:

  - пути, полученного из унаследованного атрибута `template_name`, если этот путь указан;
  - пути вида *<псевдоним приложения>\<имя модели><суффикс>*.html (так, для модели `Bb` из приложения `bboard` будет сформирован путь bboard\bb_list.html).

10.4.3. Контроллер *ListView*: все вместе

Контроллер-класс `ListView` наследует классы `View`, `TemplateResponseMixin`, `MultipleObjectMixin` и `MultipleObjectTemplateResponseMixin`. Он самостоятельно извлекает из модели набор записей, записывает его в атрибут `object_list` (чтобы

успешно работали наследуемые им примеси) и выводит на экран страницу со списком записей.

В листинге 10.4 показан код контроллера-класса `BbByRubricView`, унаследованного от `ListView` и выводящего страницу со списком объявлений, которые относятся к выбранной посетителем рубрике.

Листинг 10.4. Использование контроллера-класса `ListView`

```
from django.views.generic.list import ListView
from .models import Bb, Rubric

class BbByRubricView(ListView):
    template_name = 'bboard/by_rubric.html'
    context_object_name = 'bbs'

    def get_queryset(self):
        return Bb.objects.filter(rubric=self.kwargs['rubric_id'])

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        context['current_rubric'] = Rubric.objects.get(
                                        pk=self.kwargs['rubric_id'])
        return context
```

Код этого контроллера у нас получился более объемным, чем у ранее написанного контроллера-функции `by_rubric()` (см. листинг 2.2). Это обусловлено особенностями используемого нами шаблона bboard\by_rubric.html (его код можно увидеть в листинге 2.3). Во-первых, нам нужно вывести в нем список рубрик и текущую рубрику, следовательно, придется добавить все эти данные в контекст шаблона, переопределив метод `get_context_data()`. Во-вторых, мы используем уже имеющийся шаблон, поэтому вынуждены указать в контроллере его имя и имя переменной контекста, в которой будет храниться список объявлений.

Обратим внимание, как было получено значение URL-параметра `rubric_id`: мы обратились к словарю, хранящемуся в атрибуте `kwargs` экземпляра и содержащему все URL-параметры, что были указаны в маршруте.

10.5. Классы, работающие с формами

Имеются обобщенные классы, рассчитанные на работу с формами, как связанными с моделями, так и обычными (формы, связанные с моделями, будут описаны в *главе 13*, а обычные — в *главе 17*). Все эти классы объявлены в модуле `django.views.generic.edit`.

10.5.1. Классы для вывода и валидации форм

Начнем мы с низкоуровневых классов, которые только и «умеют», что вывести форму, проверить занесенные в нее данные на корректность и, в случае ошибки, вывести повторно, вместе с предупреждающими сообщениями.

10.5.1.1. Примесь *FormMixin*: создание формы

Класс-примесь `FormMixin` является производным от класса `ContextMixin`. Он может создать форму (неважно, связанную с моделью или обычную), проверить введенные в нее данные, выполнить перенаправление, если данные прошли проверку, или инициировать повторный вывод формы в противном случае.

Вот набор поддерживаемых атрибутов и методов:

- `form_class` — атрибут, хранит ссылку на класс формы, что будет использоваться наследуемым контроллером;
- `get_form_class()` — метод, должен возвращать ссылку на класс используемой формы. В изначальной реализации возвращает значение атрибута `form_class`;
- `initial` — атрибут, хранящий словарь с изначальными данными для занесения в только что созданную форму. Ключи элементов этого словаря должны соответствовать полям формы, а значения элементов зададут значения полей. По умолчанию хранит пустой словарь;
- `get_initial()` — метод, должен возвращать словарь с изначальными данными для занесения в только что созданную форму. В изначальной реализации просто возвращает значение атрибута `initial`;
- `success_url` — атрибут, хранит интернет-адрес, на который будет выполнено перенаправление, если введенные в форму данные прошли проверку на корректность;
- `get_success_url()` — метод, должен возвращать интернет-адрес для перенаправления в случае, если введенные в форму данные прошли валидацию. В изначальной реализации возвращает значение атрибута `success_url`;
- `prefix` — атрибут, задает строковый префикс для имени формы, который будет присутствовать в создающем форму HTML-коде. Префикс стоит задавать только в том случае, если мы планируем поместить несколько однотипных форм в одном теге `<form>`. По умолчанию хранит значение `None` (отсутствие префикса);
- `get_prefix()` — метод, должен возвращать префикс для имени формы. В изначальной реализации возвращает значение из атрибута `prefix`;
- `get_form([form_class=None])` — метод, возвращающий объект используемой контроллером формы.

  В изначальной реализации, если класс формы указан в параметре `form_class`, создает и возвращает экземпляр этого класса, в противном случае — экземпляр класса, возвращенного методом `get_form_class()`. При этом конструктору класса формы передаются параметры, возвращенные методом `get_form_kwargs()`;

- `get_form_kwargs()` — метод, должен создавать и возвращать словарь с параметрами, которые будут переданы конструктору класса формы при создании его экземпляра в методе `get_form()`.

  В изначальной реализации создает словарь с элементами:

  - `initial` — словарь с изначальными данными, возвращенный методом `get_initial()`;
  - `prefix` — префикс для имени формы, возвращенный методом `get_prefix()`.

    Следующие два элемента создаются только в том случае, если для отправки запроса применялись HTTP-методы POST и PUT;
  - `data` — словарь с данными, занесенными в форму посетителем;
  - `files` — словарь с файлами, отправляемыми посетителем через форму;
- `get_context_data([<дополнительные элементы контекста шаблона>])` — переопределенный метод, создающий и возвращающий контекст данных.

  В изначальной реализации добавляет в контекст шаблона переменную `form`, хранящую созданную форму;
- `form_valid(<форма>)` — метод, должен выполнять обработку введенных в *форму* данных в том случае, если они прошли валидацию.

  В изначальной реализации просто выполняет перенаправление на интернет-адрес, возвращенный методом `get_success_url()`;
- `form_invalid(<форма>)` — метод, должен выполнять обработку ситуации, когда введенные в *форму* данные не проходят валидацию. В изначальной реализации повторно выводит форму на экран.

10.5.1.2. Контроллер *ProcessFormView*: вывод и обработка формы

Контроллер-класс `ProcessFormView`, производный от класса `View`, выводит форму на экран, принимает введенные в нее данные и проводит их валидацию.

Этот класс переопределяет три метода, доставшиеся ему в наследство от базового класса:

- `get(<запрос>[, <значения URL-параметров>])` — выводит форму на экран;
- `post(<запрос>[, <значения URL-параметров>])` — получает введенные в форму данные и выполняет их валидацию. Если валидация прошла успешно, вызывает метод `form_valid()`, в противном случае — метод `form_invalid()` (см. *разд. 10.5.1.1*);
- `put(<запрос>[, <значения URL-параметров>])` — то же, что и `post()`.

10.5.1.3. Контроллер-класс *FormView*: создание, вывод и обработка формы

Контроллер-класс `FormView`, являясь производным от `View`, `FormMixin`, `ProcessFormView` и `TemplateResponseMixin`, предоставляет все инструменты для обработки форм. Он

создает форму, выводит ее на экран с применением указанного шаблона, проверяет на корректность занесенные в нее данные и, в случае отрицательного результата проверки, выводит страницу с формой повторно. Нам остается только реализовать обработку корректных данных, переопределив метод `form_valid()`.

В листинге 10.5 показан код контроллера-класса `BbAddView`, добавляющего на виртуальную доску новое объявление.

Листинг 10.5. Использование контроллера-класса `FormView`

```
from django.views.generic.edit import FormView
from django.urls import reverse
from .models import Bb, Rubric

class BbAddView(FormView):
    template_name = 'bboard/create.html'
    form_class = BbForm
    initial = {'price': 0.0}

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context

    def form_valid(self, form):
        form.save()
        return super().form_valid(form)

    def get_form(self, form_class=None):
        self.object = super().get_form(form_class)
        return self.object

    def get_success_url(self):
        return reverse('bboard:by_rubric',
            kwargs={'rubric_id': self.object.cleaned_data['rubric'].pk})
```

При написании этого класса мы столкнемся с проблемой. Чтобы после сохранения объявления сформировать интернет-адрес для перенаправления, нам нужно получить значение ключа рубрики, к которой относится добавленное объявление. Поэтому мы переопределили метод `get_form()`, в котором сохранили созданную форму в атрибуте `object`. После этого мы в коде метода `get_success_url()` без проблем сможем получить доступ к форме и занесенным в нее данным.

Сохранение введенных в форму данных мы выполняем в переопределенном методе `form_valid()`.

10.5.2. Классы для работы с записями

Пользоваться низкоуровневыми классами, описанными в *разд. 10.5.1*, удобно далеко не всегда. Эти классы не предусматривают продолжительного хранения объекта формы, вследствие чего мы не сможем извлечь занесенные в нее данные в любой нужный нам момент времени. Да и перенос самих данных из формы в модель нам тоже придется выполнять самостоятельно.

Но Django предоставляет нам набор классов более высокого уровня, которые сами сохранят, исправят или удалят запись.

10.5.2.1. Примесь *ModelFormMixin*: создание формы, связанной с моделью

Класс-примесь `ModelFormMixin`, наследующий от классов `SingleObjectMixin` и `FormMixin`, полностью аналогичен последнему, но нацелен на работу с формами, связанными с моделями.

Вот список атрибутов и методов, которые поддерживаются этим классом:

- `model` — атрибут, задает ссылку на класс модели, на основе которой будет создана форма;
- `fields` — атрибут, указывает последовательность имен полей модели, которые должны присутствовать в форме.

  Можно либо указать модель и список полей из нее посредством атрибутов `model` и `fields`, либо непосредственно класс формы в атрибуте `form_class`, унаследованном от класса `FormMixin`, но никак не одновременно и то, и другое.

  Если указан атрибут `model`, обязательно следует задать также и атрибут `fields`. Если этого не сделать, возникнет ошибка;
- `get_form_class()` — переопределенный метод, должен возвращать ссылку на класс используемой формы.

  В изначальной реализации возвращает значение атрибута `form_class`, если он присутствует в классе. В противном случае возвращается ссылка на класс формы, автоматически созданный на основе модели, которая взята из атрибута `model` или извлечена из набора записей, заданного в унаследованном атрибуте `queryset`. Для создания класса формы также используется список полей, взятый из атрибута `fields`;
- `success_url` — атрибут, хранит интернет-адрес, на который будет выполнено перенаправление, если введенные в форму данные прошли проверку на корректность.

  В отличие от одноименного атрибута базового класса `FormMixin`, он поддерживает указание непосредственно в строке с интернет-адресом специальных последовательностей символов вида `{<имя поля таблицы в базе данных>}`. Вместо такой последовательности самим классом будет подставлено значение поля с указанным *именем*.

Отметим, что в упомянутые ранее последовательности должно подставляться имя не поля модели, а таблицы базы данных, которая обрабатывается моделью. Так, для получения ключа следует использовать поле `id`, а не `pk`, а для получения внешнего ключа — поле `rubric_id`, а не `rubric`.

Примеры:

```
class BbCreateView(CreateView):
    . . .
    success_url = '/bboard/detail/{id}'

class BbCreateView(CreateView):
    . . .
    success_url = '/bboard/{rubric_id}'
```

- `get_success_url()` — переопределенный метод, возвращает интернет-адрес для перенаправления в случае, если введенные данные прошли валидацию.

  В изначальной реализации возвращает значение атрибута `success_url`, в котором последовательности вида *{<имя поля таблицы в базе данных>}* заменены значениями соответствующих полей. Если атрибут отсутствует, пытается получить интернет-адрес вызовом метода `get_absolute_url()` модели;

- `get_form_kwargs()` — переопределенный метод, создает и возвращает словарь с параметрами, которые будут переданы конструктору класса формы при создании его экземпляра в методе `get_form()`.

  В изначальной реализации добавляет в словарь, сформированный унаследованным методом, элемент `instance`, хранящий обрабатываемую формой запись модели (если она существует, т. е. форма используется не для добавления записи). Эта запись извлекается из атрибута `object`;

- `form_valid(<форма>)` — переопределенный метод, должен выполнять обработку введенных в *форму* данных в том случае, если они прошли валидацию.

  В изначальной реализации сохраняет содержимое формы в модели, тем самым создавая или исправляя запись, присваивает новую запись атрибуту `object`, после чего вызывает унаследованный метод `form_valid()`.

10.5.2.2. Контроллер *CreateView*: создание новой записи

Контроллер-класс `CreateView` наследует от классов `View`, `ModelFormMixin`, `ProcessFormView`, `SingleObjectMixin`, `SingleObjectTemplateResponseMixin` и `TemplateResponseMixin`. Он обладает всей необходимой функциональностью для создания, вывода, валидации формы и создания записи на основе занесенных в форму данных.

Атрибут `template_name_suffix` этого класса хранит строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_form"`).

Также в классе доступен атрибут `object`, в котором хранится созданная в модели запись или `None`, если таковая еще не была создана.

Пример контроллера-класса, производного от `CreateView`, можно увидеть в листинге 2.7.

10.5.2.3. Контроллер *UpdateView*: исправление записи

Контроллер-класс `UpdateView` наследует от классов `View`, `ModelFormMixin`, `ProcessFormView`, `SingleObjectMixin`, `SingleObjectTemplateResponseMixin` и `TemplateResponseMixin`. Он самостоятельно найдет в модели запись по полученным из URL-параметра ключу или слагу, выведет страницу с формой для ее правки, проверит и сохранит исправленные данные.

Атрибут `template_name_suffix` этого класса хранит строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_form"`).

Также в классе доступен атрибут `object`, в котором хранится исправляемая запись модели.

Поскольку класс `UpdateView` предварительно выполняет поиск записи в модели, в нем необходимо указать модель (в унаследованном атрибуте `model`), набор записей (в атрибуте `queryset`) или переопределить метод `get_queryset()`.

В листинге 10.6 показан код контроллера-класса `BbEditView`, который выполняет исправление объявления.

Листинг 10.6. Использование контроллера-класса `UpdateView`

```
from django.views.generic.edit import UpdateView
from .models import Bb, Rubric

class BbEditView(UpdateView):
    model = Bb
    form_class = BbForm
    success_url = '/'

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Шаблон bboard\bb_form.html вы можете написать самостоятельно, взяв за основу уже имеющийся шаблон bboard\create.html (там всего лишь придется поменять текст **Добавление** на **Исправление**, а подпись кнопки — с **Добавить** на **Сохранить**). Также самостоятельно вы можете записать маршрут для этого контроллера и вставить в шаблоны bboard\index.html и bboard\by_rubric.html код, создающий гиперссылки, что ведут на страницы исправления записей.

10.5.2.4. Примесь *DeletionMixin*: удаление записи

Класс-примесь `DeletionMixin` добавляет наследующему ее контроллеру инструменты для удаления записи. Он предполагает, что запись, подлежащая удалению, уже найдена и сохранена в атрибуте `object`.

Класс объявляет атрибут `success_url` и метод `get_success_url()`, аналогичные тем, что присутствуют в примеси `ModelFormMixin` (см. *разд. 10.5.2.1*).

10.5.2.5. Контроллер *DeleteView*: удаление записи с подтверждением

Контроллер-класс `DeleteView` наследует от классов `View`, `DeletionMixin`, `DetailView`, `SingleObjectMixin`, `SingleObjectTemplateResponseMixin` и `TemplateResponseMixin`. Он самостоятельно найдет в модели запись по полученному из URL-параметра ключу или слагу, выведет страницу подтверждения, включающую в себя форму с кнопкой удаления, и удалит запись.

Атрибут `template_name_suffix` этого класса хранит строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_confirm_delete"`).

Также в классе доступен атрибут `object`, в котором хранится удаляемая запись модели.

Поскольку класс `DeleteView` предварительно выполняет поиск записи в модели, в нем необходимо указать модель (в атрибуте `model`), набор записей (в атрибуте `queryset`) или переопределить метод `get_queryset()`.

В листинге 10.7 показан код контроллера-класса `BbDeleteView`, который выполняет удаление объявления.

Листинг 10.7. Использование контроллера-класса `DeleteView`

```
from django.views.generic.edit import DeleteView
from .models import Bb, Rubric

class BbDeleteView(DeleteView):
    model = Bb
    success_url = '/'

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Что касается шаблона bboard\bb_confirm_delete.html, выводящего страницу подтверждения и форму с кнопкой **Удалить**, то его код показан в листинге 10.8.

Листинг 10.8. Код шаблона, выводящего страницу для подтверждения удаления записи

```
{% extends "layout/basic.html" %}

{% block title %}Удаление объявления{% endblock %}

{% block content %}
<h2>Удаление объявления</h2>
<p>Рубрика: {{ bb.rubric.name }}</p>
<p>Товар: {{ bb.title }}</p>
<p>{{ bb.content }}</p>
<p>Цена: {{ bb.price }}</p>
<form method="post">
    {% csrf_token %}
    <input type="submit" value="Удалить">
</form>
{% endblock %}
```

10.6. Классы для вывода хронологических списков

Целый набор обобщенных классов служит для вывода хронологических списков: за определенный год, месяц, неделю или дату, за текущее число. Все эти классы объявлены в модуле `django.views.generic.dates`.

10.6.1. Вывод последних записей

Сначала рассмотрим классы, предназначенные для вывода хронологического списка наиболее «свежих» записей.

10.6.1.1. Примесь *DateMixin*: фильтрация записей по дате

Класс-примесь `DateMixin` предоставляет наследующим контроллерам возможность фильтровать записи по значениям даты (или даты и времени), хранящимся в заданном поле.

Вот атрибуты и методы, поддерживаемые этим классом:

- `date_field` — атрибут, указывает имя поля модели типа `DateField` или `DateTimeField`, где хранятся значения даты, по которым будет выполняться фильтрация записей. Имя поля должно быть задано в виде строки;
- `get_date_field()` — метод, должен возвращать имя поля модели со значениями даты, по которым будет выполняться фильтрация записей. В изначальной реализации возвращает значение атрибута `date_field`;
- `allow_future` — атрибут. Значение `True` указывает включить в результирующий набор записей поля, у которых значение даты, записанное в возвращенное мето-

дом `get_date_field()` поле, больше текущего («будущие» записи). Значение `False` запрещает включать такие записи в набор (поведение по умолчанию);

- `get_allow_future()` — метод, должен возвращать значение `True` или `False`, говорящее, следует ли включать в результирующий набор «будущие» записи. В изначальной реализации возвращает значение атрибута `allow_future`.

10.6.1.2. Контроллер *BaseDateListView*: базовый класс

Контроллер-класс `BaseDateListView` наследует от классов `DateMixin` и `MultipleObjectMixin`. Он предоставляет базовую функциональность для других, более специализированных классов, в частности, задает сортировку записей по убыванию значения поля, возвращенного методом `get_date_field()` класса `DateMixin` (см. *разд. 10.6.1.1*).

В нем объявлены такие атрибуты и методы:

- `allow_empty` — атрибут. Значение `True` разрешает извлечение «пустой», т. е. не содержащей ни одной записи, части пагинатора. Значение `False`, напротив, предписывает при попытке извлечения «пустой» части возбудить исключение `Http404` (поведение по умолчанию);
- `date_list_period` — атрибут, указывающий, по какой части следует урезать значения даты. Должен содержать значение `"year"` (год, значение по умолчанию), `"month"` (месяц) или `"day"` (число, т. е. дата не будет урезаться);
- `get_date_list_period()` — метод, должен возвращать обозначение части, по которой нужно урезать дату. В изначальной реализации возвращает значение атрибута `date_list_period`;
- `get_dated_items()` — метод, должен возвращать кортеж из трех элементов:
  - список значений дат, для которых в наборе существуют записи;
  - набор записей;
  - словарь, элементы которого будут добавлены в контекст шаблона.

  В изначальной реализации возбуждает исключение `NotImplementedError`. Предназначен для переопределения в подклассах;
- `get_dated_queryset(`*<условия фильтрации>*`)` — метод, возвращает набор записей, отфильтрованный согласно заданным *условиям*;
- `get_date_list(`*<набор записей>*`[, date_type=None][, ordering='ASC'])` — метод, возвращает список значений даты, урезанной по части, что задана в параметре `date_type` (если он отсутствует, будет использовано значение, возвращенное методом `get_date_list_period()`), для которых в заданном *наборе* есть записи. Параметр `ordering` задает направление сортировки: `"ASC"` (по возрастанию, поведение по умолчанию) или `"DESC"` (по убыванию).

Класс добавляет в контекст шаблона два дополнительных элемента:

- `object_list` — результирующий набор записей;
- `date_list` — список урезанных значений дат, для которых в наборе есть записи.

10.6.1.3. Контроллер *ArchiveIndexView*: вывод последних записей

Контроллер-класс `ArchiveIndexView` наследует от классов `View`, `DateMixin`, `BaseDateListView`, `TemplateResponseMixin`, `MultipleObjectMixin` и `MultipleObjectTemplateResponseMixin`. Он выводит хронологический список записей, отсортированных по убыванию значения заданного поля.

Для хранения результирующего набора выводимых записей в контексте шаблона создается переменная `latest`. В переменной `date_list` контекста шаблона хранится список значений дат, урезанных до года. К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс `_archive`.

Листинг 10.9 показывает код контроллера-класса `BbIndexView`, основанного на классе `ArchiveIndexView`. Для вывода страницы он может использовать шаблон bboard\index.html, написанный нами в *главах 1* и *2*.

Листинг 10.9. Применение контроллера-класса `ArchiveIndexView`

```
from django.views.generic.dates import ArchiveIndexView
from .models import Bb, Rubric

class BbIndexView(ArchiveIndexView):
    model = Bb
    date_field = 'published'
    template_name = 'bboard/index.html'
    context_object_name = 'bbs'
    allow_empty = True

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

А вот так мы можем использовать хранящийся в переменной `date_list` контекста шаблона список дат, урезанных до года:

```
<p>
    {% for d in date_list %}
    {{ d.year }}
    {% endfor %}
</p>
```

В результате на экране появится список разделенных пробелами годов, за которые в наборе имеются записи.

10.6.2. Вывод записей по годам

Следующая пара классов выводит список записей, относящихся к определенному году.

10.6.2.1. Примесь *YearMixin*: извлечение года

Класс-примесь `YearMixin` извлекает из URL- или GET-параметра с именем `year` значение года, которое будет использовано для последующей фильтрации записей.

Этот класс поддерживает следующие атрибуты и методы:

- `year_format` — атрибут, указывает строку с форматом значения года, поддерживаемым функцией `strftime()` языка Python. Значение года будет извлекаться из интернет-адреса и преобразовываться в нужный вид согласно этому формату. Значение по умолчанию: `"%Y"` (год из четырех цифр);
- `get_year_format()` — метод, должен возвращать строку с форматом значения года. В изначальной реализации возвращает значение атрибута `year_format`;
- `year` — атрибут, задает значение года в виде строки. Если `None`, год будет извлекаться из интернет-адреса (поведение по умолчанию);
- `get_year()` — метод, должен возвращать значение года в виде строки. В изначальной реализации пытается вернуть значение (если оно задано): атрибута класса `year`, URL-параметра `year`, GET-параметра `year`. Если все попытки завершились неудачами, возбуждает исключение `Http404`;
- `get_previous_year(<дата>)` — метод, возвращает значение даты, представляющей собой первый день года, который предшествует указанной *дате*. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`;
- `get_next_year(<дата>)` — метод, возвращает значение даты, представляющей собой первый день года, который следует за указанной *датой*. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`.

10.6.2.2. Контроллер *YearArchiveView*: вывод записей за год

Контроллер-класс `YearArchiveView` наследует классы `View`, `DateMixin`, `YearMixin`, `BaseDateListView`, `TemplateResponseMixin`, `MultipleObjectMixin` и `MultipleObject-TemplateResponseMixin`. Он выводит хронологический список записей, относящихся к указанному году.

Класс поддерживает дополнительные атрибут и метод:

- `make_object_list` — атрибут. Если `True`, будет сформирован и добавлен в контекст шаблона набор всех записей за заданный год. Если `False`, в контексте шаблона будет присутствовать «пустой» набор записей (поведение по умолчанию);
- `get_make_object_list()` — метод, должен возвращать логический признак того, формировать ли полноценный набор записей, относящихся к заданному году. В изначальной реализации возвращает значение атрибута `make_object_list`.

В контексте шаблона будут созданы следующие переменные:

- `date_list` — список значений дат, за которые в наборе существуют записи, урезанных до месяца и выстроенных в порядке возрастания;
- `year` — объект типа `date`, представляющий заданный год;
- `previous_year` — объект типа `date`, представляющий предыдущий год;
- `next_year` — объект типа `date`, представляющий следующий год.

Набор записей, относящихся к заданному году, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: `object_list`.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс `_archive_year`.

10.6.3. Вывод записей по месяцам

Далее мы познакомимся с классами, выводящими список записей, что относятся к определенному месяцу определенного года.

10.6.3.1. Примесь *MonthMixin*: извлечение месяца

Класс-примесь `MonthMixin` извлекает из URL- или GET-параметра с именем `month` значение месяца, которое будет использовано для последующей фильтрации записей.

Поддерживаются следующие атрибуты и методы:

- `month_format` — атрибут, указывает строку с форматом значения месяца, поддерживаемым функцией `strftime()` языка Python. Значение месяца будет извлекаться из интернет-адреса и преобразовываться в нужный вид согласно этому формату. Значение по умолчанию: `"%b"` (сокращенное наименование, записанное согласно текущим языковым настройкам);
- `get_month_format()` — метод, должен возвращать строку с форматом значения месяца. В изначальной реализации возвращает значение атрибута `month_format`;
- `month` — атрибут, задает значение месяца в виде строки. Если `None`, месяц будет извлекаться из интернет-адреса (поведение по умолчанию);
- `get_month()` — метод, должен возвращать значение месяца в виде строки. В изначальной реализации пытается вернуть значение (если оно задано): атрибута класса `month`, URL-параметра `month`, GET-параметра `month`. Если все попытки завершились неудачами, возбуждает исключение `Http404`;
- `get_previous_month(<дата>)` — метод, возвращает значение даты, представляющей собой первый день месяца, который предшествует указанной *дате*. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`;
- `get_next_month(<дата>)` — метод, возвращает значение даты, представляющей собой первый день месяца, который следует за указанной *датой*. В зависимости

от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`.

10.6.3.2. Контроллер *MonthArchiveView*: вывод записей за месяц

Контроллер-класс `MonthArchiveView` наследует классы `View`, `DateMixin`, `MonthMixin`, `YearMixin`, `BaseDateListView`, `TemplateResponseMixin`, `MultipleObjectMixin` и `MultipleObjectTemplateResponseMixin`. Он выводит хронологический список записей, относящихся к указанному месяцу указанного года.

В контексте шаблона будут созданы следующие переменные:

- `date_list` — список значений дат, за которые в наборе существуют записи, урезанных до значения числа и выстроенных в порядке возрастания;
- `month` — объект типа `date`, представляющий заданный месяц;
- `previous_month` — объект типа `date`, представляющий предыдущий месяц;
- `next_month` — объект типа `date`, представляющий следующий месяц.

Набор записей, относящихся к заданному месяцу, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: `object_list`.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс `_archive_month`.

Вот небольшой пример использования контроллера `MonthArchiveView`:

```
urlpatterns = [
    # Пример интернет-адреса: /2018/06/
    path('<int:year>/<int:month>/', BbMonthArchiveView.as_view()),
]
. . .
class BbMonthArchiveView(MonthArchiveView):
    model = Bb
    date_field = "published"
    month_format = '%m'             # Порядковый номер месяца
```

10.6.4. Вывод записей по неделям

Далее мы познакомимся с классами, выводящими список записей, что относятся к неделе с заданным порядковым номером.

10.6.4.1. Примесь *WeekMixin*: извлечение номера недели

Класс-примесь `WeekMixin` извлекает из URL- или GET-параметра с именем `week` номер недели, который будет использован для фильтрации записей.

Поддерживаются такие атрибуты и методы:

- `week_format` — атрибут, указывает строку с форматом номера недели, поддерживаемым функцией `strftime()` языка Python. Значение номера недели будет извлекаться из интернет-адреса и преобразовываться в нужный вид согласно этому

формату. Значение по умолчанию: "%U" (номер недели, если первый день недели — воскресенье).

Для обработки более привычного формата номера недели, когда первым днем является понедельник, нужно указать в качестве значения формата строку "%W";

- get_week_format() — метод, должен возвращать строку с форматом значения недели. В изначальной реализации возвращает значение атрибута week_format;
- week — атрибут, задает значение недели в виде строки. Если None, номер недели будет извлекаться из интернет-адреса (поведение по умолчанию);
- get_week() — метод, должен возвращать значение недели в виде строки. В изначальной реализации пытается вернуть значение (если оно задано): атрибута класса week, URL-параметра week, GET-параметра week. Если все попытки завершились неудачами, возбуждает исключение Http404;
- get_previous_week(*<дата>*) — метод, возвращает значение даты, представляющей собой первый день недели, которая предшествует указанной *дате*. В зависимости от значений атрибутов класса allow_empty и allow_future может возбуждать исключение Http404;
- get_next_week(*<дата>*) — метод, возвращает значение даты, представляющей собой первый день недели, которая следует за указанной *датой*. В зависимости от значений атрибутов класса allow_empty и allow_future может возбуждать исключение Http404.

10.6.4.2. Контроллер *WeekArchiveView*: вывод записей за неделю

Контроллер-класс WeekArchiveView наследует классы View, DateMixin, WeekMixin, YearMixin, BaseDateListView, TemplateResponseMixin, MultipleObjectMixin и MultipleObjectTemplateResponseMixin. Он выводит хронологический список записей, относящихся к указанной неделе указанного года.

В контексте шаблона будут созданы следующие переменные:

- week — объект типа date, представляющий заданную неделю;
- previous_week — объект типа date, представляющий предыдущую неделю;
- next_week — объект типа date, представляющий следующую неделю.

Набор записей, относящихся к заданной неделе, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: object_list.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс _archive_week.

Пример использования контроллера WeekArchiveView:

```
urlpatterns = [
    # Пример интернет-адреса: /2018/week/24/
    path('<int:year>/week/<int:week>/',
         WeekArchiveView.as_view(model=Bb, date_field="published")),
]
```

10.6.5. Вывод записей по дням

Два класса реализуют вывод списков записей, которые относятся к определенному дню заданного года. Познакомимся с ними.

10.6.5.1. Примесь *DayMixin*: извлечение заданного числа

Класс-примесь `DayMixin` извлекает из URL- или GET-параметра с именем `day` число, что будет использовано для фильтрации записей.

Атрибуты и методы, поддерживаемые классом, таковы:

- `day_format` — атрибут, указывает строку с форматом числа, поддерживаемым функцией `strftime()` языка Python. Значение числа будет извлекаться из интернет-адреса и преобразовываться в нужный вид согласно этому формату. Значение по умолчанию: `"%d"` (число с начальным нулем);
- `get_day_format()` — метод, должен возвращать строку с форматом значения числа. В изначальной реализации возвращает значение атрибута `day_format`;
- `day` — атрибут, задает значение числа в виде строки. Если `None`, число будет извлекаться из интернет-адреса (поведение по умолчанию);
- `get_day()` — метод, должен возвращать значение числа в виде строки. В изначальной реализации пытается вернуть значение (если оно задано): атрибута класса `day`, URL-параметра `day`, GET-параметра `day`. Если все попытки завершились неудачами, возбуждает исключение `Http404`;
- `get_previous_day(<дата>)` — метод, возвращает значение даты, которая предшествует указанной *дате*. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`;
- `get_next_day(<дата>)` — метод, возвращает значение даты, которая следует за указанной *датой*. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`.

10.6.5.2. Контроллер *DayArchiveView*: вывод записей за день

Контроллер-класс `DayArchiveView` наследует классы `View`, `DateMixin`, `DayMixin`, `MonthMixin`, `YearMixin`, `BaseDateListView`, `TemplateResponseMixin`, `MultipleObjectMixin` и `MultipleObjectTemplateResponseMixin`. Он выводит хронологический список записей, относящихся к указанному числу заданных месяца и года.

В контексте шаблона будут созданы следующие переменные:

- `day` — объект типа `date`, представляющий заданный день;
- `previous_day` — объект типа `date`, представляющий предыдущий день;
- `next_day` — объект типа `date`, представляющий следующий день.

Набор записей, относящихся к заданному дню, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: `object_list`.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс _archive_day.

Пример использования контроллера DayArchiveView:

```
urlpatterns = [
    # Пример интернет-адреса: /2018/06/24/
    path('<int:year>/<int:month>/<int:day>/',
         DayArchiveView.as_view(model=Bb, date_field="published",
                                month_format='%m')),
]
```

10.6.6. Контроллер *TodayArchiveView*: вывод записей за текущее число

Контроллер-класс TodayArchiveView наследует классы View, DateMixin, DayMixin, MonthMixin, YearMixin, BaseDateListView, DayArchiveView, TemplateResponseMixin, MultipleObjectMixin и MultipleObjectTemplateResponseMixin. Он выводит хронологический список записей, относящихся к текущему числу.

В контексте шаблона будут созданы следующие переменные:

- day — объект типа date, представляющий текущее число;
- previous_day — объект типа date, представляющий предыдущий день;
- next_day — объект типа date, представляющий следующий день.

Набор записей, относящихся к текущему числу, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: object_list.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс _archive_today.

10.6.7. Контроллер *DateDetailView*: вывод одной записи за указанное число

Контроллер-класс DateDetailView наследует классы View, DateMixin, DayMixin, MonthMixin, YearMixin, DetailView, TemplateResponseMixin, SingleObjectMixin и SingleObjectTemplateResponseMixin. Он выводит одну-единственную запись, относящуюся к текущему числу, и может быть полезен в случаях, когда поле даты, по которому выполняется поиск записи, хранит уникальные значения.

Запись, относящаяся к текущему числу, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: object.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс _detail.

Листинг 10.10 представляет код контроллера BbDetailView, основанного на классе DateDetailView.

Листинг 10.10. Использование контроллера-класса DateDetailView

```
from django.views.generic.dates import DateDetailView
from .models import Bb, Rubric

class BbDetailView(DateDetailView):
    model = Bb
    date_field = 'published'
    month_format = '%m'

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Для его использования в список маршрутов мы добавим маршрут такого вида:

```
path('detail/<int:year>/<int:month>/<int:day>/<int:pk>/',
     BbDetailView.as_view(), name='detail'),
```

К сожалению, в маршрут, указывающий на контроллер `DateDetailView` или его подкласс, следует записать URL-параметр ключа (`pk`) или слага (`slug`). Это связано с тем, что упомянутый ранее класс является производным от примеси `SingleObjectMixin`, которая требует для нормальной работы один из этих URL-параметров. Такая особенность сильно ограничивает область применения контроллера-класса `DateDetailView`.

10.7. Контроллер *RedirectView*: перенаправление

Контроллер-класс `RedirectView`, производный от класса `View`, выполняет перенаправление на указанный интернет-адрес. Он объявлен в модуле `django.views.generic.base`.

Этот класс поддерживает такие атрибуты и методы:

- `url` — атрибут, задает строку с интернет-адресом, на который следует выполнить перенаправление.

  Этот интернет-адрес может включать спецификаторы, поддерживаемые оператором `%` языка Python (за подробностями — на страницу **https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting**). В таких спецификаторах следует указывать имена URL-параметров — в этом случае в результирующий интернет-адрес будут подставлены их значения;

- `pattern_name` — атрибут, задает имя именованного маршрута. Обратное разрешение интернет-адреса будет произведено с теми же URL-параметрами, что получил контроллер;

- `query_string` — атрибут. Если `True`, все GET-параметры, присутствующие в текущем интернет-адресе, будут добавлены к интернет-адресу, на который выполняется перенаправление. Если `False`, GET-параметры передаваться не будут (поведение по умолчанию);
- `get_redirect_url(`*<значения URL-параметров>*`)` — метод, должен возвращать строку с интернет-адресом, на который следует выполнить перенаправление.

  В реализации по умолчанию сначала извлекает значение атрибута `url` и выполняет форматирование, передавая *значения полученных URL-параметров* оператору `%`. Если атрибут `url` хранит значение `None`, производит обратное разрешение на основе значения атрибута `pattern_name` и, опять же, *значений URL-параметров*. Если и эта попытка увенчалась неудачей, отправляет ответ с кодом 410 (запрошенная страница более не существует);
- `permanent` — атрибут. Если `True`, будет выполнено постоянное перенаправление (с кодом статуса 301). Если `False`, будет выполнено временное перенаправление (с кодом статуса 302). Значение по умолчанию: `False`.

В качестве примера давайте организуем перенаправление с интернет-адресов вида **/detail/*<год>*/*<месяц>*/*<число>*/*<ключ>*/** на интернет-адреса **/detail/*<ключ>*/**. Для этого мы добавим в список маршруты:

```
path('detail/<int:pk>/', BbDetailView.as_view(), name='detail'),
path('detail/<int:year>/<int:month>/<int:day>/<int:pk>/',
     BbRedirectView.as_view(), name='old_detail'),
```

Код контроллера `BbRedirectView`, который мы используем для этого, очень прост и показан в листинге 10.11.

Листинг 10.11. Применение контроллера-класса `RedirectView`

```
class BbRedirectView(RedirectView):
    url = '/detail/%(pk)d/'
```

Для формирования целевого интернет-адреса мы использовали атрибут `url` и строку со спецификатором, обрабатываемым оператором `%`. Вместо этого спецификатора в строку будет подставлено значение URL-параметра `pk`, т. е. ключ записи.

10.8. Контроллеры-классы смешанной функциональности

Бóльшая часть функциональности контроллеров-классов наследуется ими от классов-примесей. Это позволяет нам, наследуя классы от нужных примесей, создавать контроллеры смешанной функциональности.

Так, мы можем объявить класс, производный от классов `SingleObjectMixin` и `ListView`. В результате получится контроллер, одновременно выводящий сведения

о выбранной записи (функциональность, унаследованная от `SingleObjectMixin`) и набор связанных с ней записей (функциональность класса `ListView`).

В листинге 10.12 показан код класса `BbByRubricView`, созданного на подобном принципе и имеющего смешанную функциональность.

Листинг 10.12. Пример контроллера-класса смешанной функциональности

```
from django.views.generic.detail import SingleObjectMixin
from django.views.generic.list import ListView
from .models import Bb, Rubric

class BbByRubricView(SingleObjectMixin, ListView):
    template_name = 'bboard/by_rubric.html'
    pk_url_kwarg = 'rubric_id'

    def get(self, request, *args, **kwargs):
        self.object = self.get_object(queryset=Rubric.objects.all())
        return super().get(request, *args, **kwargs)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['current_rubric'] = self.object
        context['rubrics'] = Rubric.objects.all()
        context['bbs'] = context['object_list']
        return context

    def get_queryset(self):
        return self.object.bb_set.all()
```

Намереваясь использовать уже существующие шаблон и маршрут, мы указали в атрибутах класса путь к нашему шаблону и имя URL-параметра, через который передается ключ рубрики.

В переопределенном методе `get()` вызовом метода `get_object()`, унаследованного от примеси `SingleObjectMixin`, мы извлекаем рубрику с заданным ключом. Эту рубрику мы сохраняем в атрибуте `object` класса — она нам еще понадобится.

В переопределенном методе `get_context_data()` мы заносим в переменную `current_rubric` контекста шаблона найденную рубрику (взяв ее из атрибута `object`), а в переменную `rubrics` — набор всех рубрик.

Здесь мы столкнемся с проблемой. Наш шаблон за набором объявлений обращается к переменной `bbs` контекста шаблона. Ранее, разрабатывая предыдущую редакцию контроллера (см. листинг 10.4), мы указали имя для этой переменной, записав ее в атрибуте класса `contect_object_name`. Но здесь такой номер не пройдет: указав новое имя для переменной в атрибуте класса `contect_object_name`, мы зададим новое имя для переменной, в которой будет храниться рубрика, а не набор объявле-

ний (это связано с особенностями механизма множественного наследования Python). В результате чего получим чрезвычайно трудно диагностируемую ошибку.

Поэтому мы поступили по-другому: в том же переопределенном методе `get_context_data()` создали переменную `bbs` контекста шаблона и присвоили ей значение переменной `object_list`, в котором по умолчанию хранится набор записей, выводимых контроллером `ListView`.

И, наконец, в переопределенном методе `get_queryset()` мы просто возвращаем набор объявлений, связанных с найденной рубрикой и полученных через диспетчер обратной связи.

Вообще, на взгляд автора, лучше избегать контроллеров смешанной функциональности. Удобнее взять за основу контроллер-класс более низкого уровня и реализовать в нем всю нужную логику самостоятельно.

ГЛАВА 11

Шаблоны и статические файлы: базовые инструменты

Шаблон — это образец для формирования веб-страницы, которая будет отправлена посетителю в ответ на его запрос. Также шаблон может применяться для формирования текстовых документов — например, сообщений электронной почты.

Шаблонизатор — подсистема фреймворка, непосредственно формирующая окончательные веб-страницы (и текстовые документы), объединяя шаблоны и предоставленные контроллерами контексты шаблонов, содержащие все необходимые данные.

11.1. Настройки проекта, касающиеся шаблонов

В дополнение к настройкам проекта, описанным в *разд. 3.3*, мы рассмотрим сейчас группу настроек, непосредственно касающихся шаблонов и их обработки шаблонизатором.

Все настройки шаблонов записываются в параметре `TEMPLATES` модуля settings.py, что находится в пакете конфигурации. Параметру присваивается массив, каждый элемент которого является словарем, задающим параметры одного из доступных во фреймворке шаблонизаторов.

> ***Внимание!***
>
> Практически всегда в Django-сайтах используется только один шаблонизатор. Применение двух и более шаблонизаторов — очень специфическая ситуация, не рассматриваемая в этой книге.

В каждом таком словаре можно задать следующие элементы:

- `BACKEND` — путь к модулю шаблонизатора, записанный в виде строки.

  В составе Django поставляются два шаблонизатора:

  - `django.template.backends.django.DjangoTemplates` — стандартный шаблонизатор, применяемый в большинстве случаев;
  - `django.template.backends.jinja2.Jinja2` — шаблонизатор Jinja2;

- NAME — псевдоним для шаблонизатора. Если не указан, для обращения к шаблонизатору используется последняя часть пути к его модулю;
- DIRS — список путей к папкам, в которых шаблонизатор будет искать шаблоны (по умолчанию — «пустой» список);
- APP_DIRS — если True, шаблонизатор дополнительно будет искать шаблоны в папках templates, располагающихся в пакетах приложений. Если False, шаблонизатор станет искать шаблоны исключительно в папках, указанных в списке DIRS. Значение по умолчанию — False, однако во вновь созданном проекте устанавливается в True;
- OPTIONS — дополнительные параметры, поддерживаемые конкретным шаблонизатором. Также указываются в виде словаря, элементы которого задают отдельные параметры.

  Шаблонизатор django.template.backends.django.DjangoTemplates поддерживает такие параметры:

  - autoescape — если True, все недопустимые знаки HTML (двойная кавычка, знаки «меньше» и «больше») при их выводе будут преобразованы в соответствующие специальные символы (поведение по умолчанию). Если False, такое преобразование выполняться не будет;
  - string_if_invalid — строка, выводящаяся на экран в случае, если попытка доступа к переменной шаблона или вычисления выражения потерпела неудачу (значение по умолчанию — «пустая» строка);
  - file_charset — обозначение кодировки, в которой записан код шаблонов, в виде строки. Если параметр не указан, используется значение параметра проекта FILE_CHARSET (см. *разд. 3.3.1*);
  - context_processors — список имен модулей, реализующих обработчики контекста, что должны использоваться совместно с заданным шаблонизатором. Имена модулей должны быть заданы в виде строк.

    Обработчик контекста — это программный модуль, добавляющий в контекст шаблона какие-либо дополнительные переменные уже после его формирования контроллером. Список всех доступных в Django обработчиков контекста будет приведен позже;
  - debug — если True, будут выводиться развернутые сообщения об ошибках в коде шаблона, если False — совсем короткие сообщения. Если параметр не указан, будет использовано значение параметра проекта DEBUG (см. *разд. 3.3.1*);
  - loaders — список имен модулей, выполняющих загрузку шаблонов.

    Django — в зависимости от значений параметров DIRS и APP_DIRS — сам выбирает, какие загрузчики шаблонов использовать, и явно указывать список загрузчиков не требуется. На всякий случай, перечень доступных во фрейм-

ворке загрузчиков шаблонов и их описания приведены на странице **https://docs.djangoproject.com/en/2.1/ref/templates/api/#template-loaders**;

- `builtins` — список строк с путями к встраиваемым библиотекам тегов, которые должны использоваться с текущим шаблонизатором. Значение по умолчанию — «пустой» список.

  Библиотека тегов — это программный модуль Python, расширяющий набор доступных тегов шаблонизатора. *Встраиваемая* библиотека тегов загружается в память непосредственно при запуске проекта, и объявленные в ней дополнительные теги доступны к использованию в любой момент времени без каких бы то ни было дополнительных действий;

- `libraries` — перечень загружаемых библиотек шаблонов. Записывается в виде словаря, ключами элементов которого станут псевдонимы библиотек тегов, а значениями элементов — строковые пути к реализующим эти библиотеки модулям. Значение по умолчанию — «пустой» словарь.

  В отличие от встраиваемой библиотеки тегов, *загружаемая* библиотека перед использованием должна быть явно загружена с помощью тега шаблонизатора `load`, после которого указывается псевдоним библиотеки.

  Параметры `builtins` и `libraries` служат для указания исключительно сторонних библиотек тегов, поставляемых отдельно от Django. Библиотеки тегов, входящие в состав фреймворка, записывать туда не нужно.

Теперь рассмотрим обработчики контекста, доступные в Django:

- `django.template.context_processors.request` — добавляет в контекст шаблона переменную `request`, хранящую объект текущего запроса (в виде экземпляра класса `Request`);
- `django.template.context_processors.csrf` — добавляет в контекст шаблона переменную `csrf_token`, хранящую электронный жетон, что используется тегом шаблонизатора `csrf_token`;
- `django.contrib.auth.context_processors.auth` — добавляет в контекст шаблона переменные `user` и `perms`, хранящие, соответственно, сведения о текущем пользователе и его правах;
- `django.template.context_processors.static` — добавляет в контекст шаблона переменную `STATIC_URL`, хранящую значение одноименного параметра проекта (мы рассмотрим его в конце этой главы);
- `django.template.context_processors.media` — добавляет в контекст шаблона переменную `MEDIA_URL`, хранящую значение одноименного параметра проекта (мы рассмотрим его в *главе 19*);
- `django.contrib.messages.context_processors.messages` — добавляет в контекст шаблона переменные `messages` и `DEFAULT_MESSAGE_LEVELS`, хранящие, соответственно, список всплывающих сообщений и словарь, сопоставляющий строковые обозначения уровней сообщений с их числовыми кодами (о работе со всплывающими сообщениями будет рассказано в *главе 22*);

- `django.template.context_processors.tz` — добавляет в контекст шаблона переменную `TIME_ZONE`, хранящую наименование текущей временной зоны;
- `django.template.context_processors.debug` — добавляет в контекст шаблона переменные:
  - `debug` — хранит значение параметра проекта `DEBUG` (см. *разд. 3.3.1*);
  - `sql_queries` — сведения о запросах к базе данных, выполненных в процессе обработки запроса. Представляют собой список словарей, каждый из которых представляет один запрос. Элемент `sql` такого словаря хранит SQL-код запроса, а элемент `time` — время его выполнения.

  Может пригодиться при отладке сайта.

Листинг 11.1 показывает код, задающий настройки шаблонов по умолчанию, которые формируются при создании нового проекта.

Листинг 11.1. Настройки шаблонов по умолчанию

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

11.2. Вывод данных. Директивы

Для вывода данных в коде шаблона применяются *директивы*. Директива записывается в формате `{{ <переменная контекста шаблона> }}` и помещается в то место шаблона, в которое нужно поместить значение указанной *переменной*. Пример вставки значения из переменной `rubric`:

```
{{ rubric }}
```

В директивах, помимо простого извлечения значения из переменной шаблона, мы можем:

- обращаться к элементу последовательности по индексу, поставив его после переменной, хранящей последовательность, и отделив от нее точкой. Вот так можно обратиться к первому (с индексом 0) элементу списка `rubrics`:

```
{{ rubrics.0 }}
```

- обращаться к элементу словаря по ключу, поставив его после переменной и отделив от нее точкой. Вот пример извлечения значения элемента `kind` словаря `current_bb`:

```
{{ current_bb.kind }}
```

- разумеется, обращаться к атрибуту класса или экземпляра, применив привычную запись «с точкой»:

```
{{ bb.rubric.name }}
```

- вызывать методы, не принимающие параметров, — при этом в шаблон будет подставлен результат, возвращенный вызванным методом. Здесь применяется такой же синтаксис, как и для доступа к атрибутам, но круглые скобки не ставятся. Вот так можно вызвать метод `get_absolute_url()`:

```
{{ rubric.get_absolute_url }}
```

> ***Внимание!***
>
> Описанным здесь способом можно вызвать только методы, не принимающие параметров. Методы, принимающие параметры, вызвать нельзя, т. к. шаблонизатор Django не предоставляет средств для указания значений параметров при вызове.

В директивах, а также в рассматриваемых далее тегах и фильтрах, можно использовать обычные константы. Они записываются в том же виде, что и в Python-коде: строки должны быть взяты в одинарные или двойные кавычки, а числа с плавающей точкой должны включать дробную часть.

> ***Внимание!***
>
> Использовать в директивах выражения не допускается.

11.3. Теги шаблонизатора

Теги шаблонизатора управляют генерированием содержимого результирующего документа. Они записываются в формате:

```
{% <собственно запись тега со всеми необходимыми параметрами> %}
```

Как и HTML-теги, теги шаблонизатора бывают одинарными и парными.

Одинарный тег просто помещается в нужное место кода шаблона. Как правило, он вставляет в это место какое-либо значение, вычисляемое самим фреймворком. Пример одинарного тега, вставляющего в код шаблона электронный жетон, который используется подсистемой безопасности фреймворка:

```
{% csrf_token %}
```

Парный тег «охватывает» целый фрагмент кода шаблона. Он применяется для выполнения над этим фрагментом каких-либо действий. Например, парный тег `for . . . in` повторяет содержащийся в нем фрагмент столько раз, сколько элементов находится в указанной в этом теге последовательности:

```
{% for bb in bbs %}
<div>
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
    <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
```

Как видим, парный тег фактически состоит из двух тегов: *открывающего* и *закрывающего* (в приведенном только что примере это теги `for . . . in` и `endfor`). Закрывающий тег имеет то же имя, что и открывающий, но с добавленным впереди префиксом `end`. Фрагмент кода, находящийся внутри парного тега, носит название *содержимого*.

Теги, поддерживаемые шаблонизатором Django:

- `url` *<имя маршрута> <список значений параметров, разделенных пробелами>* [`as` *<переменная>*] — формирует интернет-адрес путем обратного разрешения. В *имени маршрута* при необходимости можно указать пространство имен. Параметры в *списке* могут быть как позиционными, так и именованными. Примеры:

  ```
  <a href="{% url 'bboard:detail' bb.pk %}">{{ bb.title }}</a>
  <a href="{% url 'bboard:by_rubric' rubric_id=bb.rubric.pk %}">
  {{ bb.rubric.name }}</a>
  ```

  По умолчанию тег вставляет сформированный интернет-адрес в шаблон. Но мы можем указать шаблонизатору сохранить интернет-адрес в *переменной*, записав ее после ключевого слова `as`. Пример:

  ```
  {% url 'bboard:detail' bb.pk as detail_url %}
  <a href="{{ detail_url }}">{{ bb.title }}</a>
  ```

- `for . . . in . . . endfor` — перебирает элементы указанной последовательности (словаря), заносит очередной элемент в переменную (переменные) и помещает в шаблон свое содержимое. В результате содержимое тега будет помещено в шаблон столько раз, сколько элементов присутствует в последовательности (словаре). Записанную в теге переменную (переменные) можно использовать в содержимом для получения значения элемента. Аналог цикла `for . . . in` языка Python. Формат записи тега:

  ```
  {% for <переменные> in <последовательность или словарь> %}
  <содержимое тега>
  [{% empty %}
  <содержимое, выводящееся, если последовательность или словарь
  не имеет элементов>]
  {% endfor %}
  ```

  Пример:

  ```
  {% for bb in bbs %}
  <div>
      <h2>{{ bb.title }}</h2>
  ```

```
    <p>{{ bb.content }}</p>
    <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
```

В *содержимом* можно использовать следующий набор переменных, создаваемых самим тегом:

- `forloop.counter` — номер текущей итерации цикла (нумерация начинается с 1);
- `forloop.counter0` — номер текущей итерации цикла (нумерация начинается с 0);
- `forloop.revcounter` — количество оставшихся итераций цикла (нумерация начинается с 1);
- `forloop.revcounter0` — количество оставшихся итераций цикла (нумерация начинается с 0);
- `forloop.first` — `True`, если это первая итерация цикла, `False`, если не первая;
- `forloop.last` — `True`, если это последняя итерация цикла, `False`, если не последняя;
- `forloop.parentloop` — у вложенного цикла указывает на цикл, в который он вложен. Пример использования: `forloop.parentloop.counter` (получение номера текущей итерации «внешнего» цикла).

Пример:

```
{% for bb in bbs %}
<div>
  <p>№№ {{ forloop.counter }}</p>
  . . .
</div>
{% endfor %}
```

- `if . . . elif . . . else . . . endif` — аналог условного выражения Python. Формат записи тега:

```
{% if <условие 1> %}
<содержимое 1>
[{% elif <условие 2> %}
<содержимое 2>
. . .
{% elif <условие n> %}
<содержимое n>]
[{% else %}
<содержимое else>]
{% endif %}
```

В условиях можно использовать операторы сравнения `==`, `!=`, `<`, `>`, `<=`, `>=`, `in`, `not in`, `is` и `is not`, логические операторы `and`, `or` и `not`.

Пример:

```
{% if bbs %}
<h2>Список объявлений</h2>
<% else %}
<p>Объявлений нет</p>
{% endif %}
```

- `ifchanged . . . endifchanged` — **применяется в циклах и выводит содержимое, только если оно изменилось после предыдущей итерации цикла. Форматы использования:**

```
{% ifchanged %} <содержимое> {% endifchanged %}

{% ifchanged <список значений, разделенных пробелами> %} <содержимое>
{% endifchanged %}
```

Первый формат выводит *содержимое*, **если оно изменилось после предыдущей итерации цикла. Второй формат выводит** *содержимое*, **если изменилось одно из значений, приведенных в** *списке*.

Примеры:

```
{% for rubric in rubrics %}
{% ifchanged %}{{ rubric.parent.name }}{% endifchanged %}
. . .
{% endfor %}
```

Выводим название рубрики, в которую вложена текущая рубрика, только если это название изменилось (т. е. если текущая рубрика вложена в другую рубрику, нежели предыдущая).

```
{% for rubric in rubrics %}
{% ifchanged rubric.parent %}
{{ rubric.parent.name }}
{% endifchanged %}
. . .
{% endfor %}
```

То же самое, только с использованием второго формата записи тега;

- `cycle <список значений, разделенных пробелами> [as <переменная>]` — **последовательно помещает в шаблон очередное значение из указанного списка. По достижении конца списка перебор начинается сначала. Количество значений в списке не ограничено. Пример:**

```
{% for bb in bbs %}
<div class="{% cycle 'bb1' 'bb2' 'bb3' %}">
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
    <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
```

Здесь при каждом проходе цикла `for . . . in` к блоку будут последовательно привязываться стилевые классы `b1`, `b2`, `b3`, потом снова `b1`, `b2` и т. д.

Текущее значение последовательности может быть занесено в переменную, указанную после ключевого слова `as`. Эту переменную можно использовать, чтобы вставить это значение в другом месте шаблона:

```
{% for bb in bbs %}
{% cycle 'bb1' 'bb2' 'bb3' as currentclass %}
<div class="{{ currentclass }}">
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
    <p class="{{ currentclass }}">{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
```

- `resetcycle [<переменная>]` — сбрасывает тег `cycle`, после чего тот начинает перебирать указанные в нем значения с начала. По умолчанию сбрасывает последний тег `cycle`, записанный в шаблоне. Если же нужно сбросить конкретный тег, следует указать *переменную*, записанную в нужном теге `cycle` после ключевого слова `as`;
- `firstof <список значений, разделенных пробелами>` — помещает в шаблон первое из приведенных в списке значений, не равное `False` (т. е. не «пустое»):

```
{% firstof bb.phone bb.email 'На деревню дедушке' %}
```

Если поле `phone` объявления `bb` не «пусто», его значение будет помещено в шаблон. Если же оно «пусто», аналогичная проверка будет произведена с полем `email`. Если же и оно «пусто», в шаблоне окажется строка `"На деревню дедушке"`;

- `with . . . endwith` — заносит какие-либо значения в переменные для дальнейшего использования. Полученные переменные становятся доступными внутри содержимого тега. Формат записи:

```
{% with <набор операций присваивания, разделенных пробелами> %}
<содержимое>
{% endwith %}
```

Операции присваивания записываются так же, как и в Python.

Этот тег может использоваться для временного сохранения в переменной значения, получаемого в результате вычислений (например, обращением к методу или свойству класса). В результате потом мы сможем извлечь готовое значение из переменной, а не производить для его получения те же самые вычисления повторно.

Пример:

```
{% with bb_count=bbs.count %}
{% if bb_count > 0 %}
<p>Всего {{ bb_count }} объявлений.</p>
{% endwith %}
```

- `regroup` *<последовательность>* `by` *<ключ элемента или атрибут>* `as` *<переменная>* — выполняет группировку указанной *последовательности* словарей или объектов по значению элемента с заданным *ключом* или атрибута с заданным *именем* и помещает созданный таким образом новый список в *переменную*. Каждый элемент созданного таким образом списка будет представлять собой объект типа `namedtuple` с элементами:

  - `grouper` — значение элемента или атрибута, по которому выполнялась группировка;
  - `list` — список словарей или объектов, относящихся к группе.

  Пример группировки объявлений по рубрике:

```
{% regroup bbs by rubric.name as grouped_bbs %}
{% for rubric_name, gbbs in grouped_bbs %}
    <h3>{{ rubric_name }}</h3>
    {% for bb in gbbs %}
    <div>
        <h2>{{ bb.title }}</h2>
        <p>{{ bb.content }}</p>
        <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
    </div>
    {% endfor %}
{% endfor %}
```

- `now` *<формат>* — вставляет в шаблон текущее значение даты и времени, оформленное согласно заданному *формату* (форматирование значений даты и времени описано далее — при рассмотрении фильтра `date`):

```
{% now 'SHORT_DATETIME_FORMAT' %}
```

- `filter` *<фильтр или фильтры>* `. . . endfilter` — применяет к содержимому указанные *фильтр или фильтры*:

```
{% filter force_escape|upper %}
<p>Текст тега filter</p>
{% endfilter %}
```

  Здесь к абзацу, находящемуся внутри тега `filter`, будут применены фильтры `force_escape` и `upper`;

- `csrf_token` — вставляет в шаблон электронный жетон, используемый подсистемой обеспечения безопасности Django. Применяется исключительно в веб-формах (в тегах `<form>`);

- `autoescape on|off . . . endautoescape` — управляет автоматическим преобразованием недопустимых знаков HTML (двойной кавычки, знаков «меньше» и «больше») в соответствующие специальные символы при их выводе. Значение `on` указывает выполнять автоматическое преобразование, значение `off` — не выполнять;

- `spaceless . . . endspaceless` — удаляет пробельные символы (в число которых входят пробел, табуляция, возврат каретки и перевод строки) между тегами в своем содержимом:

```
{% spaceless %}
<h3>
    <em>Последние объявления</em>
</h3>
{% endspaceless %}
```

В результате в шаблоне окажется код:

```
<h3><em>Последние объявления</em></h3>
```

- `templatetag` *<обозначение последовательности символов>* — вставляет в шаблон последовательность символов, которую иначе поместить туда не получится (примеры: `{{, }}`, `{%, %}`). Поддерживаются следующие *обозначения последовательностей символов*:
  - `openblock: {%;`
  - `closeblock: %};`
  - `openvariable: {{;`
  - `closevariable: }};`
  - `openbrace: {;`
  - `closebrace: };`
  - `opencomment: {#;`
  - `closecomment: #};`

- `verbatim . . . endverbatin` — вставляет содержимое как есть, не обрабатывая записанные в нем директивы, теги и фильтры шаблонизатора:

```
{% verbatim %}
<p>Текущие дата и время помещаются на страницу тегом {% now %}.</p>
{% endverbatim %}
```

- `load` *<список псевдонимов библиотек тегов, разделенных пробелами>* — загружает библиотеки тегов с указанными *псевдонимами*:

```
{% load static %}
```

- `widthratio` *<текущее значение> <максимальное значение> <максимальная ширина>* — применяется для создания диаграмм. *Текущее значение* делится на *максимальное значение*, после чего получившееся частное умножается на *максимальную ширину*, и результат всего этого вставляется в шаблон. Вот пример того, как можно использовать этот довольно странный тег:

```
<img src="bar.png"
style="height:10px;width:{% widthratio current_value 200 100 %}px">
```

- `comment [`*<заголовок>*`] . . . encomment` — помещает в код шаблона комментарий, не обрабатываемый шаблонизатором. Возможно указать для комментария необязательный *заголовок*, записав его в качестве параметров тега. Пример:

```
{% comment 'Не забыть доделать!' %}
<p>Здесь будет список объявлений</p>
{% endcomment %}
```

Если комментарий занимает всего одну строку или часть ее, его можно создать посредством парного тега {# . . . #}:

```
{# <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p> #}
```

- debug — выводит на странице разнообразную отладочную информацию, включая содержимое контекста шаблона и список отладочных модулей. Эта информация весьма объемна и не очень удобна на практике.

11.4. Фильтры

Фильтры шаблонизатора выполняют определенные преобразования значения перед его выводом.

Фильтр записывается непосредственно в директиве, после имени переменной, из которой извлекается текущее значение для обработки, и отделяется от нее символом вертикальной черты (|). Пример:

```
{{ bb.published|date:"d.m.Y H:i:s" }}
```

Фильтры можно объединять, в результате чего они будут обрабатываться последовательно, слева направо:

```
{{ bb.content|lower|default:'--описания нет--' }}
```

Вот список поддерживаемых шаблонизатором Django фильтров:

- date:*<формат даты и времени>* — форматирует выводимое значение даты и времени согласно заданному *формату*. В строке *формата* можно использовать следующие специальные символы:
  - j — число без начального нуля;
  - d — число с начальным нулем;
  - m — номер месяца с начальным нулем;
  - n — номер месяца без начального нуля;
  - F и N — полное название месяца в именительном падеже с большой буквы;
  - E — полное название месяца в родительном падеже и в нижнем регистре;
  - M — сокращенное название месяца с большой буквы;
  - b — сокращенное название месяца в нижнем регистре;
  - Y и o — год из четырех цифр;
  - y — год из двух цифр;
  - L — True, если это високосный год, False, если обычный;
  - w — номер дня недели от 0 (воскресенье) до 6 (суббота);
  - l — сокращенное название дня недели с большой буквы;
  - D — полное название дня недели в именительном падеже с большой буквы;

- `G` — часы в 24-часовом формате без начального нуля;
- `H` — часы в 24-часовом формате с начальным нулем;
- `g` — часы в 12-часовом формате без начального нуля;
- `h` — часы в 12-часовом формате с начальным нулем;
- `i` — минуты;
- `s` — секунды с начальным нулем;
- `u` — микросекунды;
- `a` — обозначение половины суток в нижнем регистре (`"д.п."` или `"п.п."`);
- `A` — обозначение половины суток в верхнем регистре (`"ДП"` или `"ПП"`);
- `I` — 1, если сейчас летнее время, 0, если зимнее;
- `P` — часы в 12-часовом формате и минуты. Если минуты равны 0, они не указываются. Вместо 00:00 выводится строка `"полночь"`, а вместо 12:00 — `"полдень"`;
- `f` — часы в 12-часовом формате и минуты. Если минуты равны 0, они не указываются;
- `t` — количество дней в текущем месяце;
- `z` — порядковый номер дня в году;
- `W` — порядковый номер недели (неделя начинается с понедельника);
- `e` — название временной зоны;
- `O` — разница между текущим и гринвичским временем в часах;
- `Z` — разница между текущим и гринвичским временем в секундах;
- `c` — дата и время в формате ISO 8601;
- `r` — дата и время в формате RFC 5322;
- `U` — время в формате UNIX (выражается как количество секунд, прошедших с полуночи 1 января 1970 года);
- `T` — название временной зоны, установленной в настройках компьютера.

Пример:

```
{{ bb.published|date:'d.m.Y H:i:s' }}
```

Также можно использовать следующие обозначения встроенных в Django форматов:

- `DATE_FORMAT` — развернутый формат даты;
- `DATETIME_FORMAT` — развернутый формат даты и времени;
- `SHORT_DATE_FORMAT` — сокращенный формат даты;
- `SHORT_DATETIME_FORMAT` — сокращенный формат даты и времени.

Пример:

```
{{ bb.published|date:'DATETIME_FORMAT' }}
```

- `time[:<формат времени>]` — форматирует выводимое значение времени согласно заданному *формату*. При написании *формата* применяются те же специальные символы, что и в случае фильтра `date`. Пример:

```
{{ bb.published|time:'H:i' }}
```

Для вывода времени с применением формата по умолчанию нужно использовать обозначение `TIME_FORMAT`:

```
{{ bb.published|time:'TIME_FORMAT' }}
```

или вообще не указывать формат:

```
{{ bb.published|time }}
```

- `timesince[:<значение для сравнения>]` — выводит промежуток времени, разделяющий выводимое значение даты и времени и заданное *значение для сравнения* (если таковое не указано, в его качестве принимается сегодняшняя дата). Здесь предполагается, что *значение для сравнения* находится по отношению к выводимому значению в будущем. Полученный промежуток времени может иметь вид, например, `"3 недели, 6 дней"`, `"6 дней 23 часа"` и т. п.

Если выяснится, что выводимое значение больше *значения для сравнения*, будет выведена строка: `"0 минут"`;

- `timeuntil[:<значение для сравнения>]` — то же самое, что и `timesince`, но предполагается, что *значение для сравнения* находится по отношению к выводимому значению в прошлом;

- `yesno[:<строка образцов>]` — преобразует значения `True`, `False` и, возможно, `None` в слова `"да"`, `"нет"` и `"может быть"`.

Можно указать свои слова для преобразования, записав их в *строке образцов* вида *<строка для* `True`*>,<строка для* `False`*>*[*,<строка для* `None`*>*]. Если *строка для* `None` не указана, вместо нее будет выводиться *строка для* `False` (поскольку `None` будет неявно преобразовываться в `False`).

Примеры:

```
{{ True|yesno }}, {{ False|yesno }}, {{ None|yesno }}
{# Результат: да, нет, может быть #}

{{ True|yesno:'так точно,никак нет,дело темное' }},
{{ False|yesno:'так точно,никак нет,дело темное' }},
{{ None|yesno:'так точно,никак нет,дело темное' }}
{# Результат: так точно, никак нет, дело темное #}

{{ True|yesno:'да,нет' }}, {{ False|yesno:'да,нет' }},
{{ None|yesno:'да,нет' }}
{# Результат: да, нет, нет #}
```

- `default:<величина>` — если выводимое значение равно `False`, возвращает указанную *величину*:

  ```
  {{ bb.price|default:'У товара нет цены' }}
  ```

  Если поле `price` товара `bb` хранит «пустое» значение (цена не указана или равна 0 — оба значения трактуются как `False`), будет выведена строка `"У товара нет цены"`;

- `default_if_none:<величина>` — то же самое, что и `default`, но возвращает *величину* только в том случае, если выводимое значение равно `None`;

- `upper` — переводит все буквы выводимого значения в верхний регистр;

- `lower` — переводит все буквы выводимого значения в нижний регистр;

- `capfirst` — переводит первую букву выводимого значения в верхний регистр;

- `title` — переводит первую букву каждого слова в выводимом значении в верхний регистр;

- `truncatechars:<длина>` — обрезает выводимое значение до указанной *длины*, помещая в конец символ многоточия (...);

- `truncatechars_html:<длина>` — то же самое, что и `truncatechars`, но сохраняет все HTML-теги, которые встретятся в выводимом значении;

- `truncatewords:<количество слов>` — обрезает выводимое значение, оставляя в нем указанное *количество слов*. В конце обрезанного значения помещается символ многоточия (...);

- `truncatewords_html:<количество слов>` — то же самое, что и `truncatewords`, но сохраняет все HTML-теги, которые встретятся в выводимом значении;

- `wordwrap:<величина>` — выполняет перенос выводимого строкового значения по словам таким образом, чтобы длина каждой получившейся в результате строки не превышала указанную *величину*;

- `cut:<удаляемая подстрока>` — удаляет из выводимого значения все вхождения заданной *подстроки*:

  ```
  {{ 'Python'|cut:'t' }}          {# Результат: 'Pyhon' #}
  {{ 'Python'|cut:'th' }}         {# Результат: 'Pyon' #}
  ```

- `slugify` — преобразует выводимое строковое значение в слаг;

- `stringformat:<формат>` — форматирует выводимое значение согласно указанному *формату*. При написании *формата* применяются специальные символы, поддерживаемые оператором `%` языка Python (см. страницу **https://docs.python.org/3/library/stdtypes.html#old-string-formatting**);

- `floatformat[:<количество знаков после запятой>]` — форматирует выводимое значение как вещественное число, округляя его до заданного *количества знаков после запятой*. Если указать положительное значение *количества знаков*, эти знаки будут выводиться всегда, если отрицательное — только при необходимости. Значение *количество знаков* по умолчанию: `-1`. Примеры:

```
{{ 34.23234|floatformat }}          {# Результат: 34.2 #}
{{ 34.00000|floatformat }}          {# Результат: 34 #}
{{ 34.26000|floatformat }}          {# Результат: 34.3 #}
{{ 34.23234|floatformat:3 }}        {# Результат: 34.232 #}
{{ 34.00000|floatformat:3 }}        {# Результат: 34.000 #}
{{ 34.26000|floatformat:3 }}        {# Результат: 34.260 #}
{{ 34.23234|floatformat:-3 }}       {# Результат: 34.232 #}
{{ 34.00000|floatformat:-3 }}       {# Результат: 34 #}
{{ 34.26000|floatformat:-3 }}       {# Результат: 34.260 #}
```

- `filesizeformat` — выводит числовую величину как размер файла (примеры: `"100 байт"`, `"8,8 КБ"`, `"47,7 МБ"`);
- `add:<величина>` — прибавляет к выводимому значению указанную *величину*. Можно складывать числа, строки и последовательности;
- `divisibleby:<делитель>` — возвращает `True`, если выводимое значение делится на указанный *делитель* без остатка, и `False` — в противном случае;
- `wordcount` — возвращает количество слов в выводимом строковом значении;
- `length` — возвращает количество элементов в выводимой последовательности. Также работает со строками;
- `length_is:<величина>` — возвращает `True`, если длина выводимой последовательности равна указанной *величине*, и `False` — в противном случае;
- `first` — возвращает первый элемент выводимой последовательности;
- `last` — возвращает последний элемент выводимой последовательности;
- `random` — возвращает случайный элемент выводимой последовательности;
- `slice:<оператор взятия среза Python>` — возвращает срез выводимой последовательности. *Оператор взятия среза* записывается без квадратных скобок. Пример:

  ```
  {{ rubric_names|slice:'1:3' }}
  ```

- `join:<разделитель>` — возвращает строку, составленную из элементов выводимой последовательности, которые отделяются друг от друга *разделителем*;
- `make_list` — преобразует выводимое значение в список. Элементами списка станут символы, присутствующие в значении;
- `dictsort:<ключ элемента>` — если выводимое значение представляет собой последовательность словарей, сортирует список по значениям элементов с указанным *ключом*. Сортировка выполняется по возрастанию значений.

  Пример вывода списка объявлений, отсортированного по цене:

  ```
  {% for bb in bbs|dictsort:'price' %}
      . . .
  {% endfor %}
  ```

!

Таким же образом можно сортировать список списков или кортежей, только вместо ключа нужно указать индекс элемента вложенного списка (кортежа), по значениям которого следует выполнить сортировку. Пример:

```
{% for el in list_of_lists|dictsort:1 %}
    . . .
{% endfor %}
```

- `dictsortreversed:<ключ элемента>` — то же самое, что `dictsort`, только сортировка выполняется по убыванию значений;
- `unordered_list` — используется, если выводимым значением является список или кортеж, чьими элементами также являются списки или кортежа. Возвращает HTML-код, создающий набор вложенных друг в друга неупорядоченных списков, без «внешних» тегов `<ul>` и `</ul>`. Пример:

```
ulist = [
         'PHP',
         ['Python', 'Django'],
         ['JavaScript', 'Node.js', 'Express']
]
. . .
<ul>
    {{ ulist:unordered_list }}
</ul>
{# Результат: #}
<ul>
    <li>PHP
        <ul>
            <li>Python</li>
            <li>Django</li>
        </ul>
        <ul>
            <li>JavaScript</li>
            <li>None.js</li>
            <li>Express</li>
        </ul>
    </li>
</ul>
```

- `linebreaksbr` — заменяет в выводимом строковом значении все символы перевода строки на HTML-теги `<br>`;
- `linebreaks` — разбивает выводимое строковое значение на отдельные строки. Если в значении встретится одинарный символ перевода строки, он будет заменен HTML-тегом `<br>`. Если встретится двойной символ перевода строки, разделяемые им части значения будут заключены в теги `<p>`;
- `urlize` — преобразует все встретившиеся в выводимом значении интернет-адреса и адреса электронной почты в гиперссылки. Каждая такая гиперссылка

создается HTML-тегом <a>. В тег, создающий обычную гиперссылку, добавляется атрибут rel со значением nofollow.

Нужно иметь в виду, что фильтр urlize нормально работает только с обычным текстом. При попытке обработать им HTML-код результат окажется непредсказуемым;

- urlizetrunc:*<длина>* — то же самое, что и urlize, но дополнительно обрезает текст гиперссылок до указанной *длины*, помещая в его конец символ многоточия (...);
- safe — подавляет у выводимого значения автоматическое преобразование недопустимых знаков HTML в соответствующие специальные символы;
- safeseq — подавляет у всех элементов выводимой последовательности автоматическое преобразование недопустимых знаков HTML в соответствующие специальные символы. Обычно применяется совместно с другими фильтрами. Пример:

```
{{ rubric_names|safeseq|join:", " }}
```

- escape — преобразует недопустимые знаки HTML в соответствующие специальные символы. Обычно применяется в содержимом парного тега autoescape с отключенным автоматическим преобразованием недопустимых знаков. Пример:

```
{% autoescape off %}
    {{ blog.content|escape }}
{% endautoescape %}
```

- force_escape — то же самое, что и escape, но выполняет преобразование принудительно. Может быть полезен, если требуется провести преобразование у результата, возвращенного другим фильтром;
- escapejs — преобразует выводимое значение таким образом, чтобы его можно было использовать как строковую константу JavaScript;
- striptags — удаляет из выводимого строкового значения все HTML-теги;
- urlencode — кодирует выводимое значение таким образом, чтобы его можно было включить в состав интернет-адреса (например, передать с GET-параметром);
- iriencode — кодирует выводимый интернационализированный идентификатор ресурса (IRI) таким образом, чтобы его можно было включить в состав интернет-адреса (например, передать с GET-параметром);
- addslashes — добавляет символы обратного слеша перед одинарными и двойными кавычками;
- ljust:*<ширина пространства в символах>* — помещает выводимое значение в левой части пространства указанной *ширины*:

```
{{ 'Python'|ljust:20 }}
```

Результатом станет строка: "Python ";

- `center:`*<ширина пространства в символах>* — помещает выводимое значение в середине пространства указанной *ширины*.

  ```
  {{ 'Python'|center:20 }}
  ```

  Результатом станет строка: " Python ";

- `rjust:`*<ширина пространства в символах>* — помещает выводимое значение в правой части пространства указанной *ширины*.

  ```
  {{ 'Python'|rjust:20 }}
  ```

  Результатом станет строка: " Python";

- `get_digit:`*<позиция цифры>* — возвращает цифру, присутствующую в выводимом числовом значении по указанной *позиции*, отсчитываемой справа. Если текущее значение не является числом, или если указанная *позиция* меньше 1 или больше общего количества цифр в числе, возвращается значение *позиции*. Пример:

  ```
  {{ 123456789|get_digit:4 }}          {# Результат: 6 #}
  ```

- `linenumbers` — выводит строковое значение, разбитое на отдельные строки посредством символов перевода строки, с номерами строк, поставленными слева.

11.5. Наследование шаблонов

Аналогично наследованию классов, практикуемому в Python, Django предлагает нам механизм наследования шаблонов.

В Python базовый класс определяет функциональность, которая должна быть общей для всех наследующих его производных классов. Точно так же и в Django: базовый шаблон определяет все элементы, которые должны быть общими для всех наследующих его производных шаблонов.

Помимо этого, базовый шаблон определяет набор *блоков*. Блок представляет собой место в коде базового шаблона, куда производный шаблон поместит свое уникальное содержимое. Таких блоков может быть сколько угодно.

Блоки в базовом шаблоне объявляются с применением парного тега шаблонизатора `block` *<имя блока>* `. . . endblock`. *Имя блока* должно быть уникальным в пределах базового шаблона. Пример:

```
<title>{% block title %}Главная{% endblock %} - Доска объявлений</title>
. . .
{% block content %}
    <p>Содержимое базового шаблона</p>
{% endblock %}
```

Мы можем записать имя блока и в закрывающем теге `endblock` — чтобы нам самим в дальнейшем было проще разобраться в коде шаблона:

```
{% block content %}
    <p>Содержимое базового шаблона</p>
{% endblock content %}
```

В коде производного шаблона необходимо явно указать, что он является производным от определенного базового шаблона. Для такого указания применяется тег шаблонизатора `extends` *<путь к базовому шаблону>*.

> **Внимание!**
>
> Тег `extends` должен находиться в самом начале кода шаблона, на отдельной строке.

Пример (подразумевается, что базовый шаблон хранится в файле layout\basic.html):

```
{% extends "layout/basic.html" %}
```

После чего в производном шаблоне точно так же объявляются блоки, но теперь уже в них заносится содержимое, уникальное для этого производного шаблона. Пример:

```
{% block content %}
    <p>Содержимое производного шаблона</p>
{% endblock content %}
```

Если в производном шаблоне не указать какой-либо из блоков, объявленных в базовом шаблоне, этот блок все равно будет выведен на экран, но получит изначальное, заданное в базовом шаблоне содержимое. Так, поскольку в нашем произвольном шаблоне мы не указали блок `title`, он получит изначальное содержимое — строку `"Главная"`.

Если в блоке производного шаблона в составе заданного там же уникального содержимого нужно вывести изначальное, последнее можно извлечь из переменной `block.super`, создаваемой в контексте шаблона самим Django. Пример:

```
{% block content %}
    <p>Содержимое производного шаблона 1</p>
    {{ block.super }}
    <p>Содержимое производного шаблона 2</p>
{% endblock content %}
```

В результате на экран будут выведены три абзаца:

```
Содержимое производного шаблона 1
Содержимое базового шаблона
Содержимое производного шаблона 2
```

Кстати, мы можем создать в производном шаблоне другие блоки и сделать его базовым для других производных шаблонов. Django такое тоже позволяет.

Конкретные примеры использования наследования шаблонов можно увидеть в листингах *разд. 2.7*.

11.6. Обработка статических файлов

В терминологии Django *статическими* называются файлы, с одной стороны, не изменяющиеся в процессе работы сайта, а с другой, не обрабатываемые программно, а отправляемые клиенту как есть. К таким файлам можно отнести таблицы сти-

лей, графические изображения, аудио- и видеоролики, помещенные на страницах, файлы статических веб-страниц и т. п.

Обработку статических файлов во фреймворке выполняет специальная подсистема, реализованная во встроенном приложении `django.contrib.staticfiles`. Оно включается в список приложений, зарегистрированных в проекте (см. *разд. 3.3.3*), уже при создании нового проекта, и, если сайт включает статические файлы, удалять его оттуда нельзя.

11.6.1. Настройка подсистемы статических файлов

Подсистемой статических файлов управляет ряд настроек, записываемых в модуле settings.py пакета конфигурации:

- `STATIC_URL` — префикс, добавляемый к интернет-адресу статического файла. Встретив в начале интернет-адреса этот префикс, Django «поймет», что это статический файл, и его нужно передать для обработки подсистеме статических файлов. Значение по умолчанию — `None`, но при создании нового проекта оно устанавливается в `"/static/"`;
- `STATIC_ROOT` — путь к основной папке, в которой хранятся все статические файлы (значение по умолчанию — `None`).

  В этой же папке будут собираться все статические файлы, если отдать команду `collectstatic` утилиты manage.py;
- `STATICFILES_DIRS` — список путей к дополнительным папкам, в которых хранятся статические файлы. Каждый путь может быть задан в двух форматах:
  - как строка с путем. Пример:

    ```
    STATICFILES_DIRS = [
        'c:/site/static',
        'c:/work/others/images',
    ]
    ```

  - как кортеж из двух элементов: префикса пути и самого пути. Чтобы сослаться на файл, хранящийся по определенному пути, нужно предварить интернет-адрес файла заданным для этого пути префиксом. Пример:

    ```
    STATICFILES_DIRS = [
        ('main', 'c:/site/static'),
        ('images', 'c:/work/others/imgs'),
    ]
    ```

    Теперь, чтобы сослаться на файл img.png, хранящийся в папке c:\work\others\imgs, следует использовать такой тег шаблонизатора:

    ```
    {% static 'images/img.png' %}
    ```

- `STATICFILES_FINDERS` — список имен классов, реализующих подсистемы поиска статических файлов. По умолчанию включает два класса, объявленные в модуле `django.contrib.staticfiles.finders`:

- `FileSystemFinder` — ищет статические файлы в папках, заданных параметрами `STATIC_ROOT` и `STATICFILES_DIRS`;
- `AppDirectoriesFinder` — ищет статические файлы в папках static, находящихся в пакетах приложений.

Если статические файлы хранятся в каком-то определенном местоположении (только в папках, заданных параметрами `STATIC_ROOT` и `STATICFILES_DIRS`, или только в папках static, находящихся в пакетах приложений), можно указать в параметре `STATICFILES_FINDERS` только один класс — соответствующий случаю. Это несколько уменьшит потребление системных ресурсов;

- `STATICFILES_STORAGE` — имя класса, реализующего хранилище статических файлов. По умолчанию используется хранилище `StaticFilesStorage` из модуля `django.contrib.staticfiles.storage`.

11.6.2. Обслуживание статических файлов

Встроенный в Django отладочный веб-сервер обслуживает статические файлы самостоятельно. Нам даже не придется указывать в настройках проекта никаких специальных параметров, чтобы обеспечить это.

Однако все сказанное ранее касалось только отладочного режима сайта. Если сайт находится в эксплуатационном режиме, нам придется позаботиться об обслуживании статических файлов веб-сервером самостоятельно. Как это сделать, будет рассказано в *главе 29*.

11.6.3. Формирование интернет-адресов статических файлов

Формировать интернет-адреса статических файлов в коде шаблонов мы можем посредством трех разных программных механизмов:

- `static` — тег шаблонизатора, вставляющий в шаблон полностью сформированный интернет-адрес статического файла. Формат записи:

  `static` *<путь к статическому файлу>* [`as` *<переменная>*]

  Путь к статическому файлу записывается в виде строки и отсчитывается от папки, чей путь записан в параметрах `STATIC_ROOT` и `STATICFILES_DIRS`, или папки static пакета приложения.

  Тег реализован в библиотеке тегов с псевдонимом `static`, поэтому требует, чтобы она была предварительно загружена тегом `load`.

  Пример:

```
{% load static %}
. . .
<link . . . href="{% static 'bboard/style.css' %}">
```

По умолчанию сформированный интернет-адрес непосредственно вставляется в шаблон. Но мы можем указать шаблонизатору сохранить интернет-адрес в *переменной*, записав ее после ключевого слова `as`. Пример:

```
{% static 'bboard/style.css' as css_url %}
<link . . . href="{{ css_url }}">
```

- `get_static_prefix` — тег шаблонизатора, который вставляет в шаблон префикс, заданный параметром `STATIC_URL`. Также реализован в библиотеке тегов `static`. Пример:

```
{% load static %}
. . .
<link . . . href="{% get_static_prefix %}bboard/style.css">
```

- `django.template.context_processors.static` — обработчик контекста, добавляющий в контекст шаблона переменную `STATIC_URL`, которая хранит префикс из одноименного параметра проекта. Поскольку этот обработчик по умолчанию не включен в список активных (элемент `context_processors` параметра `OPTIONS` — см. *разд. 11.1*), его нужно добавить туда:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            'context_processors': [
                . . .
                'django.template.context_processors.static',
            ],
        },
    },
]
```

После чего можно обращаться к созданной этим обработчиком в контексте шаблона переменной `STATIC_URL`:

```
<link . . . href="{{ STATIC_URL }}bboard/style.css">
```

ГЛАВА 12

Пагинатор

При выводе на веб-страницы больших списков их практически всегда разбивают на отдельные части, включающие не более определенного количества позиций. Это позволяет сделать страницы относительно небольшими и быстро загружающимися. Разумеется, на страницах предусматривается набор гиперссылок для перехода на следующую и предыдущую части списка, а нередко и на конкретную часть — по ее порядковому номеру.

Разбиением списков на части занимается программный механизм, носящий название *пагинатора*. Применять пагинатор явно имеет смысл только в контроллерах-функциях (см. *главу 9*) и в контроллерах-классах самого низкого уровня (см. *главу 10*). Более высокоуровневые контроллеры-класса (наподобие `ListView`, описанного в *разд. 10.4.3*) используют пагинатор неявно.

12.1. Класс *Paginator*: сам пагинатор. Создание пагинатора

Класс `Paginator` из модуля `django.core.paginator` представляет сам пагинатор. Экземпляр именно этого класса нам необходимо создать, чтобы реализовать пагинацию. Формат его конструктора:

```
Paginator(<набор записей>, <количество записей в части>[, orphans=0][,
                          allow_empty_first_page=True])
```

Первый параметр задает набор записей, который должен разбиваться на части, второй — количество записей в части.

Необязательный параметр `orphans` указывает минимальное количество записей, что могут присутствовать в последней части пагинатора. Если последняя часть пагинатора содержит меньше записей, все эти записи будут выведены в составе предыдущей части. Если задать значение 0, в последней части может присутствовать сколько угодно записей (поведение по умолчанию).

Необязательный параметр `allow_empty_first_page` указывает, будет ли создаваться «пустая» часть, если набор не содержит записей. в случае, Значение `True` разрешает создание «пустой» части пагинатора (поведение по умолчанию). Значение `False`, напротив, предписывает при попытке извлечения «пустой» части возбудить исключение `EmptyPage` из модуля `django.core.paginator`.

Класс `Paginator` поддерживает три атрибута:

- `count` — общее количество записей во всех частях пагинатора;
- `num_pages` — количество частей, на которые разбит набор записей;
- `page_range` — итератор, последовательно возвращающий номера всех частей пагинатора. Нумерация частей начинается с 1.

Однако для нас полезнее будут два метода этого класса:

- `get_page(<номер части>)` — возвращает экземпляр класса `Page` (он будет описан далее), представляющий часть с указанным *номером*. Нумерация частей начинается с 1.

  Если *номер части* не является числовой величиной, возвращается первая страница. Если *номер части* является отрицательным числом или превышает общее количество частей в пагинаторе, возвращается последняя часть.

  Если *номер части* не является целым числом, возбуждается исключение `PageNotAnInteger` из модуля `django.core.paginator`. Если получаемая часть «пуста», а при создании экземпляра класса `Paginator` параметру `allow_empty_first_page` было дано значение `False`, возбуждается исключение `EmptyPage`;

- `page(<номер части>)` — то же самое, что и `get_page()`, но в любом случае, если *номер части* не является целым числом, является числом отрицательным или большим, чем общее количество частей в пагинаторе, возбуждается исключение `InvalidPage` из модуля `django.core.paginator`.

  Этот метод оставлен для совместимости с предыдущими версиями Django.

Листинг 12.1 показывает пример использования пагинатора. В нем приведен код контроллера-функции `index()`, которая выводит список объявлений с разбиением на части (подразумевается, что номер части передается через GET-параметр `page`).

Листинг 12.1. Пример использования пагинатора

```
from django.shortcuts import render
from django.core.paginator import Paginator
from .models import Bb, Rubric

def index(request):
    rubrics = Rubric.objects.all()
    bbs = Bb.objects.all()
    paginator = Paginator(bbs, 2)
    if 'page' in request.GET:
        page_num = request.GET['page']
```

```
    else:
        page_num = 1
    page = paginator.get_page(page_num)
    context = {'rubrics': rubrics, 'page': page, 'bbs': page.object_list}
    return render(request, 'bboard/index.html', context)
```

Мы проверяем, присутствует ли в наборе GET-параметров, полученных в запросе, параметр `page`. Если это так, мы извлекаем из него номер части, которую нужно вывести на странице. В противном случае подразумевается, что посетитель запрашивает первую часть, которую мы и выводим.

В контексте шаблона мы создаем переменную `bbs`, которой присваиваем список записей, входящих в запрошенную часть (его можно извлечь из атрибута `object_list` части пагинатора). Это позволит нам использовать уже имеющийся у нас шаблон bboard\index.html.

12.2. Класс *Page*: часть пагинатора. Вывод пагинатора

Класс `Page` из модуля `django.core.paginator` представляет отдельную часть пагинатора, возвращенную методом `get_page()` или `page()` (они были описаны в *разд. 12.1*).

Класс `Page` поддерживает следующие атрибуты:

- `object_list` — список записей, входящих в состав текущей части;
- `number` — порядковый номер текущей части пагинатора (не забываем, что нумерация частей начинается с 1);
- `paginator` — пагинатор (в виде экземпляра класса `Paginator`), создавший эту часть.

А вот набор методов этого класса:

- `has_next()` — возвращает `True`, если существуют следующие части пагинатора, и `False` в противном случае;
- `has_previous()` — возвращает `True`, если существуют предыдущие части пагинатора, и `False` в противном случае;
- `has_other_pages()` — возвращает `True`, если существуют предыдущие или следующие части пагинатора, и `False` в противном случае;
- `next_page_number()` — возвращает номер следующей части пагинатора. Если это последняя часть (т. е. следующей части не существует), возбуждает исключение `InvalidPage`;
- `previous_page_number()` — возвращает номер предыдущей части пагинатора. Если это первая часть (т. е. предыдущей части не существует), возбуждает исключение `InvalidPage`;

- `start_index()` — возвращает порядковый номер первой записи, присутствующей в текущей части. Нумерация записей в этом случае начинается с 1;
- `end_index()` — возвращает порядковый номер последней записи, присутствующей в текущей части. Нумерация записей в этом случае начинается с 1.

Далее показан фрагмент кода шаблона **bboard\index.html**, выводящий набор гиперссылок для перехода между частями пагинатора:

```
<div>
    {% if page.has_previous %}
    <a href="?page={{ page.previous_page_number }}">&lt;</a>
      |  
    {% endif %}
    Часть №{{ page.number }} из {{ page.paginator.num_pages }}
    {% if page.has_next %}
      |  
    <a href="?page={{ page.next_page_number }}">&gt;</a>
    {% endif %}
</div>
```

ГЛАВА 13

Формы, связанные с моделями

Форма в терминологии Django — это сущность, предназначенная для получения данных от посетителя и проверки их на корректность. Форма определяет набор полей, в которые будут вводиться отдельные значения, типы заносимых значений, элементы управления, посредством которых будет осуществляться ввод данных, и правила валидации.

Форма, связанная с моделью, отличается от обычной формы тем, что представляет какую-либо запись модели — уже существующую или еще не существующую. В частности, поля такой формы соответствуют одноименным полям модели. Помимо этого, такая форма поддерживает метод `save()`, вызвав который, мы с легкостью сохраним занесенные в форму данные в информационной базе.

Поскольку бóльшая часть форм, создаваемых разработчиками Django-сайтов, служит для занесения в информационную базу новых записей или правки уже существующих, мы рассмотрим формы, связанные с моделями, прямо сейчас.

13.1. Создание форм, связанных с моделями

Существуют три способа создать форму, связанную с моделью: два простых и сложный. Рассмотрим их по очереди.

13.1.1. Создание форм посредством фабрики классов

Первый, самый простой способ создать форму, связанную с моделью, — использовать функцию `modelform_factory()` из модуля `django.forms`. Вот формат ее вызова:

```
modelform_factory(<модель>[, fields=None][, exclude=None][,
                  labels=None][, help_texts=None][,
                  error_messages=None][, field_classes=None][,
                  widgets=None][, form=<форма, связанная с моделью>])
```

В первом параметре указывается ссылка на класс модели, на основе которой нужно создать форму.

Параметр `fields` задает последовательность имен полей модели, которые должны быть включены в создаваемую форму. Любые поля, не включенные в эту последовательность, не войдут в состав формы. Чтобы указать все поля модели, нужно присвоить этому параметру строку `"__all__"`.

Параметр `exclude` задает последовательность имен полей модели, которые, напротив, не должны включаться в форму. Соответственно, все поля, отсутствующие в этой последовательности, войдут в состав формы.

> ***Внимание!***
>
> В вызове функции `modelform_factory()` должен присутствовать либо параметр `fields`, либо параметр `exclude`. Указание сразу обоих параметров приведет к ошибке.

Параметр `labels` задает надписи для полей формы. Его значение должно представлять собой словарь, ключи элементов которого определяют поля формы, а значения — надписи для них.

Параметр `help_texts` указывает дополнительный поясняющий текст для полей формы (такой текст будет выводиться возле элементов управления). Значение этого параметра должно представлять собой словарь, ключи элементов которого определяют поля формы, а значения — поясняющие подписи для них.

Параметр `error_messages` указывает строковые сообщения об ошибках. Его значением должен быть словарь, ключи элементов которого определяют поля формы, а значениями элементов также должны быть словари. Во вложенных словарях ключи элементов определяют строковые коды ошибок (их можно найти в *разд. 4.8.2*), а значения как раз и зададут для них строковые сообщения.

Параметр `field_classes` указывает, поле какого типа должно быть создано в форме для соответствующего ему поля модели. Значением должен быть словарь, ключи элементов которого представляют имена полей модели, а значениями элементов станут ссылки на соответствующие им классы полей формы.

Параметр `widgets` позволяет задать элемент управления, которым будет представляться на веб-странице то или иное поле модели. Значением должен быть словарь, ключи элементов которого представляют имена полей формы, а значениями элементов станут экземпляры классов, представляющих элементы управления, или ссылки на сами эти классы.

Если какой-либо параметр не указан, то либо его значение будет взято из модели, либо для его параметра установится значение по умолчанию. Так, если не указать надпись для поля формы, будет использовано название сущности, указанное в параметре `verbose_name` конструктора поля модели, а если не указать тип элемента управления, будет использован элемент управления по умолчанию для поля этого типа.

И наконец, параметр `form` служит для указания *формы, связанной с моделью*, на основе которой будет создана новая форма. Заданная в параметре *форма* может указывать какие-либо общие для целой группы форм параметры.

Функция `modelform_factory()` в качестве результата возвращает готовый к использованию класс формы, связанной с моделью (подобные функции, генерирующие целые классы, называются *фабриками классов*).

Листинг 13.1 показывает код, создающий на основе модели `Bb` класс формы `BbForm` с применением фабрики классов.

Листинг 13.1. Использование фабрики классов `modelform_factory()`

```
from django.forms import modelform_factory, DecimalField
from django.forms.widgets import Select

from .models import Bb, Rubric

BbForm = modelform_factory(Bb,
         fields=('title', 'content', 'price', 'rubric'),
         labels={'title': 'Название товара'},
         help_texts={'rubric': 'Не забудьте выбрать рубрику!'},
         field_classes={'price': DecimalField},
         widgets={'rubric': Select(attrs={'size': 8})})
```

Ради эксперимента мы изменили надпись у поля названия товара, задали поясняющий текст у поля рубрики, изменили тип поля цены на `DecimalField` и указали для поля рубрики представление в виде обычного списка высотой в 8 пунктов.

Класс, сохраненный в переменной `BbForm`, мы можем использовать точно так же, как и любой написанный нами собственноручно, — например, указать его в высокоуровневом контроллере-классе:

```
class BbCreateView(CreateView):
    form_class = BbForm
    . . .
```

Фабрики классов удобно использовать в контроллерах-функциях для создания редко используемых форм. В этом случае бóльшую часть времени класс формы не занимает оперативную память, поскольку создается только при необходимости и уничтожается сразу же, как только перестанет существовать хранящая его переменная.

13.1.2. Создание форм путем быстрого объявления

Если же форма, связанная с моделью, нужна нам на продолжительное время, имеет смысл прибегнуть ко второму способу — объявить класс формы вручную. В этом случае он будет всегда готов к использованию, и Django не станет тратить время на его создание с помощью фабрики классов.

Класс формы, связанной с моделью, должен быть производным от класса `ModelForm`, объявленного в модуле `django.forms`. В этом классе объявляется вложен-

ный класс `Meta`, в котором записывается набор атрибутов класса, имеющих те же имена, что параметры функции `modelform_factory()`, и то же назначение.

Такой способ объявления класса, при котором мы не выписываем все параметры представляемой им сущности вручную, а ограничиваемся общими указаниями, носит название *быстрого объявления*.

В листинге 13.2 можно увидеть код класса формы `BbForm`, созданный посредством быстрого объявления.

Листинг 13.2. Использование быстрого объявления формы, связанной с моделью

```
from django.forms import ModelForm, DecimalField
from django.forms.widgets import Select
from .models import Bb

class BbForm(ModelForm):
    class Meta:
        model = Bb
        fields = ('title', 'content', 'price', 'rubric')
        labels = {'title': 'Название товара'}
        help_texts = {'rubric': 'Не забудьте задать рубрику!'}
        field_classes = {'price': DecimalField}
        widgets = {'rubric': Select(attrs={'size': 8})}
```

13.1.3. Создание форм путем полного объявления

Оба описанных ранее способа создания форм имели недостаток, который может оказаться серьезным, — они позволяли задать для полей формы весьма ограниченный набор параметров. Если же мы желаем получить доступ ко всему набору параметров, поддерживаемому полями формы, нам ничего не остается, как прибегнуть к третьему, сложному способу объявления форм.

13.1.3.1. Как выполняется полное объявление

Объявление формы таким способом сильно напоминает объявление модели (см. *главу 4*). Мы записываем непосредственно в классе формы набор атрибутов класса, представляющих отдельные поля, присваиваем им экземпляры классов, представляющих поля нужных типов, и указываем параметры полей непосредственно в конструкторах этих классов. Также не забываем объявить в классе формы вложенный класс `Meta` и указать в нем модель, с которой должна быть связана форма, и список полей модели, которые должны присутствовать в форме (или, наоборот, отсутствовать в ней).

В таком случае мы вручную пишем все параметры формы, связанной с моделью, включая набор ее полей. Это *полное объявление* класса.

Пример создания формы с применением полного объявления можно увидеть в листинге 13.3. Там показан код, создающий класс формы `BbForm` на основе модели `Bb`.

Листинг 13.3. Полное объявление всех полей формы

```
from django import forms
from .models import Bb, Rubric

class BbForm(forms.ModelForm):
    title = forms.CharField(label='Название товара')
    content = forms.CharField(label='Описание',
              widget=forms.widgets.Textarea())
    price = forms.DecimalField(label='Цена', decimal_places=2)
    rubric = forms.ModelChoiceField(queryset=Rubric.objects.all(),
             label='Рубрика', help_text='Не забудьте задать рубрику!',
             widget=forms.widgets.Select(attrs={'size': 8}))

    class Meta:
        model = Bb
        fields = ('title', 'content', 'price', 'rubric')
```

Можно выполнить полное объявление не всех полей формы, а только тех, для которых нужно задать параметры из расширенного набора или у которых нужно радикально сменить поведение. Параметры остальных полей формы можно указать путем быстрого объявления (см. *разд. 13.1.2*).

Такой подход иллюстрирует листинг 13.4. Здесь выполнено создание путем полного объявления только полей `price` и `rubric`, а остальные поля создаются с применением быстрого объявления.

Листинг 13.4. Полное объявление отдельных полей формы

```
from django import forms
from .models import Bb, Rubric

class BbForm(forms.ModelForm):
    price = forms.DecimalField(label='Цена', decimal_places=2)
    rubric = forms.ModelChoiceField(queryset=Rubric.objects.all(),
             label='Рубрика', help_text='Не забудьте задать рубрику!',
             widget=forms.widgets.Select(attrs={'size': 8}))

    class Meta:
        model = Bb
        fields = ('title', 'content', 'price', 'rubric')
        labels = {'title': 'Название товара'}
```

Применение полного объявления позволило нам, в частности, задать количество знаков после запятой для поля типа `DecimalField` (параметр `decimal_places`). Быстрое объявление не даст нам это сделать.

Внимание!

Если в классе формы присутствует и полное, и быстрое объявление какого-либо поля, будет обработано только полное объявление. Параметры, записанные в быстром объявлении поля, будут проигнорированы.

Применяя полное объявление, можно добавить в форму дополнительные поля, отсутствующие в связанной с формой модели. Этот прием иллюстрирует листинг 13.5, где в форму регистрации нового пользователя `RegisterUserForm` добавляются поля `password1` и `password2`, которые не существуют в модели `User`.

Листинг 13.5. Объявление в форме полей, не существующих в связанной модели

```
class RegisterUserForm(forms.ModelForm):
    password1 = forms.CharField(label='Пароль')
    password2 = forms.CharField(label='Пароль (повторно)')

    class Meta:
        model = User
        fields = ('username', 'email', 'first_name', 'last_name')
```

Поля, созданные таким образом, необязательно приводить в списке включаемых в форму полей, что указывается в атрибуте `fields` вложенного класса `Meta`. Однако в этом случае такие поля при выводе формы на экран окажутся в ее конце. Чтобы задать для них нужное местоположение, их следует включить в список атрибута `fields`. Пример:

```
class RegisterUserForm(forms.ModelForm):
    . . .
    class Meta:
        . . .
        fields = ('username', 'email', 'password1', 'password2',
                  'first_name', 'last_name')
```

13.1.3.2. Параметры, поддерживаемые всеми типами полей

Поле формы, созданное путем полного объявления, представляется отдельным атрибутом класса формы. Ему присваивается экземпляр класса, представляющего поле определенного типа. Дополнительные параметры создаваемого поля можно указать в соответствующих им именованных параметрах конструктора класса этого поля.

Рассмотрим относительно немногочисленные параметры, поддерживаемые полями всех типов:

- `label` — надпись для поля. Если не указан, в качестве надписи будет использовано имя текущего поля;
- `help_text` — дополнительный поясняющий текст для текущего поля, который будет выведен возле элемента управления;

- `label_suffix` — суффикс, который будет добавлен к надписи для текущего поля.

  Если параметр не указан, будет взято значение одноименного параметра, поддерживаемого конструктором класса формы (эти параметры мы рассмотрим в *главе 17*). Если и тот не указан, будет использовано значение по умолчанию — символ двоеточия;

- `initial` — начальное значение для поля формы. Если не указан, поле не будет иметь начального значения;

- `required` — если `True`, в это поле обязательно должно быть занесено значение, если `False`, поле может быть «пустым». Значение по умолчанию — `True`;

- `widget` — элемент управления, которым текущее поле будет представляться на веб-странице. Значение может представлять собой либо ссылку на класс элемента управления, либо экземпляр этого класса.

  Если параметр не указан, будет использован элемент управления по умолчанию, применяемый для поля такого типа;

- `validators` — валидаторы для текущего поля. Задаются в таком же формате, что и для поля модели (см. *разд. 4.8.1*);

- `error_messages` — сообщения об ошибках. Задаются в таком же формате, что и аналогичные сообщения для поля модели (см. *разд. 4.8.2*);

- `disabled` — если `True`, соответствующий текущему полю элемент управления будет недоступен для взаимодействия, если `False`, будет доступным. Значение по умолчанию — `False`.

13.1.3.3. Доступные классы полей форм

Классов полей форм Django предоставляет почти столько же, сколько классов полей моделей (см. *разд. 4.3.2*). По большей части они аналогичны друг другу и предназначены для занесения данных строго определенных типов. Все классы полей форм объявлены в модуле `django.forms`:

- `CharField` — строковое или текстовое поле. Конструктором поддерживаются дополнительные параметры:
  - `min_length` — минимальная длина значения, заносимого в поле, в символах;
  - `max_length` — максимальная длина значения, заносимого в поле, в символах;
  - `strip` — если `True`, из заносимого в поле значения будут удалены начальные и конечные пробелы, если `False`, пробелы удаляться не будут. Значение по умолчанию — `True`;
  - `empty_value` — величина, которой будет представляться «пустое» поле. Значение по умолчанию — «пустая» строка;
- `EmailField` — корректно сформированный адрес электронной почты в строковом виде. Дополнительные параметры, поддерживаемые конструктором:

- min_length — минимальная длина значения, заносимого в поле, в символах;
- max_length — максимальная длина значения, заносимого в поле, в символах;

❒ URLField — корректно сформированный интернет-адрес. Дополнительные параметры, поддерживаемые конструктором:

- min_length — минимальная длина значения, заносимого в поле, в символах;
- max_length — максимальная длина значения, заносимого в поле, в символах;

❒ SlugField — слаг.

Конструктор поддерживает дополнительный параметр allow_unicode. Если его значение равно True, хранящийся в поле слаг может содержать символы Unicode, если False — только символы из кодировки ASCII. Значение по умолчанию — False.

❒ RegexField — строковое значение, совпадающее с заданным регулярным выражением. Конструктор поддерживает дополнительные параметры:

- regex — само регулярное выражение, с которым должно совпадать заносимое в поле значение. Может быть задано как в виде строки, так и в виде объекта типа re;
- min_length — минимальная длина значения, заносимого в поле, в символах;
- max_length — максимальная длина значения, заносимого в поле, в символах;
- strip — если True, из заносимого в поле значения будут удалены начальные и конечные пробелы, если False, пробелы удаляться не будут. Значение по умолчанию — False;

❒ BooleanField — логическое поле;

❒ NullBooleanField — то же самое, что BooleanField, но дополнительно позволяет хранить значение null;

❒ IntegerField — знаковое целочисленное поле обычной длины (32-разрядное). Конструктор поддерживает дополнительные параметры:

- min_value — минимальное значение, которое можно занести в поле;
- max_value — максимальное значение, которое можно занести в поле;

❒ FloatField — вещественное число. Конструктор поддерживает дополнительные параметры:

- min_value — минимальное значение, которое можно занести в поле;
- max_value — максимальное значение, которое можно занести в поле;

❒ DecimalField — вещественное число фиксированной точности, представленное объектом типа Decimal из модуля decimal Python. Конструктор поддерживает четыре дополнительных параметра:

- min_value — минимальное значение, которое можно занести в поле;
- max_value — максимальное значение, которое можно занести в поле;

- max_digits — максимальное количество цифр в числе;
- decimal_places — количество цифр в дробной части числа;

- DateField — значение даты, представленное в виде объекта типа date из модуля datetime Python.

  Конструктор класса поддерживает дополнительный параметр input_formats, задающий последовательность форматов, в которых в это поле можно будет занести дату. Значение этого параметра по умолчанию задается языковыми настройками или параметром DATE_INPUT_FORMATS настроек проекта (см. *разд. 3.3.5*);

- DateTimeField — значение даты и времени в виде объекта типа datetime из модуля datetime.

  Конструктор класса поддерживает дополнительный параметр input_formats, задающий последовательность форматов, в которых в это поле можно будет занести дату и время. Значение этого параметра по умолчанию задается языковыми настройками или параметром DATETIME_INPUT_FORMATS настроек проекта;

- TimeField — значение времени, представленное в виде объекта типа time из модуля datetime Python.

  Конструктор класса поддерживает дополнительный параметр input_formats, задающий последовательность форматов, в которых в это поле можно будет занести время. Значение этого параметра по умолчанию задается языковыми настройками или параметром TIME_INPUT_FORMATS настроек проекта;

- SplitDateTimeField — то же самое, что и DateTimeField, но для занесения значений даты и времени применяются разные элементы управления. Дополнительные параметры конструктора:
  - input_date_formats — последовательность форматов, в которых в это поле можно будет занести дату. Значение этого параметра по умолчанию задается языковыми настройками или параметром DATE_INPUT_FORMATS настроек проекта;
  - input_time_formats — последовательность форматов, в которых в это поле можно будет занести время. Значение этого параметра по умолчанию задается языковыми настройками или параметром TIME_INPUT_FORMATS настроек проекта;

- DurationField — промежуток времени, представленный объектом типа timedelta из модуля datetime Python;

- ModelChoiceField — поле внешнего ключа вторичной модели, создающее связь «один-со-многими» или «один-с-одним». Позволяет выбрать в списке только одну связываемую запись первичной модели. Дополнительные параметры конструктора:
  - queryset — набор записей, извлеченных из первичной модели, на основе которого будет формироваться список;
  - empty_label — строка, обозначающая «пустой» пункт в списке связываемых записей. Значение по умолчанию: "---------". Также можно убрать «пустой» пункт из списка, присвоив этому параметру значение None;

- `to_field_name` — имя поля первичной модели, значение из которого будет сохраняться в текущем поле внешнего ключа. Значение по умолчанию — `None` (обозначает ключевое поле);

- `ModelMultipleChoiceField` — поле внешнего ключа ведущей модели, создающее связь «многие-со-многими». Позволяет выбрать в списке произвольное количество связываемых записей. Дополнительные параметры конструктора:
  - `queryset` — набор записей, извлеченных из ведомой модели, на основе которого будет формироваться список;
  - `to_field_name` — имя поля ведомой модели, значение из которого будет сохраняться в текущем поле внешнего ключа. Значение по умолчанию — `None` (обозначает ключевое поле);

- `ChoiceField` — поле со списком, в которое можно занести только те значения, что приведены в списке. Значение записывается в поле в строковом формате.

  Конструктор класса поддерживает дополнительный параметр `choices`, задающий последовательность значений, которые будут представлены в списке. Последовательность указывается в том же формате, что и у параметра `choices` конструктора полей моделей (см. *разд. 4.3.1*);

- `TypedChoiceField` — то же самое, что `ChoiceField`, но позволяет хранить в поле значение любого типа, а не только строкового. Дополнительные параметры конструктора:
  - `choices` — последовательность значений, которые будут представлены в списке. Указывается в том же формате, что и у параметра `choices` конструктора полей моделей;
  - `coerce` — ссылка на функцию, выполняющую преобразование типа значения, предназначенного для сохранения в поле. Должна принимать с единственным параметром исходное значение и возвращать в качестве результата то же значение, но преобразованное к нужному типу;
  - `empty_value` — значение, которым будет представляться «пустое» поле. Значение по умолчанию — «пустая» строка.

    Значение, представляющее «пустое» поле, можно записать непосредственно в последовательность, задаваемую параметром `choices`;

- `MultipleChoiceField` — то же самое, что `ChoiceField`, но позволяет выбрать в списке произвольное количество пунктов;

- `TypedMultipleChoiceField` — то же самое, что и `TypedChoiceField`, но позволяет выбрать в списке произвольное количество пунктов;

- `GenericIPAddressField` — IP-адрес, записанный для протокола IPv4 или IPv6, в виде строки. Конструктор поддерживает два дополнительных параметра:
  - `protocol` — допустимый протокол для записи IP-адресов, представленный в виде строки. Доступны значения: `"IPv4"`, `"IPv6"` и `"both"` (поддерживаются оба протокола). Значение по умолчанию: `"both"`;

- `inpack_ipv4` — если `True`, IP-адреса протокола IPv4, записанные в формате IPv6, будут преобразованы к виду, применяемому в IPv4. Если `False`, такое преобразование не выполняется. Значение по умолчанию — `False`. Этот параметр принимается во внимание только если для параметра `protocol` указано значение `"both"`.

- `UUIDField` — уникальный универсальный идентификатор, представленный объектом типа `UUID` из модуля `uuid` Python, в виде строки.

> ***На заметку***
>
> Также поддерживаются классы полей формы `ComboField` и `MultiValueField`, из которых первый применяется в крайне специфических случаях, а второй служит для разработки на его основе других классов полей. Описание этих двух классов можно найти на странице **https://docs.djangoproject.com/en/2.1/ref/forms/fields/**.

13.1.3.4. Классы полей форм, применяемые по умолчанию

Каждому классу поля модели поставлен в соответствие определенный класс поля формы. Этот класс поля формы по умолчанию используется для представления в форме поля модели соответствующего класса при создании формы посредством фабрики классов или быстрого объявления, если мы не указали класс явно.

Табл. 13.1 показывает классы полей модели и соответствующие им классы полей формы, используемые по умолчанию.

***Таблица 13.1.** Классы полей модели и соответствующие им классы полей формы, используемые по умолчанию*

Классы полей модели	Классы полей формы
`CharField`	`CharField`. Параметр `max_length` получает значение от параметра `max_length` конструктора поля модели. Если параметр `null` конструктора поля модели имеет значение `True`, параметр `empty_value` конструктора поля формы получит значение `None`
`TextField`	`CharField`, у которого в качестве элемента управления указана область редактирования (параметр `widget` имеет значение `Textarea`)
`EmailField`	`EmailField`
`URLField`	`URLField`
`SlugField`	`SlugField`
`BooleanField`	`BooleanField`
`NullBooleanField`	`NullBooleanField`
`IntegerField`	`IntegerField`
`SmallIntegerField`	
`BigIntegerField`	`IntegerField`, у которого параметр `min_value` имеет значение –9223372036854775808, а параметр `max_value` — 9223372036854775807

Таблица 13.1 (окончание)

Классы полей модели	Классы полей формы
PositiveIntegerField	IntegerField
PositiveSmallIntegerField	
FloatField	FloatField
DecimalField	DecimalField
DateField	DateField
DateTimeField	DateTimeField
TimeField	TimeField
DurationField	DurationField
GenericIPAddressField	GenericIPAddressField
AutoField	Не представляются в формах
BigAutoField	
ForeignKey	ModelChioceField
ManyToManyField	ModelMultipleChoiceField

13.1.4. Задание элементов управления

Для любого поля формы мы можем указать элемент управления, посредством которого посетитель будет заносить в него значение. Это выполняется с помощью знакомого нам параметра `widget` конструктора поля формы.

13.1.4.1. Классы элементов управления

Django предлагает довольно много классов, представляющих различные элементы управления. Часть этих классов представляет стандартные элементы управления, непосредственно поддерживаемые языком HTML (поля ввода, списки, флажки, переключатели и др.), остальные обеспечивают работу более сложных элементов управления, составленных из ряда более простых (в качестве примера можно привести набор флажков).

Все классы элементов управления являются производными от класса `Widget` из модуля `django.forms.widgets`. Этот класс (а значит, и все производные от него классы) поддерживает параметр конструктора `attrs`, указывающий значения атрибутов тега, создающего элемент управления. Значением должен быть словарь, ключи элементов которого должны совпадать с именами атрибутов тега, а значения элементов зададут значения этих атрибутов.

Вот пример указания значения `8` для атрибута `size` тега `<select>`, создающего список (в результате будет создан обычный список высотой 8 пунктов):

```
widget = forms.widgets.Select(attrs={'size': 8})
```

Далее приведен список поддерживаемых Django классов элементов управления. Все эти классы также объявлены в модуле `django.forms.widgets`:

- `TextInput` — обычное поле ввода;
- `NumberInput` — поле для ввода числа;
- `EmailInput` — поле для ввода адреса электронной почты;
- `URLInput` — поле для ввода интернет-адреса;
- `PasswordInput` — поле для ввода пароля.

  Конструктор класса поддерживает дополнительный параметр `render_value`. Если присвоить ему значение `True`, после неудачной валидации и повторного вывода на экран для исправления ошибок в поле ввода пароля будет находиться набранный ранее пароль. Если дать параметру значение `False`, поле в этом случае будет выведено «пустым». Значение по умолчанию — `False`;
- `HiddenInput` — скрытое поле;
- `DateInput` — поле для ввода значения даты.

  Конструктор класса поддерживает дополнительный параметр `format`, указывающий формат, в котором будет выведено изначальное значение даты. Если параметр не указан, будет использовано значение, заданное языковыми настройками, или первый формат из списка, хранящегося в параметре `DATE_INPUT_FORMATS` настроек проекта (см. *разд. 3.3.5*);
- `SelectDateWidget` — то же, что и `DateInput`, но ввод даты осуществляется с помощью трех раскрывающихся списков (для числа, месяца и года соответственно). Конструктор поддерживает следующий набор дополнительных параметров:
  - `years` — список или кортеж значений года, которые будут выводиться в раскрывающемся списке, задающем год. Если не указан, будет использован набор из текущего года и 9 следующих за ним годов;
  - `months` — словарь месяцев, которые будут выводиться в раскрывающемся списке, задающем месяц. Ключами элементов этого словаря должны быть порядковые номера месяцев, начиная с 1 (1 — январь, 2 — февраль и т. д.), а значениями элементов — названия месяцев. Если параметр не указан, будет использован словарь, содержащий все месяцы;
  - `empty_label` — задает строку, представляющую «пустое» значение числа, месяца и даты. Значением этого параметра может быть строка (она будет использована для представления «пустых» даты, месяца и года), список или кортеж из трех строковых элементов (первый элемент будет представлять «пустой» год, второй — «пустой» месяц, третий — «пустое» число). Значение параметра по умолчанию: `"---"`. Пример:

```
published = forms.DateField(
            widget=forms.widgets.SelectDateWidget(
                  empty_label=('Выберите год', 'Выберите месяц',
                               'Выберите число')))
```

- `DateTimeInput` — поле для ввода значения даты и времени.

  Конструктор класса поддерживает дополнительный параметр `format`, указывающий формат, в котором будет выведено изначальное значение даты и времени. Если параметр не указан, будет использовано значение, заданное языковыми настройками, или первый формат из списка, хранящегося в параметре `DATETIME_INPUT_FORMATS` настроек проекта (см. *разд. 3.3.5*);

- `SplitDateTimeWidget` — то же, что и `DateTimeInput`, но ввод значений даты и времени осуществляется в разные поля. Конструктор класса поддерживает дополнительные параметры:

  - `date_format` — формат, в котором будет выведено изначальное значение даты. Если не указан, будет использовано значение, заданное языковыми настройками, или первый формат из списка, хранящегося в параметре `DATE_INPUT_FORMATS` настроек проекта;
  - `time_format` — формат, в котором будет выведено изначальное значение времени. Если не указан, будет использовано значение, заданное языковыми настройками, или первый формат из списка, хранящегося в параметре `TIME_INPUT_FORMATS` настроек проекта;
  - `date_attrs` — значения атрибутов тега, создающего поля ввода даты;
  - `time_attrs` — значения атрибутов тега, создающего поля ввода времени.

  Значения обоих параметров задаются в том же виде, что и у описанного ранее параметра `attrs`;

- `TimeInput` — поле для ввода значения времени.

  Конструктор класса поддерживает дополнительный параметр `format`, указывающий формат, в котором будет выведено изначальное значение времени. Если параметр не указан, будет использовано значение, заданное языковыми настройками, или первый формат из списка, хранящегося в параметре `TIME_INPUT_FORMATS` настроек проекта;

- `Textarea` — область редактирования;

- `CheckboxInput` — флажок.

  Конструктор класса поддерживает дополнительный параметр `check_test`. Ему присваивается ссылка на функцию, которая в качестве параметра принимает значение флажка и возвращает `True`, если флажок должен быть выведен установленным, и `False`, если сброшенным;

- `Select` — список, обычный или раскрывающийся (зависит от значения атрибута `size` тега `<select>`), с возможностью выбора только одного пункта. Сведения о пунктах, которые должны выводиться в нем, он берет из параметра `choices` конструктора поля формы, с которым связан.

  Конструктор класса `Select` поддерживает дополнительный параметр `choices`. Ему можно присвоить последовательность пунктов, которые должны выводиться в списке. Эта последовательность задается в том же формате, что и у пара-

метра `choices` конструктора полей моделей (см. *разд. 4.3.1*). Значение, заданное в параметре `choices` конструктора элемента управления, будет иметь приоритет перед таковым, указанным в конструкторе поля формы;

- `RadioSelect` — аналогичен `Select`, но выводится в виде группы переключателей;
- `SelectMultiple` — то же самое, что и `Select`, но позволяет выбрать произвольное количество пунктов;
- `CheckboxSelectMultiple` — аналогичен `SelectMultiple`, но выводится в виде набора флажков;
- `NullBooleanSelect` — раскрывающийся список с пунктами **Да**, **Нет** и **Неизвестно**.

13.1.4.2. Элементы управления, применяемые по умолчанию

Для каждого класса поля формы существует класс элемента управления, применяемый для его представления по умолчанию. Эти классы приведены в табл. 13.2.

***Таблица 13.2.** Классы полей формы и соответствующие им классы элементов управления, используемые по умолчанию*

Классы полей формы	Классы элементов управления
`CharField`	`TextInput`
`EmailField`	`EmailInput`
`URLField`	`URLInput`
`SlugField`	`TextInput`
`RegexField`	
`BooleanField`	`CheckboxInput`
`NullBooleanField`	`NullBooleanSelect`
`IntegerField`	`NumberInput`
`FloatField`	
`DecimalField`	
`DateField`	`DateInput`
`DateTimeField`	`DateTimeInput`
`TimeField`	`TimeInput`
`SplitDateTimeField`	`SplitDateTimeWidget`
`DurationField`	`TextInput`
`ModelChoiceField`	`Select`
`ModelMultipleChoiceField`	`SelectMultiple`
`ChoiceField`	`Select`
`TypedChoiceField`	

Таблица 13.2 (окончание)

Классы полей формы	Классы элементов управления
MultipleChoiceField	SelectMultiple
TypedMultipleChoiceField	
GenericIPAddressField	TextInput
UUIDField	

13.2. Обработка форм

Объявив класс формы, мы можем применить его для обработки и сохранения полученных от посетителя данных.

Если мы используем высокоуровневые контроллеры-классы (см. *главу 11*), последние обработают и сохранят данные из формы без какого бы то ни было нашего участия. Но если мы предпочитаем контроллеры-классы низкого уровня или вообще контроллеры-функции, нам самим придется заняться обработкой форм.

13.2.1. Добавление записи посредством формы

Начнем мы с добавления в модель новой записи с применением формы, связанной с этой моделью (подразумевается, что класс формы мы уже объявили).

13.2.1.1. Создание формы для добавления записи

Для добавления записи нам нужно создать экземпляр класса формы, поместить его в контекст шаблона и выполнить рендеринг шаблона, представляющего страницу добавления записи, тем самым выведя форму на экран. Все эти действия выполняются при отправке клиентом запроса с помощью HTTP-метода GET.

Создать экземпляр класса формы мы можем вызовом конструктора этого класса без параметров:

```
bbf = BbForm()
```

Если нам нужно поместить в форму какие-то изначальные данные, мы можем использовать необязательный параметр `initial` конструктора класса формы. Этому параметру присваивается словарь, ключи элементов которого задают имена полей формы, а значения элементов — изначальные значения для этих полей. Пример:

```
bbf = BbForm(initial={kind='b'})
```

Пример простейшего контроллера-функции, создающего форму и выводящего ее на экран, можно увидеть в листинге 9.1. А листинг 9.3 показывает более сложный контроллер-функцию, который при получении запроса, отправленного методом GET, выводит форму на экран, а при получении POST-запроса сохраняет данные из формы в модели, там самым создавая новую запись.

13.2.1.2. Повторное создание формы

После ввода посетителем данных в форму и нажатия кнопки отправки веб-обозреватель выполняет POST-запрос, содержащий введенные в форму данные. Получив такой запрос, контроллер сразу «поймет», что сейчас следует выполнить валидацию данных из формы и, если она пройдет успешно, сохранить эти данные в новой записи модели.

Но сначала нужно создать экземпляр класса формы еще раз и поместить в него данные, полученные в составе запроса. Для этого следует, опять же, вызвать конструктор класса формы, передав ему в качестве первого позиционного параметра словарь с полученными данными. Словарь этот можно извлечь из атрибута `POST` запроса, представленного экземпляром класса `Request` (см. *разд. 9.4*). Пример:

```
bbf = BbForm(request.POST)
```

После этого форма произведет обработку полученных из запроса данных и подготовится к выполнению валидации.

Имеется возможность проверить, были ли в форму при ее создании помещены данные из запроса. Для этого достаточно вызвать метод `is_bound()`, поддерживаемый классом `ModelForm`. Метод вернет `True`, если при создании формы в нее были помещены данные, полученные в составе POST-запроса (т. е. выполнялось повторное создание формы), и `False`, если данные в форму не помещались (форма создавалась впервые).

13.2.1.3. Валидация данных, занесенных в форму

Чтобы запустить валидацию формы, достаточно выполнить одно из двух действий:

- вызвать метод `is_valid()` формы. Он вернет `True`, если занесенные в форму данные корректны, и `False` — в противном случае. Пример:

  ```
  if bbf.is_valid():
      # Данные корректны, и их можно сохранять
  else:
      # Данные некорректны
  ```

- обратиться к атрибуту `errors` формы. Он хранит перечень сообщений об ошибках, допущенных посетителем при вводе данных в форму. Ключи элементов этого словаря совпадают с именами полей формы, а значениями являются списки текстовых сообщений об ошибках.

  Ключ, совпадающий со значением переменной `NON_FIELD_ERRORS` из модуля `django.core.exceptions`, хранит сообщения об ошибках, относящихся не к определенному полю формы, а ко всей форме.

  Если данные, занесенные в форму, корректны, атрибут `errors` будет хранить «пустой» словарь.

  Пример:

  ```
  from django.core.exceptions import NON_FIELDS_ERRORS
  . . .
  ```

```
if bbf.errors:
    # Данные некорректны
    # Получаем список сообщений об ошибках, допущенных при вводе
    # названия товара
    title_errors = bbf.errors['title']
    # Получаем список ошибок, относящихся ко всей форме
    form_errors = bbf.errors[NON_FIELDS_ERRORS]
else:
    # Данные корректны, и их можно сохранять
```

Если данные корректны, их следует сохранить. После этого обычно выполняется перенаправление на страницу со списком записей или сведениями о только что добавленной записи, чтобы посетитель сразу смог увидеть, увенчалась ли его попытка успехом.

Если же данные некорректны, необходимо повторно вывести страницу с формой на экран. Тогда рядом с элементами управления формы будут выведены все относящиеся к ним сообщения об ошибках, и посетитель сразу поймет, что он сделал не так.

13.2.1.4. Сохранение данных, занесенных в форму

Сохранить данные, что были занесены в связанную с моделью форму, очень просто. Для этого достаточно вызвать метод `save()` формы:

```
bbf.save()
```

Перед сохранением данных из формы настоятельно рекомендуется выполнить их валидацию. Если этого не сделать, метод `save()` перед сохранением выполнит валидацию самостоятельно и, если она не увенчалась успехом, возбудит исключение `ValueError`. А обрабатывать результат, возвращенный методом `is_valid()`, удобнее, чем исключение (по крайней мере, на взгляд автора).

Метод `save()` в качестве результата возвращает объект созданной или исправленной записи модели, связанной с текущей формой.

Есть возможность получить только что созданную, но еще не сохраненную, запись модели с целью внести в нее какие-либо правки. Сделать это можно, записав в вызове метода `save()` необязательный параметр `commit` и присвоив ему значение `False`. Объект записи будет возвращен методом `save()` в качестве результата, но сама запись сохранена не будет. Пример:

```
bb = bbf.save(commit=False)
```

После этого с записью можно произвести нужные действия и сохранить ее обычным способом, рассмотренным еще в *главе 6*:

```
if not bb.kind:
    bb.kind = 's'
bb.save()
```

В описанном только что случае при сохранении записи модели, связанной с другой моделью связью «многие-со-многими», нужно иметь в виду один момент. Чтобы

связь между записями была успешно создана, связываемая запись должна иметь ключ (поскольку именно ключ связываемой записи записывается в связующей таблице). Однако, пока запись не сохранена, ключа у нее нет. Поэтому сначала нужно сохранить запись вызовом метода `save()` у самой модели, а потом создать связь вызовом метода `save_m2m()` формы. Вот пример:

```
mf = MachineForm(request.POST)
if mf.is_valid():
    machime = mf.save(commit=False)
    # Выполняем какие-либо дополнительные действия с записью
    machine.save()
    mf.save_m2m()
```

Отметим, что метод `save_m2m()` нужно вызывать только в том случае, если сохранение записи выполнялось вызовом метода `save()` формы с параметром `commit`, равным `False`, и последующим вызовом метода `save()` модели. Если запись сохранялась вызовом `save()` формы без параметра `commit` (или если для этого параметра было указано значение по умолчанию `True`), метод `save_m2m()` вызывать не нужно — форма сама сохранит запись и создаст связь. Пример:

```
mf = MachineForm(request.POST)
if mf.is_valid():
    mf.save()
```

13.2.1.5. Доступ к данным, занесенным в форму

Иногда бывает необходимо извлечь из формы занесенные в нее данные, чтобы, скажем, сформировать правильный интернет-адрес перенаправления. Эти данные, приведенные к нужному типу (строковому, целочисленному, логическому и др.), можно извлечь из атрибута `cleaned_data` формы. Значением этого атрибута является словарь, ключи элементов которого совпадают с именами полей формы, а значениями элементов станут значения, введенные в эти поля.

Вот пример использования ключа рубрики, указанной в только что созданном объявлении, для формирования интернет-адреса перенаправления:

```
return HttpResponseRedirect(reverse('bboard:by_rubric',
       kwargs={'rubric_id': bbf.cleaned_data['rubric'].pk}))
```

Полный пример контроллера, выполняющего создание записи с применением формы, можно увидеть в листинге 9.2. А листинг 9.3 показывает пример более сложного контроллера, который и выводит форму, и сохраняет занесенную в нее запись.

13.2.2. Правка записи посредством формы

Чтобы исправить уже имеющуюся в модели запись посредством формы, связанной с моделью, следует выполнить следующие шаги:

1. При получении запроса по HTTP-методу GET создать форму для правки записи. В этом случае нужно указать запись, которая будет править в форме, задав ее в параметре `instance` конструктора класса формы. Пример:

```
bb = Bb.objects.get(pk=pk)
bbf = BbForm(instance=bb)
```

2. Вывести страницу с формой на экран.
3. После получения POST-запроса, содержащего исправленные данные, создать форму во второй раз, указав первым позиционным параметром полученные данные, извлеченные из атрибута `POST` запроса, а параметром `instance` — исправляемую запись.
4. Выполнить валидацию формы;
   - если валидация прошла успешно, сохранить запись. После этого обычно выполняется перенаправление;
   - если валидация завершилась неудачей, повторно вывести страницу с формой.

Листинг 13.6 показывает код контроллера-функции, производящего правку записи, чей ключ был получен с URL-параметром `pk`. Здесь используется шаблон **bboard\bb_form.html**, написанный в *главе 10*.

Листинг 13.6. Контроллер-функция, производящий правку записи

```
def edit(request, pk):
    bb = Bb.objects.get(pk=pk)
    if request.method == 'POST':
        bbf = BbForm(request.POST, instance=bb)
        if bbf.is_valid():
            bbf.save()
            return HttpResponseRedirect(reverse('bboard:by_rubric',
                    kwargs={'rubric_id': bbf.cleaned_data['rubric'].pk}))
        else:
            context = {'form': bbf}
            return render(request, 'bboard/bb_form.html', context)
    else:
        bbf = BbForm(instance=bb)
        context = {'form': bbf}
        return render(request, 'bboard/bb_form.html', context)
```

При правке записи (и, в меньшей степени, при ее создании) нам может пригодиться метод `has_changed()`. Он возвращает `True`, если данные в форме были изменены посетителем, и `False` — в противном случае. Пример:

```
if bbf.is_valid():
    if bbf.has_changed():
        bbf.save()
```

Атрибут `changed_data` формы хранит список имен полей формы, чьи значения были изменены посетителем.

13.2.3. Некоторые соображения касательно удаления записей

Для удаления записи нам не нужна форма, связанная с моделью. Более того, на уровне контроллера нам вообще не придется обрабатывать никаких форм.

Для удаления записи нужно выполнить такие шаги:

1. Извлечь запись, подлежащую удалению.
2. Вывести на экран страницу с предупреждением об удалении записи.

   Эта страница должна содержать форму с кнопкой отправки данных. После нажатия кнопки веб-обозреватель отправит POST-запрос, который послужит контроллеру сигналом того, что посетитель подтвердил удаление записи.
3. После получения POST-запроса в контроллере произвести удаление записи.

Код контроллера-функции, удаляющего запись, чей ключ был получен через URL-параметр `pk`, показан в листинге 13.7. Код шаблона **bboard\bb_confirm_delete.html** можно найти в листинге 10.8.

Листинг 13.7. Контроллер-функция, производящий удаление записи

```
def delete(request, pk):
    bb = Bb.objects.get(pk=pk)
    if request.method == 'POST':
        bb.delete()
        return HttpResponseRedirect(reverse('bboard:by_rubric',
               kwargs={'rubric_id': bb.rubric.pk}))
    else:
        context = {'bb': bb}
        return render(request, 'bboard/bb_confirm_delete.html', context)
```

13.3. Вывод форм на экран

Форма, связанная с моделью, предусматривает мощные и простые в использовании механизмы для вывода на экран. Сейчас мы с ними познакомимся.

13.3.1. Быстрый вывод форм

Быстрый вывод форм осуществляется вызовом одного-единственного метода из трех, поддерживаемых Django:

- `as_p()` — вывод по абзацам. Надпись для элемента управления и сам элемент управления, представляющий какое-либо поле формы, выводятся в отдельном абзаце и отделяются друг от друга пробелами. Пример использования:

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
```

```
    <input type="submit" value="Добавить">
</form>
```

- `as_ul()` — вывод в виде маркированного списка. Надпись для элемента управления и сам элемент управления, представляющий какое-либо поле формы, выводятся в отдельном пункте списка и отделяются друг от друга пробелами. Теги `<ul>` и `</ul>`, создающие сам список, не формируются. Пример:

```
<form method="post">
    {% csrf_token %}
    <ul>
        {{ form.as_ul }}
    </ul>
    <input type="submit" value="Добавить">
</form>
```

- `as_table()` — вывод в виде таблицы. Таблица содержит два столбца: в левом выводятся надписи для элементов управления, в правом — сами элементы управления, представляющие поля формы. Каждая пара «надпись — элемент управления» занимает отдельную строку таблицы. Теги `<table>` и `</table>`, создающие саму таблицу, не выводятся. Пример:

```
<form method="post">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" value="Добавить">
</form>
```

Мы можем просто указать переменную, хранящую форму, — в этом случае будет автоматически вызван метод `as_table()`:

```
<form method="post">
    {% csrf_token %}
    <table>
        {{ form }}
    </table>
    <input type="submit" value="Добавить">
</form>
```

Обязательно уясним следующие моменты:

- парный тег `<form>`, создающий саму форму, не создается в любом случае. Нам самим придется вставить его в код шаблона и задать для него соответствующие параметры (метод отправки данных, целевой интернет-адрес, метод кодирования и др.);
- кнопка отправки данных также не создается, и нам придется поместить ее в форму самостоятельно. Такая кнопка формируется одинарным тегом `<input>` с атрибутом `type`, значение которого равно `"submit"`;

- не забываем поместить в форму тег шаблонизатора `csrf_token`. Он создаст в форме скрытое поле с электронным жетоном, по которому Django проверит, пришел ли запрос с того же самого сайта. Это сделано ради безопасности.

Теперь обратимся к методу кодирования данных. По умолчанию в веб-формах применяется метод `application/x-www-form-urlencoded`. Но, если форма отправляет файлы, в ней нужно указать метод `multipart/form-data`.

Выяснить, какой метод следует указать в теге `<form>`, нам поможет метод `is_multipart()`, поддерживаемый формой. Он возвращает `True`, если форма содержит поля, предназначенные для хранения файлов (мы познакомимся с ними в *главе 19*), и, соответственно, требует указания метода `multipart/form-data`, и `False` — в противном случае. Пример:

```
{% if form.is_multipart %}
    <form enctype="multipart/form-data" method="post">
{% else %}
    <form method="post">
{% endif %}
```

13.3.2. Расширенный вывод форм

При выводе формы по абзацам и в виде списка тег `<label>`, создающий надпись к элементу управления, отделяется от самого элемента управления пробелом. Получившаяся в результате форма почти всегда выглядит на редкость неаккуратно (пример такой формы можно увидеть на рис. 2.5).

Однако Django предоставляет нам более развитые инструменты для вывода форм на экран. Мы можем применить так называемый *расширенный вывод*, позволяющий располагать отдельные элементы формы как нравится нам, а не фреймворку.

Прежде всего, экземпляр класса `ModelForm`, представляющий связанную с моделью форму, поддерживает функциональность словаря. Ключи элементов этого словаря совпадают с именами полей формы, а значениями элементов являются экземпляры класса `BoundField`, которые представляют поля формы в виде, доступном для помещения в шаблон.

Если указать в директиве непосредственно экземпляр класса `BoundField`, он будет выведен как HTML-код, создающий элемент управления для текущего поля. Вот так можно вывести HTML-код, создающий область редактирования для указания описания товара:

```
{{ form.content }}
```

В результате мы получим такой HTML-код:

```
<textarea name="content" cols="40" rows="10" id="id_content"></textarea>
```

Помимо этого, класс `BoundField` поддерживает такие атрибуты:

- `label_tag` — HTML-код, создающий надпись для элемента управления, включая тег `<label>`. Вот пример вывода кода, создающего надпись для области редактирования, в которую заносится описание товара:

  ```
  {{ form.content.label_tag }}
  ```

Результирующий HTML-код:

```
<label for="id_title">Название товара:</label>
```

- `label` — только текст надписи;
- `help_text` — дополнительный поясняющий текст;
- `errors` — список сообщений об ошибках, относящихся к текущему полю.

Список ошибок можно вывести в шаблоне непосредственно:

```
{{ form.content.errors }}
```

В этом случае будет сформирован маркированный список с привязанным стилевым классом `errorlist`, а отдельные ошибки будут выведены как пункты этого списка:

```
<ul class="errorlist">
    <li>Укажите описание продаваемого товара</li>
</ul>
```

Также можно перебрать список ошибок в цикле и вывести отдельные его элементы с применением любых других HTML-тегов.

Чтобы получить список сообщений об ошибках, относящихся ко всей форме, следует вызвать метод `non_field_errors()` формы:

```
{{ form.non_field_errors }}
```

- `is_hidden` — `True`, если это скрытое поле, `False`, если какой-либо иной элемент управления.

Еще класс `ModelForm` поддерживает два метода:

- `visible_fields()` — возвращает список видимых полей, которые представляются на экране обычными элементами управления;
- `hidden_fields()` — возвращает список невидимых полей, представляющимися скрытыми полями HTML.

Давайте рассмотрим пример, приведенный в листинге 13.8. Показанный в нем код создает форму добавления объявления, в которой сообщения об ошибках выводятся курсивом, надписи, элементы управления и поясняющие тексты разделяются разрывом строки (HTML-тегом `<br>`), а невидимые поля (если таковые есть), выводятся отдельно от видимых. Такую форму можно увидеть на рис. 13.1 — согласитесь, что она смотрится лучше представленной на рис. 2.5.

Листинг 13.8. Отображение формы средствами расширенного вывода

```
<form method="post">
    {% csrf_token %}
    {% for hidden in form.hidden_fields %}
        {{ hidden }}
    {% endfor %}
    {% if form.non_field_errors %}
```

```
    <ul>
        {% for error in form.non_field_errors %}
        <li><em>{{ error|escape }}</em></li>
        {% endfor %}
    </ul>
    {% endif %}
    {% for field in form.visible_fields %}
        {% if field.errors %}
        <ul>
            {% for error in field.errors %}
            <li><em>{{ error|escape }}</em></li>
            {% endfor %}
        </ul>
        {% endif %}
        <p>{{ field.label_tag }}<br>{{ field }}<br>
        {{ field.help_text }}</p>
    {% endfor %}
    <p><input type="submit" value="Добавить"></p>
</form>
```

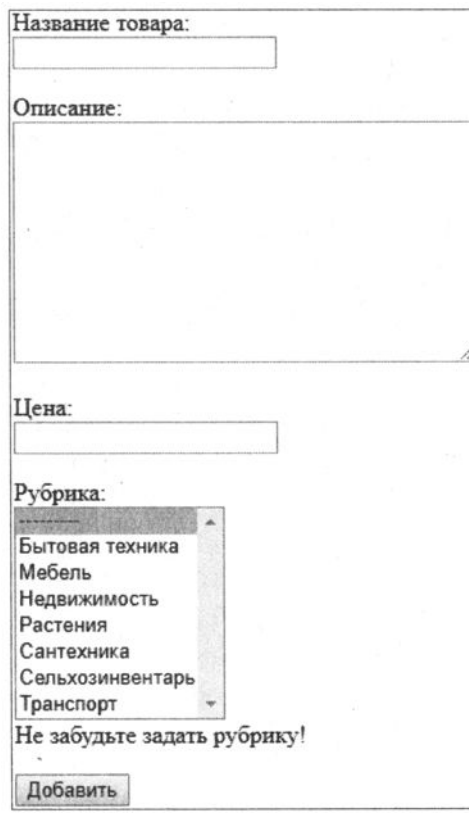


Рис. 13.1. Веб-форма, код которой показан в листинге 13.7

13.4. Валидация в формах

Из *разд. 4.8* мы знаем, что в моделях можно выполнять валидацию на уровне отдельных полей или же сразу всей модели. Мы можем использовать имеющиеся в Django валидаторы, писать свои валидаторы и вставлять валидацию непосредственно в модель, переопределяя ее методы.

А еще мы можем реализовывать валидацию на уровне форм, связанных с моделью, применяя аналогичные инструменты.

13.4.1. Валидация полей формы

Валидацию отдельных полей формы мы можем реализовать двумя способами: с применением валидаторов и путем переопределения методов формы.

13.4.1.1. Валидация с применением валидаторов

Валидация с помощью валидаторов в полях формы выполняется так же, как и в полях модели (см. *разд. 4.8*). Вот пример проверки, содержит ли название товара больше четырех символов, с выводом собственного сообщения об ошибке:

```
from django.core import validators

class BbForm(forms.ModelForm):
    title = forms.CharField(label='Название товара',
            validators=[validators.RegexValidator(regex='^.{4,}$')],
            error_messages={'invalid': 'Неправильное название товара'})
    . . .
```

Разумеется, мы можем использовать не только стандартные валидаторы, объявленные в модуле `django.core.validators`, но и свои собственные.

13.4.1.2. Валидация путем переопределения методов формы

Если нам нужно выполнить более сложную валидацию, чем та, что предоставляется валидаторами, мы можем реализовать ее непосредственно в классе формы, в переопределенном методе с именем вида `clean_<имя поля>`. Этот метод будет выполнять валидацию значения поля, чье *имя* указано в его имени.

Упомянутый ранее метод должен получать значение поля, валидацию которого выполняет, из словаря, хранящегося в знакомом нам атрибуте `cleaned_data`. Метод не должен принимать параметров и всегда должен возвращать значение проверяемого поля. Если значение не проходит валидацию, в методе следует возбудить исключение `ValidationError`.

Вот пример проверки, не собирается ли посетитель выставить на продажу прошлогодний снег:

```
from django.core.exceptions import ValidationError

class BbForm(forms.ModelForm):
    . . .
```

```
    def clean_title(self):
        val = self.cleaned_data['title']
        if val == 'Прошлогодний снег':
            raise ValidationError('Прошлогодний снег продавать ' +\
            'не допускается')
        return val
```

13.4.2. Валидация формы

И наконец, если нам нужно выполнить более сложную проверку или проверить значения сразу нескольких полей формы, т. е., говоря иными словами, произвести *валидацию формы*, мы переопределим в классе формы метод `clean()`.

Как и в случае модели, метод не должен принимать параметров, возвращать результата, а обязан при неудачной валидации возбудить исключение `ValidationError`, чтобы указать на возникшую ошибку. Единственное отличие — предварительно он должен вызвать одноименный метод базового класса, чтобы он заполнил словарь, хранящийся в атрибуте `cleaned_data` (если мы этого не сделаем, то не сможем получить данные, занесенные в форму).

Далее показан пример реализации проверки, аналогичной той, что мы делали в *разд. 4.8.4* (описание товара должно быть занесено, а значение цены должно быть неотрицательным).

```
from django.core.exceptions import ValidationError

class BbForm(forms.ModelForm):
    . . .
    def clean(self):
        super().clean()
        errors = {}
        if not self.cleaned_data['content']:
            errors['content'] = ValidationError(
                                'Укажите описание продаваемого товара')
        if self.cleaned_data['price'] < 0:
            errors['price'] = ValidationError(
                              'Укажите неотрицательное значение цены')
        if errors:
            raise ValidationError(errors)
```

ГЛАВА 14

Наборы форм, связанные с моделями

Если обычная форма, связанная с моделью, позволяет работать лишь с одной записью, то *набор форм, связанный с моделью*, предоставляет возможность работы сразу с несколькими записями. Внешне он представляет собой группу форм, выведенных одна за другой, и в каждой такой форме отображается содержимое одной записи. Помимо того, там могут быть выведены «пустые» формы для добавления записей, а также специальные средства для переупорядочивания и удаления записей.

Можно сказать, что один набор форм, связанный с моделью, заменяет несколько страниц: страницу списка записей и страницы добавления, правки и удаления записей. Вот только, к сожалению, с наборами форм удобно работать лишь тогда, когда количество отображающихся в них записей модели невелико.

14.1. Создание наборов форм, связанных с моделями

Для создания наборов форм, связанных с моделями, применяется быстрое объявление посредством фабрики классов. В качестве фабрики классов выступает функция `modelformset_factory()` из модуля `django.forms`. Вот формат ее вызова:

```
modelformset_factory(<модель>[, form=<форма, связанная с моделью>][,
                     fields=None][, exclude=None][, labels=None][,
                     help_texts=None][, error_messages=None][,
                     field_classes=None][, widgets=None][, extra=1][,
                     can_order=False][, can_delete=False][,
                     min_num=None][, validate_min=False][,
                     max_num=None][, validate_max=False][,
                     formset=<набор форм, связанных с моделью>])
```

Параметров здесь очень много:

- первый, позиционный, — модель, на основе которой будет создан набор форм;
- `form` — *форма, связанная с моделью*, на основе которой будет создан набор форм. Если не указан, форма будет создана автоматически;
- `fields` — последовательность имен полей модели, которые должны быть включены в автоматически создаваемую для набора форму. Чтобы указать все поля модели, нужно присвоить этому параметру строку `"__all__"`;
- `exclude` — последовательность имен полей модели, которые, напротив, не должны включаться в форму, автоматически создаваемую для набора;

> ***Внимание!***
>
> В вызове функции `modelformset_factory()` должен присутствовать только один из следующих параметров: `form`, `fields`, `exclude`. Одновременное указание двух или более параметров приведет к ошибке.

- `labels` — надписи для полей формы. Указываются в виде словаря, ключи элементов которого определяют поля формы, а значения — надписи для них;
- `help_texts` — дополнительный поясняющий текст для полей формы. Указывается в виде словаря, ключи элементов которого определяют поля формы, а значения — поясняющие надписи для них;
- `error_messages` — строковые сообщения об ошибках. Задаются в виде словаря, ключи элементов которого определяют поля формы, а значениями элементов также должны быть словари. Во вложенных словарях ключи элементов определяют строковые коды ошибок (их можно найти в *разд. 4.8.2*), а значения укажут сообщения об ошибках;
- `field_classes` — типы полей формы, которыми будут представляться в создаваемой форме различные поля модели. Значением должен быть словарь, ключи элементов которого представляют имена полей модели, а значениями элементов станут ссылки на классы полей формы;
- `widgets` — элементы управления, с помощью которых в различные поля формы будут заноситься данные. Значение — словарь, ключи элементов которого представляют имена полей формы, а значениями элементов станут экземпляры классов, представляющих элементы управления, или ссылки на сами эти классы;

> ***Внимание!***
>
> Параметры `labels`, `help_texts`, `error_messages`, `field_classes` и `widgets` указываются только в том случае, если форма для набора создается автоматически (параметр `form` не указан). В противном случае все необходимые сведения о форме должны быть записаны в ее классе.

- `extra` — количество «пустых» форм, предназначенных для добавления новых записей, которые будут присутствовать в наборе (по умолчанию — 1);
- `can_order` — если `True`, посредством создаваемого набора форм можно переупорядочивать записи связанной с ним модели, если `False` — нельзя (поведение по умолчанию);

- `can_delete` — если `True`, посредством создаваемого набора форм можно удалять записи связанной с ним модели, если `False` — нельзя (поведение по умолчанию);
- `min_num` — минимальное количество форм в наборе, за вычетом помеченных на удаление (по умолчанию не ограничено);
- `validate_min` — если `True`, Django в процессе валидации будет проверять, не меньше ли количество форм в наборе значения, указанного в параметре `min_num`, если `False` — не будет (поведение по умолчанию). Если количество форм меньше указанного минимального, будет выведено сообщение об ошибке с кодом `"too_few_forms"` (этот код можно использовать для указания своего сообщения об ошибке, более подробно об этом см. в *разд. 4.8.2*);
- `max_num` — максимальное количество форм в наборе, за вычетом помеченных на удаление (по умолчанию не ограничено);
- `validate_max` — если `True`, Django в процессе валидации будет проверять, не превосходит ли количество форм в наборе значение, указанное в параметре `max_num`, если `False` — не будет (поведение по умолчанию). Если количество форм больше указанного максимального, будет выведено сообщение об ошибке с кодом `"too_many_forms"`;
- `formset` — *набор форм, связанный с моделью*, на основе которого будет создан новый набор форм. Заданный в параметре *набор форм* может указывать какие-либо общие для целой группы наборов форм параметры.

Рассмотрим код, который создает для работы со списком рубрик набор форм, имеющий минимальную функциональность, не позволяющий переупорядочивать, удалять записи и разрешающий добавить за один раз только одну рубрику:

```
from django.forms import modelformset_factory
from .models import Rubric

RubricFormSet = modelformset_factory(Rubric, fields=('name',))
```

Созданный этим кодом набор форм на экране будет выглядеть как простая последовательность форм, каждая из которых служит для правки одной записи модели (рис. 14.1).

А теперь рассмотрим код, создающий для правки рубрик набор форм, который позволяет переупорядочивать и удалять записи:

```
RubricFormSet = modelformset_factory(Rubric, fields=('name',),
                                     can_order=True, can_delete=True)
```

Набор форм с подобного рода расширенной функциональностью показан на рис. 14.2. Видно, что в составе каждой формы находятся поле ввода **Порядок** и флажок **Удалить**. В поле ввода заносится целочисленное значение, по которому записи модели могут быть отсортированы. А установка флажка приведет к тому, что после нажатия на кнопку отправки данных соответствующие записи будут удалены из модели.

Рис. 14.1. Набор форм с базовой функциональностью

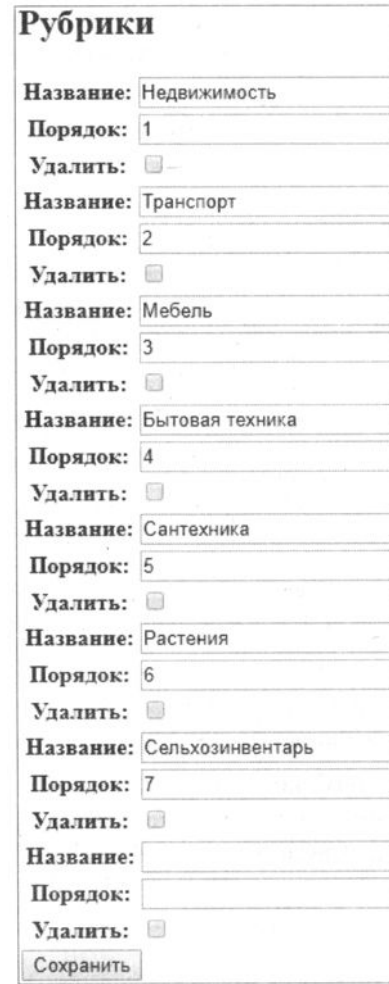


Рис. 14.2. Набор форм с расширенной функциональностью

14.2. Обработка наборов форм, связанных с моделями

Из *главы 10* мы знаем, что Django предлагает ряд высокоуровневых контроллеров-классов, выполняющих обработку форм самостоятельно, без нашего участия. Однако это не касается наборов форм — нам самим придется заниматься их обработкой.

14.2.1. Создание набора форм, связанного с моделью

Как и в случае формы, при получении GET-запроса нам нужно создать объект набора форм, добавить его в контекст шаблона и выполнить рендеринг соответст-

вующего шаблона. Таким способом мы выведем на экран страницу с набором форм, и посетитель сможет начать работу с приведенными в нем записями модели.

Создание экземпляра класса, представляющего набор форм, выполняется вызовом конструктора этого класса без параметров:

```
formset = RubricFormSet()
```

Конструктор класса набора форм поддерживает два необязательных параметра, которые могут нам пригодиться:

- `initial` — изначальные данные, которые будут помещены в «пустые» (предназначенные для добавления новых записей) формы. Значение параметра должно представлять собой последовательность, каждый элемент которой задаст изначальные значения для одной из «пустых» форм. Этим элементом должен выступать словарь, ключи элементов которого задают имена полей «пустой» формы, а значения элементов — изначальные значения для этих полей;
- `queryset` — набор записей, откуда будут взяты записи для вывода в наборе форм.

Для примера зададим в качестве изначального значения для поля названия в первой «пустой» форме строку `"Новая рубрика"`, во второй — строку `"Еще одна новая рубрика"` и сделаем так, чтобы в наборе форм выводились только первые пять рубрик:

```
formset = RubricFormSet(initial=[{'name': 'Новая рубрика'},
                                 {'name': 'Еще одна новая рубрика'}],
                        queryset=Rubric.objects.all()[0:5])
```

14.2.2. Повторное создание набора форм

Закончив работу, посетитель нажмет кнопку отправки данных, в результате чего веб-обозреватель отправит POST-запрос с данными, занесенными в набор форм. Этот запрос нам нужно обработать.

Прежде всего, нам следует создать набор форм повторно. Это выполняется так же, как и в случае формы, — вызовом конструктора класса с передачей ему единственного позиционного параметра, значением которого станет словарь из атрибута `POST` объекта запроса. Пример:

```
formset = RubricFormSet(request.POST)
```

Если при первом создании набора форм мы указали набор записей в параметре `queryset`, при повторном создании не забудем сделать то же самое:

```
formset = RubricFormSet(request.POST, queryset=Rubric.objects.all()[0:5])
```

14.2.3. Валидация и сохранение набора форм

Теперь мы можем выполнить валидацию и сохранение набора форм. Для этого мы воспользуемся знакомыми нам по *разд. 13.2* методами `is_valid()`, `save()` и `save_m2m()`, поддерживаемыми классом набора форм:

```
if formset.is_valid():
    formset.save()
```

Метод `save()` в качестве результата возвращает последовательность всех записей модели, что представлены в текущем наборе форм. Мы можем перебрать эту последовательность в цикле и выполнить над записями какие-либо действия (что может пригодиться при вызове метода `save()` с параметром `commit`, равным `False`).

После вызова метода `save()` мы можем воспользоваться тремя атрибутами, поддерживаемыми классом набора форм:

- `new_objects` — последовательность добавленных записей модели, связанной с текущим набором форм;
- `changed_objects` — последовательность исправленных записей модели, связанной с текущим набором форм;
- `deleted_objects` — последовательность удаленных записей модели, связанной с текущим набором форм.

Метод `save()` самостоятельно обрабатывает удаление записей (если при вызове конструктора набора форм был указан параметр `can_delete` со значением `True`). Если посетитель в форме установит флажок **Удалить**, соответствующая запись модели будет удалена.

Однако нужно помнить один важный момент. Если вызов метода `save()` содержит параметр `commit` со значением `False`, помеченные на удаление записи удалены не будут. Нам самим придется перебрать все удаленные записи, что приведены в списке из атрибута `deleted_objects`, и вызвать у каждой метод `delete()`:

```
formset.save(commit=False)
for rubric in formset.deleted_objects:
    rubric.delete()
```

14.2.4. Доступ к данным, занесенным в набор форм

Каждая форма, входящая в состав набора, поддерживает знакомый нам по *разд. 13.2.1.5* атрибут `cleaned_data`. Его значением является словарь, хранящий все данные, что были занесены в текущую форму, в виде объектов языка Python.

Сам набор форм поддерживает функциональность последовательности. На каждой итерации он возвращает очередную форму, входящую в его состав. Вот так можно перебрать в цикле входящие в набор формы:

```
for form in formset:
    # Что-либо делаем с формой и введенными в нее данными
```

Однако здесь нужно иметь в виду, что в наборе, возможно, будет присутствовать «пустая» форма, в которую не были занесены никакие данные. Такая форма будет хранить в атрибуте `cleaned_data` «пустой» словарь. Мы можем обработать такую ситуацию следующим образом:

```
for form in formset:
    if form.cleaned_data:
        # Форма не пуста, и мы можем получить занесенные в нее данные
```

Листинг 14.1 показывает полный код контроллера-функции, обрабатывающего набор форм и позволяющего удалять записи.

Листинг 14.1. Обработка набора форм, связанного с моделью

```
from django.shortcuts import render, redirect
from django.forms import modelformset_factory
from .models import Rubric

def rubrics(request):
    RubricFormSet = modelformset_factory(Rubric, fields=('name',),
                                         can_delete=True)
    if request.method == 'POST':
        formset = RubricFormSet(request.POST)
        if formset.is_valid():
            formset.save()
            return redirect('bboard:index')
    else:
        formset = RubricFormSet()
    context = {'formset': formset}
    return render(request, 'bboard/rubrics.html', context)
```

14.2.5. Реализация переупорядочивания записей

Метод `save()` набора форм обрабатывает удаление записей сам, без какого-либо нашего вмешательства. Чего не скажешь о переупорядочивании записей. И, к сожалению, этот момент не документирован в официальном руководстве по Django...

Переупорядочивание записей достигается тем, что в каждой форме, составляющей набор, автоматически создается целочисленное поле. Оно имеет имя, заданное в переменной `ORDERING_FIELD_NAME`, что объявлена в модуле `django.forms.formsets`. А хранится в этом поле целочисленная величина, указывающая порядковый номер текущей записи в последовательности.

При выводе набора форм в такие поля будут подставлены порядковые номера записей, начиная с 1. Меняя эти номера, посетитель может переупорядочивать записи.

Чтобы сохранить заданный порядок записей, нам нужно куда-то записать указанные для них значения порядковых номеров. Понятно, что для этого следует:

1. Добавить в модель поле целочисленного типа.
2. Указать порядок сортировки по значению этого поля (обычно по возрастанию — так будет логичнее).

Со всем этим мы справимся, т. к. получили все необходимые знания о моделях еще в *главах 4* и *7*. Так, в модели `Rubric` мы можем предусмотреть поле `order`:

```
class Rubric(models.Model):
    . . .
    order = models.SmallIntegerField(default=0, db_index=True)
    . . .
    class Meta:
        . . .
        ordering = ['order', 'name']
```

Досадно другое: набор форм не сохраняет порядковые номера в моделях, и нам придется реализовывать сохранение самостоятельно.

Полный код обрабатывающего набор форм контроллера-функции, позволяющего удалять и переупорядочивать записи, показан в листинге 14.2. Обратим внимание, как выполняется сохранение порядковых номеров записей в поле `order` модели `Rubric`.

Листинг 14.2. Обработка набора форм, позволяющего переупорядочивать записи

```
from django.shortcuts import render, redirect
from django.forms import modelformset_factory
from django.forms.formsets import ORDERING_FIELD_NAME
from .models import Rubric

def rubrics(request):
    RubricFormSet = modelformset_factory(Rubric, fields=('name',),
                                         can_order=True, can_delete=True)
    if request.method == 'POST':
        formset = RubricFormSet(request.POST)
        if formset.is_valid():
            for form in formset:
                if form.cleaned_data:
                    rubric = form.save(commit=False)
                    rubric.order = form.cleaned_data[ORDERING_FIELD_NAME]
                    rubric.save()
            return redirect('bboard:index')
    else:
        formset = RubricFormSet()
    context = {'formset': formset}
    return render(request, 'bboard/rubrics.html', context)
```

14.3. Вывод наборов форм на экран

В целом, вывод наборов форм на экран выполняется так же и теми же средствами, что и вывод обычных форм (см. *разд. 13.3*).

14.3.1. Быстрый вывод наборов форм

Быстрый вывод наборов форм выполняется тремя следующими методами:

- `as_p()` — вывод по абзацам. Надпись для элемента управления и сам элемент управления каждой формы, что входит в состав набора, выводятся в отдельном абзаце и отделяются друг от друга пробелами. Пример использования:

```
<form method="post">
    {% csrf_token %}
    {{ formset.as_p }}
    <input type="submit" value="Сохранить">
</form>
```

- `as_ul()` — вывод в виде маркированного списка. Надпись для элемента управления и сам элемент управления каждой формы, что входит в состав набора, выводятся в отдельном пункте списка и отделяются друг от друга пробелами. Теги `<ul>` и `</ul>`, создающие сам список, не формируются. Пример:

```
<form method="post">
    {% csrf_token %}
    <ul>
        {{ formset.as_ul }}
    </ul>
    <input type="submit" value="Сохранить">
</form>
```

- `as_table()` — вывод в виде таблицы. Таблица содержит два столбца: в левом выводятся надписи для элементов управления, в правом — сами элементы управления каждой формы из набора. Каждая пара «надпись — элемент управления» занимает отдельную строку таблицы. Теги `<table>` и `</table>`, создающие саму таблицу, не выводятся. Пример:

```
<form method="post">
    {% csrf_token %}
    <table>
        {{ formset.as_table }}
    </table>
    <input type="submit" value="Сохранить">
</form>
```

Мы можем просто указать переменную, хранящую набор форм, — в этом случае будет автоматически вызван метод `as_table()`:

```
<form method="post">
    {% csrf_token %}
    <table>
        {{ formset }}
    </table>
    <input type="submit" value="Сохранить">
</form>
```

Парный тег <form>, создающий саму форму, в этом случае не создается, равно как и кнопка отправки данных. Также не забываем вставить в форму тег шаблонизатора csrf_token, который создаст скрытое поле с электронным жетоном безопасности.

14.3.2. Расширенный вывод наборов форм

Инструментов для расширенного вывода наборов форм Django предоставляет совсем немного.

Прежде всего, это атрибут management_form, поддерживаемый всеми классами наборов форм. Он хранит ссылку на служебную форму, входящую в состав набора и хранящую необходимые для работы класса служебные данные.

Далее, не забываем, что набор форм поддерживает функциональность итератора, возвращающего на каждом проходе очередную входящую в него форму.

И, наконец, метод non_form_errors() набора форм возвращает список сообщений об ошибках, относящихся ко всему набору.

Исходя из этого, мы можем написать шаблон bboard\rubrics.html, используемый в контроллерах из листингов 14.1 и 14.2. Код этого шаблона показан в листинге 14.3.

Листинг 14.3. Вывод набора форм в шаблоне

```
<form method="post">
    {% csrf_token %}
    {{ formset.management_form }}
    {% if formset.non_form_errors %}
    <ul>
        {% for error in formset.non_form_errors %}
        <li><em>{{ error|escape }}</em></li>
        {% endfor %}
    </ul>
    {% endif %}
    {% for form in formset %}
        {% for hidden in form.hidden_fields %}
            {{ hidden }}
        {% endfor %}
        {% if form.non_field_errors %}
        <ul>
            {% for error in form.non_field_errors %}
            <li><em>{{ error|escape }}</em></li>
            {% endfor %}
        </ul>
        {% endif %}
        {% for field in form.visible_fields %}
            {% if field.errors %}
            <ul>
                {% for error in field.errors %}
```

```
                <li><em>{{ error|escape }}</em></li>
                {% endfor %}
            </ul>
            {% endif %}
            <p>{{ field.label_tag }}<br>{{ field }}<br>
            {{ field.help_text }}</p>
        {% endfor %}
    {% endfor %}
    <input type="submit" value="Сохранить">
</form>
```

14.4. Валидация в наборах форм

Если уж зашла речь о валидации, ее удобнее всего реализовать:

- в модели, с которой связан набор форм;
- в форме, связанной с моделью. Эту форму следует указать в вызове функции `modelformset_factory()`, в параметре `form`.

В самом наборе форм имеет смысл выполнять валидацию лишь тогда, когда необходимо проверить на правильность весь массив данных, введенных в этот набор.

Валидация в наборе форм реализуется так:

- объявляется класс, производный от класса `BaseModelFormSet` из модуля `django.forms`;
- в этом классе переопределяется метод `clean()`, в котором и выполняется валидация. Этот метод должен удовлетворять тем же требованиям, что и одноименный метод класса формы, связанной с моделью (см. *разд. 13.4.2*);
- выполняется создание класса набора форм путем использования функции `modelformset_factory()`. Объявленный ранее класс указывается в параметре `formset` вызова этой функции.

Атрибут `forms`, унаследованный от класса `BaseModelFormSet`, хранит последовательность всех форм, что имеются в наборе.

Сообщения об ошибках, генерируемые таким валидатором, будут присутствовать в списке ошибок, относящихся ко всему набору форм (возвращается методом `non_form_errors()`).

Вот пример кода, выполняющего валидацию на уровне набора форм и требующего обязательного присутствия рубрик «Недвижимость», «Транспорт» и «Мебель»:

```
class RubricBaseFormSet(BaseModelFormSet):
    def clean(self):
        super().clean()
        names = [form.cleaned_data['name'] for form in self.forms \
                 if 'name' in form.cleaned_data]
```

```
        if ('Недвижимость' not in names) or ('Транспорт' not in names) \
           or ('Мебель' not in names):
            raise ValidationError('Добавьте рубрики недвижимости, ' +\
                                  'транспорта и мебели')
. . .
def rubrics(request):
    RubricFormSet = modelformset_factory(Rubric, fields=('name',),
                    can_order=True, can_delete=True,
                    formset=RubricBaseFormSet)
    . . .
```

14.5. Встроенные наборы форм

Встроенные наборы форм служат для работы с наборами записей вторичной модели, связанными с указанной записью первичной модели.

14.5.1. Создание встроенных наборов форм

Для создания встроенных наборов форм применяется функция `inlineformset_factory()` из модуля `django.forms`. Вот формат ее вызова:

```
inlineformset_factory(<первичная модель>, <вторичная модель>[,
                      form=<форма, связанная с моделью>][,
                      fk_name=None][, fields=None][, exclude=None][,
                      labels=None][, help_texts=None][,
                      error_messages=None][, field_classes=None][,
                      widgets=None][, extra=3][,
                      can_order=False][, can_delete=True][,
                      min_num=None][, validate_min=False][,
                      max_num=None][, validate_max=False][,
                      formset=<набор форм, связанных с моделью>])
```

В первом позиционном параметре указывается ссылка на класс первичной модели, а во втором позиционном параметре — ссылка на класс вторичной модели.

Параметр `fk_name` указывает имя ключевого поля вторичной модели, по которому устанавливается связь с первичной моделью. Он приводится только в том случае, если во вторичной модели установлено более одной связи с первичной моделью, и требуется выбрать, какую именно связь нужно использовать.

Остальные параметры имеют такое же назначение, что и у функции `modelformset_factory()` (см. *разд. 14.1*). Отметим только, что у параметра `extra` (задает количество «пустых» форм) значение по умолчанию 3, а у параметра `can_delete` (указывает, можно ли удалять связанные записи) — `True`.

14.5.2. Обработка встроенных наборов форм

Обработка встроенных наборов форм выполняется так же, как и обычных наборов форм, связанных с моделями. Единственное исключение: при создании объекта набора форм, как в первый, так и во второй раз, нужно передать конструктору с пара-

метром `instance` запись первичной модели. После этого набор форм выведет записи вторичной модели, связанные с этой записью.

В листинге 14.4 приведен код контроллера, который выводит на экран страницу со встроенным набором форм. Этот набор форм показывает все объявления, относящиеся к выбранной посетителем рубрике, и позволяет править и удалять их. В нем используется форма `BbForm`, написанная нами в *главе 2* (см. листинг 2.6).

Листинг 14.4. Применение встроенного набора форм

```
from django.shortcuts import render, redirect
from django.forms import inlineformset_factory

from .models import Bb, Rubric
from .forms import BbForm

def bbs(request, rubric_id):
    BbsFormSet = inlineformset_factory(Rubric, Bb, form=BbForm, extra=1)
    rubric = Rubric.objects.get(pk=rubric_id)
    if request.method == 'POST':
        formset = BbsFormSet(request.POST, instance=rubric)
        if formset.is_valid():
            formset.save()
            return redirect('bboard:index')
    else:
        formset = BbsFormSet(instance=rubric)
    context = {'formset': formset, 'current_rubric': rubric}
    return render(request, 'bboard/bbs.html', context)
```

Вывод встроенного набора форм на экран выполняется с применением тех же программных инструментов, что и вывод обычного набора форм, связанного с моделью (см. *разд. 14.3*).

И точно такими же средствами мы можем реализовать во встроенном наборе форм валидацию. Единственное исключение: объявляемый для этого класс должен быть производным от класса `BaseInlineFormSet` из модуля `django.forms`.

ГЛАВА 15

Разграничение доступа: базовые инструменты

К внутренним данным сайта, хранящимся в его информационной базе, не следует допускать кого попало. К ним можно допускать только тех посетителей, кто записан в особом списке — *зарегистрированных пользователей*, или просто *пользователей*. Нужно также учитывать, что какому-либо пользователю может быть запрещено работать с определенными данными — иначе говоря, принимать во внимание *права*, или *привилегии* пользователя.

Допущением или недопущением посетителей к работе с внутренними данными сайта на основе того, зарегистрирован ли он в *списке пользователей*, и его прав, в Django-сайте занимается подсистема *разграничения доступа*.

15.1. Как работает подсистема разграничения доступа

Если посетитель желает получить доступ к внутренним данным сайта (например, чтобы добавить объявление или создать новую рубрику), он предварительно должен войти на особую веб-страницу и занести в представленную там форму свои регистрационные данные: имя, под которым он был зарегистрирован, и пароль. Подсистема разграничения доступа проверит, записан ли пользователь с такими именем и паролем в списке пользователей (т. е. является ли он зарегистрированным пользователем). Если такой пользователь в списке обнаружился, подсистема помечает его как выполнившего процедуру *аутентификации*, или *входа* на сайт. В противном случае посетитель получит сообщение о том, что его в списке нет, и данные сайта ему недоступны.

Когда посетитель пытается попасть на страницу для работы с внутренними данными сайта, подсистема разграничения доступа проверяет, выполнил ли он процедуру входа и имеет ли он права на работу с этими данными, — выполняет *авторизацию*. Если посетитель выполнил вход и имеет необходимые права (прошел авторизацию), он допускается к странице, в противном случае получает соответствующее сообщение.

Закончив работу с внутренними данными сайта, пользователь выполняет процедуру *выхода* с сайта. В процессе выполнения выхода подсистема разграничения доступа помечает его как не выполнившего вход. После чего, чтобы в очередной раз добраться до внутренних данных сайта, посетитель вновь должен выполнить процедуру входа.

Но как посетитель может добавить себя в список пользователей? Есть два варианта. Во-первых, посетителей может заносить в список (выполнять их *регистрацию*) один из пользователей, имеющих права на работу с этим списком, — такое обычно практикуется в корпоративных решениях с ограниченным кругом пользователей. Во-вторых, посетитель сможет занести себя в список самостоятельно, зайдя на страницу регистрации и введя необходимые данные — в первую очередь, свои имя и пароль. Этот способ применяется на общедоступных интернет-ресурсах.

На тот случай, если какой-либо из зарегистрированных пользователей забыл свой пароль, на сайтах часто предусматривают процедуру *восстановления пароля*. Забывчивый пользователь заходит на особую страницу и вводит свой адрес электронной почты. Подсистема разграничения доступа ищет в списке пользователя с таким адресом и отправляет ему особое письмо с гиперссылкой, ведущей на страницу, где пользователь сможет задать новый пароль.

15.2. Подготовка подсистемы разграничения доступа

Прежде чем задействовать на своем сайте подсистему разграничения доступа, нам необходимо выполнить некоторые подготовительные действия.

15.2.1. Настройка подсистемы разграничения доступа

Настройки подсистемы разграничения доступа записываются, как обычно, в модуле settings.py пакета конфигурации.

Чтобы эта подсистема успешно работала, нужно сделать следующее:

- проверить, записаны ли в списке зарегистрированных в проекте приложений (параметр `INSTALLED_APPS`) приложения `django.contrib.auth` и `django.contrib.contenttypes`;
- проверить, записаны ли в списке зарегистрированных посредников (параметр `MIDDLEWARE`) посредники `django.contrib.sessions.middleware.SessionMiddleware` и `django.contrib.auth.middleware.AuthenticationMiddleware`.

Впрочем, во вновь созданном проекте все эти приложения и посредники уже занесены в соответствующие списки.

Кроме того, на работу подсистемы влияют следующие параметры:

- `LOGIN_URL` — интернет-адрес, на который будет выполнено перенаправление после попытки попасть на страницу, закрытую от неавторизованных посетителей

(*гостей*). Здесь можно указать как собственно интернет-адрес, так и имя маршрута (см. *разд. 8.3*). Значение по умолчанию: `"/accounts/login/"`.

Обычно в этом параметре указывается интернет-адрес, ведущий на страницу входа на сайт;

- `LOGIN_REDIRECT_URL` — интернет-адрес, на который будет выполнено перенаправление после успешного входа на сайт. Здесь можно указать как собственно интернет-адрес, так и имя маршрута. Значение по умолчанию: `"/accounts/profile/"`.

  Если переход на страницу входа на сайт был вызван попыткой попасть на страницу, закрытую от неавторизованных посетителей, Django автоматически выполнит перенаправление на страницу входа, добавив к ее интернет-адресу GET-параметр `next`, в котором запишет интернет-адрес страницы, на которую хотел попасть посетитель. После успешного входа будет выполнено перенаправление на интернет-адрес, сохраненный в этом GET-параметре. Если же такого параметра обнаружить не удалось, перенаправление выполнится на интернет-адрес из параметра `LOGIN_REDIRECT_URL`;

- `LOGOUT_REDIRECT_URL` — интернет-адрес, на который будет выполнено перенаправление после успешного выхода с сайта. Можно указать как собственно интернет-адрес, так и имя маршрута.

  Также можно указать значение `None` (это, кстати, значение параметра по умолчанию). В таком случае никакого перенаправления выполняться не будет, а будет выведена страница выхода с сайта;

- `PASSWORD_RESET_TIMEOUT_DAYS` — количество дней, в течение которых будет действителен интернет-адрес сброса пароля, отправляемый посетителю в электронном письме.

Здесь мы рассмотрели не все параметры подсистемы разграничения доступа Django. Еще с несколькими параметрами, касающимися работы низкоуровневых механизмов аутентификации и авторизации, мы познакомимся в *главе 20*.

> ***Внимание!***
>
> Перед использованием подсистемы разграничения доступа необходимо хотя бы раз провести процедуру выполнения миграций (за подробностями — к *разд. 5.3*). Это необходимо для того, чтобы Django создал в базе данных таблицы списков пользователей, групп и прав.

15.2.2. Создание суперпользователя

Один из пользователей, хранящихся в списке зарегистрированных пользователей, является *суперпользователем*. Такой пользователь имеет права на работу со всеми данными сайта, включая список пользователей.

Для создания суперпользователя применяется команда `createsuperuser` утилиты manage.py:

```
manage.py createsuperuser [--username <имя суперпользователя>]
[--email <адрес электронной почты>]
```

После отдачи этой команды утилита manage.py запросит имя создаваемого пользователя, его адрес электронной почты и пароль, который потребуется ввести дважды.

Поддерживаются два необязательных ключа командной строки:

- `--username` — задает *имя* создаваемого суперпользователя. Если указан, утилита manage.py не будет запрашивать имя;
- `--email` — задает *адрес электронной почты* суперпользователя. Если указан, утилита manage.py не будет запрашивать этот адрес.

15.2.3. Смена пароля пользователя

В процессе разработки сайта может потребоваться сменить пароль у какого-либо из хранящихся в списке пользователей (вследствие забывчивости или по иной причине). Для такого случая утилита manage.py предусматривает команду `changepassword`:

```
manage.py changepassword [<имя пользователя>]
```

После отдачи этой команды будет выполнена смена пароля пользователя с указанным *именем* или, если таковое не указано, текущего пользователя (выполнившего вход на сайт в данный момент). Новый пароль будет запрошен дважды — для надежности.

15.3. Работа со списками пользователей и групп

Административный веб-сайт Django предоставляет удобные средства для работы со списками пользователей и групп (разговор о них пойдет позже). Оба эти списка находятся в приложении **Пользователи и группы** на главной странице административного сайта (см. рис. 1.6).

15.3.1. Список пользователей

Для каждого пользователя из списка пользователей мы можем указать следующие сведения:

- имя, которое он будет вводить в соответствующее поле ввода в форме входа;
- пароль.

  При создании пользователя на странице будут присутствовать два поля для указания пароля. В эти поля нужно ввести один и тот же пароль (это сделано для надежности).

  При правке существующего пользователя вместо поля ввода пароля будет выведен сам пароль в закодированном виде. Сменить пароль будет можно, щелкнув на расположенной под закодированным паролем гиперссылке;

- настоящее имя (необязательно);
- настоящая фамилия (необязательно);
- адрес электронной почты;
- является ли пользователь активным. Только *активные* пользователи могут выполнять вход на сайт;
- имеет ли пользователь статус персонала. Только пользователи со статусом *персонала* имеют доступ к административному сайту Django. Однако для получения доступа к страницам, не относящимся к административному сайту, в том числе закрытым для гостей, статус персонала не нужен;
- является ли пользователь суперпользователем;
- список прав, имеющихся у пользователя.

Для указания списка прав используется элемент управления в виде двух списков и четырех кнопок (рис. 15.1).

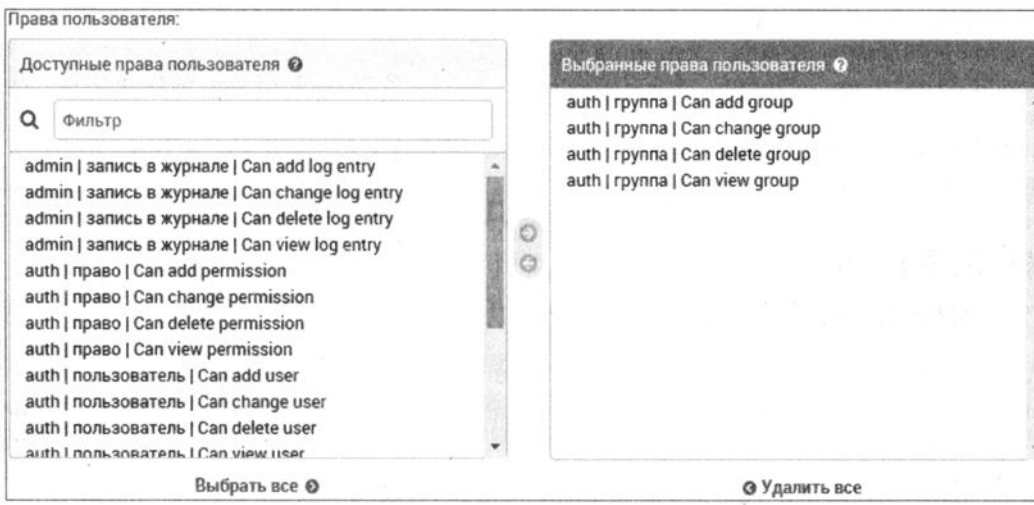


Рис. 15.1. Элемент управления для указания списка прав пользователя

В левом списке выводятся все доступные для пользователей права. Представляющие их пункты списка имеют следующий вид:

<приложение>* | *<модель>* | Can *<операция>* *<модель>

Приложение всегда выводится в виде своего псевдонима. *Модель* выводится либо в виде имени класса, либо как заданное для него название (указывается в параметре `verbose_name` модели, описанном в *разд. 4.5*). *Операция* обозначается словом **view** (просмотр), **add** (добавление), **change** (правка) или **delete** (удаление).

Рассмотрим несколько примеров:

- **auth | пользователь | Can add user** — право добавлять пользователей (**auth** — это псевдоним приложения, реализующего систему разграничения доступа);

- **auth | группа | Can delete group** — право удалять группы пользователей;
- **bboard | Объявление | Can add bb** — право добавлять объявления;
- **bboard | Рубрика | Can change Рубрика** — право править рубрики.

Вернемся к элементу управления с рис. 15.1. В правом списке показываются права, уже имеющиеся у пользователя. Выводятся они в точно таком же виде, что и права из левого списка.

Оба списка: и левый, и правый — предоставляют возможность выбора произвольного количества пунктов:

- чтобы предоставить пользователю какие-либо права, следует выбрать их в левом списке и нажать расположенную между списками кнопку со стрелкой вправо;
- чтобы удалить права, данные пользователю ранее, нужно выбрать их в правом списке и нажать расположенную между списками кнопку со стрелкой влево;
- чтобы дать пользователю какое-либо одно право, достаточно найти его в левом списке и щелкнуть на нем двойным щелчком;
- чтобы удалить у пользователя какое-либо одно право, следует найти его в правом списке и щелкнуть на нем двойным щелчком;
- чтобы дать пользователю все доступные права, нужно нажать находящуюся под левым списком кнопку **Выбрать все**;
- чтобы удалить у пользователя все права, нужно нажать находящуюся под правым списком кнопку **Удалить все**.

> ***Внимание!***
>
> Пользователь может выполнять только те операции над внутренними данными сайта, на которые он явно получил права. Модели, на которые он не имеет никаких прав, при этом вообще не будут отображаться в административном сайте.
>
> Однако суперпользователь может выполнять любые операции над любыми моделями, независимо от того, какие права он имеет.

15.3.2. Группы пользователей. Список групп

На сайтах с большим количеством зарегистрированных пользователей, выполняющих разные задачи и имеющих для этого разные права, для быстрого указания прав у пользователей можно включать последних в *группы*. Каждая такая группа объединяет произвольное количество пользователей и задает для них одинаковый набор прав. Любой пользователь может входить в любое количество групп.

Для каждой группы на Django-сайте мы указываем ее имя и список прав, которые будут иметь входящие в группу пользователи. Список прав для группы задается точно так же и с помощью точно такого же элемента управления, что и аналогичный параметр у отдельного пользователя (см. *разд. 15.3.1*).

Для указания групп, в которые входит пользователь, применяется такой же элемент управления.

Внимание!

При проверке прав, предоставленных пользователю, принимаются в расчет как права, заданные непосредственно для него, так и права всех групп, в которые он входит.

На заметку

Для своих нужд Django создает в базе данных таблицы `auth_user` (список пользователей), `auth_group` (список групп), `auth_permission` (список прав), `auth_user_groups` (связующая таблица, реализующая связь «многие-со-многими» между списками пользователей и групп), `auth_user_user_permissions` (связующая между списками пользователей и прав) и `auth_group_permissions` (связующая между списками групп и прав).

15.4. Аутентификация и служебные процедуры

Заполнив списки группами и пользователями, мы можем начать реализацию аутентификации, т. е. входа на сайт, и различных служебных процедур (выхода, смены и сброса пароля). Для этого Django предусматривает несколько удобных в использовании контроллеров-классов, объявленных в модуле `django.contrib.auth.views`.

15.4.1. Контроллер *LoginView*: вход на сайт

Контроллер-класс `LoginView`, наследующий от `FormView` (см. *разд. 10.5.1.3*), реализует вход на сайт. При получении запроса по HTTP-методу GET он выводит на экран страницу входа с формой, в которую следует занести имя и пароль пользователя. При получении POST-запроса (т. е. после отправки формы) он ищет в списке пользователя с указанными именем и паролем. Если такой пользователь обнаружился, выполняется перенаправление по интернет-адресу, взятому из GET- или POST-параметра `next`, или, если такой параметр отсутствует, из параметра `LOGIN_REDIRECT_URL` настроек проекта. Если же подходящего пользователя не нашлось, страница входа выводится повторно.

Класс `LoginView` поддерживает следующие атрибуты:

- `template_name` — путь к шаблону страницы входа в виде строки (значение по умолчанию: `"registration/login.html"`);
- `redirect_field_name` — имя GET- или POST-параметра, из которого будет извлекаться интернет-адрес для перенаправления после успешного входа, в виде строки (значение по умолчанию: `"next"`);
- `redirect_authenticated_user` — если `True`, пользователи, уже выполнившие вход, при попытке попасть на страницу входа будут перенаправлены по интернет-адресу, взятому из GET- или POST-параметра `next` или параметра `LOGIN_REDIRECT_URL` настроек проекта. Если `False`, пользователи, выполнившие вход, все же смогут попасть на страницу входа.

Значение этого атрибута по умолчанию — `False`, и менять его на `True` следует с осторожностью, т. к. это может вызвать нежелательные эффекты. В частности, если пользователь попытается попасть на страницу с данными, для работы с которыми у него нет прав, он будет перенаправлен на страницу входа. Но если атрибут `redirect_authenticated_user` имеет значение `True`, сразу же после этого будет выполнено перенаправление на страницу, с которой пользователь попал на страницу входа. В результате возникнет зацикливание, которое закончится аварийным завершением работы сайта;

- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `success_url_allowed_hosts` — множество, задающее хосты, на которые можно выполнить перенаправление после успешного входа, в дополнение к текущему хосту (значение по умолчанию — «пустое» множество);
- `authentication_form` — ссылка на класс формы входа (значение по умолчанию — класс `AuthenticationForm` из модуля `django.contrib.auth.forms`).

Странице входа в составе контекста шаблона передаются следующие переменные:

- `form` — форма для ввода имени и пароля;
- `next` — интернет-адрес, на который будет выполнено перенаправление после успешного входа.

> ***Внимание!***
>
> Шаблон *registration\login.html* изначально не существует ни в одном из зарегистрированных в проекте приложений, включая встроенные во фреймворк. Поэтому мы можем просто создать такой шаблон в одном из своих приложений.
>
> Шаблоны, указанные по умолчанию во всех остальных контроллерах-классах, что будут рассмотрены в этом разделе, уже существуют во встроенном приложении `django.contrib.admin`, реализующем работу административного сайта Django. Поэтому в остальных контроллерах нам придется указать другие имена шаблонов (в противном случае будут использоваться шаблоны административного сайта).

Чтобы реализовать на своем сайте процедуру аутентификации (входа), нам достаточно добавить в список маршрутов уровня проекта (он объявлен в модуле urls.py пакета конфигурации) такой элемент:

```
from django.contrib.auth.views import LoginView

urlpatterns = [
    . . .
    path('accounts/login/', LoginView.as_view(), name='login'),
]
```

А код простейшего шаблона страницы входа registration\login.html можно увидеть в листинге 15.1.

Листинг 15.1. Код шаблона страницы входа

```
{% extends "layout/basic.html" %}

{% block title %}Вход{% endblock %}

{% block content %}
<h2>Вход</h2>
{% if user.is_authenticated %}
<p>Вы уже выполнили вход.</p>
{% else %}
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="hidden" name="next" value="{{ next }}">
    <input type="submit" value="Войти">
</form>
{% endif %}
{% endblock %}
```

Поскольку разработчики Django предостерегают от автоматического перенаправления со страницы входа пользователей, уже выполнивших вход, нам придется не допускать выполнения ими повторного входа другими средствами. Для этого в контекст любого шаблона помещается переменная `user`, хранящая объект текущего пользователя. Мы можем обратиться к атрибуту `is_authenticated` этого объекта, и он вернет `True`, если пользователь уже вошел на сайт, и `False`, если еще нет. Исходя из этого, мы выводим либо текст, сообщающий, что пользователь уже вошел на сайт, или форму для входа.

И не забываем создать в форме входа скрытое поле с именем `next` и указать ему в качестве значения интернет-адрес, на который будет выполнено перенаправление при успешном входе. Это значение позже понадобится контроллеру.

15.4.2. Контроллер *LogoutView*: выход с сайта

Контроллер-класс `LogoutView`, наследующий от `TemplateView` (см. *разд. 10.2.4*), реализует выход с сайта. При получении GET-запроса он выполняет выход с сайта. Затем он производит перенаправление на интернет-адрес, указанный в GET-параметре `next` или, если такового нет, в атрибуте `next_page`. Если значение этого атрибута равно `None`, он выводит страницу с сообщением об успешном выходе.

Класс поддерживает атрибуты:

- `next_page` — интернет-адрес, на который будет выполнено перенаправление после успешного выхода с сайта (значение по умолчанию берется из параметра `LOGOUT_REDIRECT_URL` настроек сайта). Также можно указать имя нужного маршрута.

Если задать этому атрибуту значение None, перенаправление выполняться не станет, а вместо этого на экран будет выведена страница с сообщением об успешном выходе;

- template_name — путь к шаблону страницы сообщения об успешном выходе в виде строки (значение по умолчанию: "registration/logged_out.html").

  Эта страница будет выведена только в том случае, если в текущем интернет-адресе отсутствует GET-параметр next, и значение атрибута next_page равно None;

- redirect_field_name — имя GET-параметра, из которого будет извлекаться интернет-адрес для перенаправления после успешного выхода, в виде строки (значение по умолчанию: "next");

- extra_context — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;

- success_url_allowed_hosts — множество, задающее хосты, на которые можно выполнить перенаправление после успешного выхода, в дополнение к текущему хосту (значение по умолчанию — «пустое» множество).

В контексте шаблона создается переменная title, в которой хранится сообщение об успешном выходе.

> ***Внимание!***
>
> Шаблоны, указанные по умолчанию в этом и во всех остальных контроллерах-классах, рассматриваемых в этом разделе, существуют во встроенном приложении django.contrib.admin, реализующем работу административного сайта Django. Нам придется указать для этих классов другие имена шаблонов (в противном случае будут использоваться шаблоны административного сайта).

Реализовать выход можно добавлением в список маршрутов уровня проекта элемента следующего вида:

```
from django.contrib.auth.views import LogoutView

urlpatterns = [
    . . .
    path('accounts/logout/',
         LogoutView.as_view(next_page='bboard:index'), name='logout'),
]
```

Также можно не задавать интернет-адрес перенаправления или, как в нашем случае, имени маршрута, в самом контроллере, а записать его в настройках сайта:

```
path('accounts/logout/', LogoutView.as_view(), name='logout'),
. . .
LOGOUT_REDIRECT_URL = 'bboard:index'
```

15.4.3. Контроллер *PasswordChangeView*: смена пароля

Контроллер-класс `PasswordChangeView`, наследующий от класса `FormView`, выполняет смену пароля у текущего пользователя. При получении GET-запроса он выводит на экран страницу с формой, где нужно ввести старый пароль и, дважды, новый пароль. При получении POST-запроса он сохраняет введенный новый пароль и перенаправляет пользователя на страницу с сообщением об успешной смене пароля.

Вот атрибуты, поддерживаемые этим классом:

- `template_name` — путь к шаблону страницы с формой для смены пароля в виде строки (значение по умолчанию: `"registration/password_change_form.html"`);
- `success_url` — интернет-адрес, по которому будет выполнено перенаправление после успешной смены пароля (по умолчанию производится перенаправление на маршрут с именем `password_change_done`);
- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `form_class` — ссылка на класс формы для ввода нового пароля (значение по умолчанию — класс `PasswordChangeForm` из модуля `django.contrib.auth.forms`).

Странице смены пароля в составе контекста шаблона передаются следующие переменные:

- `form` — форма для ввода нового пароля;
- `title` — текст вида `"Смена пароля"`, который можно использовать в заголовке страницы.

Реализовать смену пароля можно добавлением в список маршрутов уровня проекта элемента:

```
from django.contrib.auth.views import PasswordChangeView

urlpatterns = [
    . . .
    path('accounts/password_change/', PasswordChangeView.as_view(
         template_name='registration/change_password.html'),
         name='password_change'),
]
```

15.4.4. Контроллер *PasswordChangeDoneView*: уведомление об успешной смене пароля

Контроллер-класс `PasswordChangeDoneView`, наследующий от `TemplateView`, просто выводит страницу с уведомлением об успешной смене пароля.

Он поддерживает атрибуты:

- `template_name` — путь к шаблону страницы с уведомлением в виде строки (значение по умолчанию: `"registration/password_change_done.html"`);
- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст.

В контексте шаблона создается переменная `title`, в которой хранится текст уведомления.

Чтобы на нашем сайте выводилось уведомление о смене пароля, мы добавим в список маршрутов уровня проекта элемент:

```
from django.contrib.auth.views import PasswordChangeDoneView

urlpatterns = [
    . . .
    path('accounts/password_change/done/',
         PasswordChangeDoneView.as_view(
         template_name='registration/password_changed.html'),
         name='password_change_done'),
]
```

15.4.5. Контроллер *PasswordResetView*: отправка письма для сброса пароля

Контроллер-класс `PasswordResetView`, производный от `FormView`, инициирует процедуру сброса пароля. При получении GET-запроса он выводит страницу с формой, в которую пользователю нужно занести свой адрес электронной почты. После получения POST-запроса он проверит существование этого адреса в списке пользователей и, если такой адрес есть, отправит по нему электронное письмо с гиперссылкой на страницу собственно сброса пароля.

Этот класс поддерживает следующие атрибуты:

- `template_name` — путь к шаблону страницы с формой для ввода адреса в виде строки (значение по умолчанию: `"registration/password_reset_form.html"`);
- `subject_template_name` — путь к шаблону темы электронного письма (значение по умолчанию: `"registration/password_reset_subject.txt"`);
- `email_template_name` — путь к шаблону тела электронного письма в формате обычного текста (значение по умолчанию: `"registration/password_reset_email.html"`);
- `html_email_template_name` — путь к шаблону тела электронного письма в формате HTML. Если `None` (это значение по умолчанию), письмо в формате HTML отправляться не будет;
- `success_url` — интернет-адрес, по которому будет выполнено перенаправление после успешной отправки электронного письма (по умолчанию производится перенаправление на маршрут с именем `password_reset_done`);

- `from_email` — адрес электронной почты отправителя, который будет вставлен в отправляемое письмо (значение по умолчанию берется из параметра `DEFAULT_FROM_EMAIL` настроек проекта);
- `extra_context` — дополнительное содержимое контекста шаблона для страницы с формой. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `extra_email_context` — дополнительное содержимое контекста шаблона для электронного письма. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `form_class` — ссылка на класс формы для ввода адреса (значение по умолчанию — класс `PasswordResetForm` из модуля `django.contrib.auth.forms`);
- `token_generator` — экземпляр класса, выполняющего формирование электронного жетона безопасности, который будет включен в интернет-адрес, ведущий на страницу собственно сброса пароля (по умолчанию — экземпляр класса `PasswordResetTokenGenerator` из модуля `django.contrib.auth.tokens`).

Странице отправки письма для сброса пароля в составе контекста шаблона передаются следующие переменные:

- `form` — форма для ввода адреса электронной почты;
- `title` — текст вида `"Сброс пароля"`, который можно использовать в заголовке страницы.

Что касается контекста шаблона для темы и тела электронного письма, то в нем создаются такие переменные:

- `protocol` — обозначение протокола (`"http"` или `"https"`);
- `domain` — строка с комбинацией IP-адреса (или доменного имени, если его удастся определить) и номера TCP-порта, через который работает веб-сервер;
- `uid` — закодированный ключ пользователя;
- `token` — электронный жетон безопасности, выступающий в качестве электронной подписи;
- `email` — адрес электронной почты пользователя, по которому высылается это письмо;
- `user` — текущий пользователь, представленный экземпляром класса `User`.

Сброс пароля реализуется добавлением в список маршрутов уровня проекта таких элементов:

```
from django.contrib.auth.views import PasswordResetView

urlpatterns = [
    . . .
    path('accounts/password_reset/',
         PasswordResetView.as_view(
         template_name='registration/reset_password.html',
```

```
        subject_template_name='registration/reset_subject.txt',
        email_template_name='registration/reset_email.html'),
        name='password_reset'),
]
```

Листинг 15.2 показывает код шаблона registration\reset_subject.txt, создающего тему электронного письма, а листинг 15.3 — код шаблона registration\reset_email.html, который создаст тело письма.

Листинг 15.2. Код шаблона, создающего тему электронного письма с гиперссылкой для сброса пароля

```
{{ user.username }}: запрос на сброс пароля
```

Листинг 15.3. Код шаблона, создающего тело электронного письма с гиперссылкой для сброса пароля

```
{% autoescape off %}
Уважаемый {{ user.username }}!
Вы отправили запрос на сброс пароля. Чтобы выполнить сброс, пройдите по
этому интернет-адресу:
{{ protocol}}://{{ domain }}{% url 'password_reset_confirm' ↳
uidb64=uid token=token %}

До свидания!
С уважением, администрация сайта "Доска объявлений".
{% endautoescape %}
```

В листинге 15.3 интернет-адрес для перехода на страницу собственно сброса пароля формируется путем обратного разрешения на основе маршрута с именем `password_reset_confirm`. Мы напишем этот маршрут чуть позже.

15.4.6. Контроллер *PasswordResetDoneView*: уведомление об отправке письма для сброса пароля

Контроллер-класс `PasswordResetDoneView`, наследующий от `TemplateView`, просто выводит страницу с уведомлением об успешной отправке электронного письма для сброса пароля.

Он поддерживает атрибуты:

- `template_name` — путь к шаблону страницы с уведомлением в виде строки (значение по умолчанию: `"registration/password_reset_done.html"`);
- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст.

В контексте шаблона создается переменная `title`, в которой хранится текст уведомления.

Чтобы на нашем сайте выводилось уведомление об отправке письма, мы добавим в список маршрутов уровня проекта такой элемент:

```
from django.contrib.auth.views import PasswordResetDoneView

urlpatterns = [
    . . .
    path('accounts/password_reset/done/',
         PasswordResetDoneView.as_view(
         template_name='registration/email_sent.html'),
         name='password_reset_done'),
]
```

15.4.7. Контроллер *PasswordResetConfirmView*: собственно сброс пароля

Контроллер-класс `PasswordResetConfirmView`, наследующий от `FormView`, выполняет собственно сброс пароля. Он запускается при переходе по интернет-адресу, отправленному в письме с сообщением о сбросе пароля. С URL-параметром `uidb64` он получает закодированный ключ пользователя, а с URL-параметром `token` — электронный жетон безопасности, значения обоих параметров должны быть строковыми. Получив GET-запрос, он выводит страницу с формой для задания нового пароля, а после получения POST-запроса производит смену пароля и выполняет перенаправление на страницу с уведомлением об успешном сбросе пароля.

Вот атрибуты, поддерживаемые этим классом:

- `template_name` — путь к шаблону страницы с формой для задания нового пароля в виде строки (значение по умолчанию: `"registration/password_reset_confirm.html"`);
- `post_reset_login` — если `True`, пользователь после занесения нового пароля автоматически выполнит вход на сайт, если `False`, автоматический вход выполнен не будет, и пользователю придется войти на сайт самостоятельно (значение по умолчанию — `False`);
- `success_url` — интернет-адрес, по которому будет выполнено перенаправление после успешной смены пароля (по умолчанию производится перенаправление на маршрут с именем `password_reset_complete`);
- `extra_context` — дополнительное содержимое контекста шаблона для страницы с формой. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `form_class` — ссылка на класс формы для ввода пароля (значение по умолчанию — класс `SetPasswordForm` из модуля `django.contrib.auth.forms`);
- `token_generator` — экземпляр класса, выполняющего формирование электронного жетона безопасности, который был включен в интернет-адрес, ведущий

на страницу собственно сброса пароля (по умолчанию — экземпляр класса `PasswordResetTokenGenerator` из модуля `django.contrib.auth.tokens`).

Странице ввода нового пароля в составе контекста шаблона передаются следующие переменные:

- `form` — форма для ввода нового пароля;
- `validlink` — если `True`, интернет-адрес, по которому пользователь попал на эту страницу, действителен и еще ни разу не использовался, если `False`, этот интернет-адрес скомпрометирован;
- `title` — текст вида `"Введите новый пароль"`, который можно использовать в заголовке страницы.

Вот такой элемент нужно добавить в список маршрутов уровня проекта, чтобы у нас заработал сброс пароля:

```
from django.contrib.auth.views import PasswordResetConfirmView

urlpatterns = [
    . . .
    path('accounts/reset/<uidb64>/<token>/',
         PasswordResetConfirmView.as_view(
         template_name='registration/confirm_password.html'),
         name='password_reset_confirm'),
]
```

15.4.8. Контроллер *PasswordResetCompleteView*: уведомление об успешном сбросе пароля

Контроллер-класс `PasswordResetCompleteView`, наследующий от `TemplateView`, выводит страницу с уведомлением об успешном сбросе пароля.

Он поддерживает атрибуты:

- `template_name` — путь к шаблону страницы с уведомлением в виде строки (значение по умолчанию: `"registration/password_reset_complete.html"`);
- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст.

В контексте шаблона создается переменная `title`, в которой хранится текст уведомления.

Чтобы на нашем сайте выводилось уведомление об успешном сбросе пароля, мы добавим в список маршрутов уровня проекта такой элемент:

```
from django.contrib.auth.views import PasswordResetCompleteView

urlpatterns = [
    . . .
```

```
    path('accounts/reset/done/',
         PasswordResetCompleteView.as_view(
         template_name='registration/password_confirmed.html'),
         name='password_reset_complete'),
]
```

15.5. Получение сведений о текущем пользователе

Любой зарегистрированный пользователь представляется в Django экземпляром класса `User` из модуля `django.contrib.auth.models`. Этот класс является моделью.

Мы можем получить доступ к экземпляру класса `User`, представляющего текущего пользователя:

- в контроллере — из атрибута `user` класса `Request`. Экземпляр этого класса, представляющий текущий запрос, передается контроллеру-функции и различным методам контроллера-класса через первый параметр, обычно имеющий имя `request`. Пример:

```
def index(request):
    # Проверяем, выполнил ли текущий пользователь вход
    if request.user.is_authenticated:
        . . .
```

- в шаблоне — через переменную контекста `user`, создаваемую обработчиком контекста `django.contrib.auth.context_processors.auth`:

```
{% if user.is_authenticated %}
<p>Вы уже выполнили вход.</p>
{% endif %}
```

Класс `User` поддерживает довольно большой набор полей, атрибутов и методов, с которыми нам нужно ознакомиться. Начнем знакомство с полей:

- `username` — имя пользователя, которое он вводит в форму входа. Обязательно к заполнению;
- `password` — пароль в закодированном виде. Обязательно к заполнению;
- `email` — адрес электронной почты;
- `first_name` — настоящее имя пользователя;
- `last_name` — настоящая фамилия пользователя;
- `is_active` — `True`, если пользователь является активным, и `False` в противном случае;
- `is_staff` — `True`, если пользователь имеет статус персонала, и `False` в противном случае;
- `is_superuser` — `True`, если пользователь является суперпользователем, и `False` в противном случае;

- `groups` — группы, в которые входит пользователь. Хранит диспетчер обратной связи, дающий доступ к записям связанной модели `Group` из модуля `django.contrib.auth.models`, в которой хранятся все группы;
- `last_login` — дата и время последнего входа на сайт;
- `date_joined` — дата и время регистрации пользователя на сайте.

Полезных атрибутов класс `User` поддерживает всего два:

- `is_authenticated` — возвращает `True`, если текущий пользователь выполнил вход, и `False`, если не выполнил (т. е. является гостем);
- `is_anonymous` — возвращает `True`, если текущий пользователь не выполнил вход на сайт (т. е. является гостем), и `False`, если выполнил.

Теперь рассмотрим методы этого класса:

- `has_perm(`*<право>*`[, obj=None])` — возвращает `True`, если текущий пользователь имеет указанное *право*, и `False` — в противном случае. *Право* задается в виде строки формата "*<приложение>*.*<операция>*_*<модель>*", где *приложение* указывается его псевдонимом, *операция* — строкой "view" (просмотр), "add" (добавление), "change" (правка) или "delete" (удаление), а *модель* — именем ее класса. Пример:

```
def index(request):
    # Проверяем, имеет ли текущий пользователь право добавлять рубрики
    if request.user.has_perm('bboard.add_rubric'):
        . . .
```

  Если в параметре `obj` указана запись модели, будут проверяться права пользователя на эту запись, а не на саму модель.

  Если пользователь неактивен, метод всегда возвращает `False`;

- `has_perms(`*<последовательность прав>*`[, obj=None])` — то же самое, что и `has_perm()`, но возвращает `True` только в том случае, если текущий пользователь имеет все права из заданной *последовательности*:

```
def index(request):
    # Проверяем, имеет ли текущий пользователь права добавлять,
    # править и удалять рубрики
    if request.user.has_perms(('bboard.add_rubric',
                    'bboard.change_rubric', 'bboard.delete_rubric')):
        . . .
```

- `get_username()` — возвращает имя пользователя, которое он набирает в форме входа на сайт;
- `get_full_name()` — возвращает строку, составленную из настоящих имени и фамилии пользователя, которые разделены пробелом;
- `get_short_name()` — возвращает настоящее имя пользователя.

Посетитель-гость представляется экземпляром класса `AnonymousUser` из того же модуля `django.contrib.auth.models`. Этот класс полностью аналогичен классу `User`,

поддерживает те же поля, атрибуты и методы. Следует только учитывать то, что гость не может выполнить вход и не имеет никаких прав.

15.6. Авторизация

Реализовав аутентификацию и научившись получать сведения о текущем пользователе, мы можем приступить к созданию *авторизации*, сделав часть страниц сайта недоступными для гостей, а часть — и для зарегистрированных пользователей с недостаточными правами.

15.6.1. Авторизация в контроллерах

Прежде всего, нам следует закрыть от нежелательных посетителей контроллеры, выводящие на экран страницы.

15.6.1.1. Императивный подход к авторизации

Самый простой подход к авторизации можно назвать императивным. Мы просто указываем Django, что нужно делать, если пользователь не выполнил вход или не имеет достаточных прав.

Рассмотрим пример, где мы проверяем, выполнил ли пользователь процедуру входа, и, если это не так, отправляем ему сообщение об ошибке 403 (доступ к запрошенной странице запрещен):

```
from django.http import HttpResponseForbidden

def rubrics(request):
    if request.user.is_authenticated:
        # Все в порядке: пользователь выполнил вход
        . . .
    else:
        return HttpResponseForbidden(
               'Вы не имеете допуска к списку рубрик')
```

Вместо отправки сообщения об ошибке мы можем перенаправлять посетителя на страницу входа:

```
def rubrics(request):
    if request.user.has_perms(('bboard.add_rubric',
                    'bboard.change_rubric', 'bboard.delete_rubric')):
        # Все в порядке: пользователь выполнил вход
        . . .
    else:
        return redirect('login')
```

Нам может пригодиться функция `redirect_to_login()`, объявленная в модуле `django.contrib.auth.views`. Формат ее вызова таков:

```
redirect_to_login(<интернет-адрес перенаправления>[,
                  redirect_field_name='next'][, login_url=None])
```

Первым параметром указывается *интернет-адрес*, на который нужно выполнить перенаправление после успешного входа. Обычно в качестве его задается интернет-адрес страницы с ограниченным доступом, на которую пытался попасть текущий пользователь.

Необязательный параметр `redirect_field_name` указывает имя для GET-параметра, передающего интернет-адрес страницы, на которую хотел попасть посетитель. По умолчанию используется имя `next`.

Параметр `login_url` задает интернет-адрес страницы входа или имя указывающего на нее маршрута. Если он опущен, задействуется интернет-адрес (или имя маршрута), записанный в параметре `LOGIN_URL` настроек проекта.

В качестве результата функция `redirect_to_login()` возвращает объект ответа, выполняющий перенаправление. Этот объект нужно вернуть из контроллера-функции.

Пример:

```
from django.contrib.auth.views import redirect_to_login
def rubrics(request):
    if request.user.is_authenticated:
        . . .
    else:
        return redirect_to_login(reverse('bboard:rubrics'))
```

15.6.1.2. Декларативная авторизация в контроллерах-функциях

Но часто удобнее оказывается декларативный подход к авторизации, когда мы говорим Django, что хотим допустить к какой-либо странице только пользователей, удовлетворяющих определенным критериям (в наиболее простом случае — выполнивших вход на сайт).

Декларативный подход реализуется набором декораторов, объявленных в модуле `django.contrib.auth.decorators`. Эти декораторы указываются у контроллера, выводящего страницу, доступ к которой мы собираемся ограничить. Всего этих декораторов три:

- `login_required([redirect_field_name='next'][,][login_url=None])` — допускает к странице только пользователей, выполнивших вход. Если пользователь не выполнил вход, выполняет перенаправление по интернет-адресу из параметра `LOGIN_URL` настроек проекта с передачей через GET-параметр `next` текущего интернет-адреса. Пример:

```
from django.contrib.auth.decorators import login_required
@login_required
def rubrics(request):
    . . .
```

Параметр `redirect_field_name` позволяет указать другое имя для GET-параметра, передающего интернет-адрес страницы, на которую хотел попасть посетитель. А параметр `login_url` позволит задать другой интернет-адрес страницы входа или другое имя указывающего на нее маршрута. Пример:

```
@login_required(login_url='/login/')
def rubrics(request):
    . . .
```

- `user_passes_test(<проверочная функция>[, redirect_field_name='next'][, login_url=None])` — допускает к странице только тех пользователей, кто выполнил вход и в чьем отношении *проверочная функция* вернет в качестве результата значение `True`. *Проверочная функция* должна принимать в качестве единственного параметра экземпляр класса `User`, представляющий текущего пользователя. Вот пример кода, допускающего к списку рубрик только пользователей, имеющих статус персонала:

```
from django.contrib.auth.decorators import user_passes_test
@user_passes_test(lambda user: user.is_staff)
def rubrics(request):
    . . .
```

Параметр `redirect_field_name` позволяет указать другое имя для GET-параметра, передающего интернет-адрес страницы, на которую хотел попасть посетитель. Параметр `login_url` позволит задать другой интернет-адрес страницы входа или другое имя указывающего на нее маршрута;

- `permission_required(<права>[, raise_exception=False][, login_url=None])` — допускает к странице только пользователей, имеющих заданные *права*. Права задаются в том же формате, что применяется в методах `has_perm()` и `has_perms()` (см. *разд. 15.5*). Можно указать:

  - одно право:

```
from django.contrib.auth.decorators import permission_required
@permission_required('bboard.view_rubric')
def rubrics(request):
    . . .
```

  - последовательность из произвольного количества прав. В этом случае у текущего пользователя должны иметься все указанные в последовательности права. Пример:

```
@permission_required(('bboard.add_rubric',
            'bboard.change_rubric', 'bboard.delete_rubric'))
def rubrics(request):
    . . .
```

Параметр `login_url` позволит задать другой интернет-адрес страницы входа или другое имя указывающего на нее маршрута.

Если параметру `raise_exception` присвоить значение `True`, декоратор вместо перенаправления пользователей, не выполнивших вход, на страницу входа будет

возбуждать исключение PermissionDenied, тем самым выводя страницу с сообщением об ошибке 403. Если это сообщение нужно выводить только для пользователей, выполнивших вход и не имеющих необходимых прав, следует применить декоратор permission_required() вместе с декоратором login_required():

```
@login_required
@permission_required(('bboard.add_rubric',
                      'bboard.change_rubric', 'bboard.delete_rubric'))
def rubrics(request):
    . . .
```

В этом случае пользователи, не выполнившие вход, попадут на страницу входа, а те из них, кто выполнил вход, но не имеет достаточных прав, получат сообщение об ошибке 403.

15.6.1.3. Декларативная авторизация в контроллерах-классах

Реализовать декларативный подход к авторизации в контроллерах-классах можно посредством трех классов-примесей, объявленных в модуле django.contrib.auth.mixins. Эти классы указываются в числе базовых в объявлении производных контроллеров-классов, причем в списках базовых классов примеси должны стоять первыми.

Все три класса-примеси являются производными от класса AccessMixin, объявленного в том же модуле. Он поддерживает ряд атрибутов и методов, предназначенных для указания важных параметров авторизации:

- login_url — атрибут, задает интернет-адрес или имя маршрута страницы входа (значение по умолчанию — None);
- get_login_url() — метод, должен возвращать интернет-адрес или имя маршрута страницы входа. В изначальной реализации возвращает значение атрибута login_url или, если оно равно None, значение параметра LOGIN_URL настроек проекта;
- permission_denied_message — атрибут, хранит строковое сообщение о возникшей ошибке (значение по умолчанию — «пустая» строка);
- get_permission_denied_message() — метод, должен возвращать сообщение об ошибке. В изначальной реализации возвращает значение атрибута permission_denied_message;
- redirect_field_name — атрибут, указывает имя GET-параметра, передающего интернет-адрес страницы с ограниченным доступом, на которую пытался попасть посетитель (значение по умолчанию: "next");
- get_redirect_field_name() — метод, должен возвращать имя GET-параметра, передающего интернет-адрес страницы, на которую пытался попасть посетитель. В изначальной реализации возвращает значение атрибута redirect_field_name;
- raise_exception — атрибут. Если его значение равно True, при попытке попасть на страницу гость или пользователь с недостаточными правами получит сооб-

щение об ошибке 403. Если значение параметра равно `False`, посетитель будет перенаправлен на страницу входа. Значение по умолчанию — `False`;

- `has_no_permission()` — метод, вызывается в том случае, если текущий пользователь не выполнил вход или не имеет необходимых прав, и на это нужно как-то отреагировать.

  В изначальной реализации, если значение атрибута `raise_exception` равно `True`, возбуждает исключение `PermissionDenied` с сообщением, возвращенным методом `get_permission_denied_message()`. Если же значение атрибута `raise_exception` равно `False`, выполняет перенаправление по интернет-адресу, возвращенному методом `get_login_url()`.

Теперь рассмотрим классы-примеси, производные от `AccessMixin`:

- `LoginRequiredMixin` — допускает к странице только пользователей, выполнивших вход:

```
from django.contrib.auth.mixins import LoginRequiredMixin
class BbCreateView(LoginRequiredMixin, CreateView):
    . . .
```

  Здесь мы разрешили добавлять новые объявления только тем пользователям, кто выполнит процедуру входа на сайт;

- `UserPassesTestMixin` — допускает к странице только тех пользователей, кто выполнил вход и в чьем отношении переопределенный метод `test_func()` вернет в качестве результата значение `True` (в изначальной реализации метод `test_func()` возбуждает исключение `NotImplementedError`, поэтому его обязательно следует переопределить). Пример:

```
from django.contrib.auth.mixins import UserPassesTestMixin
class BbCreateView(UserPassesTestMixin, CreateView):
    . . .
    def test_func(self):
        return self.request.user.is_staff
```

- `PermissionRequiredMixin` — допускает к странице только пользователей, имеющих заданные права. Класс поддерживает дополнительные атрибут и методы:

  - `permission_required` — атрибут, задает требуемые права, которые указываются в том же формате, что применяется в методах `has_perm()` и `has_perms()` (см. *разд. 15.5*). Можно задать одно право или последовательность прав;

  - `get_permission_required()` — метод, должен возвращать требуемые права. В изначальной реализации возвращает значение атрибута `permission_required`;

  - `has_permission()` — метод, должен возвращать `True`, если текущий пользователь имеет заданные права, и `False` — в противном случае. В изначальной реализации возвращает результат вызова метода `has_perms()` у текущего пользователя.

Пример:

```
from django.contrib.auth.mixins import PermissionRequiredMixin
class BbCreateView(PermissionRequiredMixin, CreateView):
    permission_required = ('bboard.add_bb', 'bboard.change_bb',
                           'bboard.delete_bb')
    . . .
```

15.6.2. Авторизация в шаблонах

Если в числе активных обработчиков контекста, указанных в параметре `context_processors` настроек шаблонизатора, имеется `django.contrib.auth.context_processors.auth` (подробности — в *разд. 11.1*), он будет добавлять в контекст каждого шаблона переменные `user` и `perms`, хранящие, соответственно, текущего пользователя и его права.

Переменная `user` хранит экземпляр класса `User`, и мы можем использовать его поля, атрибуты и методы для получения сведений о текущем пользователе. Например, таким способом мы можем выяснить, выполнил ли пользователь вход на сайт, и, если выполнил, вывести его имя:

```
{% if user.is_authenticated %}
<p>Добро пожаловать, {{ user.username }}!</p>
{% endif %}
```

Переменная `perms` хранит особый объект, который мы можем использовать для выяснения обладаемых пользователем прав, причем двумя способами:

- первый способ — применением оператора `in` или `not in`. Права записываются в виде строки в том же формате, что применяется в вызовах методов `has_perm()` и `has_perms()` (см. *разд. 15.5*). Вот так мы можем проверить, имеет ли пользователь право на добавление объявлений, и, если имеет, вывести гиперссылку на страницу добавления объявления:

```
{% if 'bboard.add_bb' in perms %}
<a href="{% url 'bboard:add' %}">Добавить</a>
{% endif %}
```

- второй способ — доступ к атрибуту с именем вида *<приложение>.<операция>_<модель>*. Такой атрибут хранит значение `True`, если пользователь имеет право на выполнение заданной *операции* в заданной *модели* указанного *приложения*, и `False` — в противном случае. Пример:

```
{% if perms.bboard.add_bb %}
<a href="{% url 'bboard:add' %}">Добавить</a>
{% endif %}
```

ЧАСТЬ III

Расширенные инструменты и дополнительные библиотеки

ГЛАВА 16

Модели: расширенные инструменты

Модели Django предоставляют ряд расширенных инструментов, которые будут полезны в некоторых специфических случаях. Это средства для управления выборкой полей, для связи с дополнительными параметрами, полиморфные связи, наследование моделей, объявление своих диспетчеров записей и наборов записей и инструменты для управления транзакциями.

16.1. Управление выборкой полей

При выборке набора записей Django извлекает из таблицы базы данных значения полей только текущей модели. При обращении к полю связанной модели фреймворк выполняет дополнительный SQL-запрос для извлечения содержимого этого поля, что может снизить производительность.

Помимо этого, выполняется выборка значений из всех полей текущей модели. Если какие-то поля хранят данные большого объема (например, большой текст объявления), их выборка займет много времени и отнимет существенный объем оперативной памяти.

Далее приведены методы, поддерживаемые диспетчером записей (классом `Manager`) и набором записей (классом `QuerySet`), которые позволят нам управлять выборкой значений полей:

- `select_related(`*<поле внешнего ключа 1>, <поле внешнего ключа 2> . . . <поле внешнего ключа n>*`)` — будучи вызван у записи вторичной модели, указывает извлечь связанную запись первичной модели. В качестве параметров записываются имена *полей внешних ключей*, устанавливающих связь с нужными первичными моделями.

  Метод извлекает единичную связанную запись. Его можно применять только в моделях, связанных связью «один-с-одним», и вторичных моделях в случае связи «один-со-многими».

Пример:

```
>>> from bboard.models import Bb
>>> b = Bb.objects.get(pk=1)
>>> # Никаких дополнительных запросов к базе данных не выполняется,
>>> # т. к. значение поля title, равно как и значения всех прочих
>>> # полей текущей модели, уже извлечены
>>> b.title
'Дача'
>>> # Но для извлечения полей записи связанной модели выполняется
>>> # отдельный запрос к базе данных
>>> b.rubric.name
'Недвижимость'
>>> # Используем метод select_related(), чтобы выбрать поля и текущей,
>>> # и связанной моделей в одном запросе
>>> b = Bb.objects.select_related('rubric').get(pk=1)
>>> # Теперь отдельный запрос к базе для извлечения значения поля
>>> # связанной модели не выполняется
>>> b.rubric.name
'Недвижимость'
```

Применяя синтаксис, описанный в *разд. 7.3.7*, можно выполнить выборку модели, связанной с первичной моделью. Предположим, что наша модель `Bb` связана с моделью `Rubric`, которая, в свою очередь, связана с моделью `SuperRubric` через поле внешнего ключа `super_rubric` и является вторичной для этой модели. Тогда мы можем выполнить выборку связанной записи модели `SuperRubric`, написав выражение вида:

```
>>> b = Bb.objects.select_related('rubric__super_rubric').get(pk=1)
```

Можно выполнять выборку сразу нескольких связанных моделей, написав такой код:

```
>>> b = Bb.objects.select_related('rubric',
                                  'rubric__super_rubric').get(pk=1)
```

или такой:

```
>>> b = Bb.objects.select_related('rubric').select_related(
                                  'rubric__super_rubric').get(pk=1)
```

Чтобы отменить выборку связанных записей, заданную предыдущими вызовами метода `select_related()`, достаточно вызвать этот метод, передав в качестве параметра значение `None`;

- `prefetch_related(`*<связь 1>*`, `*<связь 2>*` . . . `*<связь n>*`)` — будучи вызван у записи первичной модели, указывает извлечь и сохранить в памяти все связанные записи вторичной модели.

  Метод извлекает набор связанных записей. Вследствие чего он применяется в моделях, связанных связью «многие-со-многими», и первичных моделях в случае связи «один-со-многими».

В качестве *связи* можно указать:

- строку с именем:
  - атрибута, применяющегося для извлечения связанных записей (см. *разд. 7.2*) — если метод вызывается у записи первичной модели, и установлена связь «один-со-многими»:

```
>>> from bboard.models import Rubric
>>> r = Rubric.objects.first()
>>> r
<Rubric: Транспорт>
>>> # Здесь для извлечения каждого объявления, связанного
>>> # с рубрикой, выполняется отдельный запрос к базе данных
>>> for bb in r.bb_set.all(): print(bb.title, end=' ')
...
Велосипед Мотоцикл
>>> # Используем метод prefetch_related(), чтобы извлечь все
>>> # связанные объявления в одном запросе
>>> r = Rubric.objects.prefetch_related('bb_set').first()
>>> for bb in r.bb_set.all(): print(bb.title, end=' ')
...
Велосипед Мотоцикл
```

  - поля внешнего ключа — если установлена связь «многие-со-многими»:

```
>>> from testapp.models import Machine, Spare
>>> # Указываем предварительно извлечь все связанные
>>> # с машиной составные части
>>> m = Machine.objects.prefetch_related('spares').first()
>>> for s in m.spares.all(): print(s.name, end=' ')
...
Гайка Винт
```

- экземпляр класса `Prefetch` из модуля `django.db.models`, хранящий все необходимые сведения для выборки записей. Конструктор этого класса вызывается в формате:

```
Prefetch(<связь>[, queryset=None][, to_attr=None])
```

Связь указывается точно так же, как было описано ранее.

Необязательный параметр `queryset` задает набор записей для выборки связанных записей. В этом наборе записей можно указать какую-либо фильтрацию, сортировку или предварительную выборку полей связанных записей методом `select_related()` (см. ранее).

Необязательный параметр `to_attr` позволяет указать имя атрибута, который будет создан в объекте текущей записи и в качестве значения получит набор выбранных записей связанной модели.

Примеры:

```
>>> from django.db.models import Prefetch
>>> # Выполняем выборку объявлений, связанных с рубрикой,
>>> # с одновременной их сортировкой по убыванию названия
>>> pr1 = Prefetch('bb_set',
                   queryset=Bb.objects.order_by('-title'))
>>> r = Rubric.objects.prefetch_related(pr1).first()
>>> for bb in r.bb_set.all(): print(bb.title, end=' ')
...
Мотоцикл Велосипед
>>> # Выполняем выборку только тех связанных объявлений,
>>> # в которых указана цена свыше 10 000 руб., и помещаем
>>> # получившийся набор записей в атрибут expensive объекта
>>> # рубрики
>>> pr2 = Prefetch('bb_set',
                   queryset=Bb.objects.filter(price__gt=10000),
                   to_attr='expensive')
>>> r = Rubric.objects.prefetch_related(pr2).first()
>>> for bb in r.expensive: print(bb.title, end=' ')
...
Велосипед
```

Можно выполнить выборку наборов записей по нескольким связям, записав их либо в одном, либо в нескольких вызовах метода `prefetch_related()`. Чтобы отменить выборку наборов записей, заданную предыдущими вызовами метода `prefetch_related()`, следует вызвать этот метод, передав ему в качестве параметра значение `None`;

- `defer(`*<имя поля 1>*`, `*<имя поля 2>*` . . . `*<имя поля n>*`)` — указывает не извлекать значения полей с заданными *именами* в текущем запросе. Для последующего извлечения значений этих полей будет выполнен отдельный запрос к базе данных. Пример:

```
>>> bb = Bb.objects.defer('content').get(pk=3)
>>> # Никаких дополнительных запросов к базе данных не выполняется,
>>> # поскольку значение поля title было извлечено в текущем запросе
>>> bb.title
'Дом'
>>> # А значение поля content будет извлечено в отдельном запросе,
>>> # т. к. это поле было указано в вызове метода defer()
>>> bb.content
'Трехэтажный, кирпич'
```

Можно указать не выполнять выборку значений сразу у нескольких полей, записав их либо в одном, либо в нескольких вызовах метода `defer()`. Чтобы отменить запрет выборки, заданный предыдущими вызовами метода `defer()`, следует вызвать этот метод, передав ему в качестве параметра значение `None`;

- only(*<имя поля 1>, <имя поля 2> . . . <имя поля n>*) — указывает не извлекать значения всех полей, кроме полей с заданными *именами*, в текущем запросе. Для последующего извлечения значений полей, не указанных в вызове метода, будет выполнен отдельный запрос к базе данных. Пример:

```
>>> bb = Bb.objects.only('title', 'price').get(pk=3)
```

  Вызов метода only() отменяет параметры выборки, заданные предыдущими вызовами методов only() и defer(). Однако после его вызова можно поставить вызов метода defer() — он укажет поля, которые не должны выбираться в текущем запросе.

16.2. Связи «многие-со-многими» с дополнительными данными

В *главе 4*, разбираясь с моделями, мы написали код моделей Machine (машина) и Spare (отдельная деталь), связанных связью «многие-со-многими». Однако совершенно забыли о том, что машина может включать в себя более одной детали каждого наименования, и нам нужно где-то хранить количество деталей, входящих в состав каждой машины.

Реализовать это на практике не так уж и сложно. Надо только создать *связь с дополнительными данными*. Делается это в два шага:

1. Сначала следует объявить связующую модель, которая, во-первых, создаст связь «многие-со-многими» между ведущей и ведомой моделями, а во-вторых, сохранит дополнительные данные этой связи (в нашем случае — количество деталей, входящих в состав машины).

   В связующей модели нужно объявить такие поля:

   - поле типа ForeignKey для связи с ведущей моделью;
   - поле типа ForeignKey для связи с ведомой моделью;
   - поля нужных типов для хранения дополнительных данных.

2. В ведущей модели нужно объявить поле типа ManyToManyField для связи с ведомой моделью. В параметре through этого поля следует указать имя связующей модели, представленное в виде строки. В параметре through_fields того же поля задается кортеж из двух элементов:

   - имени поля связующей модели, по которому устанавливается связь с ведущей моделью;
   - имени поля связующей модели, по которому устанавливается связь с ведомой моделью.

   На заметку

   Вообще-то, параметр through_fields обязательно указывается только в том случае, если связующая модель связана с ведущей или ведомой несколькими связями, и, соответственно, в ней присутствуют несколько полей внешнего ключа, устанавливающих эти связи. Но знать о нем все равно полезно.

Ведомая модель объявляется так же, как и в случае обычной связи «многие-со-многими».

Листинг 16.1 показывает код, объявляющий три модели: ведомую `Spare`, ведущую `Machine` и связующую `Kit`.

Листинг 16.1. Создание связи «многие-со-многими» с дополнительными данными

```
from django.db import models

class Spare(models.Model):
    name = models.CharField(max_length=40)

class Machine(models.Model):
    name = models.CharField(max_length=30)
    spares = models.ManyToManyField(Spare, through='Kit',
                                    through_fields=('machine', 'spare'))

class Kit(models.Model):
    machine = models.ForeignKey(Machine, on_delete=models.CASCADE)
    spare = models.ForeignKey(Spare, on_delete=models.CASCADE)
    count = models.IntegerField()
```

Написав классы моделей, сформировав и выполнив миграции, мы можем работать с данными с применением способов, хорошо знакомых нам по *главе 6*. Сначала мы создадим записи в моделях `Spare` и `Machine`:

```
>>> from testapp.models import Spare, Machine, Kit
>>> s1 = Spare.objects.create(name='Болт')
>>> s2 = Spare.objects.create(name='Гайка')
>>> s3 = Spare.objects.create(name='Шайба')
>>> s4 = Spare.objects.create(name='Шпилька')
>>> m1 = Machine.objects.create(name='Самосвал')
>>> m2 = Machine.objects.create(name='Тепловоз')
```

Потом создадим связи. Пусть самосвал содержит 10 болтов, 100 гаек и две шпильки (сообщения, выводимые консолью, пропущены ради краткости):

```
>>> Kit.objects.create(machine=m1, spare=s1, count=10)
>>> Kit.objects.create(machine=m1, spare=s2, count=100)
>>> Kit.objects.create(machine=m1, spare=s4, count=2)
```

А тепловоз — 49 шайб и один болт:

```
>>> Kit.objects.create(machine=m2, spare=s2, count=49)
>>> Kit.objects.create(machine=m2, spare=s1, count=1)
```

Проверим, какие детали содержит тепловоз:

```
>>> for s in m1.spares.all(): print(s.name, end=' ')
...
Болт Гайка Шпилька
```

Выведем список деталей, которые содержит самосвал, с указанием их количества:

```
>>> for s in m1.spares.all():
...     kit = s.kit_set.get(machine=m1)
...     print(s.name, ' (', kit.count, ')', sep='')
...
Болт (10)
Гайка (100)
Шпилька (2)
```

Уменьшаем количество входящих в состав самосвала болтов до пяти и удаляем все шпильки:

```
>>> k1 = Kit.objects.get(machine=m1, spare=s1)
>>> k1.count
10
>>> k1.count = 5
>>> k1.save()
>>> k1.count
5
>>> k2 = Kit.objects.get(machine=m1, spare=s4)
>>> k2.delete()
(1, {'testapp.Kit': 1})
>>> for s in m1.spares.all(): print(s.name, end=' ')
...
Болт Гайка
```

Мы не можем использовать для установки и удаления связей такого типа методы `add()`, `create()`, `set()` и `remove()`, описанные в *разд. 6.5.3*. Однако рассмотренный там же метод `clear()` будет работать и в этом случае.

16.3. Полиморфные связи

Обычную связь «один-со-многими» можно установить только между двумя конкретными моделями. В параметрах поля внешнего ключа, создаваемого во вторичной модели, в обязательном порядке записывается класс первичной модели, с которой устанавливается связь. Стало быть, в нашем случае, если во вторичной модели `Bb` создана связь с первичной моделью `Rubric`, то запись модели `Bb` может быть связана с записью только из модели `Rubric` — и более ни из какой другой.

Но предположим, что нам нужно добавить возможность оставлять заметки к машинам и деталям, хранящимся в моделях `Machine` и `Spare` соответственно. Мы можем создать модели `MachineNotes` и `SpareNotes` и связать их с нужными моделями обычными связями «один-со-многими». Но это не лучший подход: во-первых, нам придется создать две совершенно одинаковые модели, хранящие совершенно однотипные данные, а во-вторых, мы можем столкнуться с проблемами при обработке массива заметок как единой сущности (например, при подсчете общего количества заметок).

Однако есть и более изящное решение. Мы можем создать всего одну модель `Notes` и установить в ней полиморфную связь.

Полиморфной, или *обобщенной*, называется связь, позволяющая связать запись вторичной модели с записью любой модели, что объявлена в приложениях проекта, без исключений. Разные записи вторичной модели при этом могут быть связаны с записями разных моделей.

Перед созданием полиморфных связей нужно убедиться, что приложение `django.contrib.contenttypes`, реализующее функциональность соответствующей подсистемы Django, присутствует в списке зарегистрированных приложений (параметр `INSTALLED_APPS` настроек проекта). После этого следует хотя бы раз провести выполнение миграций, чтобы Django создал в базе данных используемые упомянутым ранее приложением структуры.

> ***На заметку***
>
> Для хранения списка созданных в проекте моделей, который используется подсистемой полиморфных связей в работе, в базе данных создается таблица `django_content_type`. Править ее вручную не рекомендуется.

Как и обычная связь «один-со-многими», полиморфная связь создается во вторичной модели. Для ее установления необходимо создать в этом классе три сущности:

- поле для хранения типа модели, связываемой с записью. Это поле должно иметь тип `ForeignKey` (т. е. внешний ключ для связи «один-со-многими»), устанавливать связь с моделью `ContentType` из модуля `django.contrib.contenttypes.models` (в этой модели хранятся сведения обо всех моделях, что есть в проекте) и указывать каскадное удаление. Обычно такому полю дается имя `content_type`;
- поле для хранения значения ключа связываемой записи. Оно должно иметь целочисленный тип — обычно `PositiveIntegerField`. Как правило, этому полю дается имя `object_id`;
- *поле полиморфной связи*, собственно и создающее эту связь. Оно реализуется экземпляром класса `GenericForeignKey` из модуля `django.contrib.contenttypes.fields`. Вот формат вызова конструктора этого класса:

```
GenericForeignKey([ct_field='content_type'][,]
                  [fk_field='object_id'][,][for_concrete_model=True])
```

Конструктор принимает следующие параметры:

- `ct_field` — указывает имя поля, хранящего тип связываемой модели, если это имя отличается от `content_type`;
- `fk_field` — указывает имя поля, хранящего ключ связываемой записи, если это имя отличается от `object_id`;
- `for_concrete_model` — следует дать значение `False`, если необходимо устанавливать связи, в том числе и с прокси-моделями (о них будет рассказано позже).

Для создания связи текущей записи с другой записью последнюю следует присвоить полю полиморфной связи.

Листинг 16.2 показывает код модели `Note`, хранящей заметки к машинам и деталям и использующей для связи с соответствующими моделями полиморфную связь.

Листинг 16.2. Пример использования полиморфной связи

```
from django.db import models
from django.contrib.contenttypes.fields import GenericForeignKey
from django.contrib.contenttypes.models import ContentType

class Note(models.Model):
    content = models.TextField()
    content_type = models.ForeignKey(ContentType,
                                     on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey(ct_field='content_type',
                                       fk_field='object_id')
```

Теперь мы можем создать заметку и связать ее с любой моделью, что объявлена в приложениях нашего проекта, без исключений. Давайте, например, оставим заметки для шпильки и тепловоза:

```
>>> from testapp.models import Spare, Machine, Note
>>> m1 = Machine.objects.get(name='Тепловоз')
>>> s1 = Spare.objects.get(name='Шпилька')
>>> n1 = Note.objects.create(content='Самая бесполезная деталь',
                             content_object=s1)
>>> n2 = Note.objects.create(content='В нем не используются шпильки',
                             content_object=m1)
>>> n1.content_object.name
'Шпилька'
>>> n2.content_object.name
'Тепловоз'
```

Из поля, хранящего тип связанной модели, мы можем получить объект записи, описывающей этот тип. А обратившись к полю `name` этой записи, мы получим сам тип связанной модели, фактически — имя ее класса, приведенное к нижнему регистру:

```
>>> n2.content_type.name
'machine'
```

Переберем в цикле все заметки и выведем их текст вместе с названием сущности, для которой оставлена та или иная заметка:

```
>>> for n in Note.objects.all(): print(n.content, n.content_object.name)
...
Шпилька Самая бесполезная деталь
Тепловоз В нем не используются шпильки
```

К сожалению, поле полиморфной связи нельзя использовать в условиях фильтрации. Так что вот такой код вызовет ошибку:

```
>>> notes = Note.objects.filter(content_object=s1)
```

Кроме того, при создании полиморфной связи Django по умолчанию не предоставляет средств для доступа из первичной модели к связанным записям вторичной модели (что вполне оправдано — ведь если с записью вторичной модели посредством полиморфной связи может быть связана запись из любой модели, что есть в проекте, придется создавать такие средства во всех моделях без исключения, а на это никаких системных ресурсов не напасешься). Нам придется создать такие средства самостоятельно в каждой модели, из которых мы хотим получать доступ к связанным записям.

Доступ из записи первичной модели к связанным записям вторичной модели в случае полиморфной связи предоставляет так называемое *поле обратной связи*. Оно реализуется экземпляром класса `GenericRelation` из модуля `django.contrib.contenttypes.fields`. Конструктор этого класса вызывается в формате:

```
GenericRelation(<вторичная модель>[, content_type_field='content_type'][,
                object_id_field='object_id'][, for_concrete_model=True][,
                related_query_name=None])
```

Вторичная модель указывается либо в виде ссылки на класс, либо в виде имени класса, представленным строкой. Параметр `content_type_field` указывает имя поля, хранящего тип связываемой модели, а параметр `object_id_field` — имя поля, в котором сохраняется ключ связываемой записи, если эти имена отличаются от используемых по умолчанию `content_type` и `object_id` соответственно. Если одна из связываемых моделей является прокси-моделью, параметру `for_concrete_model` нужно присвоить значение `False`. Назначение параметра `related_query_name` аналогично таковому у поля типа `ForeignKey` (см. *разд. 4.4.1*).

Из созданного таким образом поля обратной связи мы и сможем получить набор связанных записей.

Чтобы из записи модели `Machine` получить список связанных заметок, нам нужно добавить в объявление ее класса такой код:

```
from django.contrib.contenttypes.fields import GenericRelation
class Machine(models.Model):
    . . .
    notes = GenericRelation('Note')
```

Теперь мы можем перебрать все заметки, оставленные для тепловоза, и вывести на экран их содержимое:

```
>>> for n in m1.notes.all(): print(n.content)
...
В нем не используются шпильки
```

Поля типа `GenericForeingKey` никак не представляются в формах, связанных с моделями (см. *главу 13*). Однако Django позволяет создать встроенный набор форм,

служащий для работы со связанными записями (подробнее о встроенных наборах форм рассказывалось в *разд. 14.5*). Он создается посредством функции `generic_inlineformset_factory()` из модуля `django.contrib.contenttypes.forms`, чей формат вызова таков:

```
generic_inlineformset_factory(<вторичная модель с полиморфной связью>[,
                        form=<форма, связанная с моделью>][,
                        ct_field='content_type'][, fk_field='object_id'][,
                        for_concrete_model=True][,
                        fields=None][, exclude=None][, extra=3][,
                        can_order=False][, can_delete=True][,
                        min_num=None][, validate_min=False][,
                        max_num=None][, validate_max=False][,
                        formset=<набор форм, связанных с моделью>])
```

Параметр `ct_field` указывает имя поля, хранящего тип связываемой модели, а параметр `fk_field` — имя поля, в котором сохраняется ключ связываемой записи, если эти имена отличаются от используемых по умолчанию `content_type` и `object_id` соответственно. Если одна из связываемых моделей является прокси-моделью, параметру `for_concrete_model` нужно дать значение `False`. Базовый набор записей, указываемый в параметре `formset`, должен быть производным от класса `BaseGenericInlineFormSet` из модуля `django.contrib.contenttypes.forms`.

16.4. Наследование моделей

Модель Django — это обычный класс Python. Следовательно, мы можем объявить модель, являющуюся подклассом другой модели. Причем фреймворк предлагает нам целых три способа сделать это.

16.4.1. Прямое наследование моделей

В случае *прямого*, или *многотабличного*, наследования мы просто наследуем один класс модели от другого.

Пример такого наследования можно увидеть в листинге 16.3. Там мы объявляем базовую модель `Message`, хранящую любое сообщение, и наследуем от него производную модель `PrivateMessage`, которая хранит частные сообщения, адресованные зарегистрированным пользователям.

Листинг 16.3. Пример прямого (многотабличного) наследования моделей

```
from django.db import models
from django.contrib.auth.models import User

class Message(models.Model):
    content = models.TextField()
```

```
class PrivateMessage(Message):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
```

В этом случае Django поступит следующим образом:

- на основе обеих моделей — и базовой, и производной — создаст в базе данных таблицы. Эти таблицы будут включать только те поля, что объявлены в моделях;
- установит между моделями связь «один-с-одним». Базовая модель станет первичной, а производная — вторичной;
- создаст в производной модели поле с именем вида *<имя базовой модели>*_ptr, реализующее связь «один-с-одним» с базовой моделью.

  Мы можем создать такое служебное поле самостоятельно, дав ему произвольное имя. При создании поля следует указать для него каскадное удаление и обязательно присвоить параметру `parent_link` значение `True`. Пример:

```
class PrivateMessage(Message):
    . . .
    message = models.OneToOneField(Message, on_delete=models.CASCADE,
                                   parent_link=True)
```

- в объекте записи базовой модели создаст атрибут с именем вида *<имя производной модели>*, хранящий объект записи производной модели;
- при сохранении данных в записи производной модели значения для полей, унаследованных от базовой модели, будут сохраняться в базовой модели, а данные для полей, объявленных в производной модели, — в производной модели;
- при удалении записи фактически удалит обе записи: из базовой и производной моделей.

  Есть возможность удалить запись производной модели, оставив связанную запись базовой модели. Для этого при вызове у записи производной модели метода `delete()` следует указать в нем параметр `keep_parents` со значением `True`.

Производная модель наследует от базовой все параметры, заданные во вложенном классе `Meta`. При необходимости эти параметры можно переопределить в производной модели.

Давайте немного поэкспериментируем. Добавим в модель `PrivateMessage` запись и попробуем получить значения ее полей:

```
>>> from testapp.models import PrivateMessage
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username='editor')
>>> pm = PrivateMessage.objects.create(content='Привет, editor!', user=u)
>>> pm.content
'Привет, editor!'
>>> pm.user
<User: editor>
```

Получим связанную запись базовой модели `Message`:

```
>>> m = pm.message_ptr
>>> m
<Message: Message object (1)>
```

И, наконец, попытаемся получить запись производной модели из записи базовой модели:

```
>>> m.privatemessage
<PrivateMessage: PrivateMessage object (1)>
```

Прямое наследование может выручить, если нам нужно хранить в базе данных набор сущностей с примерно одинаковым набором полей. Тогда поля, общие для всех типов сущностей, можно вынести в базовую модель, а в производных оставить только поля, уникальные для каждого конкретного типа сущностей.

16.4.2. Абстрактные модели

Второй способ наследовать одну модель от другой — объявить базовую модель как *абстрактную*. Для этого достаточно задать для нее параметр `abstract` со значением `True`. Пример:

```
class Message(models.Model):
    . . .
    class Meta:
        abstract = True

class PrivateMessage(Message):
    . . .
```

Для абстрактной базовой модели в базе данных не создается никаких таблиц. Напротив, таблица, созданная для производной от нее модели, будет содержать весь набор полей, объявленных как в базовой, так и в производной модели.

Поля, объявленные в базовой абстрактной модели, могут быть переопределены в производной. Можно даже удалить объявленное в базовой модели поле, объявив в производной модели атрибут класса с тем же именем и присвоив ему значение `None`. Пример такого переопределения и удаления полей представлен в листинге 16.4.

Листинг 16.4. Переопределение и удаление полей, объявленных в базовой абстрактной модели

```
class Message(models.Model):
    content = models.TextField()
    name = models.CharField(max_length=20)
    email = models.EmailField()

    class Meta:
        abstract = True
```

```
class PrivateMessage(Message):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    # Переопределяем поле name
    name = models.CharField(max_length=40)
    # Удаляем поле email
    email = None
```

Параметры базовой модели, объявленные во вложенном классе `Meta`, не наследуются производной моделью автоматически. Однако есть возможность унаследовать их, объявив вложенный класс `Meta` как производный от класса `Meta` базовой модели. Пример:

```
class Message(models.Model):
    . . .
    class Meta:
        abstract = True
        ordering = ['name']

class PrivateMessage(Message):
    . . .
    class Meta(Message.Meta):
        pass
```

Наследование с применением абстрактной базовой модели применяется в тех же случаях, что и прямое наследование (см. *разд. 16.4.1*). Оно предлагает более гибкие возможности по объявлению набора полей в производных моделях — так, мы можем переопределить или даже удалить какое-либо поле, объявленное в базовой модели. К тому же, поскольку все данные хранятся в одной таблице, работа с ними выполняется быстрее, чем в случае прямого наследования (обращение к двум таблицам требует больше времени).

16.4.3. Прокси-модели

Третий способ — объявить производную модель как *прокси-модель*. Классы такого типа не предназначены для расширения или изменения набора полей базовой модели, а служат для расширения или изменения ее функциональности.

Прокси-модель объявляется заданием в параметрах производной модели (т. е. во вложенном классе `Meta`) параметра `proxy` со значением `True`. В базовой модели при этом никаких дополнительных параметров записывать не нужно.

Для прокси-модели не создается никаких таблиц в базе данных. Все данные хранятся в базовой модели.

Листинг 16.5 показывает код прокси-модели `RevRubric`, производной от модели `Rubric` и задающей сортировку рубрик по убыванию названия.

Листинг 16.5. Пример прокси-модели

```
class RevRubric(Rubric):
    class Meta:
        proxy = True
        ordering = ['-name']
```

Давайте посмотрим, что у нас получилось. Выведем список рубрик из только что созданной модели:

```
>>> from bboard.models import RevRubric
>>> for r in RevRubric.objects.all(): print(r.name, end=' ')
...
Транспорт Сельхозинвентарь Сантехника Растения Недвижимость Мебель
Бытовая техника
```

16.5. Создание своих диспетчеров записей

Диспетчер записей — это объект, предоставляющий доступ к набору записей, которые хранятся в модели. По умолчанию он представляет собой экземпляр класса `Manager` из модуля `django.db.models` и хранится в атрибуте `objects` модели.

16.5.1. Создание диспетчеров записей

Создание своих диспетчеров записей выполняется путем наследования от класса `Manager`. В получившемся подклассе мы можем как переопределить имеющиеся методы, так и объявить новые.

Переопределять в новом диспетчере записей имеет смысл только метод `get_queryset()`, который должен возвращать набор записей, хранящихся в текущей модели, в виде экземпляра класса `QuerySet` из модуля `django.db.models`. Обычно в теле переопределенного метода сначала вызывают тот же метод базового класса, чтобы получить изначальный набор записей, устанавливают у него фильтрацию, сортировку, добавляют вычисляемые поля и возвращают в качестве результата.

Листинг 16.6 показывает код диспетчера записей `RubricManager`, который возвращает набор рубрик уже отсортированным по полям `order` и `name`. Помимо того, он объявляет дополнительный метод `order_by_bb_count()`, который возвращает набор рубрик, отсортированный по убыванию количества относящихся к ним объявлений.

Листинг 16.6. Пример объявления собственного диспетчера записей

```
from django.db import models

class RubricManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().order_by('order', 'name')
```

```
    def order_by_bb_count(self):
        return super().get_queryset().annotate(
                        cnt=models.Count('bb')).order_by('-cnt')
```

Теперь нам нужно указать свежесозданный диспетчер записей в модели. Сделать это можно в трех вариантах:

- указать в качестве единственного диспетчера записей, объявив в классе модели атрибут `objects` и присвоив ему экземпляр класса диспетчера записей:

```
class Rubric(models.Model):
    . . .
    objects = RubricManager()
```

Теперь, обратившись к атрибуту `objects` модели, мы получим доступ к нашему диспетчеру:

```
>>> from bboard.models import Rubric
>>> # Получаем набор записей, возвращенный методом get_queryset()
>>> Rubric.objects.all()
<QuerySet [<Rubric: Транспорт>, <Rubric: Недвижимость>,
<Rubric: Мебель>, <Rubric: Бытовая техника>, <Rubric: Сантехника>,
<Rubric: Растения>, <Rubric: Сельхозинвентарь>]>
>>> # Получаем набор записей, возвращенный вновь добавленным методом
>>> # order_by_bb_count()
>>> Rubric.objects.order_by_bb_count()
<QuerySet [<Rubric: Бытовая техника>, <Rubric: Недвижимость>,
<Rubric: Сельхозинвентарь>, <Rubric: Транспорт>, <Rubric: Мебель>,
<Rubric: Сантехника>, <Rubric: Растения>]>
```

- то же самое, только с использованием атрибута класса с другим именем:

```
class Rubric(models.Model):
    . . .
    bbs = RubricManager()
. . .
>>> # Теперь для доступа к диспетчеру записей мы используем
>>> # атрибут класса bbs
>>> Rubric.bbs.all()
```

- указать написанный нами диспетчер записей в качестве дополнительного:

```
class Rubric(models.Model):
    . . .
    objects = models.Manager()
    bbs = RubricManager()
```

Здесь мы присвоили атрибуту `objects` экземпляр класса `Manager`, т. е. диспетчер записей, применяемый по умолчанию, а атрибуту `bbs` — наш диспетчер записей. Так что теперь мы можем пользоваться сразу двумя диспетчерами записей. Пример:

```
>>> Rubric.objects.all()
<QuerySet [<Rubric: Недвижимость>, <Rubric: Транспорт>,
<Rubric: Бытовая техника>, <Rubric: Сельхозинвентарь>,
<Rubric: Мебель>, <Rubric: Сантехника>, <Rubric: Растения>]>
>>> Rubric.bbs.all()
<QuerySet [<Rubric: Транспорт>, <Rubric: Недвижимость>,
<Rubric: Мебель>, <Rubric: Бытовая техника>, <Rubric: Сантехника>,
<Rubric: Растения>, <Rubric: Сельхозинвентарь>]>
```

Здесь нужно учитывать один момент. Первый объявленный в классе модели диспетчер записей будет рассматриваться Django как диспетчер записей по умолчанию (в нашем случае это диспетчер записей `Manager`, присвоенный атрибуту `objects`). Диспетчер записей по умолчанию используется для доступа к записям модели при выполнении различных служебных задач. Поэтому ни в коем случае нельзя задавать в таком диспетчере данных фильтрацию, которая может оставить часть записей «за бортом».

На заметку

Есть возможность указать атрибуту, применяемому для доступа к диспетчеру записей, другое имя без задания для него нового диспетчера записей:

```
class Rubric(models.Model):
    . . .
    bbs = models.Manager()
```

16.5.2. Создание диспетчеров обратной связи

Аналогично мы можем создать свой собственный диспетчер обратной связи, который вернет набор записей вторичной модели, связанный с текущей записью первичной модели. Его класс также объявляется как производный от класса `Manager` из модуля `django.db.models` и также указывается в модели присваиванием его экземпляра атрибуту класса модели. В целом, диспетчер обратной связи в этом смысле ничем не отличается от диспетчера записей и, более того, может быть использован в качестве такового.

Листинг 16.7 показывает код диспетчера обратной связи `BbManager`. Он сортирует объявления по убыванию значения даты и времени их публикации.

Листинг 16.7. Пример диспетчера обратной связи

```
from django.db import models

class BbManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().order_by('-published')
```

Чтобы из записи первичной модели получить набор связанных записей вторичной модели с применением созданного нами диспетчера обратной связи, нам придется

явно указать этот диспетчер. Для этого у объекта первичной записи мы вызовем метод, чье имя совпадает с именем атрибута, хранящего диспетчер связанных записей. Этому методу мы передадим параметр `manager`, которому в качестве значения присвоим строку с именем атрибута класса вторичной модели, которому был присвоен объект написанного нами диспетчера. Метод вернет в качестве результата набор записей, сформированный написанным нами диспетчером.

Так, указать объявленный нами диспетчер обратной связи `BbManager` в классе модели `Bb` мы можем следующим образом (на всякий случай не забыв задать диспетчер, что будет использоваться по умолчанию):

```
class Bb(models.Model):
    . . .
    objects = models.Manager()
    reverse = BbManager()
```

И проверим его в деле:

```
>>> from bboard.models import Rubric
>>> r = Rubric.objects.first()
>>> r
<Rubric: Недвижимость>
>>> # Используем диспетчер обратной связи по умолчанию. Объявления будут
>>> # отсортированы по возрастанию значений ключа, т. е. в том порядке,
>>> # в котором они были созданы
>>> r.bb_set.all()
<QuerySet [<Bb: Bb object (1)>, <Bb: Bb object (3)>,
<Bb: Bb object (17)>]>
>>> # Используем диспетчер обратной связи, написанный нами. Объявления
>>> # сортируются по убыванию даты их публикации
>>> r.bb_set(manager='reverse').all()
<QuerySet [<Bb: Bb object (17)>, <Bb: Bb object (3)>,
<Bb: Bb object (1)>]>
```

16.6. Создание своих наборов записей

Еще мы можем объявить свой класс набора записей. В такие классы обычно добавляют методы, указывающие какие-либо часто используемые и одновременно достаточно сложные условия фильтрации, сортировки, агрегатные вычисления или вычисляемые поля.

Набор записей должен быть производным от класса `QuerySet` из модуля `django.db.models`.

Пример собственного класса набора записей можно увидеть в листинге 16.8. Набор записей `RubricQuerySet` содержит дополнительный метод, вычисляющий количество имеющихся в каждой рубрике объявлений и сортирующий рубрики по убыванию этого количества.

Листинг 16.8. Пример собственного набора записи

```
from django.db import models

class RubricQuerySet(models.QuerySet):
    def order_by_bb_count(self):
        return self.annotate(cnt=models.Count('bb')).order_by('-cnt')
```

Для того чтобы модель возвращала набор записей, представленный экземпляром объявленного нами класса, нам понадобится также объявить свой диспетчер записей (как это сделать, было рассказано в *разд. 16.5*). Прежде всего, в методе `get_queryset()` он сформирует и вернет в качестве результата экземпляр объявленного нами класса набора записей. Мы передадим конструктору этого класса в качестве первого позиционного параметра используемую модель, которую можно извлечь из атрибута `model`, а в качестве параметра `using` — базу данных, в которой хранятся записи модели и которая извлекается из атрибута `_db`.

Помимо этого, нам нужно предусмотреть вариант, когда объявленные в новом наборе записей дополнительные методы вызываются не у набора записей:

```
rs = Rubric.objects.all().order_by_bb_count()
```

а непосредственно у диспетчера записей:

```
rs = Rubric.objects.order_by_bb_count()
```

Для этого нам придется объявить одноименные методы еще и в классе набора записей и выполнять в этих методах вызовы соответствующих им методов класса набора записей.

Пример класса диспетчера записей `RubricManager`, призванного обслуживать набор записей `RubricQuerySet` (см. листинг 16.8), приведен в листинге 16.9.

Листинг 16.9. Пример диспетчера записей, обслуживающего набор записей из листинга 16.8

```
from django.db import models

class RubricManager(models.Manager):
    def get_queryset(self):
        return RubricQuerySet(self.model, using=self._db)

    def order_by_bb_count(self):
        return self.get_queryset().order_by_bb_count()
```

Написанный диспетчер записей мы указываем в классе модели уже известным нам из *разд. 16.5* способом:

```
class Rubric(models.Model):
    . . .
    objects = RubricManager()
```

Теперь мы можем проверить созданный нами набор записей в действии (вывод пропущен ради краткости):

```
>>> from bboard.models import Rubric
>>> Rubric.objects.all()
. . .
>>> Rubric.objects.order_by_bb_count()
. . .
```

Писать свой класс диспетчера записей только для того, чтобы «подружить» модель с новым классом набора записей, может быть утомительно (особенно если таковых классов наберется с десяток). Поэтому Django предусматривает целых две фабрики классов, которые создают классы диспетчеров записей на основе классов наборов записей и реализованы в виде методов:

- `as_manager()` — метод, вызываемый у класса набора записей и возвращающий экземпляр класса диспетчера записей, обслуживающего текущий набор записей:

```
class Rubric(models.Model):
    . . .
    objects = RubricQuerySet.as_manager()
```

- `from_queryset(`*<класс набора записей>*`)` — метод, вызываемый у класса диспетчера записей и возвращающий ссылку на производный класс диспетчера записей, обслуживающего переданный методу *набор записей*:

```
class Rubric(models.Model):
    . . .
    objects = models.Manager.from_queryset(RubricQuerySet)()
```

  Метод `from_queryset()` удобно использовать в случае, если уже существует класс диспетчера записей, и на его основе нужно создать производный класс, обслуживающий какой-либо набор записей.

Можно сказать, что обе фабрики классов создают новый класс набора записей и переносят в него методы из набора записей. В обоих случаях действуют следующие правила переноса методов:

- обычные методы переносятся по умолчанию;
- псевдочастные методы (те, чьи имена предваряются символом подчеркивания) не переносятся по умолчанию;
- записанный у метода атрибут `queryset_only` со значением `False` указывает перенести метод;
- записанный у метода атрибут `queryset_only` со значением `True` указывает не переносить метод.

Пример:

```
class SomeQuerySet(models.QuerySet):
    # Этот метод будет перенесен
    def method1(self):
        . . .
```

```
    # Этот метод не будет перенесен
    def _method2(self):
        . . .

    # Этот метод будет перенесен
    def _method3(self):
        . . .
    _method3.queryset_only = False

    # Этот метод не будет перенесен
    def method4(self):
        . . .
    _method4.queryset_only = True
```

16.7. Управление транзакциями

При разработке сложных бизнес-решений, в которых в процессе выполнения одной операции изменения вносятся сразу в несколько моделей, приходится решать вопрос неделимости этих изменений. Говоря другими словами, по завершении операции в модель должны быть гарантированно внесены все запланированные изменения — или же, если в процессе работы возникла ошибка, все модели должны остаться в неизменном состоянии.

Такая одновременность и неделимость изменений гарантируется использованием *транзакций*. Да, транзакции — это механизм, поддерживаемый СУБД, а не Django, но наш любимый фреймворк предоставляет удобные инструменты для управления транзакциями, и эти инструменты обязательно нам пригодятся.

16.7.1. Всё или ничего: два высокоуровневых режима управления транзакциями

Прежде всего, Django-сайт можно настроить на работу в двух высокоуровневых режимах обработки транзакций, которые можно охарактеризовать словами «всё» и «ничего».

16.7.1.1. Ничего: режим по умолчанию

Сразу после создания Django-сайт настраивается для работы в высокоуровневом режиме под названием «ничего». В этом режиме:

- каждый отдельный запрос к базе данных на чтение, добавление, правку или удаление выполняется в отдельной транзакции;
- каждая запущенная транзакция завершается автоматически.

Чтобы вручную настроить сайт на работу в этом режиме, достаточно в настройках нужной базы данных (см. *разд. 3.3.2*):

- дать параметру ATOMIC_REQUEST значение False или вообще удалить этот параметр, поскольку False — его значение по умолчанию.

  Параметр ATOMIC_REQUEST управляет неделимостью операций, производимых с базой данных во время работы одного контроллера. Значение True заставляет все производимые в контроллере запросы выполняться в контексте одной-единственной транзакции, а значение False запускает каждый запрос в своей собственной транзакции;

- дать параметру AUTOCOMMIT значение True или вообще удалить этот параметр, поскольку True — его значение по умолчанию.

  Параметр AUTOCOMMIT управляет автоматическим завершением транзакций. Значение True указывает Django завершать каждую запущенную транзакцию автоматически, а значение False отключает этот режим.

> ***Внимание!***
>
> **Отключать автоматическое завершение транзакций настоятельно не рекомендуется, поскольку в таком режиме нормально обрабатываются базы данных не всех форматов.**

Режим «ничего» неплохо подходит для случаев, когда бóльшая часть запросов к базе данных, выполняемых в контроллере, производят чтение из базы, и в контроллере одновременно производится не более одного запроса на изменение данных. Этим критериям удовлетворяют простые сайты наподобие нашей доски объявлений.

16.7.1.2. Всё: режим для максималистов

Режим, называемый «всё», подходит для максималистов, желающих, чтобы все запросы к базе данных, что выполняются в контроллере, производились в одной-единственной транзакции. В этом случае транзакция запускается перед началом выполнения контроллера, а по окончании его выполнения завершается с подтверждением (если не возникло ошибок) или откатом (если ошибки все-таки случились).

Переключить Django в такой режим можно заданием параметру ATOMIC_REQUEST настроек нужной базы данных (см. *разд. 3.3.2*) значения True. Пример:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
        'AUTOCOMMIT': True,
    }
}
```

Режим «всё» подходит для случаев, когда бóльшая часть контроллеров изменяет данные, хранящиеся в разных таблицах базы. Этот режим гарантирует, что или в базу будут внесены все требуемые изменения, или, в случае возникновения нештатной ситуации, база останется в своем изначальном состоянии.

16.7.2. Управление транзакциями на низком уровне

Помимо двух описанных ранее высокоуровневых режимов, Django предлагает средства для управления транзакциями на низком уровне. Их довольно много.

16.7.2.1. Включение режима «всё» на уровне контроллера

Если на уровне базы данных включен режим управления транзакциями «ничего», мы все еще имеем возможность включить режим «всё» на уровне какого-либо одного контроллера. Для этого применяется функция `atomic([savepoint=True])` из модуля `django.db.transaction`. Ее можно использовать как:

- декоратор, указываемый перед контроллером-функцией, для которого следует включить режим «всё»:

```
from django.db.transaction import atomic

@atomic
def edit(request, pk):
    # Все запросы к базе данных в этом контроллере будут выполняться
    # в одной транзакции
    . . .
```

- менеджер контекста в блоке `with`, для содержимого которого нужно включить режим «всё»:

```
if formset.is_valid():
    with atomic():
        # Выполняем сохранение всех форм набора в одной транзакции
    return redirect('bboard:index')
```

Можно использовать вложенные блоки `with`:

```
if formset.is_valid():
    with atomic():
        for form in formset:
            if form.cleaned_data:
                with atomic():
                    # Выполняем сохранение каждой формы набора
                    # в отдельном вложенном блоке with
```

В этом случае при входе во внешний блок `with` будет запущена собственно транзакция, а при входе во вложенный блок `with` — создана точка сохранения. При выходе из вложенного блока производится освобождение точки сохранения (если все прошло успешно) или же откат до состояния на момент ее создания (в случае возникновения ошибки). Наконец, после выхода из внешнего блока `with` происходит подтверждение или откат самой транзакции.

Каждая созданная точка сохранения отнимает некоторое количество системных ресурсов. Поэтому предусмотрена возможность отключить их создание, для чего достаточно в вызове функции `atomic()` указать параметр `savepoint` со значением `False`.

16.7.2.2. Обработка подтверждения транзакции

Если для базы данных включен режим автоматического подтверждения транзакции, есть возможность обработать момент ее подтверждения. Для этого достаточно вызвать функцию `on_commit(<функция-обработчик>)` из модуля `django.db.transaction`, передав ей ссылку на *функцию-обработчик*. Последняя не должна ни принимать параметров, ни возвращать результат. Пример:

```
from django.db import transaction

def commit_handler():
    # Выполняем какие-либо действия после подтверждения транзакции

transaction.on_commit(commit_handler)
```

Указанная функция будет вызвана только после подтверждения транзакции, но не после ее отката, подтверждения или отката точки сохранения. Если в момент вызова функции `on_commit()` активная транзакция отсутствует, функция-обработчик выполнена не будет.

16.7.2.3. Выключение режима «всё» для контроллера

Если на уровне базы данных включен режим «всё», мы можем отключить его на уровне нужного нам контроллера, тем самым задействовав в нем режим «ничего». Сделать это можно, указав перед контроллером декоратор `non_atomic_requests()` из модуля `django.db.transaction`. Пример:

```
from django.db import transaction

@transaction.non_atomic_requests
def my_view(request):
    # В этом контроллере действует режим обработки транзакций "ничего"
```

16.7.2.4. Управление транзакциями вручную

И наконец, существует набор функций для управления транзакциями вручную: их подтверждения, отката, работы с точками сохранения и др. Все эти функции объявлены в модуле `django.db.transaction`:

- `get_autocommit()` — возвращает `True`, если для базы данных включен режим автоматического завершения транзакции, и `False` — в противном случае;
- `set_autocommit(<режим>)` — включает или отключает режим автоматического завершения транзакции для базы данных. *Режим* указывается в виде логической величины: `True` включает автоматическое завершение транзакций, `False` отключает;
- `commit()` — выполняет подтверждение активной транзакции;
- `rollback()` — выполняет откат активной транзакции;

- `savepoint()` — создает новую точку сохранения и возвращает ее идентификатор в качестве результата;
- `savepoint_commit(<идентификатор точки сохранения>)` — выполняет подтверждение точки сохранения с указанным *идентификатором*;
- `savepoint_rollback(<идентификатор точки сохранения>)` — выполняет откат до точки останова с указанным *идентификатором*;
- `clean_savepoints()` — сбрасывает счетчик, применяемый для генерирования уникальных идентификаторов точек сохранения.

Вот пример организации ручного управления транзакциями при сохранении записи:

```
from django.db import transaction
. . .
if form.is_valid():
    try:
        form.save()
        transaction.commit()
    except:
        transaction.rollback()
```

А вот пример ручного управления транзакциями при сохранении набора форм с использованием точек сохранения:

```
if formset.is_valid():
    for form in formset:
        if form.cleaned_data:
            sp = transaction.savepoint()
            try:
                form.save()
                transaction.savepoint_commit(sp)
            except:
                transaction.savepoint_rollback(sp)
            transaction.commit()
```

ГЛАВА 17

Формы и наборы форм: расширенные инструменты и дополнительная библиотека

Помимо форм и наборов форм, связанных с моделями, Django предлагает аналогичные инструменты, которые не связаны с моделями. Они могут применяться для указания как данных, не предназначенных для занесения в базу данных (например, ключевого слова для поиска), так и сведений, которые должны быть сохранены в базе после дополнительной обработки.

Кроме того, фреймворк предлагает расширенные инструменты для вывода форм на экран, и эти инструменты могут нам пригодиться. А еще мы изучим весьма полезную дополнительную библиотеку Django Simple Captcha, предназначенную для обработки CAPTCHA.

17.1. Формы, не связанные с моделями

Формы, не связанные с моделями, создаются точно так же, как их «коллеги», что связаны с моделями. Есть только несколько исключений:

- форма, не связанная с моделью, объявляется как подкласс класса `Form` из модуля `django.forms`;
- все поля, которые должны присутствовать в форме, необходимо объявлять в виде атрибутов класса формы;
- вложенный класс `Meta` в такой форме не объявляется (что вполне понятно — ведь в этом классе задаются сведения о модели, с которой связана форма);
- средства сохранения введенных в форму данных в базе, включая параметр `instance` конструктора и метод `save()`, не поддерживаются.

В остальном работа с формами такого рода выполняется точно так же, как и с формами, связанными с моделями.

Пример объявления формы, не связанной с моделью, можно увидеть в листинге 17.1. Эта форма предназначена для указания искомого ключевого слова и рубрики, в которой будет производиться поиск.

Листинг 17.1. Форма, не связанная с моделью

```
from django import forms

class SearchForm(forms.Form):
    keyword = forms.CharField(max_length=20, label='Искомое слово')
    rubric = forms.ModelChoiceField(queryset=Rubric.objects.all(),
             label='Рубрика')
```

В листинге 17.2 показан код контроллера, который извлекает данные из созданной ранее формы и использует их для указания параметров фильтрации объявлений (предполагается, что форма пересылает данные методом POST).

Листинг 17.2. Контроллер, который использует форму, не связанную с моделью

```
def search(request):
    if request.method == 'POST':
        sf = SearchForm(request.POST)
        if sf.is_valid():
            keyword = sf.cleaned_data['keyword']
            rubric_id = sf.cleaned_data['rubric'].pk
            bbs = Bs.objects.filter( . . . )
            context = {'bbs': bbs}
            return render(request, 'bboard/search_result.html', context)
    else:
        sf = SearchForm()
    context = {'form': sf}
    return render(request, 'bboard/search.html', context)
```

17.2. Наборы форм, не связанные с моделями

Наборы форм, не связанные с моделями, создаются с применением функции `formset_factory()` из модуля `django.forms`:

```
formset_factory(form=<форма>[, extra=1][,
                can_order=False][, can_delete=False][,
                min_num=None][, validate_min=False][,
                max_num=None][, validate_max=False][,
                formset=<набор форм>])
```

Здесь *форма*, на основе которой создается набор форм, является обязательным параметром. А базовый *набор форм* должен представлять собой подкласс класса `BaseFormSet` из модуля `django.forms.formsets`. Назначение остальных параметров было описано в *разд. 14.1*.

Вот пример создания набора форм на основе формы `SearchForm`, чей код показан в листинге 17.1:

```
from django.forms import formset_factory
fs = formset_factory(SearchForm, extra=3, can_delete=True)
```

Здесь нужно иметь в виду, что набор форм, не связанный с моделью, не поддерживает средств для сохранения занесенных в него данных в модели. Также не следует забывать, что атрибуты `new_objects`, `changed_objects` и `deteled_objects`, хранящие списки, соответственно, новых, исправленных и удаленных записей, недоступны. Нам самим придется писать код, который будет сохранять введенные данные в модели, или, что случается гораздо чаще, обрабатывать эти данные иным образом.

В остальном же работа с такими наборами форм протекает аналогично работе с наборами форм, связанными с моделями. Мы можем перебирать формы, содержащиеся в наборе, и извлекать из них данные.

Если набор форм поддерживает переупорядочение форм (т. е. при его создании в вызове функции `formset_factory()` был указан параметр `can_order` со значением `True`), в составе каждой формы появится поле `ORDER` типа `IntegerField`. Это поле будет хранить порядковый номер текущей формы. Мы можем использовать его, чтобы выстроить в нужном порядке какие-либо сущности, или иным образом.

Если набор форм поддерживает удаление отдельных форм (добавить эту поддержку можно, записав в вызове функции `formset_factory()` параметр `can_delete` со значением `True`), в составе каждой формы появится поле `DELETE` типа `BooleanField`. Это поле будет хранить значение `True`, если форма была помечена на удаление, и `False` — в противном случае.

Класс набора форм, не связанного с моделью, поддерживает два полезных атрибута:

- `ordered_forms` — последовательность форм, которые были переупорядочены;
- `deleted_forms` — последовательность удаленных форм.

В листинге 17.3 показан код контроллера, который обрабатывает набор форм, не связанный с моделью.

Листинг 17.3. Контроллер, который обрабатывает набор форм, не связанный с моделью

```
def formset_processing(request):
    FS = formset_factory(SearchForm, extra=3, can_order=True,
                                              can_delete=True)
    if request.method == 'POST':
        formset = FS(request.POST)
        if formset.is_valid():
            for form in formset:
                if form.cleaned_data and not form.cleaned_data['DELETE']:
                    keyword = form.cleaned_data['keyword']
                    rubric_id = form.cleaned_data['rubric'].pk
                    order = form.cleaned_data['ORDER']
                    # Выполняем какие-либо действия над полученными
                    # данными
```

```
            return render(request, 'bboard/process_result.html')
    else:
        formset = FS()
    context = {'formset': formset}
    return render(request, 'bboard/formset.html', context)
```

17.3. Расширенные средства для вывода форм и наборов форм

Фреймворк имеет средства для тонкой настройки выводимых форм. Они поддерживаются обеими разновидностями форм: и связанными с моделями, и не связанными с ними.

17.3.1. Указание CSS-стилей для форм

Для указания CSS-стилей, которые будут применены к отдельным элементам выводимой формы, классы форм поддерживают два атрибута класса:

- `required_css_class` — имя стилевого класса, которым будут помечаться элементы управления, куда нужно обязательно ввести значение;
- `error_css_class` — имя стилевого класса, которым будут помечаться элементы управления с некорректными данными.

Эти стилевые классы будут привязываться к тегам `<p>`, `<li>` или `<tr>`, в зависимости от того, посредством каких HTML-тегов была выведена на экран форма.

Пример:

```
class SearchForm(forms.Form):
    error_css_class = 'error'
    required_css_class = 'required'
```

17.3.2. Настройка выводимых форм

Некоторые настройки, затрагивающие выводимые на экран формы, указываются в виде именованных параметров конструктора класса формы. Вот эти параметры:

- `field_order` — задает порядок следования полей формы при ее выводе на экран. В качестве значения указывается последовательность имен полей, представленных в виде строк. Если задать `None`, поля будут следовать друг за другом в том же порядке, в котором они были объявлены в классе формы. Значение по умолчанию — `None`. Пример:

  ```
  bf = BbForm(field_order=('rubric', 'rubric', 'price', 'content'))
  ```

- `label_suffix` — строка с суффиксом, который будет добавлен к тексту надписи при выводе. Значение по умолчанию — символ двоеточия;
- `auto_id` — управляет формированием идентификаторов элементов управления, которые указываются в атрибутах `id` тегов, формирующих эти элементы управ-

ления, и тегов <label>, создающих для них надписи. В качестве значения параметра можно указать:

- строку формата — идентификаторы будут формироваться согласно указанной строке. Символ-заменитель %s указывает местоположение в строке формата имени поля, соответствующего элементу управления. Пример:

```
sf = SearchForm(auto_id='id_for_%s')
```

  Для поля keyword будет сгенерирован идентификатор id_for_keyword, а для поля rubric — идентификатор id_for_rubric;

- True — в качестве идентификаторов будут использоваться имена полей, соответствующих элементам управления;
- False — идентификаторы вообще не будут формироваться. Также не будут формироваться теги <label>, а надписи будут представлять собой простой текст.

Значение параметра по умолчанию: "id_%s";

- use_required_attribute — если True, в теги, формирующие обязательные для заполнения элементы управления, будут помещены атрибуты required, если False, этого не произойдет. Значение по умолчанию — True;
- prefix — указывает префикс для имен полей в выводимой форме. Применяется, если в один тег <form> нужно поместить несколько форм. Значение по умолчанию — None (префикс отсутствует).

17.3.3. Настройка наборов форм

Конструкторы классов наборов форм поддерживают именованные параметры auto_id и prefix:

```
formset = FS(auto_id=False)
```

Поле порядкового номера ORDER, посредством которого выполняется переупорядочивание форм, и поле удаления формы DELETE доступны через одноименные элементы. Мы можем использовать это, чтобы вывести служебные поля отдельно от остальных. Пример:

```
<h2>Рубрики</h2>
<form method="post">
    {% csrf_token %}
    {{ formset.management_form }}
    <table>
        {% for form in formset %}
        <tr><td colspan="2">{{ form.name }}</td></tr>
        <tr><td>{{ form.ORDER }}</td><td>{{ form.DELETE }}</td></tr>
        {% endfor %}
    </table>
    <p><input type="submit" value="Сохранить"></p>
</form>
```

17.4. Библиотека Django Simple Captcha: поддержка CAPTCHA

Если планируется, что в какую-либо форму данные будут заноситься незарегистрированными пользователями (простыми посетителями, или гостями), не помешает как-то обезопасить форму от нашествия программ-ботов. Одним из решений является применение *CAPTCHA* (Completely Automated Public Turing test to tell Computers and Humans Apart, полностью автоматизированный публичный тест Тьюринга для различения компьютеров и людей).

CAPTCHA выводится на веб-страницу в виде графического изображения, содержащего сильно искаженный или зашумленный текст, который нужно прочитать и занести в расположенное рядом поле ввода. Если результат оказался верным, то, скорее всего, данные занесены человеком, поскольку программам такие сложные задачи пока еще не по плечу.

Для Django существует довольно много библиотек, реализующих в формах поддержку CAPTCHA. Одна из них — django-simplecaptcha, которую мы сейчас и рассмотрим.

> ***НА ЗАМЕТКУ***
>
> Полная документация по библиотеке Django Simple Captcha находится здесь: **http://django-simple-captcha.readthedocs.io/**.

17.4.1. Установка Django Simple Captcha

Установка этой библиотеки выполняется отдачей в командной строке следующей команды:

```
pip install django-simple-captcha
```

Django Simple Captcha для успешной работы требует наличия известной библиотеки обработки графики Pillow. Если таковая на компьютере отсутствует, она будет установлена автоматически.

Библиотека содержит в своем составе Django-приложение `captcha`, включающее модель, миграцию, список маршрутов, контроллеры, класс поля для формы и пр.

Чтобы задействовать библиотеку после установки, необходимо:

- добавить входящее в ее состав приложение `captcha` в список приложений, зарегистрированных в проекте (параметр `INSTALLED_APPS` настроек проекта, подробнее — в *разд. 3.3.3*):

```
INSTALLED_APPS = [
    . . .
    'captcha',
]
```

- выполнить миграции:

```
manage.py migrate
```

- в списке маршрутов уровня проекта (в модуле urls.py пакета конфигурации) создать маршрут, связывающий префикс `captcha` и список маршрутов из модуля `captcha.urls`:

```
urlpatterns = [
    . . .
    path('captcha/', include('captcha.urls')),
]
```

На заметку

Для своих нужд приложение `captcha` создает в базе данных таблицу `captcha_captchastore`. Она хранит сгенерированные на данный момент CAPTCHA и время их устаревания.

17.4.2. Использование Django Simple Captcha

Использовать библиотеку для обработки CAPTCHA очень просто. Достаточно объявить в форме, в которой должна присутствовать CAPTCHA, поле типа `CaptchaField`, объявленного в модуле `captcha.fields`. Это может быть как форма, связанная с моделью:

```
from django import forms
from captcha.fields import CaptchaField

class CommentForm(forms.ModelForm):
    . . .
    captcha = CaptchaField()
    class Meta:
        model = Comment
```

так и обычная форма, наподобие тех, с которыми мы познакомились в этой главе:

```
class BbForm(forms.Form):
    . . .
    captcha = CaptchaField(label='Введите текст с картинки',
              error_messages={'invalid': 'Неправильный текст'})
```

Как видим, поле этого типа поддерживает все параметры, единые для всех типов полей (см. *разд. 13.1.3.2*).

На веб-странице элемент управления, представляющий CAPTCHA, выглядит так, как показано на рис. 17.1.

Проверка правильности ввода CAPTCHA будет выполняться при валидации формы. Если был введен неправильный текст, форма не пройдет валидацию и будет повторно выведена на экран с указанием сообщения об ошибке.

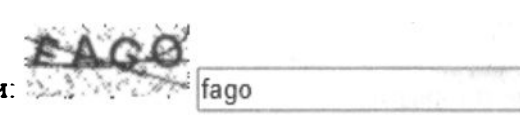


Рис. 17.1. CAPTCHA на веб-странице

17.4.3. Настройка Django Simple Captcha

Библиотека Django Simple Captcha предоставляет довольно богатые возможности настройки. Все поддерживаемые ею параметры можно увидеть в документации, доступной на домашнем сайте библиотеки, а далее приведены наиболее полезные из них:

- `CAPTCHA_CHALLENGE_FUNCT` — полное имя функции, генерирующей текст для CAPTCHA, в виде строки. В библиотеке доступны следующие функции:
  - `captcha.helpers.random_char_challenge` — классическая CAPTCHA, представляющая собой случайный набор из четырех букв. Не чувствительна к регистру;
  - `captcha.helpers.math_challenge` — математическая CAPTCHA, в которой посетителю нужно вычислить результат арифметического выражения и ввести получившийся результат;
  - `captcha.helpers.word_challenge` — словарная CAPTCHA, представляющая собой случайно выбранное слово из заданного словаря.

  Пример:

  ```
  CAPTCHA_CHALLENGE_FUNCT = 'captcha.helpers.math_challenge'
  ```

  Значение параметра по умолчанию: `"captcha.helpers.random_char_challenge"`;
- `CAPTCHA_LENGTH` — длина CAPTCHA в символах текста. Принимается во внимание только при использовании классической CAPTCHA. Значение по умолчанию: `4`;
- `CAPTCHA_MATH_CHALLENGE_OPERATOR` — строка с символом, обозначающим оператор умножения. Принимается во внимание только при использовании математической CAPTCHA. Значение по умолчанию: `"*"`. Пример указания крестика в качестве оператора умножения:

  ```
  CAPTCHA_MATH_CHALLENGE_OPERATOR = 'x'
  ```
- `CAPTCHA_WORDS_DICTIONARY` — полный путь к файлу со словарем, используемым в случае выбора словарной CAPTCHA. Словарь должен представлять собой текстовый файл, в котором каждое слово находится на отдельной строке;
- `CAPTCHA_DICTIONARY_MIN_LENGTH` — минимальная длина слова, взятого из словаря, в символах. Используется в случае выбора словарной CAPTCHA. Значение по умолчанию: `0`;
- `CAPTCHA_DICTIONARY_MAX_LENGTH` — максимальная длина слова, взятого из словаря, в символах. Используется в случае выбора словарной CAPTCHA. Значение по умолчанию: `99`;
- `CAPTCHA_TIMEOUT` — промежуток времени в минутах, в течение которого сгенерированная CAPTCHA останется действительной. Значение по умолчанию: `5`;
- `CAPTCHA_FONT_PATH` — полный путь к файлу шрифта, который будет использоваться для вывода текста. Значение по умолчанию — путь `"<папка, в которой`

установлен Python>\Lib\site-packages\captcha\fonts\Vera.ttf" (шрифт Vera, хранящийся в файле по этому пути, является свободным для распространения);

- CAPTCHA_FONT_SIZE — размер шрифта текста в пикселах. Значение по умолчанию: 22;
- CAPTCHA_LETTER_ROTATION — диапазон углов поворота букв в тексте CAPTCHA в виде кортежа, элементы которого укажут предельные углы поворота в градусах. Значение по умолчанию: (-35, 35);
- CAPTCHA_FOREGROUND_COLOR — цвет текста на изображении CAPTCHA в любом формате, поддерживаемом CSS. Значение по умолчанию: "#001100" (очень темный, практически черный цвет);
- CAPTCHA_BACKGROUND_COLOR — цвет фона изображения CAPTCHA в любом формате, поддерживаемом CSS. Значение по умолчанию: "#ffffff" (белый цвет);
- CAPTCHA_IMAGE_SIZE — геометрические размеры изображения в виде кортежа, первым элементом которого должна быть ширина, вторым — высота. Размеры исчисляются в пикселах. Если указать None, размер изображения будет устанавливаться самой библиотекой. Значение по умолчанию — None.

17.4.4. Дополнительные команды *captcha_clean* и *captcha_create_pool*

Библиотека Django Simple Captcha добавляет утилите manage.py поддержку двух дополнительных команд, которые мы сейчас рассмотрим.

Команда captcha_clean удаляет из хранилища устаревшие CAPTCHA. Ее формат очень прост:

```
manage.py captcha_clean
```

Команда captcha_create_pool создает набор готовых CAPTCHA для дальнейшего использования, что позволит потом сэкономить время и системные ресурсы на их вывод. Формат команды:

```
manage.py captcha_create_pool
[--pool-size <количество создаваемых CAPTCHA>] [--cleanup-expired]
```

Поддерживаются следующие дополнительные ключи:

- --pool-size — задает количество предварительно создаваемых CAPTCHA. Если не указан, будет создано 1000 CAPTCHA;
- --cleanup-expired — заодно удаляет из хранилища устаревшие CAPTCHA.

17.5. Дополнительные настройки проекта, имеющие отношение к формам

Осталось рассмотреть пару параметров, указываемых в настройках проекта (т. е. в модуле settings.py пакета конфигурации) и имеющих прямое отношение к обработке форм и полученных из них данных:

- DATA_UPLOAD_MAX_MEMORY_SIZE — максимально допустимый размер полученных от посетителя данных в виде числа в байтах. Если этот размер был превышен, генерируется исключение SuspiciousOperation из модуля django.core.exceptions. Если указать значение None, проверка на превышение допустимого объема выполняться не будет. Значение по умолчанию: 2621440 (2,5 Мбайт);
- DATA_UPLOAD_MAX_NUMBER_FIELDS — максимально допустимое количество POST-параметров в полученном запросе (т. е. полей в выведенной форме). Если это количество было превышено, генерируется исключение SuspiciousOperation. Если указать значение None, проверка на превышение допустимого количества значений выполняться не будет. Значение по умолчанию: 1000.

Эти параметры и ограничения установлены для предотвращения сетевых атак *DOS* (Denial Of Service, отказ от обслуживания). Увеличивать значения параметров следует только в крайнем случае, если сайт должен обрабатывать данные большого объема.

ГЛАВА 18

Шаблоны: расширенные инструменты и дополнительные библиотеки

В *главе 11* мы изучили практически все возможности, предлагаемые шаблонизатором Django. Однако для этого фреймворка существуют несколько полезных библиотек, которые расширяют его возможности, — им и посвящена текущая глава. Вдобавок неплохо было бы выяснить, как расширить набор тегов и фильтров, поддерживаемых шаблонизатором Django, — это тоже может пригодиться.

18.1. Библиотека django-precise-bbcode: поддержка BBCode

BBCode (Bulletin Board Code, код досок объявлений) — это язык разметки, который используется для форматирования текста на многих форумах и блогах. Форматирование выполняется с применением набора тегов, схожих с тегами языка HTML, но заключаемых в квадратные скобки. Программное ядро сайта преобразует такие теги в обычный HTML-код.

Для поддержки BBCode в Django-сайтах удобно использовать дополнительную библиотеку django-precise-bbcode. Ее достоинство заключается не только в простоте применения, но и в удобстве расширения набора поддерживаемых тегов, поскольку перечень дополнительных BBCode-тегов, создаваемых разработчиком сайта под свои нужды, хранится в особой модели, которую можно править средствами административного сайта Django.

Помимо поддержки BBCode-тегов, эта библиотека выполняет еще одну важную задачу — заменяет символы перевода строк HTML-тегами `<br>`. В результате чего текст, в процессе ввода разбитый на абзацы, выводится на страницу также разбитым на абзацы, а не в одну строку.

> ***НА ЗАМЕТКУ***
>
> Полную документацию по django-precise-bbcode можно найти здесь: **https://django-precise-bbcode.readthedocs.io/en/stable/index.html**.

18.1.1. Установка django-precise-bbcode

Библиотека django-precise-bbcode содержит приложение `precise_bbcode`, которое и реализует всю функциональность по выводу BBCode-тегов. Оно включает в себя модель и все необходимые классы.

Для установки библиотеки необходимо отдать команду:

```
pip install django-precise-bbcode
```

Помимо самой django-precise-bbcode, будет установлена библиотека обработки графики Pillow, которая необходима для вывода смайликов.

Перед использованием библиотеки следует выполнить следующие шаги:

- добавить приложение `precise_bbcode` в список приложений, зарегистрированных в проекте (параметр `INSTALLED_APPS` настроек проекта, подробнее — в *разд. 3.3.3*):

```
INSTALLED_APPS = [
    . . .
    'precise_bbcode',
]
```

- выполнить миграции.

> ***На заметку***
>
> Для своих нужд приложение `precise_bbcode` создает в базе данных таблицы `precise_bbcode_bbcodetag` и `precise_bbcode_smileytag`. Первая хранит список дополнительных BBCode-тегов, а вторая — список смайликов, которые задаются самим разработчиком сайта.

18.1.2. Поддерживаемые BBCode-теги

Изначально django-precise-bbcode поддерживает лишь общеупотребительные BBCode-теги, которые уже стали стандартом де-факто в Интернете:

- `[b]`*<текст>*`[/b]` — выводит *текст* полужирным шрифтом;
- `[i]`*<текст>*`[/i]` — выводит *текст* курсивным шрифтом;
- `[u]`*<текст>*`[/u]` — выводит *текст* подчеркнутым;
- `[s]`*<текст>*`[/s]` — выводит *текст* зачеркнутым;
- `[img]`*<интернет-адрес>*`[/img]` — выводит изображение, загруженное с заданного *интернет-адреса*;
- `[url]`*<интернет-адрес>*`[/url]` — выводит *интернет-адрес* как гиперссылку;
- `[url=`*<интернет-адрес>*`]`*<текст>*`[/url]` — выводит *текст* как гиперссылку, указывающую на заданный *интернет-адрес*;
- `[color=<цвет>]`*<текст>*`[/color]` — выводит *текст* указанным *цветом*. *Цвет* может быть задан в любом формате, поддерживаемом CSS. Примеры:

```
[color=green]Зеленый текст[/color]
[color=#cccccc]Серый текст[/color]
```

- `[center]<текст>[/center]` — выравнивает *текст* по середине;
- `[list]<набор пунктов>[/list]` — создает маркированный список с указанным *набором пунктов*;
- `[list=<тип нумерации>]<набор пунктов>[/list]` — создает нумерованный список с указанным *набором пунктов*. В качестве *типа нумерации* можно указать:
  - `1` — нумерация арабскими цифрами;
  - `01` — нумерация арабскими цифрами с начальным нулем;
  - `I` — нумерация римскими цифрами в верхнем регистре;
  - `i` — нумерация римскими цифрами в нижнем регистре;
  - `A` — нумерация латинскими буквами в верхнем регистре;
  - `a` — нумерация латинскими буквами в нижнем регистре;
- `[*]<текст пункта списка>` — создает отдельный пункт списка, формируемого тегом `[list]`:

```
[list]
[*]Python
[*]Django
[*]django-precise-bbcode
[/list]
```

- `[quote]<текст>[/quote]` — выводит *текст* как цитату, с отступом слева;
- `[code]<текст>[/code]` — выводит *текст* моноширинным шрифтом.

18.1.3. Обработка BBCode

Библиотека django-precise-bbcode предоставляет два способа обработки BBCode: обработку в процессе вывода и непосредственное хранение в поле модели.

18.1.3.1. Обработка BBCode в процессе вывода

Если мы уже имеем модель, в поле которой хранится текст, отформатированный с применением BBCode-тегов, мы можем выполнить его преобразование в эквивалентный HTML-код. Причем сделать это можно как непосредственно в шаблоне, так и в контроллере.

Чтобы выполнить преобразование BBCode в HTML-код в шаблоне, нужно предварительно загрузить библиотеку тегов с псевдонимом `bbcode_tags`:

```
{% load bbcode_tags %}
```

После чего мы можем воспользоваться одним из двух следующих инструментов:

- тегом `bbcode <выводимый текст>`:

```
{% bbcode bb.content %}
```

- фильтром `bbcode`:

```
{{ bb.content|bbcode|safe }}
```

Недостатком этого фильтра является то, что его необходимо использовать совместно с фильтром `safe`. Если фильтр `safe` не указать, все содержащиеся в выводимом тексте недопустимые знаки HTML будут преобразованы в специальные символы, и мы получим на экране непосредственно сам HTML-код, а не результат его обработки.

Еще мы можем выполнить преобразование BBCode в HTML-код прямо в контроллере. Для этого мы выполним следующие шаги:

- вызовем функцию `get_parser()` из модуля `precise_bbcode.bbcode`. В качестве результата она вернет объект преобразователя;
- вызовем метод `render(<текст BBCode>)` полученного преобразователя. Метод вернет HTML-код, полученный преобразованием заданного *текста BBCode*.

Пример:

```
from precise_bbcode.bbcode import get_parser
def detail(request, pk):
    parser = get_parser()
    bb = Bb.objects.get(pk=pk)
    parsed_content = parser.render(bb.content)
    . . .
```

18.1.3.2. Хранение BBCode в модели

Текст, отформатированный тегами BBCode, можно хранить в обычном поле типа `TextField`. Однако если модель только создается и еще не заполнена данными, лучше выделить под хранение такого текста поле типа `BBCodeTextField` из модуля `precise_bbcode.fields`. Пример:

```
from precise_bbcode.fields import BBCodeTextField

class Bb(models.Model):
    . . .
    content = BBCodeTextField(null=True, blank=True,
                              verbose_name='Описание')
    . . .
```

Поле этого типа поддерживает все параметры, общие для всех классов полей (см. *разд. 4.3.1*), и дополнительные параметры конструктора класса `TextField` (поскольку является производным от этого класса).

Для вывода содержимого такого поля, преобразованного в HTML-код, в шаблоне следует воспользоваться атрибутом `rendered`:

```
{{ bb.content.rendered }}
```

Фильтр `safe` здесь указывать не нужно.

На заметку

При выполнении миграции на каждое поле типа `BBCodeTextField`, объявленное в модели, создаются два поля таблицы. Первое поле хранит изначальный текст, занесенный в соответствующее поле модели, а его имя совпадает с именем этого поля модели. Второе поле хранит HTML-код, полученный в результате преобразования изначального текста, а его имя имеет вид `_<имя первого поля>_rendered`. При выводе содержимого поля типа `BBCodeTextField` в шаблоне путем обращения к атрибуту `rendered` выводимый HTML-код берется из второго поля.

Внимание!

Поле типа `BBCodeTextField` стоит объявлять только в том случае, если в модели нет ни одной записи. Если же его добавить в модель, в которой есть хотя бы одна запись, после выполнения миграции второе поле, хранящее полученный в результате преобразования HTML-код, окажется пустым, и при попытке вывести его содержимое на экран мы ничего не увидим. Единственный способ заполнить второе поле — выполнить правку и сохранение каждой записи модели.

18.1.4. Создание дополнительных BBCode-тегов

Как говорилось ранее, дополнительные BBCode-теги, созданные самим разработчиком сайта, записываются в модели приложения `precise_bbcode` и хранятся в базе данных. Следовательно, мы можем расширить набор поддерживаемых библиотекой тегов без всякого программирования.

На заметку

Библиотека django-precise-bbcode также предоставляет возможность создания дополнительных BBCode-тегов с более сложной реализацией, однако для этого потребуется программирование. Необходимые инструкции вы можете найти на домашнем сайте библиотеки.

Список дополнительных тегов доступен в административном сайте Django. Он находится в приложении **Precise BBCode** и носит название **BBCode tags**. Изначально он пуст.

Для каждого вновь создаваемого дополнительного тега необходимо ввести следующие сведения:

- **Tag definition** — объявление BBCode-тега в том виде, в котором он будет записываться в форматируемом тексте.

  Теги практически всегда имеют какое-либо содержимое или параметры. Содержимое указывается между открывающим и закрывающим тегами (внутри тега), а параметр — в открывающем теге, после его имени и отделяется от него знаком =. Для указания содержимого и параметров применяются следующие специальные символы:

  - `{TEXT}` — совпадает с любым фрагментом текста:

    ```
    [right]{TEXT}[/right]
    [spoiler={TEXT}]{TEXT}[/spoiler]
    ```

  - `{SIMPLETEXT}` — совпадает с текстом, состоящим из букв латиницы, цифр, пробелов, точек, запятых, знаков «плюс», дефисов и подчеркиваний;

- {COLOR} — совпадает с цветом, записанным в одном из форматов, которые поддерживаются CSS:

```
[color={COLOR}]{TEXT}[/color]
```

- {URL} — совпадает с корректным интернет-адресом;
- {EMAIL} — совпадает с корректным адресом электронной почты;
- {NUMBER} — совпадает с числом;
- {RANGE=*<минимум>*,*<максимум>*} — совпадает с числом, находящемся в диапазоне от *минимума* до *максимума*:

```
[digit]{RANGE=0,9}[/digit]
```

- {CHOICE=*<перечень значений, разделенных запятыми>*} — совпадает с любым из указанных *значений*:

```
[platform]{CHOICE=Python,Django,SQLite}[/platform]
```

❐ **Replacement HTML code** — HTML-код, эквивалентный объявляемому BBCode-тегу. Для указания мест в HTML-коде, куда следует вставить содержимое или какие-либо параметры объявляемого тега, используются те же самые специальные символы, что были приведены ранее. Несколько примеров можно увидеть в табл. 18.1;

***Таблица 18.1.** Примеры объявлений тегов BBCode и написания эквивалентного HTML-кода*

Объявление BBCode-тега	Эквивалентный HTML-код

`[right]{TEXT}[/right]`	`<div style="text-align:right;">`
	`  {TEXT}`
	`</div>`

`[spoiler={TEXT}]{TEXT}[/spoiler]`	`<div class="spoiler">`
	`  <div class="header">`
	`  {TEXT}`
	`  </div>`
	`  <div class="content">`
	`  {TEXT}`
	`  </div>`
	`</div>`

`[color={COLOR}]{TEXT}[/color]`	`<span style="color:{COLOR};">` `{TEXT}</span>`

❐ **Standalone tag** — флажок, который нужно установить, если создается одинарный тег, и сбросить, если создаваемый тег — парный. Изначально он сброшен.

Остальные параметры находятся под спойлером **Advanced options**;

❐ **Newline closing** — установка флажка предпишет закрывать тег перед началом новой строки. Изначально он сброшен;

- **Same tag closing** — установка этого флажка предпишет закрывать тег, если в тексте далее встретится такой же тег. Изначально флажок сброшен;
- **End tag closing** — установка этого флажка предпишет закрывать тег перед окончанием содержимого тега, в который он вложен. Изначально флажок сброшен;
- **Transform line breaks** — установка флажка вызовет преобразование переводов строк в соответствующие HTML-теги. Изначально флажок установлен;
- **Render embedded tags** — если флажок установлен, все теги, вложенные в текущий тег, будут соответственно обработаны. Изначально флажок установлен;
- **Escape HTML characters** — установка флажка укажет преобразовывать любые недопустимые знаки HTML («меньше», «больше», амперсанд) в соответствующие специальные символы. Изначально флажок установлен;
- **Replace links** — если флажок установлен, все интернет-адреса, присутствующие в содержимом текущего тега, будут преобразованы в гиперссылки. Изначально флажок установлен;
- **Strip leading and trailing whitespace** — установка флажка приведет к тому, что начальные и конечные пробелы в содержимом тега будут удалены. Изначально флажок сброшен;
- **Swallow trailing newline** — будучи установленным, флажок предпишет удалять первый из следующих за тегом переводов строки. Изначально флажок сброшен.

> ***На заметку***
>
> В перечне задаваемых для нового BBCode-тега сведений присутствуют также поле ввода **Help text for this tag** и флажок **Display on editor**, не описанные в документации по библиотеке. В поле ввода заносится, судя по всему, необязательное краткое описание тега. Назначение флажка неясно.

18.1.5. Создание смайликов

Помимо тегов BBCode, библиотека django-precise-bbcode поддерживает вывод смайликов. Однако, к сожалению, встроенных смайликов она не содержит, и нам придется создать весь желаемый набор самостоятельно.

> ***Внимание!***
>
> Для создания набора смайликов и их вывода будет задействована подсистема Django, обрабатывающая выгруженные пользователями файлы. Чтобы эта подсистема успешно заработала, следует соответственно настроить ее. Как это сделать, будет описано в *главе 19*.

Работа со списком смайликов также выполняется на административном сайте Django. Список этот можно найти в приложении **Precise BBCode** и под названием **Smilies**.

Для каждого создаваемого в списке смайлика нужно указать следующие сведения:

- **Smiley code** — текстовое представление смайлика (примеры: `":-)"`, `";-)"`, `":-("`);

- **Smiley icon** — графическое изображение, представляющее смайлик и выводящееся вместо его текстового представления;
- **Smiley icon width** — необязательная ширина смайлика, выведенного на экран, в пикселах. Если не указана, смайлик при выводе будет иметь свою изначальную ширину;
- **Smiley icon height** — необязательная высота смайлика, выведенного на экран, в пикселах. Если не указана, смайлик при выводе будет иметь свою изначальную высоту.

> ***На заметку***
>
> В перечне задаваемых для нового BBCode-тега сведений присутствуют также поле ввода **Related emotions** и флажок **Display on editor**, не описанные в документации по библиотеке. Их назначение неясно.

18.1.6. Настройка django-precise-bbcode

Библиотека django-precise-bbcode прекрасно работает с настройками по умолчанию. Однако знать о них все же не помешает. Все эти настройки указываются в модуле settings.py пакета конфигурации:

- `BBCODE_NEWLINE` — строка с HTML-кодом, которым при выводе будет заменен перевод строки. Значение по умолчанию: `"<br>"` (тег разрыва строки);
- `BBCODE_ESCAPE_HTML` — набор знаков, которые при выводе должны заменяться специальными символами HTML. Указывается в виде последовательности, каждым элементом которой должна быть последовательность из двух элементов: самого заменяемого знака и соответствующего ему специального символа. Значением по умолчанию является последовательность:

```
( ('&', '&amp;'), ('<', '&lt;'), ('>', '&gt;'), ('"', '&quot;'),
  ('\'', '&#39;'), )
```

- `BBCODE_DISABLE_BUILTIN_TAGS` — если `True`, BBCode-теги, поддержка которых встроена в библиотеку (см. *разд. 18.1.2*), не будут обрабатываться, если `False` — будут. Значение по умолчанию — `False`.

  Этот параметр может пригодиться, если стоит задача полностью изменить набор поддерживаемых библиотекой тегов. В таком случае следует присвоить ему значение `True` и создать все нужные теги самостоятельно, пользуясь инструкциями из *разд. 18.1.4*;

- `BBCODE_ALLOW_CUSTOM_TAGS` — если `True`, дополнительные теги, созданные самим разработчиком сайтов согласно инструкциям из *разд. 18.1.4*, будут обрабатываться библиотекой, если `False` — не будут. Значение по умолчанию — `True`.

  Установка этого параметра в `False` может сэкономить немного системных ресурсов, но только при условии, что будут использоваться только теги, поддержка которых встроена в саму библиотеку (см. *разд. 18.1.2*);

- `BBCODE_ALLOW_CUSTOM_TAGS` — если `True`, все символы `\n`, представляющие переводы строк, будут заменяться последовательностями символов `\r\n`, если `False` — не будут. Значение по умолчанию — `True`;
- `BBCODE_ALLOW_SMILIES` — если `True`, библиотека будет обрабатывать смайлики, если `False` — не будет. Значение по умолчанию — `True`.

  Опять же, установка этого параметра в `False` сэкономит немного системных ресурсов, но только в том случае, если функциональность по выводу смайликов не нужна;
- `SMILIES_UPLOAD_TO` — путь к папке, куда будут записываться выгружаемые файлы с изображениями смайликов. Эта папка должна располагаться в папке выгрузки файлов, поэтому путь к ней указывается относительно пути к этой папке. Значение по умолчанию: `"precise_bbcode/smilies"` (о настройке папки для выгрузки файлов будет рассказано в *главе 19*).

18.2. Библиотека django-bootstrap4: интеграция с Bootstrap

Bootstrap — популярный CSS-фреймворк для быстрой верстки веб-страниц и создания всевозможных интерфейсных элементов наподобие меню, спойлеров и пр. Использовать Bootstrap для оформления Django-сайта, что называется, малой кровью позволяет дополнительная библиотека django-bootstrap4.

Эта библиотека позволяет выводить с использованием Bootstrap веб-формы, пагинатор, предупреждения и всплывающие сообщения (о них разговор пойдет в *главе 22*). Также она включает средства для привязки к формируемым веб-страницам необходимых таблиц стилей и файлов веб-сценариев.

> ***На заметку***
>
> Документацию по библиотеке Bootstrap можно найти здесь: **http://getbootstrap.com/**, а полную документацию по библиотеке django-bootstrap4 здесь: **http://django-bootstrap4.readthedocs.io/en/latest/index.html**.

18.2.1. Установка django-bootstrap4

Действующим началом django-bootstrap4 является приложение `bootstrap4`, которое реализует всю функциональность по выводу совместимого с Bootstrap HTML-кода.

Для установки библиотеки необходимо отдать команду:

```
pip install django-bootstrap4
```

Перед использованием библиотеки необходимо добавить приложение `bootstrap4` в список приложений, зарегистрированных в проекте (параметр `INSTALLED_APPS` настроек проекта, подробнее — в *разд. 3.3.3*):

```
INSTALLED_APPS = [
    . . .
    'bootstrap4',
```

18.2.2. Использование django-bootstrap4

Перед использованием django-bootstrap4 следует загрузить библиотеку тегов с псевдонимом `bootstrap4`:

```
{% load bootstrap4 %}
```

Далее приведены все теги, поддерживаемые этой библиотекой:

- `bootstrap_css` — вставляет в шаблон HTML-код, привязывающий к странице таблицу стилей Bootstrap;
- `bootstrap_javascript [jquery=<редакция jQuery>]` — вставляет в шаблон HTML-код, привязывающий к странице файлы веб-сценариев Bootstrap и jQuery (эта библиотека требуется для реализации некоторых эффектов и, к тому же, может оказаться полезной при программировании веб-сценариев). В качестве *редакции jQuery*, привязываемой к странице, можно указать одну из следующих строк:
  - `"falsy"` — вообще не привязывать jQuery (поведение по умолчанию). Однако при этом не будут работать сложные элементы страниц, создаваемые средствами Bootstrap, наподобие раскрывающихся меню;
  - `"slim"` — привязать сокращенную редакцию jQuery, из которой исключены инструменты для работы с AJAX и реализации анимационных эффектов. Тем не менее сложные элементы страниц, которые можно создать средствами Bootstrap, работать будут;
  - `"full"` — привязать полную редакцию jQuery.

  Пример:

```
<html>
    <head>
        . . .
        {% bootstrap_css %}
        {% bootstrap_javascript jquery='full' %}
        . . .
    </head>
    <body>
        . . .
    </body>
</html>
```

- `bootstrap_css_url` — вставляет в шаблон интернет-адрес файла, в котором хранится таблица стилей Bootstrap;
- `bootstrap_jquery_url` — вставляет в шаблон интернет-адрес файла веб-сценариев полной редакции jQuery;
- `bootstrap_jquery_slim_url` — вставляет в шаблон интернет-адрес файла веб-сценариев сокращенной редакции jQuery;
- `bootstrap_popper_url` — вставляет в шаблон интернет-адрес файла веб-сценариев jQuery-плагина Popper, необходимого для работы Bootstrap;

- `bootstrap_javascript_url` — вставляет в шаблон интернет-адрес файла веб-сценариев Bootstrap.

  Отметим, что файлы веб-сценариев должны быть привязаны в следующем порядке: jQuery, Popper, Bootstrap. Если этот порядок нарушен, библиотека работать не будет.

  Пример:

```
<link href="{% bootstrap_css %}" rel="stylesheet">
<script src="{% bootstrap_jquery_slim_url %}"></script>
<script src="{% bootstrap_popper_url %}"></script>
<script src="{% bootstrap_javascript_url %}"></script>
```

- `bootstrap_form` *<форма>* [exclude=*<список имен полей, которые не должны выводиться на экран>*] [*<параметры оформления>*] — выводит указанную *форму*:

```
{% load bootstrap4 %}

<form method="post">
    {% csrf_token %}
    {% bootstrap_form form %}
    {% buttons submit='Добавить' %}{% endbuttons %}
</form>
```

  В необязательном параметре `exclude` можно указать *список имен полей, которые не должны выводиться на экран*, приведя их через запятую:

```
{% bootstrap_form form exclude='price,rubric' %}
```

  Параметров оформления поддерживается весьма много. Они будут действовать на все поля формы, выводящиеся на экран (т. е. не упомянутые в параметре `exclude`)

  - `layout` — указывает разметку полей формы. Для указания доступны следующие строковые значения:
    - `"vertical"` — надписи выводятся над элементами управления (разметка по умолчанию);
    - `"horizontal"` — надписи выводятся слева от элементов управления;
    - `"inline"` — надписи выводятся непосредственно в элементах управления. Отметим, что такая разметка поддерживается только для полей ввода и областей редактирования, у всех остальных элементов управления надписи не будут выведены вообще;
  - `form_group_class` — имя стилевого класса, что будет привязан к блокам (тегам `<div>`), в которые заключается каждый элемент управления вместе с относящейся к нему надписью. Значение по умолчанию: `"form_group"`;
  - `field_class` — имя стилевого класса, что будет привязан к блокам, в которые заключается каждый элемент управления (но не относящаяся к нему надпись). Значение по умолчанию — «пустая» строка (т. е. никакой стилевой класс не будет привязан к блокам);

- `label_class` — имя стилевого класса, что будет привязан к тегам `<label>`, создающим надписи. Значение по умолчанию — «пустая» строка;
- `show_help` — если `True`, для полей будет выведен дополнительный поясняющий текст (разумеется, если он задан), если `False` — не будет выведен. Значение по умолчанию — `True`;
- `show_label` — если `True`, у элементов управления будут выводиться надписи, если `False` — не будут. Значение по умолчанию — `True`;
- `size` — размер элемента управления и его надписи. Доступны для указания строковые значения `"small"` (маленькие элементы управления), `"medium"` (средние) и `"large"` (большие). Значение по умолчанию: `"medium"`;
- `horizontal_label_class` — имя стилевого класса, который будет привязан к тегам `<label>`, создающим надписи, если используется разметка `"horizontal"`. Значение по умолчанию: `"col-md-3"` (может быть изменено в настройках библиотеки);
- `horizontal_field_class` — имя стилевого класса, который будет привязан к тегам `<div>`, что заключают в себе элементы управления, если используется разметка `"horizontal"`. Значение по умолчанию: `"col-md-9"` (может быть изменено в настройках библиотеки);
- `required_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который обязательно следует занести значение. Значение по умолчанию — «пустая» строка;
- `bound_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который занесено корректное значение. Значение по умолчанию: `"has-success"`;
- `error_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который занесены некорректные данные. Значение по умолчанию: `"has-error"`.

Пример:

```
{% bootstrap_form form layout='horizontal' show_help=False ↵
size='small' %}
```

- `bootstrap_form_errors` *<форма>* `[type=`*<тип ошибок>*`]` — выводит список ошибок, допущенных посетителем при занесении данных в указанную *форму*. В качестве *типа ошибок* можно указать одну из следующих строк:
  - `"all"` — все ошибки (значение по умолчанию);
  - `"fields"` — ошибки, относящиеся к полям формы;
  - `"non_fields"` — ошибки, относящиеся к форме в целом.

Этот тег имеет смысл применять только в том случае, если нужно вывести сообщения об ошибках в каком-то определенном месте страницы. В противном

случае можно положиться на тег `bootstrap_form`, который выводит такие сообщения непосредственно в форме;

- `bootstrap_formset` *<набор форм>* [*<параметры оформления>*] — выводит указанный *набор форм*:

```
{% load bootstrap4 %}

<form method="post">
    {% csrf_token %}
    {% bootstrap_formset formset %}
    {% buttons submit='Сохранить' %}{% endbuttons %}
</form>
```

Здесь поддерживаются те же *параметры оформления*, что были рассмотрены в рассказе о теге `bootstrap_form`;

- `bootstrap_formset_errors` *<набор форм>* — выводит список ошибок, допущенных посетителем при занесении данных в указанный *набор форм*.

Этот тег имеет смысл применять только в том случае, если нужно вывести сообщения об ошибках в каком-то определенном месте страницы. В противном случае можно положиться на тег `bootstrap_formset`, который выводит такие сообщения непосредственно в наборе форм;

- `buttons submit=`*<текст надписи>* [`reset=`*<текст надписи>*] . . . `endbuttons` — выводит кнопку отправки данных и, возможно, кнопку сброса формы. Параметр `submit` задает текст надписи для кнопки отправки данных. Необязательный параметр `reset` задает текст надписи для кнопки сброса формы; если он не указан, такая кнопка не будет создана. Пример:

```
{% buttons submit='Добавить объявление' %}{% endbuttons %}
```

- `bootstrap_button` *<параметры кнопки>* — выводит кнопку. Поддерживаются следующие *параметры* создаваемой кнопки:
  - `content` — текст надписи для кнопки;
  - `button_type` — тип кнопки. Доступны строковые значения `"submit"` (кнопка отправки данных), `"reset"` (кнопка сброса формы), `"button"` (обычная кнопка) и `"link"` (кнопка-гиперссылка);
  - `href` — интернет-адрес для кнопки-гиперссылки;
  - `button_class` — имя стилевого класса, который будет привязан к кнопке. Значение по умолчанию: `"btn-default"`;
  - `extra_classes` — имена дополнительных стилевых классов, которые следует привязать к кнопке, перечисленные через пробел. Значение по умолчанию — «пустая» строка;
  - `size` — размер кнопки. Доступны для указания строковые значения `"xs"` (самый маленький размер), `"sm"`, `"small"` (маленький), `"md"`, `"medium"` (средний), `"lg"`, `"large"` (большой). По умолчанию создается кнопка среднего размера;

- `name` — значение для атрибута `name` тега `<button>`, создающего кнопку;
- `value` — значение для атрибута `value` тега `<button>`, создающего кнопку.

Пример:

```
{% bootstrap_button content='Сохранить' button_type='submit' ↳
button_class='btn-primary' %}
```

- `bootstrap_field` *<поле формы>* [*<параметры оформления>*] [*<дополнительные параметры оформления>*] — выводит указанное *поле формы*:

```
{% bootstrap_field form.title %}
```

Здесь поддерживаются те же *параметры оформления*, что были рассмотрены в рассказе о теге `bootstrap_form`. Помимо этого, поддерживаются *дополнительные параметры оформления*:

- `placeholder` — текст, который будет выводиться непосредственно в элементе управления, если используется разметка, отличная от `"inline"`. Поддерживается только для полей ввода и областей редактирования;
- `addon_before` — текст, который будет помещен перед элементом управления. Значение по умолчанию — «пустая» строка;
- `addon_before_class` — имя стилевого класса, который будет привязан к тегу `<span>`, охватывающему текст, что помещается перед элементом управления. Если указать значение `None`, тег `<span>` создаваться не будет. Значение по умолчанию: `"input-group-text"`;
- `addon_after` — текст, который будет помещен после элемента управления. Значение по умолчанию — «пустая» строка;
- `addon_after_class` — имя стилевого класса, который будет привязан к тегу `<span>`, охватывающему текст, что помещается после элемента управления. Если указать значение `None`, тег `<span>` создаваться не будет. Значение по умолчанию: `"input-group-text"`.

Пример:

```
{% bootstrap_field form.title placeholder='Товар' show_label=False %}
```

- `bootstrap_messages` — выводит всплывающие сообщения (см. *главу 22*):

```
{% bootstrap_messages %}
```

- `bootstrap_alert` *<параметры предупреждения>* — выводит предупреждение с заданными *параметрами*:
  - `content` — HTML-код, создающий содержимое предупреждения;
  - `alert_type` — тип предупреждения. Доступны строковые значения `"info"` (простое сообщение), `"warning"` (предупреждение о некритической ситуации), `"danger"` (предупреждение о критической ситуации) и `"success"` (сообщение об успехе выполнения какой-либо операции). Значение по умолчанию: `"info"`;

- `dismissable` — если `True`, в сообщении будет присутствовать кнопка закрытия в виде крестика, щелкнув на которой, посетитель уберет предупреждение со страницы. Если `False`, кнопка закрытия не выведется, и предупреждение будет присутствовать на странице постоянно. Значение по умолчанию — `True`.

Пример:

```
{% bootstrap_alert content='<p>Рубрика добавлена</p>' ↳
alert_type='success' %}
```

- `bootstrap_label` *<параметры надписи>* — выводит надпись. Поддерживаются следующие *параметры* создаваемой надписи:
  - `content` — текст надписи;
  - `label_for` — значение, которое будет присвоено атрибуту `for` тега `<label>`, создающего надпись;
  - `label_class` — имя стилевого класса, который будет привязан к тегу `<label>`, создающему надпись;
  - `label_for` — текст всплывающей подсказки для надписи;
- `bootstrap_pagination` *<часть пагинатора>* [*<параметры пагинатора>*] — выводит пагинатор на основе заданной *части* (представленной экземпляром класса `Page`, описанного в *разд. 12.2*). Поддерживаются дополнительные *параметры пагинатора*:
  - `pages_to_show` — количество гиперссылок, указывающих на части пагинатора, которые будут выведены на страницу (остальные будут скрыты). Значение по умолчанию: `11` (текущая часть плюс по 5 частей предыдущих и следующих);
  - `url` — интернет-адрес, на основе которого будут формироваться интернет-адреса отдельных частей пагинатора. Если указать значение `None`, будет использован текущий интернет-адрес. Значение по умолчанию — `None`;
  - `size` — размер гиперссылок, ведущих на части пагинатора. Можно указать значения `"small"` (маленькие гиперссылки), `None` (среднего размера) или `"large"` (большие). Значение по умолчанию — `None`;
  - `parameter_name` — имя GET-параметра, через который передается номер текущей части. Значение по умолчанию: `"page"`.

Пример:

```
{% bootstrap_pagination page size="small" %}
```

18.2.3. Настройка django-bootstrap4

Все настройки библиотеки django-bootstrap4 записываются в параметре `BOOTSTRAP4` модуля settings.py пакета конфигурации. Значением этого параметра должен быть словарь, отдельные элементы которого представляют отдельные параметры библиотеки. Пример:

```
BOOTSTRAP4 = {
    'required_css_class': 'required',
    'success_css_class': 'has-success',
    'error_css_class': 'has-error',
}
```

Список наиболее полезных параметров приведен далее.

- `horizontal_label_class` — имя стилевого класса, который будет привязан к тегам `<label>`, создающим надписи, если используется разметка `"horizontal"`. Значение по умолчанию: `"col-md-3"`;
- `horizontal_field_class` — имя стилевого класса, который будет привязан к тегам `<div>`, что заключают в себе элементы управления, если используется разметка `"horizontal"`. Значение по умолчанию: `"col-md-9"`;
- `required_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который обязательно следует занести значение. Значение по умолчанию — «пустая» строка;
- `success_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который занесено корректное значение. Значение по умолчанию: `"has-success"`;
- `error_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который занесены некорректные данные. Значение по умолчанию: `"has-error"`;
- `base_url` — базовый интернет-адрес фреймворка Bootstrap, с которого будут загружаться все необходимые файлы таблицы стилей и веб-сценариев, составляющих этот фреймворк. Значение по умолчанию: `"//maxcdn.bootstrapcdn.com/bootstrap/4.0.0/"`;
- `css_url` — интернет-адрес файла, в котором хранится таблица стилей фреймворка Bootstrap. Если указать значение `None`, интернет-адрес будет формироваться автоматически на основе базового адреса, заданного в параметре `base_url`. Значение по умолчанию — `None`;
- `javascript_url` — интернет-адрес файла веб-сценария с кодом фреймворка Bootstrap. Если указать значение `None`, интернет-адрес будет формироваться автоматически на основе базового адреса, заданного в параметре `base_url`. Значение по умолчанию — `None`;
- `jquery_url` — интернет-адрес файла с библиотекой jQuery. Значение по умолчанию: `"//code.jquery.com/jquery.min.js"`.

Эти параметры могут пригодиться, если нужно загрузить файлы фреймворка Bootstrap с какого-либо другого местоположения.

18.3. Написание своих фильтров и тегов

Мы можем расширить функциональные возможности шаблонизатора, в частности, набор поддерживаемых им фильтров и тегов. Сейчас мы выясним, как написать свой собственный фильтр и самую простую разновидность тега (более сложные разновидности приходится разрабатывать много реже).

18.3.1. Организация исходного кода

Модули с кодом, объявляющим фильтры и теги шаблонизатора, должны находиться в пакете templatetags пакета приложения. Поэтому сразу же создадим в пакете приложения папку с именем templatetags, а в ней — «пустой» модуль __init__.py.

> ***Внимание!***
>
> Если отладочный веб-сервер Django запущен, после создания пакета *templatetags* его следует перезапустить, чтобы он перезагрузил обновленный код сайта с вновь созданными модулями.

Вот схема организации исходного кода для приложения `bboard` (предполагается, что код фильтров и тегов хранится в модуле filtersandtags.py):

```
<папка проекта>
    bboard
        __.init__.py
        . . .
        templatetags
            __.init__.py
            filtersandtags.py
        . . .
```

Каждый модуль, объявляющий фильтры и теги, становится библиотекой тегов. Псевдоним этой библиотеки совпадает с именем модуля.

18.3.2. Написание фильтров

Проще всего написать фильтр, который принимает какое-либо значение и, возможно, набор параметров, после чего возвращает то же самое значение в преобразованном виде.

18.3.2.1. Написание и использование простейших фильтров

Фильтр — это обычная функция Django, которая:

- в качестве первого параметра принимает обрабатываемое значение;
- в качестве последующих параметров принимает значения параметров, указанных у фильтра. Эти параметры могут иметь значения по умолчанию;
- возвращает в качестве результата преобразованное значение.

Но написанная нами функция — еще не фильтр. Ее еще нужно зарегистрировать в шаблонизаторе в качестве фильтра.

Сначала необходимо создать экземпляр класса `Library` из модуля `django.template`. Потом у этого экземпляра класса следует вызвать метод `filter()` в следующем формате:

```
filter(<имя регистрируемого фильтра>,
       <ссылка на функцию, реализующую фильтр>)
```

Зарегистрированный фильтр будет доступен в шаблоне под указанным *именем*.

Листинг 18.1 показывает пример объявления и регистрации фильтра `currency`. Он принимает числовое значение и, в качестве необязательного параметра, обозначение денежной единицы. В качестве результата он возвращает строку с числовым значением, отформатированном как денежная сумма.

Листинг 18.1. Пример создания фильтра

```
from django import template

register = template.Library()

def currency(value, name='руб.'):
    return '%1.2f %s' % (value, name)

register.filter('currency', currency)
```

Вызов метода `filter()` можно оформить как декоратор. В таком случае он указывается у функции, реализующей фильтр, и вызывается без параметров. Пример:

```
@register.filter
def currency(value, name='руб.'):
    . . .
```

В необязательном параметре `name` можно указать другое имя, под которым фильтр будет доступен в шаблоне:

```
@register.filter(name='cur')
def currency(value, name='руб.'):
    . . .
```

Может случиться так, что фильтр в качестве обрабатываемого должен принимать значение исключительно строкового типа, но ему было передано значение, чей тип отличается от строки (например, число). В этом случае при попытке обработать такое значение, как строку (скажем, при вызове у него метода, который поддерживается только строковым типом), возникнет ошибка. Но мы можем указать Django предварительно преобразовать нестроковое значение в строку. Для этого достаточно указать для функции, реализующей фильтр, декоратор `stringfilter` из модуля `django.template.defaultfilters`. Пример:

```
from django.template.defaultfilters import stringfilter
. . .
@register.filter
@stringfilter
def somefilter(value):
    . . .
```

Если фильтр в качестве обрабатываемого значения принимает дату и время, мы можем указать, чтобы это значение было автоматически преобразовано в местное время в текущей временной зоне. Для этого нужно указать у декоратора `filter` параметр `expects_localtime` со значением `True`. Пример:

```
@register.filter(expects_localtime=True)
def datetimefilter(value):
    . . .
```

Объявив фильтр, мы можем использовать его в шаблонах. Ранее говорилось, что модуль, объявляющий фильтры, становится библиотекой тегов, чей псевдоним совпадает с именем модуля. Следовательно, чтобы использовать фильтр, нам нужно предварительно загрузить нужную библиотеку тегов с помощью тега `load`. Пример (предполагается, что фильтр `currency` объявлен в модуле **filtersandtags.py**):

```
{% load filtersandtags %}
```

Объявленный нами фильтр используется так же, как и любой из встроенных в Django:

```
{{ bb.price|currency }}
{{ bb.price|currency:'p.' }}
```

18.3.2.2. Управление заменой недопустимых знаков HTML

Если в выводимом на страницу значении присутствует какой-либо из недопустимых знаков HTML: символ «меньше», «больше», двойная кавычка, амперсанд, — он должен быть преобразован в соответствующий ему специальный символ. В коде фильтров для этого можно использовать две функции из модуля `django.utils.html`:

- `escape(`*<строка>*`)` — выполняет замену всех недопустимых знаков в *строке* и возвращает обработанную строку в качестве результата;
- `conditional_escape(`*<строка>*`)` — то же самое, что `escape()`, но выполняет замену только в том случае, если в переданной ему *строке* такая замена еще не производилась.

Результат, возвращаемый фильтром, должен быть помечен как строка, в которой была выполнена замена недопустимых знаков. Сделать это можно, вызвав функцию `mark_safe(`*<помечаемая строка>*`)` из модуля `django.utils.safestring` и вернув из фильтра возвращенный ей результат. Пример:

```
from django.utils.safestring import mark_safe
. . .
@register.filter
```

```
def somefilter(value):
    . . .
    return mark_safe(result_string)
```

Строка, помеченная как прошедшая замену недопустимых знаков, представляется экземпляром класса `SafeText` из того же модуля `django.utils.safestring`. Так что мы можем проверить, проходила ли полученная фильтром в качестве параметра обрабатываемая строка процедуру замены или еще нет. Пример:

```
from django.utils.html import escape
from django.utils.safestring import SafeText
. . .
@register.filter
def somefilter(value):
    if not isinstance(value, SafeText):
        # Полученная строка не прошла замену. Выполняем ее сами
        value = escape(value)
    . . .
```

Еще нужно учитывать тот факт, что разработчик шаблонов может отключить автоматическую замену недопустимых знаков для какого-либо фрагмента кода, заключив его в тег шаблонизатора `autoescape . . . endautoescape`. Чтобы в коде фильтра выяснить, была ли отключена автоматическая замена, следует указать в вызове декоратора `filter()` параметр `needs_autoescape` со значением по умолчанию `True` и добавить в список параметров функции, реализующий фильтр, параметр `autoescape` также со значением по умолчанию `True`. В результате последний получит значение `True`, если автоматическая замена активна, и `False`, если она была отключена. Пример:

```
@register.filter(needs_autoescape=True)
def somefilter(value, autoescape=True):
    if autoescape:
        value = escape(value)
    . . .
```

И, наконец, можно уведомить Django, что он сам должен выполнять замену в значении, возвращенном нашим фильтром. Для этого достаточно указать в вызове декоратора `filter()` параметр `is_safe` со значением `True`. Пример:

```
@register.filter(is_safe=True)
def currency(value, name='руб.'):
    . . .
```

18.3.3. Написание тегов

Объявить простейший одинарный тег шаблонизатора, вставляющий какие-либо данные в то место, где он находится, немногим сложнее, чем создать фильтр.

18.3.3.1. Написание тегов, выводящих элементарные значения

Если объявляемый тег должен выводить какое-либо элементарное значение: строку, число или дату, — наша работа заключается лишь в объявлении функции, которая реализует этот тег.

Функция, реализующая тег, может принимать произвольное количество параметров, как обязательных, так и необязательных. В качестве результата она должна возвращать строковое значение, которое будет помещено в то место кода, в котором находится сам тег.

Подобного рода тег регистрируется почти так же, как и фильтр: созданием экземпляра класса `Library` из модуля `template` и вызовом у этого экземпляра метода `simple_tag([name=None][,][takes_context=False])`, причем вызов нужно оформить в виде декоратора у функции, реализующей тег.

Листинг 18.2 показывает объявление тега `lst`. Он принимает произвольное количество параметров, из которых первый — строка-разделитель — является обязательным, выводит на экран значения остальных параметров, отделяя их друг от друга строкой-разделителем, а в конце ставит количество выведенных значений, взятое в скобки.

Листинг 18.2. Пример объявления тега, выводящего элементарное значение

```
from django import template

register = template.Library()

@register.simple_tag
def lst(sep, *args):
    return '%s (итого %s)' % (sep.join(args), len(args))
```

По умолчанию созданный таким образом тег доступен в коде шаблона под своим изначальным именем, которое совпадает с именем функции, реализующей тег.

В вызове метода `simple_tag()` мы можем указать два необязательных именованных параметра:

- `name` — имя, под которым тег будет доступен в коде шаблона. Используется, если нужно указать для тега другое имя;
- `takes_context` — если `True`, первым параметром в функцию, реализующую тег, будет передан контекст шаблона:

```
@register.simple_tag(takes_context=True)
def lst(context, sep, *args):
    . . .
```

Значение, возвращенное таким тегом, подвергается автоматической замене недопустимых знаков HTML на специальные символы. Так что нам самим это делать не придется.

Если же замену недопустимых знаков проводить не нужно (например, мы решили написать тег шаблонизатора, который помещает на страницу фрагмент HTML-кода), возвращаемое функцией значение можно «пропустить» через функцию `mark_safe()`, описанную в *разд. 18.3.1.2*.

Объявив тег, мы можем использовать его в шаблоне (не забыв загрузить модуль, в котором он реализован):

```
{% load filtersandtags %}
. . .
{% lst ', ' '1' '2' '3' '4' '5' '6' '7' '8' '9' %}
```

18.3.3.2. Написание шаблонных тегов

Если тег, который мы пишем, должен выводить не просто строку, а фрагмент HTML-кода, мы можем, как говорилось ранее, «пропустить» возвращаемую строку через функцию `mark_safe()`. Вот пример подобного рода тега:

```
@register.simple_tag
def lst(sep, *args):
    return mark_safe('%s (итого <strong>%s</strong>)' %
                     (sep.join(args), len(args)))
```

После чего сразу увидим, что количество позиций, что мы передали тегу в качестве параметров, будет выведено полужирным шрифтом.

Но если нам нужно выводить более сложные фрагменты HTML-кода, удобнее объявить *шаблонный тег*. Возвращаемое им значение формируется так же, как и обычная веб-страница Django-сайта, — объединением указанного шаблона и контекста данных.

Функция, реализующая такой тег, должна возвращать в качестве результата контекст шаблона. В качестве декоратора, указываемого для этой функции, нужно поместить вызов метода `inclusion_tag()` экземпляра класса `Library`. Вот формат вызова этого метода:

```
inclusion_tag(<путь к шаблону>[, name=None][, takes_context=False])
```

Листинг 18.3 показывает пример шаблонного тега `ulist`. Он аналогичен объявленному в листинге 18.2 тегу `lst`, но выводит перечень переданных ему позиций в виде маркированного списка HTML, а количество позиций помещает под списком и выделяет курсивом.

Листинг 18.3. Пример шаблонного тега

```
from django import template

register = template.Library()

@register.inclusion_tag('tags/ulist.html')
def ulist(*args):
    return {'items': args}
```

Шаблон для такого тега ничем не отличается от шаблонов веб-страниц и располагается там же. Код шаблона tags\ulist.html для нашего тега `ulist` показан в листинге 18.4.

Листинг 18.4. Шаблон для тега из листинга 18.3

```
<ul>
    {% for item in items %}
    <li>{{ item }}</li>
    {% endfor %}
</ul>
<p>Итого <em>{{ items|length }}</em></p>
```

Метод `inclusion_tag()` поддерживает необязательные именованные параметры `name` и `takes_context`, знакомые нам по *разд. 18.3.2.1*. При указании значения `True` для параметра `takes_context` контекст шаблона страницы также будет доступен в шаблоне тега.

Используется шаблонный тег так же, как и обычный, возвращающий элементарное значение:

```
{% load filtersandtags %}
. . .
{% ulist '1' '2' '3' '4' '5' '6' '7' '8' '9' %}
```

> ***Внимание!***
>
> Django также поддерживает объявление более сложных тегов, являющихся парными и выполняющих над своим содержимым различные манипуляции (в качестве примера можно привести тег `for . . . endfor`). Поскольку потребность в разработке новых тегов такого рода возникает нечасто, их объявление не описывается в этой книге. Интересующиеся могут найти руководство по этой теме на странице **https://docs.djangoproject.com/en/2.1/howto/custom-template-tags/**.

18.3.4. Регистрация фильтров и тегов

Если при объявлении фильтров и тегов мы соблюдали принятые в Django соглашения, а именно поместили модуль с объявлениями в пакете templatetags пакета приложения, нам не придется регистрировать объявленные фильтры и теги где-либо в настройках проекта. В таком случае модуль с объявлениями будет обработан фреймворком как библиотека тегов, имя модуля станет псевдонимом этой библиотеки, и нам останется лишь загрузить ее, воспользовавшись тегом `load` шаблонизатора.

Но если мы не последовали этим соглашениям или хотим изменить псевдоним библиотеки тегов, нам придется внести исправления в настройки проекта, касающиеся обработки шаблонов. Эти настройки описывались в *разд. 11.1*.

Если мы хотим зарегистрировать модуль с фильтрами и тегами как загружаемую библиотеку тегов (т. е. требующую загрузки тегом `load` шаблонизатора), мы занесем его в список загружаемых библиотек тегов. Этот список хранится в дополни-

тельных настройках шаблонизатора, задаваемых параметром OPTIONS в параметре libraries.

Предположим, что модуль filtersandtags.py, в котором мы записали наши фильтры и теги, находится непосредственно в пакете приложения (что нарушает соглашения Django). Тогда зарегистрировать его мы можем, записав в модуле settings.py пакета конфигурации такой код (выделен полужирным шрифтом):

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            . . .
            'libraries': {
                'filtersandtags': 'bboard.filtersandtags',
            }
        },
    },
]
```

Нам даже не придется переделывать код шаблонов, т. к. написанная нами библиотека тегов будет доступна под тем же псевдонимом filtersandtags.

Мы можем изменить псевдоним этой библиотеки тегов, скажем, на ft:

```
'libraries': {
    'ft': 'bboard.filtersandtags',
}
```

и сможем использовать для ее загрузки новое, более короткое имя:

```
{% load ft %}
```

Если же у нас нет ни малейшего желания писать в каждом шаблоне тег load, чтобы загрузить библиотеку тегов, мы можем оформить его как встраиваемую — и объявленные в ней фильтры и теги станут доступными без каких бы то ни было дополнительных действий. Для этого достаточно указать путь к модулю библиотеки тегов в списке дополнительного параметра builtins. Вот пример (добавленный код выделен полужирным шрифтом):

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            . . .
            'builtins': [
                'bboard.filtersandtags',
            ],
        },
    },
]
```

После чего мы сможем просто использовать все объявленные в библиотеке теги, когда и где нам заблагорассудится.

18.4. Переопределение шаблонов

Предположим, нам нужно реализовать выход с сайта с выводом страницы с сообщением о выходе. Для этого мы решаем использовать стандартный контроллер-класс `LogoutView`, описанный в *разд. 15.4.2*, и указанный для него по умолчанию шаблон registration\logged_out.html. Мы пишем шаблон logged_out.html, помещаем его в папку registration, вложенную в папку templates пакета приложения, запускаем отладочный веб-сервер, выполняем вход на сайт, переходим на страницу выхода... и наблюдаем на экране не страницу, созданную на основе написанного нами шаблона, а какую-то другую. Судя по внешнему виду, эта страница принадлежит административному сайту Django...

Все дело в том, что Django в поисках нужного шаблона просматривает папки templates, находящиеся в пакетах *всех* приложений, что зарегистрированы в проекте, и прекращает поиски, как только найдет первый подходящий шаблон. В нашем случае таким оказался шаблон registration\logged_out.html, принадлежащий стандартному приложению `django.contrib.admin`, т. е. административному сайту.

Мы можем избежать этой проблемы, просто задав для контроллера-класса шаблон с другим именем. Это можно сделать либо в маршруте, вставив нужный параметр в вызов метода `as_view()` контроллера-класса, либо в его подклассе, в соответствующем атрибуте. Но есть и другой способ — использовать *переопределение шаблонов*, «подсунув» Django наш шаблон до того, как он доберется до стандартного.

Есть два способа реализовать переопределение шаблонов. Какой из них использовать, каждый разработчик решает для себя сам:

- приложения в поисках шаблонов просматриваются в том порядке, в котором они указаны в списке зарегистрированных приложений, что хранится в параметре `INSTALLED_APPS` настроек проекта. Следовательно, мы можем просто поместить наше приложение перед стандартным. Пример (предполагается, что нужный шаблон находится в приложении `bboard`):

```
INSTALLED_APPS = [
    'bboard',
    'django.contrib.admin',
    . . .
]
```

  После чего, поместив шаблон registration\logged_out.html в папку templates пакета приложения `bboard`, мы можем быть уверены, что наш шаблон будет найден раньше, чем стандартный;

- папки, пути к которым приведены в списке параметра `DIRS` настроек шаблонизатора (см. *разд. 11.1*), просматриваются перед папками templates пакетов приложений. Мы можем создать в папке проекта папку main_templates и поместить

шаблон registration\logged_out.html в нее. После чего нам останется изменить настройки шаблонизатора следующим образом:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'main_templates')],
        . . .
    },
]
```

Значение переменной BASE_DIR вычисляется в модуле settings.py ранее и представляет собой полный путь к папке проекта.

Здесь мы указали в списке параметра DIRS единственный элемент — путь к только что созданной нами папке. И, опять же, мы можем быть уверены, что контроллер-класс будет использовать наш, а не «чужой» шаблон.

Однако при переопределении шаблонов нужно иметь в виду один весьма неприятный момент. Если мы переопределим какой-либо шаблон, используемый стандартным приложением, то это стандартное приложение будет использовать переопределенный нами шаблон, а не свой собственный. Например, если мы переопределим шаблон registration\logged_out.html, принадлежащий стандартному приложению django.contrib.admin, последнее будет использовать именно переопределенный шаблон. Так что в некоторых случаях, возможно, будет целесообразнее воздержаться от переопределения шаблонов стандартных приложений, а написать свой шаблон.

ГЛАВА 19

Обработка выгруженных файлов

При разработке сайтов без применения фреймворков обработка файлов, выгруженных посетителями, может стать большой проблемой. Но мы использует Django — лучший программный продукт в этой категории, — и для нас такой проблемы просто не существует!

19.1. Подготовка подсистемы обработки выгруженных файлов

Чтобы успешно обрабатывать в своем сайте выгруженные посетителями файлы, нам придется выполнить некоторые подготовительные действия.

19.1.1. Настройка подсистемы обработки выгруженных файлов

Настройки этой подсистемы записываются в модуле settings.py пакета конфигурации. Вот наиболее интересные для нас настройки:

- `MEDIA_URL` — префикс, добавляемый к интернет-адресу выгруженного файла. Встретив в начале интернет-адреса этот префикс, Django поймет, что это выгруженный файл, и его нужно передать для обработки подсистеме выгруженных файлов. Значение по умолчанию — «пустая» строка;
- `MEDIA_ROOT` — полный путь к папке, в которой будут храниться файлы, выгруженные посетителем. Значение по умолчанию — «пустая» строка.

  Это единственные обязательные для указания параметры подсистемы выгруженных файлов. Вот пример их задания:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

  Значение переменной `BASE_DIR` вычисляется в том же модуле settings.py и представляет собой полный путь к папке проекта;

- FILE_UPLOAD_HANDLERS — последовательность имен классов обработчиков выгрузки (*обработчик выгрузки* извлекает файл из отправленных посетителем данных и временно сохраняет его на диске серверного компьютера или в оперативной памяти). В Django доступны два класса обработчика выгрузки, объявленные в модуле django.core.files.uploadhandler:

  - MemoryFileUploadHandler — сохраняет выгруженный файл в оперативной памяти и задействуется, если размер файла не превышает 2,5 Мбайт (это значение настраивается в другом параметре);
  - TemporaryFileUploadHandler — сохраняет выгруженный файл на диске серверного компьютера в папке, предназначенной для хранения временных файлов. Задействуется, если размер выгруженного файла превышает 2,5 Мбайт.

  Значение параметра по умолчанию: список, содержащий два элемента: имена обоих этих классов, которые выбираются автоматически, в зависимости от размера выгруженного файла;

- FILE_UPLOAD_MAX_MEMORY_SIZE — максимальный размер выгруженного файла, сохраняемого в оперативной памяти, в байтах. Если размер превышает это значение, файл будет сохранен на диске. Значение по умолчанию: 2621440 байтов (2,5 Мбайт);

- FILE_UPLOAD_TEMP_DIR — полный путь к папке, в которой будут сохраняться файлы, чей размер превышает величину, указанную в параметре FILE_UPLOAD_MAX_MEMORY_SIZE. Если указано значение None, будет использована стандартная папка для хранения временных файлов в операционной системе. Значение по умолчанию — None;

- FILE_UPLOAD_PERMISSIONS — числовой код прав доступа, даваемых выгруженным файлам. Если задано значение None, права доступа даст сама операционная система, — в большинстве случаев это будут права 0o600 (владелец может читать и записывать файлы, его группа и остальные пользователи вообще не имеют доступа к файлам). Значение по умолчанию — None;

- FILE_UPLOAD_DIRECTORY_PERMISSIONS — числовой код прав доступа, даваемых папкам, которые создаются при сохранении выгруженных файлов. Если задано значение None, права доступа даст сама операционная система, — в большинстве случаев это будут права 0o600 (владелец может читать и записывать файлы в папках, его группа и остальные пользователи вообще не имеют доступа к папкам). Значение по умолчанию — None;

- DEFAULT_FILE_STORAGE — имя класса файлового хранилища, используемого по умолчанию, в виде строки (*файловое хранилище* обеспечивает сохранение файла в выделенной для этого папке, получение его интернет-адреса, параметров и пр.). Значение по умолчанию: "django.core.files.storage.FileSystemStorage" (это единственное файловое хранилище, поставляемое в составе Django).

19.1.2. Указание маршрута для выгруженных файлов

Практически всегда посетители выгружают файлы на сайт для того, чтобы сделать их доступными для других посетителей. Следовательно, на веб-страницах сайта позже будут выводиться сами эти файлы или указывающие на них гиперссылки. Для того чтобы посетители смогли их просмотреть или загрузить, нам придется создать соответствующий маршрут (о маршрутах и маршрутизации рассказывалось в *главе 8*).

Маршрут, указывающий на выгруженный файл, записывается в списке уровня проекта, т. е. в модуле urls.py пакета конфигурации. Для его указания мы используем функцию `static()` из модуля `django.conf.urls.static`:

```
static(<префикс>, document_root=<путь к папке с файлами>)
```

Эта функция создает маршрут, связывающий заданные *префикс* и папку с указанным *путем*. Встретив интернет-адрес, начинающийся с заданного *префикса*, маршрутизатор передает управление особому контроллеру-функции Django, а тот загружает запрошенный файл из папки с указанным *путем* и отправляет его посетителю.

Мы же в качестве префикса укажем значение параметра `MEDIA_URL`, а в качестве пути к папке — значения параметра `MEDIA_ROOT` настроек проекта. Вот так:

```
from django.conf import settings
from django.conf.urls.static import static
. . .
urlpatterns = [
    . . .
]
urlpatterns += static(settings.MEDIA_URL,
                      document_root=settings.MEDIA_ROOT)
```

Этот маршрут необходим, только если сайт работает в отладочном режиме (см. *разд. 3.3.1*). При переводе сайта в эксплуатационный режим, когда сайт начинает работать под сторонним веб-сервером, и выгруженные файлы также начинают обслуживаться сторонним веб-сервером, этот маршрут уже не нужен (разговор о публикации готового сайта пойдет в *главе 29*). Поэтому для создания такого маршрута удобнее использовать выражение:

```
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

Оно добавляет маршрут в список только в том случае, если параметру `DEBUG` настроек проекта присвоено значение `True` (т. е. сайт работает в отладочном режиме).

19.2. Хранение файлов в моделях

Автор книги настоятельно рекомендует пользоваться для обработки выгружаемых файлов высокоуровневыми средствами, предполагающими хранение таких файлов непосредственно в моделях. Применяя такие средства, можно существенно упростить разработку сайтов.

19.2.1. Типы полей модели, предназначенные для хранения файлов

Для хранения файлов в моделях Django предусматривает два типа полей, представляемые описанными далее классами из модуля `django.db.models`:

- `FileField` — файл любого типа. Фактически хранит путь к выгруженному файлу, указанный относительно папки, путь к которой задан в параметре `MEDIA_ROOT` настроек проекта.

  Необязательный параметр `max_length` указывает максимальную длину заносимого в поле пути в виде целого числа в символах. Значение этого параметра по умолчанию: `100`.

  Необязательный параметр `upload_to` указывает папку, куда будет выгружен файл, и которая должна находиться в папке, путь к которой задан в параметре `MEDIA_ROOT`. В этом параметре можно записать:

  - строку с путем, заданным относительно пути из параметра `MEDIA_ROOT`, — файл будет выгружен во вложенную папку, находящуюся по этому пути:

    ```
    archive = models.FileField(upload_to='archives/')
    ```

    Для формирования пути можно использовать составные части текущих даты и времени: год, число, номер месяца и т. п. Для этого в строке с путем применяются специальные символы, используемые в строках формата, указываемых в вызовах функций `strftime()` и `strptime()` Python. Перечень этих специальных символов можно найти в документации по Python или на странице **https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior**. Пример:

    ```
    archive = models.FileField(upload_to='archives/%Y/%m/%d/')
    ```

    В результате выгруженные файлы будут сохраняться во вложенных папках с именами формата `archives\<год>\<номер месяца>\<число>`, где *год*, *номер месяца* и *число* взяты из текущей даты.

    Чтобы выгруженные файлы сохранялись непосредственно в папку из параметра `MEDIA_ROOT`, достаточно указать в параметре `upload_to` «пустую» строку. Это, кстати, его значение по умолчанию;

  - функцию, возвращающую путь для сохранения файла, который включает и имя файла, — файл будет сохранен во вложенной папке, расположенной по полученному от функции пути, под полученным именем. Функция должна принимать два параметра: текущую запись модели и изначальное имя файла.

    Этот способ задания пути сохранения можно использовать для сохранения файлов под какими-либо отвлеченными именами, в качестве которых используется, например, текущая отметка времени. Вот пример такой функции:

    ```
    from datetime import datetime
    from os.path import splitext
    ```

```
def get_timestamp_path(instance, filename):
    return '%s%s' % (datetime.now().timestamp(),
                     splitext(filename)[1])
. . .
file = models.FileField(upload_to=get_timestamp_path)
```

- `ImageField` — графический файл. Фактически хранит путь к выгруженному файлу, указанный относительно папки, путь к которой задан в параметре `MEDIA_ROOT` настроек проекта.

  Внимание!

  Для успешной обработки полей типа `ImageField` необходимо установить графическую библиотеку Pillow. Сделать это можно отдачей команды:

  ```
  pip install pillow
  ```

  Поддерживаются дополнительные параметры `max_length` и `upload_to`, описанные ранее. Помимо них, имеется поддержка следующих дополнительных параметров:

  - `width_field` — имя поля модели, в котором будет храниться ширина изображения, сохраненного в выгруженном файле. Если не указан, ширина изображения нигде храниться не будет;
  - `height_field` — имя поля модели, в котором будет храниться высота изображения, сохраненного в выгруженном файле. Если не указан, высота изображения нигде храниться не будет.

  Эти поля будут созданы неявно. Можно указать создание как обоих полей, так и лишь одного из них (если зачем-то понадобится хранить лишь один размер изображения).

В листинге 19.1 приведен код модели, в которой используется поле типа `ImageField`.

Листинг 19.1. Пример модели, содержащей поле для хранения выгруженного файла

```
from django.db import models

class Img(models.Model):
    img = models.ImageField(verbose_name='Изображение',
                            upload_to=get_timestamp_path)
    desc = models.TextField(verbose_name='Описание')

    class Meta:
        verbose_name='Изображение'
        verbose_name_plural='Изображения'
```

19.2.2. Поля, валидаторы и элементы управления форм, служащие для указания файлов

По умолчанию поле модели `FileField` представляется в форме полем типа `FileField`, а поле модели `ImageField` — полем формы `ImageField`. Оба этих типа полей формы объявлены в модуле `django.forms`:

- `FileField` — поле для ввода файла произвольного типа. Конструктором поддерживаются дополнительные параметры:
  - `max_length` — максимальная длина пути к файлу, заносимого в поле, в символах;
  - `allow_empty_file` — если `True`, к выгрузке будут допускаться даже «пустые» файлы (с нулевым размером), если `False` — только файлы с содержимым (ненулевого размера). Значение по умолчанию — `False`;
- `ImageField` — поле для ввода графического файла. Конструктором поддерживаются дополнительные параметры `max_length` и `allow_empty_file`, описанные ранее.

Помимо валидаторов, описанных в *разд. 4.8.1*, в полях этих типов мы можем использовать и валидаторы, приведенные далее. Все они объявлены в модуле `django.core.validators`:

- `FileExtensionValidator` — класс, проверяет, входит ли расширение сохраняемого в поле файла в список допустимых. Формат конструктора:

  ```
  FileExtensionValidator(allowed_extensions=<допустимые расширения>[,
                         message=None][, code=None])
  ```

  Он принимает следующие параметры:
  - `allowed_extensions` — последовательность, содержащая допустимые расширения файлов. Каждое расширение представляется в виде строки без начальной точки;
  - `message` — строка с сообщением об ошибке. Если не указан, используется стандартное сообщение;
  - `code` — код ошибки. Если не указан, используется код по умолчанию `"invalid_extension"`;
- `validate_image_file_extension` — переменная, хранит экземпляр класса `FileExtensionValidator`, настроенный считать допустимыми только расширения графических файлов. За основу берется список форматов файлов, поддерживаемых библиотекой Pillow.

Также поддерживаются дополнительные коды ошибок в дополнение к приведенным в *разд. 4.8.2*:

- `"invalid"` — применительно к полю типа `FileField` или `ImageField` сообщает, что задан неверный метод кодирования данных для формы. Следует указать метод `multipart/form-data`, воспользовавшись атрибутом `enctype` тега `<form>`;

- "missing" — файл по какой-то причине не был выгружен;
- "empty" — выгруженный файл «пуст» (имеет нулевой размер);
- "contradiction" — следует либо выбрать файл для выгрузки, либо установить флажок удаления файла из поля, но не одновременно;
- "invalid_extension" — расширение выбранного файла не входит в список допустимых;
- "invalid_image" — графический файл сохранен в неподдерживаемом формате или поврежден.

Для непосредственного представления полей формы на странице мы можем использовать такие классы элементов управления, объявленные в модуле `django.forms.widgets`:

- `FileInput` — обычное поле ввода файла;
- `ClearableFileInput` — поле ввода файла с возможностью очистки. Представляется как комбинация обычного поля ввода файла и флажка очистки, при установке которого сохраненный в поле файл удаляется.

Пример формы, включающей в себя поле для выгрузки графического изображения `ImageField`, можно увидеть в листинге 19.2. Эта форма связана с моделью `Img` (см. листинг 19.1).

Листинг 19.2. Пример формы, содержащей поле для выгрузки файла

```
from django import forms
from django.core import validators

from .models import Img

class ImgForm(forms.ModelForm):
    img = forms.ImageField(label='Изображение',
          validators=[validators.FileExtensionValidator(
          allowed_extensions=('gif', 'jpg', 'png'))],
          error_messages={'invalid_extension': 'Этот формат файлов ' + \
          'не поддерживается'})
    desc = forms.CharField(label='Описание',
                           widget=forms.widgets.Textarea())

    class Meta:
        model = Img
        fields = '__all__'
```

19.2.3. Обработка выгруженных файлов

Обработка выгруженных файлов в контроллерах производится так же, как обработка любых других данных, полученных от посетителя (см. *разд. 13.2*). Есть лишь два момента, которые нужно иметь в виду:

- при выводе формы на экран необходимо указать для нее метод кодирования данных `multipart/form-data`, воспользовавшись атрибутом `enctype` тега `<form>`:

  ```
  <form . . . enctype="multipart/form-data">
  ```

  Если этого не сделать, файл не будет выгружен;

- при повторном создании формы конструктору ее класса следует передать вторым позиционным параметром значение атрибута `FILES` объекта запроса. Этот атрибут содержит словарь со всеми выгруженными из формы файлами.

  Пример кода контроллера, который выполняет сохранение выгруженного графического файла в модели, представлен в листинге 19.3. Для выгрузки файла он использует форму `ImgForm` (см. листинг 19.2).

Листинг 19.3. Контроллер, выполняющий сохранение выгруженного посетителем файла

```
from django.shortcuts import render, redirect

from .models import Img
from .forms import ImgForm

def add(request):
    if request.method == 'POST':
        form = ImgForm(request.POST, request.FILES)
        if form.is_valid():
            form.save()
            return redirect('testapp:index')
    else:
        form = ImgForm()
    context = {'form': form}
    return render(request, 'testapp/add.html', context)
```

Отметим, что сохранение выгруженного файла в папке, путь к которой указан в параметре `MEDIA_ROOT` настроек проекта (см. *разд. 19.1*), выполняет сама модель при вызове метода `save()` связанной с ней формы. Нам самим заниматься сохранением файла не придется.

Если для выгрузки файла используется форма, не связанная с моделью, нам понадобится самостоятельно занести выгруженный файл в нужное поле записи модели. Этот файл можно найти в элементе словаря, хранящегося в атрибуте `cleaned_data` объекта формы. Вот пример:

```
form = ImgNonModelForm(request.POST, request.FILES)
if form.is_valid():
    img = Img()
    img.img = form.cleaned_data['img']
    img.desc = form.cleaned_data['desc']
    img.save()
```

И в этом случае сохранение файла выполняется самой моделью.

Аналогичным способом мы можем обрабатывать выгрузку сразу нескольких файлов. Сначала следует указать для поля ввода файла возможность выбора произвольного количества файлов, добавив в создающий его тег `<input>` атрибут без значения `multiple`. Пример:

```
class ImgNonModelForm(forms.Form):
    img = forms.ImageField( . . .
       widget=forms.widgets.ClearableFileInput(attrs={'multiple': True}))
    . . .
```

Далее у нас возникнут некоторые трудности. Дело в том, что элемент словаря из атрибута `cleaned_data` в нашем случае будет хранить только один из выгруженных файлов. Поэтому, чтобы получить все файлы, мы обратимся непосредственно к словарю, что хранится в атрибуте `FILES` объекта запроса, и вызовем у него метод `getlist(`*<элемент словаря>*`)`. Этот метод в качестве результата вернет последовательность файлов, хранящихся в указанном элементе. Вот пример:

```
form = ImgNonModelForm(request.POST, request.FILES)
if form.is_valid():
    for file in request.FILES.getlist('img'):
        img = Img()
        img.desc = form.cleaned_data['desc']
        img.img = file
        img.save()
```

19.2.4. Вывод выгруженных файлов

И, наконец, осталось вывести выгруженные файлы в шаблоне, сформировав указывающие на них интернет-адреса.

При обращении непосредственно к полю типа `FileField`, хранящему выгруженный файл, мы получим экземпляр класса `FieldFile`, представляющий различные сведения о выгруженном файле. Он поддерживает следующие атрибуты:

- `url` — интернет-адрес файла:

```
{% for img in imgs %}
    <div><img src="{{ img.img.url }}"></div>
    <div><a href="{{ img.img.url }}">Загрузить картинку</a></div>
{% endfor %}
```

- `name` — путь к файлу, записанный относительно папки, в которую он выгружен (путь к этой папке указывается в параметре `MEDIA_ROOT` настроек проекта).
- `size` — размер файла в байтах.

При обращении к полю `ImageField` мы получим экземпляр класса `ImageFieldFile`. Он является производным от класса `FieldFile`, поддерживает все его атрибуты и добавляет два своих собственных:

- `width` — ширина хранящегося в файле изображения в пикселах;
- `height` — высота хранящегося в файле изображения в пикселах.

19.2.5. Удаление выгруженного файла

К сожалению, при удалении записи модели, в которой хранится выгруженный файл, сам этот файл удален не будет. Нам придется удалить его самостоятельно.

Для удаления файла применяется метод `delete([save=True])` класса `FieldFile`. Помимо этого, он очищает поле записи, в котором хранится файл. Необязательный параметр `save` указывает сохранять запись модели после удаления файла (значение `True`, используемое по умолчанию) или нет (значение `False`).

Листинг 19.4 показывает код контроллера, который выполняет удаление файла вместе с записью модели, в которой он хранится. Этот контроллер принимает ключ удаляемой записи с URL-параметром `pk`.

Листинг 19.4. Контроллер, выполняющий удаление выгруженного файла вместе с записью модели, в которой он хранится

```
from django.shortcuts import redirect

def delete(request, pk):
    img = Img.objects.get(pk=pk)
    img.img.delete()
    img.delete()
    return redirect('testapp:index')
```

Хотя зачастую удобнее реализовать удаление сохраненных файлов непосредственно в классе модели, переопределив метод `delete()`. Вот подходящий пример:

```
class Img(models.Model):
    . . .
    def delete(self, *args, **kwargs):
        self.img.delete(save=False)
        super().delete(*args, **kwargs)
```

19.3. Хранение путей к файлам в моделях

Еще у нас есть возможность сохранить в поле модели путь к какому-либо файлу, уже существующему на компьютере и хранящемуся в указанной папке. Причем Django сделает так, чтобы мы в принципе не смогли указать путь к файлу, которого нет на диске.

Для указания пути к файлу применяется поле типа `FilePathField` из модуля `django.db.models`. Конструктор этого класса поддерживает следующие дополнительные параметры:

- `path` — полный путь к папке. В поле могут сохраняться только пути к файлам, находящимся в этой папке;
- `match` — регулярное выражение, записанное в виде строки. Если указано, в поле могут быть сохранены только пути к файлам, чьи имена (не пути целиком!) сов-

падают с этим регулярным выражением. Значение по умолчанию — None (в поле можно сохранить путь к любому файлу из заданной папки);

- recursive — если True, в поле можно сохранить путь не только к файлу, хранящемуся непосредственно в заданной папке, но и к любому файлу из вложенных в нее папок. Если False, в поле можно сохранить только путь к файлу из указанной папки. Значение по умолчанию — False;
- allow_files — если True, в поле можно сохранять пути к файлам, что хранятся в указанной папке, если False — нельзя. Значение по умолчанию — True;
- allow_folders — если True, в поле можно сохранять пути к папкам, что вложены в указанную папку, если False — нельзя. Значение по умолчанию — False.

> ***Внимание!***
>
> Допускается указание значения True только для одного из параметров: allow_files или allow_folders.

Для форм предусмотрено поле типа FilePathField из модуля django.forms. Конструктор этого поля поддерживает те же самые параметры path, match, recursive, allow_files и allow_folders.

Для представления поля типа FilePathField на странице применяется список (элемент управления Select).

19.4. Низкоуровневые средства для сохранения выгруженных файлов

Если использование моделей для сохранения выгруженных файлов по какой-либо причине неприемлемо, можно воспользоваться низкоуровневыми средствами для сохранения таких файлов. Однако в этом случае все задачи по сохранению, загрузке и выдаче таких файлов посетителю нам придется решать самостоятельно.

19.4.1. Класс *UploadedFile*: выгруженный файл. Сохранение выгруженных файлов

Как мы уже знаем, все выгруженные посетителем файлы хранятся в словаре, доступном через атрибут FILES объекта запроса. Так вот, каждый из этих файлов представляется экземпляром класса UploadedFile.

Класс UploadedFile поддерживает ряд атрибутов и методов, которые обязательно пригодятся нам. Начнем с атрибутов;

- name — изначальное имя выгруженного файла;
- size — размер выгруженного файла в байтах;
- content_type — MIME-тип файла в виде строки;

- `content_type_extra` — дополнительные параметры MIME-типа файла, представленные в виде словаря;
- `charset` — кодировка, если файл текстовый.

Теперь — список методов:

- `multiple_chunks([chunk_size=None])` — возвращает `True`, если файл настолько велик, что для обработки его придется разбивать на отдельные части, и `False`, если он может быть обработан как единое целое.

  Необязательный параметр `chunk_size` указывает размер отдельной части (собственно, файл считается слишком большим, если его размер превышает размер части). Если этот параметр не указан, размер принимается равным 64 Кбайт;
- `read()` — считывает и возвращает в качестве результата все содержимое файла.

  Этот метод следует использовать, если файл не слишком велик (метод `multiple_chunks()` возвращает `False`);
- `chunks([chunk_size=None])` — возвращает итератор, который на каждой итерации выдает очередную часть файла.

  Необязательный параметр `chunk_size` указывает размер отдельной части. Если он не указан, размер принимается равным 64 Кбайт.

  Этот метод рекомендуется использовать, если файл слишком велик, чтобы быть обработанным за один раз (метод `multiple_chunks()` возвращает `True`). На практике же его можно применять в любом случае — это позволит упростить код.

Пример контроллера, который выполняет сохранение выгруженного файла, показан в листинге 19.5.

Листинг 19.5. Контроллер, сохраняющий выгруженный файл низкоуровневыми средствами Django

```
from django.shortcuts import render, redirect
from samplesite.settings import BASE_DIR
from datetime import datetime
import os

from .forms import ImgForm

FILES_ROOT = os.path.join(BASE_DIR, 'files')

def add(request):
    if request.method == 'POST':
        form = ImgForm(request.POST, request.FILES)
        if form.is_valid():
            uploaded_file = request.FILES['img']
            fn = '%s%s' % (datetime.now().timestamp(),
                           os.path.splitext(uploaded_file.name)[1])
```

```
            fn = os.path.join(FILES_ROOT, fn)
            with open(fn, 'wb+') as destination:
                for chunk in uploaded_file.chunks():
                    destination.write(chunk)
            return redirect('testapp:index')
    else:
        form = ImgForm()
    context = {'form': form}
    return render(request, 'testapp/add.html', context)
```

Для сохраняемого файла мы указываем в качестве имени текущую отметку времени и изначальное расширение. Сами файлы сохраняются в папке files, находящейся в папке проекта. Как видим, применяя низкоуровневые средства, мы можем сохранить файл в любой папке, а не только в той, путь к которой указан в параметре `MEDIA_ROOT`.

19.4.2. Вывод выгруженных файлов низкоуровневыми средствами

Если выгруженные файлы не сохраняются в моделях, задача по их выводу несколько усложняется.

Чтобы вывести список выгруженных файлов, мы можем выполнить поиск всех файлов в нужной папке и сформировать их список. Для этого удобно применять функцию `scandir()` из модуля `os`.

Листинг 19.6 показывает код контроллера, который выводит список выгруженных файлов, что хранятся в папке files папки проекта.

Листинг 19.6. Контроллер, выводящий список выгруженных файлов

```
from django.shortcuts import render
from samplesite.settings import BASE_DIR
import os

FILES_ROOT = os.path.join(BASE_DIR, 'files')

def index(request):
    imgs = []
    for entry in os.scandir(FILES_ROOT):
        imgs.append(os.path.basename(entry))
    context = {'imgs': imgs}
    return render(request, 'testapp/index.html', context)
```

В шаблоне testapp\index.html нам нужно вывести изображения, хранящиеся в выгруженных файлах. Обратиться к атрибуту `url` мы не можем по вполне понятной причине. Однако мы можем написать еще один контроллер, который получит через

URL-параметр имя выгруженного файла и сформирует на его основе ответ — экземпляр класса `FileResponse` (описан в *разд. 9.8.2*). Код этого контроллера можно увидеть в листинге 19.7.

Листинг 19.7. Контроллер, отправляющий выгруженный файл клиенту

```
from django.http import FileResponse

def get(request, filename):
    fn = os.path.join(FILES_ROOT, filename)
    return FileResponse(open(fn, 'rb'),
                        content_type='application/octet-stream')
```

Маршрут, ведущий к этому контроллеру, может быть таким:

```
path('get/<path:filename>', get, name='get'),
```

Обратите внимание, что здесь используется обозначение формата `path`, т. е. любая непустая строка, могущая включать в себя любые символы.

И, наконец, для вывода списка файлов мы напишем в шаблоне testapp\index.html следующий код:

```
{% for img in imgs %}
<div class="image">
    <p><img src="{% url 'testapp:get' filename=img %}"></p>
</div>
{% endfor %}
```

Низкоуровневые средства выгрузки файлов, поддерживаемые Django, по большей части основаны на инструментах Python, предназначенных для работы с файлами и папками (как, собственно, мы только что убедились). Для хранения файлов они не требуют создания модели и имеют более высокое быстродействие. Однако им присущ ряд недостатков: невозможность сохранения дополнительной информации о выгруженном файле (например, описания или выгрузившего их пользователя) и трудности в реализации фильтрации и сортировки файлов по произвольным критериям.

19.5. Библиотека django-cleanup: автоматическое удаление ненужных файлов

В *разд. 19.2.5* мы уже касались одной проблемы, присущей Django. При удалении записи модели, которая содержит поле типа `FileField` или `ImageField`, файл, сохраненный в этом поле, не удаляется. Аналогично, при записи в такое поле другого файла старый файл также не удаляется, а остается на диске. По какой-то причине разработчики фреймворка считают, что такими делами должны заниматься мы (как будто нам совершенно нечем заняться...).

Однако существует дополнительная библиотека django-cleanup, которая отслеживает появление ненужных файлов и сама их удаляет. Установить ее можно отдачей команды:

```
pip install django-cleanup
```

Ядром этой библиотеки является приложение `django_cleanup`, которое следует добавить в список зарегистрированных в проекте:

```
INSTALLED_APPS = [
    . . .
    'django_cleanup',
]
```

На этом какие-либо действия с нашей стороны закончены. Далее библиотека django-cleanup начнет работать самостоятельно.

> ***На заметку***
>
> Документацию по этой библиотеке можно найти на странице **https://github.com/un1t/django-cleanup**.

19.6. Библиотека easy-thumbnails: вывод миниатюр

Очень часто при выводе списка графических изображений, хранящихся на сайте, показывают их *миниатюры* — уменьшенные копии. А при переходе на страницу со сведениями о выбранном изображении уже демонстрируют полную редакцию этого изображения.

Для автоматического формирования миниатюр удобно применять дополнительную библиотеку easy-thumbnails. Она самостоятельно создает миниатюры, кэширует их на диске и даже позволяет задавать параметры их формирования.

> ***На заметку***
>
> Полную документацию по библиотеке easy-thumbnails можно найти здесь: **http://easy-thumbnails.readthedocs.io/en/latest/**.

19.6.1. Установка easy-thumbnails

Для установки библиотеки следует набрать в командной строке команду:

```
pip install easy-thumbnails
```

Для успешной работы требуется наличие известной библиотеки обработки графики Pillow. Если таковая на компьютере отсутствует, она будет установлена.

Программное ядро библиотеки реализовано в виде приложения `easy-thumbnails`. Это приложение необходимо добавить в список зарегистрированных в проекте:

```
INSTALLED_APPS = [
    . . .
    'easy_thumbnails',
]
```

Наконец, чтобы все заработало, нужно выполнить миграции.

> ***На заметку***
>
> Для своих нужд easy-thumbnails создает в базе данных таблицы `easy_thumbnails_source`, `easy_thumbnails_thumbnail` и `easy_thumbnails_thumbnaildimensions`.

19.6.2. Настройка easy-thumbnails

Для успешной работы библиотека должна быть должным образом настроена. Настройки, как обычно, записываются в модуле settings.py пакета конфигурации.

19.6.2.1. Пресеты миниатюр

Прежде всего необходимо указать набор *пресетов* (предопределенных комбинаций настроек), на основе которых будут создаваться миниатюры. Все параметры, которые можно указать в таких пресетах и которые затрагивают создаваемые библиотекой миниатюры, приведены далее:

- `size` — размеры миниатюры. Значением должен быть кортеж из двух элементов: ширины и высоты, обе величины измеряются в пикселах.

  Допускается вместо одного из размеров указывать число `0`. Тогда библиотека сама подберет значение этого размера таким образом, чтобы пропорции изображения не искажались.

  Примеры:

  ```
  "size": (400, 300) # Миниатюра размерами 400x300 пикселов
  "size": (400, 0)   # Миниатюра получит ширину в 400 пикселов,
                     # а высота будет подобрана так, чтобы не допустить
                     # искажения пропорций
  "size": (0, 300)   # Миниатюра получит высоту в 300 пикселов,
                     # а ширина будет подобрана так, чтобы не допустить
                     # искажения пропорций
  ```

- `crop` — управляет обрезкой или масштабированием изображения до размеров, указанных в параметре `size`. Значением может быть одна из строк:

  - `"scale"` — изображение будет масштабироваться до указанных размеров. Обрезка производиться не будет;
  - `"smart"` — будут обрезаны малозначащие, с точки зрения библиотеки, края изображения («умная» обрезка);
  - `"<смещение слева>,<смещение сверху>"` — явно указывает местоположение фрагмента изображения, который будет вырезан и превращен в миниатюру. Величины *смещения слева* и *сверху* задаются в % от ширины и высоты изо-

бражения соответственно. Положительные значения указывают, собственно, смещение слева и сверху левой или верхней границы миниатюры, а отрицательные — смещения справа и снизу ее правой или нижней границы. Если задать значение `0`, соответствующая граница миниатюры будет находиться на границе исходного изображения.

Значение параметра по умолчанию: `"50,50"`;

- `autocrop` — если `True`, белые поля на границах изображения будут обрезаны;
- `bw` — если `True`, миниатюра станет черно-белой;
- `replace_alpha` — цвет, которым будет замещен прозрачный цвет в исходном изображении. Цвет указывается в формате `#`*RRGGBB*, где *RR* — доля красной составляющей, *GG* — зеленой, *BB* — синей. По умолчанию преобразование прозрачного цвета не выполняется;
- `quality` — качество миниатюры в виде числа от `1` (наихудшее качество) до `100` (наилучшее качество). Значение по умолчанию: `85`;
- `subsampling` — обозначение уровня подвыборки цвета в виде числа `2` (значение по умолчанию), `1` (более четкие границы, небольшое увеличение размера файла) или `0` (очень четкие границы, значительное увеличение размера файла).

Пресеты записываются в параметре `THUMBNAIL_ALIASES` в виде словаря. Ключи элементов этого словаря указывают области действия пресетов, записанные в одном из следующих форматов:

- «пустая» строка — пресет действует во всех приложениях проекта;
- "*<псевдоним приложения>*" — пресет действует только в приложении с указанным *псевдонимом*;
- "*<псевдоним приложения>.<имя модели>*" — пресет действует только в модели с заданным *именем* в приложении с указанным *псевдонимом*;
- "*<псевдоним приложения>.<имя модели>.<имя поля>*" — пресет действует только для поля с указанным *именем*, в модели с заданным *именем*, в приложении с указанным *псевдонимом*.

Значениями элементов этого словаря должны быть словари, указывающие собственно пресеты. Ключи элементов зададут имена пресетов, а элементы, также словари, укажут настройки, относящиеся к соответствующему пресету.

Вот пример указания пресетов для библиотеки easy-thumbnails:

```
THUMBNAIL_ALIASES = {
    'bboard.Bb.picture': {
        'default": {
            'size': (500, 300),
            'crop': 'scale',
        },
    },
```

```
    'testapp': {
        'default': {
            'size': (400, 300),
            'crop': 'smart',
            'bw': True,
        },
    },
    '': {
        'default': {
            'size': (180, 240),
            'crop': 'scale',
        },
        'big': {
            'size': (480, 640),
            'crop': '10,10',
        },
    },
}
```

Для поля `picture` модели `Bb` приложения `bboard` мы создали пресет `default`, в котором указали размеры миниатюры 500×300 пикселов и масштабирование без обрезки. Для приложения `testapp` мы также создали пресет `default`, где задали размеры 400×300 пикселов, «умную» обрезку и преобразование в черно-белый вид. А для всего проекта мы расстарались на два пресета: `default` (размеры 180×240 пикселов и масштабирование) и `big` (размеры 480×640 пикселов и обрезка, причем миниатюра будет находиться на расстоянии 10% от левой и верхней границ исходного изображения).

Параметр `THUMBNAIL_DEFAULT_OPTIONS` указывает параметры по умолчанию, применяемые ко всем пресетам, в которых они не были переопределены. Значением этого параметра должен быть словарь, аналогичный тому, который задает параметры отдельного пресета. Пример:

```
THUMBNAIL_DEFAULT_OPTIONS = {'quality': 90, 'subsampling': 1,}
```

19.6.2.2. Остальные параметры библиотеки

Теперь рассмотрим остальные параметры библиотеки easy-thumbnails, которые могут нам пригодиться:

- `THUMBNAIL_MEDIA_URL` — префикс, добавляемый к интернет-адресу файла со сгенерированной миниатюрой. Если указать «пустую» строку, будет использоваться префикс из параметра `MEDIA_URL`. Значение по умолчанию — «пустая» строка;
- `THUMBNAIL_MEDIA_ROOT` — полный путь к папке, в которой будут храниться файлы со сгенерированными миниатюрами. Если указать «пустую» строку, будет использована папка из параметра `MEDIA_ROOT`. Значение по умолчанию — «пустая» строка.

В случае указания параметров THUMBNAIL_MEDIA_URL и THUMBNAIL_MEDIA_ROOT необходимо записать соответствующий маршрут, чтобы Django смог загрузить созданные библиотекой миниатюры. Код, создающий этот маршрут, аналогичен тому, что был представлен в *разд. 19.1.2*, и может выглядеть так:

```
urlpatterns += static(settings.THUMBNAIL_MEDIA_URL,
                      document_root=settings.THUMBNAIL_MEDIA_ROOT)
```

- THUMBNAIL_BASEDIR — путь к папке, в которой будут сохраняться все файлы с миниатюрами. Путь должен быть задан относительно папки из параметра THUMBNAIL_MEDIA_ROOT. Значение по умолчанию — «пустая» строка (т. е. файлы миниатюр будут сохраняться непосредственно в папке из параметра THUMBNAIL_MEDIA_ROOT);
- THUMBNAIL_SUBDIR — имя папки, в которой будут сохраняться все файлы с миниатюрами, представляющими изображения из какой-либо папки. Так, если задать значение "thumbs", для файла images\img1.jpg миниатюра будет сохранена в папке images\thumbs. Значение по умолчанию — «пустая» строка (т. е. файлы миниатюр будут сохраняться непосредственно в папке из параметра MEDIA_ROOT);
- THUMBNAIL_PREFIX — префикс, который будет добавляться в начало имен файлов с миниатюрами. Значение по умолчанию — «пустая» строка;
- THUMBNAIL_EXTENSION — формат файлов для сохранения миниатюр без поддержки прозрачности. Значение по умолчанию: "jpg";
- THUMBNAIL_TRANSPARENCY_EXTENSION — формат файлов для сохранения миниатюр с поддержкой прозрачности. Значение по умолчанию: "png";
- THUMBNAIL_PRESERVE_EXTENSIONS — последовательность расширений файлов, для которых следует создать миниатюры в тех же форматах, в которых были сохранены оригинальные файлы. Расширения должны быть указаны без начальных точек в нижнем регистре. Пример:

  ```
  THUMBNAIL_PRESERVE_EXTENSIONS = ('png',)
  ```

  Теперь для файлов с расширением png будут созданы миниатюры также в формате PNG, а не формате, указанном в параметре THUMBNAIL_EXTENSION.

  Если указать значение True, для файлов всех форматов будут создаваться миниатюры в тех же форматах, что и исходные файлы.

  Значение параметра по умолчанию — None;
- THUMBNAIL_PROGRESSIVE — величина размера изображения в пикселах, при превышении которой изображение будет сохранено в прогрессивном формате JPEG. Учитывается любой размер — как ширина, так и высота. Если указать значение False, прогрессивный JPEG вообще не будет использоваться. Значение по умолчанию: 100;
- THUMBNAIL_QUALITY — качество изображения JPEG в диапазоне от 1 до 100. Значение по умолчанию: 85;
- THUMBNAIL_WIDGET_OPTIONS — параметры миниатюры, генерируемой для элемента управления ImageClearableFileInput (будет описан позже). Записываются в виде

словаря в том же формате, что и параметры отдельного пресета (см. *разд. 19.6.2.1*). Значение по умолчанию: `{'size': (80, 80)}` (размеры 80×80 пикселов).

19.6.3. Вывод миниатюр в шаблонах

Прежде чем выводить в шаблонах сгенерированные библиотекой easy-thumbnails миниатюры, нужно загрузить библиотеку тегов с псевдонимом `thumbnail`:

```
{% load thumbnail %}
```

Для вывода миниатюры, сгенерированной на основе сохраненного в модели изображения, мы можем использовать:

- `thumbnail_url:`*<название пресета>* — фильтр, вставляет в шаблон интернет-адрес файла с миниатюрой, созданной на основе пресета с указанным *названием* и исходного изображения, которое берется из поля типа `FileField` или `ImageField`. Если пресета с указанным *названием* нет, будет выведена «пустая» строка. Пример:

  ```
  <img src="{{ img.img|thumbnail_url:'default' }}">
  ```

- `thumbnail` — тег, вставляет в шаблон интернет-адрес файла с миниатюрой, созданной на основе *исходного изображения*, которое берется из поля типа `FileField` или `ImageField`. Формат тега:

  `thumbnail` *<исходное изображение> <название пресета>*|*<размеры>* [*<параметры>*] [as *<переменная>*]

  Размеры могут быть указаны либо в виде строки формата "*<ширина>*x*<высота>*", либо в виде переменной, которая может содержать строку описанного ранее формата или кортеж из двух элементов, из которых первый укажет ширину, а второй — высоту.

  Параметры записываются в том же формате, что и в объявлении пресетов (см. *разд. 19.6.2.1*). Если вместо *размеров* указано *название пресета*, указанные *параметры* переопределят значения, записанные в пресете.

  Примеры:

  ```
  {# Используем пресет default #}
  <img src="{% thumbnail img.img 'default' %}">
  {# Используем пресет default и дополнительно указываем преобразование
  миниатюр в черно-белый вид #}
  <img src="{% thumbnail img.img 'default' bw=True %}">
  {# Явно указываем размеры миниатюр и "умный" режим обрезки #}
  <img src="{% thumbnail img.img '300x200' crop='smart' %}">
  ```

  Мы можем поместить созданную тегом миниатюру в *переменную*, чтобы использовать ее впоследствии. Эта миниатюра представляется экземпляром класса `ThumbnailFile`, являющегося производным от знакомого нам класса `ImageFieldFile` (см. *разд. 19.2.4*) и поддерживающего те же атрибуты.

Пример:

```
{% thumbnail img.img 'default' as thumb %}
<img src="{{ thumb.url }}">
```

19.6.4. Хранение миниатюр в моделях

Библиотека easy-thumbnails поддерживает два класса полей, которые можно использовать в моделях и которые объявлены в модуле `easy_thumbnails.fields`:

- `ThumbnailerField` — подкласс класса `FileField`. Выполняет часть работ по генерированию миниатюры непосредственно при сохранении записи, а при ее удалении также удаляет все миниатюры, сгенерированные на основе сохраненного в поле изображения. В остальном ведет себя так же, как знакомое нам поле `FileField`;
- `ThumbnailerImageField` — подкласс классов `ImageField` и `ThumbnailerField`.

  Конструктор класса поддерживает дополнительный параметр `resize_source`, задающий параметры генерируемой на основе сохраняемого изображения миниатюры. Эти параметры записываются в виде словаря в том же формате, что и при объявлении пресета (см. *разд. 19.6.2.1*).

  > ***Внимание!***
  >
  > **Если в конструкторе класса `ThumbnailerImageField` указать параметр `resize_source`, в поле будет сохранено не исходное изображение, а сгенерированная на его основе миниатюра.**

  Пример:

```
from easy_thumbnails.fields import ThumbnailerImageField
. . .
class Img(models.Model):
    img = ThumbnailerImageField(
          resize_source={'size': (400, 300), 'crop': 'scale'})
    . . .
```

  Теперь для вывода миниатюры, сохраненной в поле, мы можем использовать средства, описанные в *разд. 19.2.4*:

```
<img src="{{ img.img.url }}">
```

  Если же параметр `resize_source` в конструкторе поля не указан, в поле будет сохранено оригинальное изображение, и для вывода миниатюры нам придется прибегнуть к средствам, описанным в *разд. 19.6.3*.

Эти поля могут представляться в форме элементом управления `ImageClearableFileInput`, класс которого объявлен в модуле `easy_thumbnails.widgets`. Он является подклассом класса `ClearableFileInput`, но дополнительно выводит миниатюру выбранного в нем изображения. Параметры этой миниатюры можно указать в необязательном параметре `thumbnail_options` в виде словаря. Пример:

```
from easy_thumbnails.widgets import ImageClearableFileInput
. . .
class ImgForm(forms.Form):
    img = forms.ImageField(widget=ImageClearableFileInput(
                           thumbnail_options={'size': (300, 200)}))
    . . .
```

Если параметры миниатюры для этого элемента не указаны, они будут взяты из параметра `THUMBNAIL_WIDGET_OPTIONS` настроек проекта (см. *разд. 19.6.2.2*).

19.6.5. Дополнительная команда *thumbnail_cleanup*

Библиотека easy-thumbnails добавляет утилите manage.py поддержку команды `thumbnail_cleanup`, удаляющей все файлы с миниатюрами, что были сгенерированы библиотекой. Формат вызова этой команды следующий:

```
manage.py thumbnail_cleanup [--last-n-days <количество дней>]
[--path <путь для очистки>] [--dry-run]
```

Поддерживаются следующие дополнительные ключи:

- `--last-n-days` — позволяет оставить миниатюры, сгенерированные в течение указанного *количества дней*. Если не задан, будут удалены все миниатюры;
- `--path` — задает *путь для очистки* от миниатюр. Если не указан, будет очищена папка из параметра `MEDIA_ROOT` настроек проекта и все вложенные в нее папки;
- `--dry-run` — выводит на экран сведения об удаляемых миниатюрах, но не удаляет их.

ГЛАВА 20

Разграничение доступа: расширенные инструменты и дополнительная библиотека

Подсистема разграничения доступа, реализованная в Django, таит в себе немало сюрпризов. И временами, в частности, при программировании на низком уровне, эти сюрпризы могут оказаться приятными.

20.1. Настройки проекта, касающиеся разграничения доступа

Прежде всего рассмотрим немногочисленные настройки, затрагивающие работу подсистемы разграничения доступа. Все они записываются в модуле settings.py пакета приложения:

- AUTH_PASSWORD_VALIDATORS — список валидаторов, применяемых при валидации введенного пользователем при регистрации пароля. Каждый элемент списка задает один валидатор и должен представлять собой словарь с элементами NAME (задает имя класса валидатора в виде строки) и OPTIONS (словарь с дополнительными параметрами валидатора).

  Валидаторы, приведенные в списке, задействуются в формах для смены и сброса пароля, в командах создания суперпользователя и смены пароля. Во всех прочих случаях они никак не используются.

  Значение по умолчанию — «пустой» список, однако сразу при создании проекта этому параметру присваивается список из всех поставляемых в составе Django валидаторов с параметрами по умолчанию;

- AUTHENTICATION_BACKENDS — список имен классов, реализующих аутентификацию и авторизацию, представленных в виде строк. Значение по умолчанию — список с единственным элементом "django.contrib.auth.backends.ModelBackend" (этот класс реализует аутентификацию и авторизацию пользователей из списка, хранящегося в модели);

- `AUTH_USER_MODEL` — имя класса модели, применяемой для хранения списка зарегистрированных пользователей, в виде строки. Значение по умолчанию: `"auth.User"` (стандартная модель `User`).

20.2. Работа с пользователями

Ряд инструментов Django предлагает для работы с пользователями: их создания, смены пароля и пр.

20.2.1. Создание пользователей

Для создания пользователя применяются два описанных далее метода, поддерживаемые классом диспетчера записей `UserManager`, который используется в модели `User`:

- `create_user(`*<имя>*`, password=`*<пароль>*`[, email=`*<адрес электронной почты>*`][, `*<дополнительные поля>*`])` — создает нового пользователя с указанными *именем*, *паролем* и *адресом электронной почты* (если он задан). Также могут быть указаны значения для *дополнительных полей*, которые будут сохранены в модели пользователя. Созданный пользователь делается активным (полю `is_active` присваивается значение `True`). После создания пользователя выполняет его сохранение и возвращает в качестве результата. Примеры:

```
user1 = User.objects.create_user('ivanov', password='1234567890',
                                 email='ivanov@site.ru')
user2 = User.objects.create_user('petrov', password='0987654321',
                                 email='petrov@site.ru',
                                 is_stuff=True)
```

- `create_superuser(`*<имя>*`, `*<адрес электронной почты>*`, `*<пароль>*`[, `*<дополнительные поля>*`])` — создает нового суперпользователя с указанными *именем*, *паролем* и *адресом электронной почты* (если он задан). Также могут быть указаны значения для *дополнительных полей*, которые будут сохранены в модели пользователя. Созданный суперпользователь делается активным (полю `is_active` присваивается значение `True`). После создания суперпользователя выполняет его сохранение и возвращает в качестве результата.

20.2.2. Работа с паролями

Еще четыре метода, поддерживаемые моделью `User`, предназначены для работы с паролями:

- `check_password(`*<пароль>*`)` — возвращает `True`, если заданный *пароль* совпадает с хранящимся в списке, и `False` — в противном случае:

```
from django.contrib.auth.models import User
admin = User.objects.get(name='admin')
if admin.check_password('password'):
    # Пароли совпадают
```

```
else:
    # Пароли не совпадают
```

- `set_password(<новый пароль>)` — задает для текущего пользователя *новый пароль*. Сохранение пользователя не выполняет. Пример:

```
admin.set_password('newpassword')
admin.save()
```

- `set_unusable_password()` — задает для текущего пользователя недействительный пароль. При проверке такого пароля функцией `check_password()` последняя всегда будет возвращать `False`. Сохранение пользователя не выполняет.

  Недействительный пароль указывается у тех пользователей, для которых процедура входа на сайт выполняется не средствами Django, а какой-либо сторонней библиотекой — например, Python Social Auth, описываемой далее;

- `has_usable_password()` — возвращает `True`, если для текущего пользователя был задан действительный пароль, и `False`, если указанный для него пароль недействителен (была вызвана функция `set_unusable_password()`).

20.3. Аутентификация и выход с сайта

Аутентификация, т. е. вход на сайт, с применением низкоуровневых инструментов выполняется в два этапа: поиск пользователя в списке и собственно вход. Здесь нам понадобятся три функции, объявленные в модуле `django.contrib.auth`.

Для выполнения поиска пользователя по указанным им на странице входа имени и паролю применяется функция `authenticate()`:

```
authenticate(<запрос>, username=<имя>, password=<пароль>)
```

Запрос должен быть представлен экземпляром класса `HttpRequest`. Если пользователь с указанными *именем* и *паролем* существует в списке, функция возвращает запись модели `User`, представляющую этого пользователя. В противном случае возвращается `None`.

Собственно вход выполняется вызовом функции `login(<запрос>, <пользователь>)`. *Запрос* должен быть представлен экземпляром класса `HttpRequest`, а *пользователь*, от имени которого выполняется вход, — записью модели `User`.

Вот пример кода, получающего в POST-запросе данные из формы входа и выполняющего вход на сайт:

```
from django.contrib.auth import authenticate, login

def my_login(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(request, username=username, password=password)
    if user is not None:
        login(request, user)
        # Вход выполнен
```

```
    else:
        # Вход не был выполнен
```

Выход с сайта выполняется вызовом функции `logout(<запрос>)`. *Запрос* должен быть представлен экземпляром класса `HttpRequest`. Пример:

```
from django.contrib.auth import logout

def my_logout (request):
    logout(request)
    # Выход был выполнен. Выполняем перенаправление на какую-либо
    # страницу
```

20.4. Валидация паролей

В *разд. 20.1* описывался параметр `AUTH_PASSWORD_VALIDATORS` настроек проекта, который задает набор валидаторов, применяемых для валидации паролей. Эти валидаторы будут работать в формах для смены и сброса пароля, в командах создания суперпользователя и смены пароля.

Значение этого параметра по умолчанию — «пустой» список. Однако сразу при создании проекта для него задается такое значение:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'NumericPasswordValidator',
    },
]
```

Это список, включающий все (четыре) стандартные валидаторы, поставляемые в составе Django.

20.4.1. Стандартные валидаторы паролей

Все стандартные валидаторы реализованы в виде классов и объявлены в модуле `django.contrib.auth.password_validation`:

- `UserAttributeSimilarityValidator()` — позволяет удостовериться, что пароль в достаточной степени отличается от остальных сведений о пользователе. Формат вызова конструктора:

```
UserAttributeSimilarityValidator(
            [user_attributes=self.DEFAULT_USER_ATTRIBUTES][,
            ][max_similarity=0.7])
```

  Необязательный параметр `user_attributes` задает последовательность имен полей модели `User`, из которых будут браться сведения о пользователе для сравнения с паролем, имена полей должны быть представлены в виде строк. По умолчанию берется кортеж, хранящийся в атрибуте `DEFAULT_USER_ATTRIBUTES` этого же класса и указывающий поля `username`, `first_name`, `last_name` и `email`.

  Другой необязательный параметр `max_similarity` задает степень схожести пароля со значением какого-либо из указанных в последовательности `user_attributes` полей. Значение параметра должно представлять собой вещественное число от `0` (будут отклоняться все пароли без исключения) до `1` (будут отклоняться только пароли, полностью совпадающие со значением поля). По умолчанию используется значение `0.7`;

- `MinimumLengthValidator([min_length=8])` — проверяет, не оказались ли длина пароля меньше заданной в параметре `min_length` (по умолчанию `8` символов);

- `CommonPasswordValidator()` — проверяет, не входит ли пароль в указанный перечень наиболее часто встречающихся паролей. Формат вызова конструктора:

```
CommonPasswordValidator(
            [password_list_path=self.DEFAULT_PASSWORD_LIST_PATH])
```

  Необязательный параметр `password_list_path` задает полный путь к файлу со списком недопустимых паролей. Этот файл должен быть сохранен в текстовом формате, а каждый из паролей должен находиться на отдельной строке. По умолчанию используется файл с порядка 1000 паролей, путь к которому хранится в атрибуте `DEFAULT_PASSWORD_LIST_PATH` класса;

- `NumericPasswordValidator` — проверяет, не содержит ли пароль одни цифры.

Вот пример кода, задающего новый список валидаторов:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'MinimumLengthValidator',
        'OPTIONS": {'min_length': 10}
    },
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'NumericPasswordValidator',
    },
]
```

Новый список содержит только валидаторы `MinimumLengthValidator` и `NumericPasswordValidator`, причем для первого указана минимальная длина пароля `10` символов.

20.4.2. Написание своих валидаторов паролей

Мы можем написать собственные валидаторы паролей, если стандартных нам не хватает.

Валидаторы паролей отличаются от таковых, выполняющих валидацию в моделях (см. *разд. 4.8.3*) и формах (см. *разд. 13.4.1*). Прежде всего, валидаторы паролей обязательно должны быть реализованы в виде классов. Кроме того, они должны объявлять два метода:

- `validate(<проверяемый пароль>[, user=None])` — выполняет валидацию *проверяемого пароля*. С необязательным параметром `user` может быть получен текущий пользователь.

  Метод не должен возвращать значения. Если пароль не проходит валидацию, следует возбудить исключение `ValidationError` из модуля `django.core.exceptions`;

- `get_help_text()` — должен возвращать строку с требованиями к вводимому паролю.

Листинг 20.1 показывает код валидатора `NoForbiddenCharsValidator`, проверяющего, не содержатся ли в пароле недопустимые символы, заданные в параметре `forbidden_chars`.

Листинг 20.1. Пример валидатора паролей

```
from django.core.exceptions import ValidationError

class NoForbiddenCharsValidator:
    def __init__(self, forbidden_chars=(' ',)):
        self.forbidden_chars = forbidden_chars

    def validate(self, password, user=None):
        for fc in self.forbidden_chars:
            if fc in password:
                raise ValidationError(
                      'Пароль не должен содержать недопустимые ' + \
                      'символы %s' % ', '.join(self.forbidden_chars),
                      code='forbidden_chars_present')

    def get_help_text(self):
        return 'Пароль не должен содержать недопустимые ' + \
               'символы %s' % ', '.join(self.forbidden_chars)
```

Такой валидатор может быть использован наряду со стандартными:

```
AUTH_PASSWORD_VALIDATORS = [
    . . .
    {
        'NAME': ' NoForbiddenCharsValidator ',
        'OPTIONS': {'forbidden_chars': (' ', ',', '.', ':', ';')},
    },
]
```

20.4.3. Выполнение валидации паролей

Валидаторы из параметра `AUTH_PASSWORD_VALIDATORS` используются в ограниченном количестве случаев. Если же нам понадобится произвести валидацию пароля там, где нам нужно (например, в написанной нами самими форме), мы прибегнем к набору функций, объявленных в модуле `django.contrib.auth.password_validation`. Вот все эти функции:

- `validate_password(<пароль>[, user=None][, password_validators=None])` — выполняет валидацию *пароля*. Если пароль не проходит валидацию, возбуждает исключение `ValidationError`;
- `password_validators_help_texts([password_validators=None])` — возвращает список строк, содержащий требования к вводимым паролям от всех валидаторов. Строка с такими требованиями возвращается методом `get_help_text()` валидатора (см. *разд. 20.4.2*);
- `password_validators_help_texts_html([password_validators=None])` — то же самое, что и `password_validators_help_text()`, но возвращает HTML-код, создающий маркированный список со всеми требованиями;
- `password_changed(<пароль>[, user=None][, password_validators=None])` — сообщает всем валидаторам, что пароль пользователя изменился.

  Вызов этой функции следует выполнять сразу после каждой смены пароля, если для этого не использовалась функция `set_password()`, описанная в *разд. 20.2.2*. После выполнения функции `set_password()` функция `password_changed()` вызывается автоматически.

Необязательный параметр `user`, принимаемый большинством функций, задает пользователя, чей пароль проходит валидацию. Это значение может понадобиться некоторым валидаторам.

Необязательный параметр `password_validators`, поддерживаемый всеми этими функциями, указывает список валидаторов, которые будут заниматься валидацией пароля. Если он не указан, используется список из параметра `AUTH_PASSWORD_VALIDATORS` настроек проекта.

Чтобы сформировать свой список валидаторов, следует применить функцию `get_password_validators(<настройки валидаторов>)`. *Настройки валидаторов* указываются в том же формате, что и значение параметра `AUTH_PASSWORD_VALIDATORS` настроек проекта. Пример:

```
from django.contrib.auth import password_validation
my_validators = [
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'NumericPasswordValidator',
    },
    {
        'NAME': ' NoForbiddenCharsValidator ',
        'OPTIONS': {'forbidden_chars': (' ', ',', '.', ':', ';')},
    },
]
validator_config = password_validation.get_password_validators(
                                                    my_validators)
password_validation.validate_password(password, validator_config)
```

20.5. Библиотека Python Social Auth: регистрация и вход через социальные сети

В настоящее время очень и очень многие пользователи Интернета являются подписчиками какой-либо социальной сети, а то и не одной. Неудивительно, что появились решения, позволяющие выполнять регистрацию в списках пользователей различных сайтов и вход на них посредством социальных сетей. Одно из таких решений — дополнительная библиотека Python Social Auth.

Python Social Auth поддерживает не только Django, но и ряд других веб-фреймворков, написанных на Python. Она предоставляет исключительно простые в использовании (но несколько мудреные в настройке) средства для регистрации на сайте и выполнения входа на него посредством более чем 100 социальных сетей и интернет-сервисов, включая «ВКонтакте», Facebook, Twitter, GitHub, Instagram и др.

В этой главе описана установка и использование библиотеки для выполнения регистрации и входа на сайт посредством социальной сети «ВКонтакте».

> ***НА ЗАМЕТКУ***
>
> Полное руководство по Python Social Auth располагается здесь: **http://python-social-auth.readthedocs.io/**.

20.5.1. Создание приложения «ВКонтакте»

Чтобы успешно выполнять регистрацию и вход на сайт посредством «ВКонтакте», необходимо предварительно зарегистрировать в этой социальной сети новое приложение. Вот шаги, которые необходимо выполнить для этого:

1. Выполнить вход в социальную сеть «ВКонтакте». Если вы не являетесь подписчиком этой сети, предварительно зарегистрируйтесь на ней.

2. Перейти на страницу списка приложений, созданных текущим пользователем, которая находится по интернет-адресу **https://vk.com/apps?act=manage**.
3. Перейти на страницу создания нового приложения, нажав кнопку **Создать приложение**.
4. Заполнить форму со сведениями о создаваемом приложении (рис. 20.1). Название приложения, заносимое в одноименное поле ввода, может быть каким угодно. В группе **Платформа** следует выбрать переключатель **Веб-сайт**. В поле ввода **Адрес сайта** заносится интернет-адрес сайта, для которого необходимо реализовать вход через «ВКонтакте», а в поле ввода **Базовый домен** — его домен. Если сайт в данный момент еще разрабатывается и развернут на локальном хосте, следует ввести интернет-адрес **http://localhost** и домен **localhost** соответственно. Введя все нужные данные, нужно нажать кнопку **Подключить сайт**.

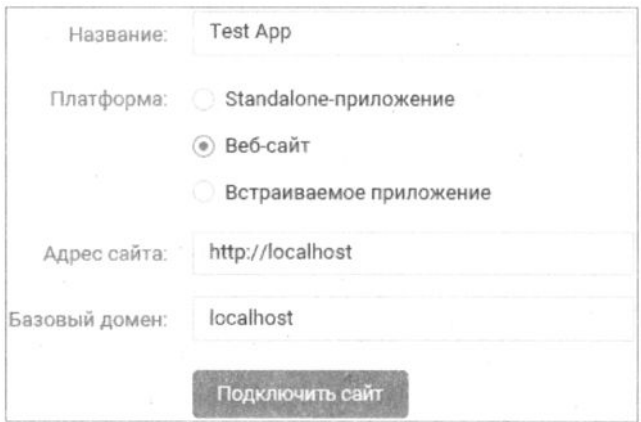


Рис. 20.1. Форма для создания нового приложения «ВКонтакте»

5. Следуя появляющимся на экран инструкциям, запросить SMS-подтверждение на создание нового приложения.
6. На следующей странице, где выводятся полные сведения о созданном приложении, в левой колонке щелкнуть на гиперссылке **Настройки**. В появившейся на экране форме настроек приложения, на самом ее верху, найти поля **ID приложения** и **Защищенный ключ** (рис. 20.2). Эти величины лучше переписать куда-нибудь — они нам скоро пригодятся.

Внимание!

ID приложения и, в особенности, его защищенный ключ необходимо хранить в тайне.

Рис. 20.2. Поля **ID приложения** и **Защищенный ключ**, находящиеся в форме настроек приложения «ВКонтакте»

20.5.2. Установка и настройка Python Social Auth

Для установки редакции библиотеки Python Social Auth, предназначенной для Django, необходимо в командной строке отдать команду:

```
pip install social-auth-app-django
```

Одновременно с самой этой библиотекой будет установлено довольно много других библиотек, используемых ею в работе.

Перед использованием Python Social Auth мы выполним следующие шаги:

1. Зарегистрируем в проекте приложение `social_django` — программное ядро библиотеки:

```
INSTALLED_APPS = [
    . . .
    'social_django',
]
```

2. Выполним миграции, чтобы приложение создало в базе данных все необходимые для своих моделей структуры.

3. Если используется СУБД PostgreSQL, добавим в модуль settings.py пакета конфигурации такой параметр:

```
SOCIAL_AUTH_POSTGRES_JSONFIELD = True
```

   Он разрешает использование для хранения данных поле типа `JSONB`, поддерживаемого этой СУБД.

4. Добавим в список классов, реализующих аутентификацию и авторизацию, класс `social_core.backends.vk.VKOAuth2`:

```
AUTHENTICATION_BACKENDS = (
    'social_core.backends.vk.VKOAuth2',
    'django.contrib.auth.backends.ModelBackend',
)
```

   Параметр `AUTHENTICATION_BACKENDS` придется добавить в модуль settings.py, т. к. изначально его там нет.

5. Добавим в список обработчиков контекста для используемого нами шаблонизатора классы `social_django.context_processors.backends` и `social_django.context_processors.login_redirect`:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            'context_processors': [
                . . .
                'social_django.context_processors.backends',
                'social_django.context_processors.login_redirect',
            ],
```

```
            . . .
        },
    },
]
```

6. Добавим в модуль settings.py параметры, указывающие полученные ранее ID приложения и защищенный ключ:

```
SOCIAL_AUTH_VK_OAUTH2_KEY = 'XXXXXXX'
SOCIAL_AUTH_VK_OAUTH2_SECRET = 'XXXXXXXXXXXXXXXXXXXX'
```

7. Если нам необходимо помимо всех прочих сведений о пользователе получать от сети «ВКонтакте» еще и его адрес электронной почты, мы добавим в модуль settings.py такой параметр:

```
SOCIAL_AUTH_VK_OAUTH2_SCOPE = ['email']
```

20.5.3. Использование Python Social Auth

Использовать библиотеку Python Social Auth для выполнения регистрации и входа на сайт очень просто.

Сначала нужно создать маршруты, которые ведут на контроллеры, выполняющие регистрацию и вход. Эти маршруты добавляются в список маршрутов уровня проекта — в модуль urls.py пакета конфигурации. Вот пример:

```
urlpatterns = [
    . . .
    path('social/', include('social_django.urls', namespace='social')),
]
```

Префикс, указываемый в первом параметре функции `path()`, может быть любым.

Далее мы добавим в шаблон страницы входа гиперссылку, которая укажет на контроллер, выполняющий вход на сайт и, если это необходимо, регистрацию нового пользователя на основе сведений, полученных от «ВКонтакте». Вот код, создающий эту гиперссылку:

```
<a href="{% url 'social:begin' 'vk-oauth2' %}">Войти через ВКонтакте</a>
```

При щелчке на такой гиперссылке на экране откроется окно с формой для входа в сеть «ВКонтакте». После успешного выполнения входа пользователь будет перенаправлен на сайт по пути, записанном в параметре `LOGIN_REDIRECT_URL` настроек проекта (см. *разд. 15.2.1*). Если же пользователь ранее выполнил вход на «ВКонтакте», он будет просто перенаправлен по пути, записанному в упомянутом ранее параметре.

20.6. Указание своей модели пользователя

Для хранения списка пользователей подсистема разграничения доступа Django использует стандартную модель `User`, объявленную в модуле `django.contrib.auth.models`. Эта модель хранит объем сведений о пользователе, вполне достаточный для многих случаев. Однако часто приходится сохранять в составе сведений

о пользователе дополнительные данные — например, номер телефона, интернет-адрес сайта, признак, хочет ли пользователь получать по электронной почте уведомления о новых сообщениях, и т. п.

Можно объявить дополнительную модель, поместить в нее поля для хранения всех нужных данных и добавить поле, устанавливающее связь «один-с-одним» со стандартной моделью пользователя. Вот пример создания подобной дополнительной модели:

```
from django.db import models
from django.contrib.auth.models import User
class Profile(models.Model):
    phone = models.CharField(max_length=20)
    user = models.OneToOneField(User, on_delete=models.CASCADE)
```

Разумеется, при создании нового пользователя нам придется явно создавать связанную с ним запись модели, хранящую дополнительные сведения. Зато у нас не будет никаких проблем с подсистемой разграничения доступа и старыми дополнительными библиотеками, поскольку для хранения основных сведений о пользователях будет использоваться стандартная модель `User`.

Другой подход заключается в написании своей собственной модели пользователя. Такую модель следует сделать производной от класса `AbstractUser`, который объявлен в модуле `django.contrib.auth.models`, реализует всю функциональность по хранению пользователей и представляет собой абстрактную модель (см. *разд. 16.4.2*) — собственно, класс стандартной модели пользователей `User` также является производным от класса `AbstractUser`. Пример:

```
from django.db import models
from django.contrib.auth.models import AbstractUser
class AdvUser(AbstractUser):
    phone = models.CharField(max_length=20)
```

Новую модель пользователя следует указать в параметре `AUTH_USER_MODEL` настроек проекта (см. *разд. 20.1*):

```
AUTH_USER_MODEL = 'testapp.models.AdvUser'
```

Преимуществом такого подхода является то, что нам не придется самостоятельно создавать связанные записи, хранящие дополнительные сведения о пользователе, — это сделает за нас Django. Однако нужно быть готовым к тому, что некоторые дополнительные библиотеки, в особенности старые, которые работают со списком пользователей, не считывают имя модели пользователя из параметра `AUTH_USER_MODEL`, а обращаются напрямую к модели `User`. Если такая библиотека добавит в список нового пользователя, он будет сохранен без дополнительных сведений, и код, который считывает эти сведения, не будет работать.

Если нужно лишь расширить или изменить функциональность модели пользователя, можно создать на его основе прокси-модель, также не забыв занести ее в параметр `AUTH_USER_MODEL`:

```
from django.db import models
from django.contrib.auth.models import User
```

```
class AdvUser(User):
    . . .
    class Meta:
        proxy = True
```

И, наконец, можно написать полностью свой класс модели. Однако такой подход применяется весьма редко из-за его трудоемкости. Интересующиеся могут обратиться к странице **https://docs.djangoproject.com/en/2.1/topics/auth/customizing/**, где приводятся все нужные инструкции.

20.7. Создание своих прав пользователя

Из *главы 15* мы знаем о правах, определяющих операции, которые пользователь может выполнять над записями какой-либо модели. Изначально для каждой модели создаются четыре стандартных права: на просмотр, добавление, правку и удаление записей.

Однако мы можем создать для любой модели дополнительный набор произвольных прав. Для этого мы воспользуемся параметром `permissions`, задаваемым для самой модели — во вложенном классе `Meta`. В качестве его значения указывается список или кортеж, каждый элемент которого описывает одно право и также представляет собой кортеж из двух элементов: обозначения, используемого самим Django, и наименования, предназначенного для вывода на экран. Пример:

```
class Comment(models.Model):
    . . .
    class Meta:
        permissions = (
            ('hide_comments', 'Можно скрывать комментарии'),
        )
```

Обрабатываются эти права точно так же, как и стандартные. В частности, мы можем программно проверить, имеет ли текущий пользователь право скрывать комментарии:

```
def hide_comment(request):
    if request.user.has_perm('bboard.hide_comments'):
        # Пользователь может скрывать комментарии
```

Можно также изменить набор стандартных прав, создаваемых для каждой модели самим Django. Правда, автору книги не понятно, зачем это может понадобиться...

Стандартные права указываются в параметре модели `default_permissions` в виде списка или кортежа, содержащего строки с их наименованиями: `"view"` (просмотр), `"add"` (добавление), `"change"` (правка) и `"delete"` (удаление). Вот пример указания у модели только прав на правку и удаление записей:

```
class Comment(models.Model):
    . . .
    class Meta:
        default_permissions = ('change', 'delete')
```

Значение параметра `default_permissions` по умолчанию: `('view', 'add', 'change', 'delete')` (т. е. будет создан полный набор стандартных прав).

ГЛАВА 21

Посредники и обработчики контекста

Посредник (middleware) Django — это программный модуль, выполняющий предварительную обработку клиентского запроса перед передачей его контроллеру и окончательную обработку ответа, выданного контроллером. Список посредников, зарегистрированных в проекте, указывается в параметре `MIDDLEWARE` настроек проекта (см. *разд. 3.3.4*).

Посредники в Django можно использовать не только для обработки запросов и ответов, но и для добавления в контекст шаблона каких-либо значений. Ту же самую задачу выполняют и обработчики контекста, список которых указывается в дополнительном параметре `context_processors` настроек шаблонизатора (см. *разд. 11.1*).

21.1. Посредники

Посредники — весьма мощный инструмент по обработке данных, пересылаемых по сети. В частности, немалая часть функциональности Django реализована именно в посредниках.

21.1.1. Стандартные посредники

Посредники, изначально включенные в список параметра `MIDDLEWARE` настроек проекта, были описаны в *разд. 3.3.4*. Помимо них, в составе Django имеется еще ряд посредников:

- `django.middleware.gzip.GZipMiddleware` — сжимает запрашиваемую страницу с применением алгоритма GZip, если размер страницы превышает 200 байтов, страница не была сжата на уровне контроллера (для чего достаточно указать у него декоратор `gzip_page()`, описанный в *разд. 9.10*), а веб-обозреватель поддерживает сжатие страниц.

  В списке зарегистрированных посредников должен находиться перед теми, которые получают доступ к содержимому ответа с целью прочитать или изменить его, и после посредника `django.middleware.cache.UpdateCacheMiddleware`;

- `django.middleware.http.ConditionalGetMiddleware` — выполняет обработку заголовков, связанных с кэшированием страниц на уровне клиента. Если ответ не имеет заголовка `E-Tag`, такой заголовок будет добавлен. Если ответ имеет заголовки `E-Tag` или `Last-Modified`, а запрос — заголовки `If-None-Match` или `If-Modified-Since`, вместо страницы будет отправлено сообщение с кодом 304 (запрашиваемая страница не была изменена).

  В списке зарегистрированных посредников должен находиться перед `django.middleware.common.CommonMiddleware`;

- `django.middleware.cache.UpdateCacheMiddleware` — обновляет кэш при включенном режиме кэширования всего сайта.

  В списке зарегистрированных посредников должен находиться перед теми, которые модифицируют заголовок `Vary` (`django.contrib.sessions.middleware.SessionMiddleware` и `django.middleware.gzip.GZipMiddleware`);

- `django.middleware.cache.FetchFromCacheMiddleware` — извлекает запрошенную страницу из кэша при включенном режиме кэширования всего сайта.

  В списке зарегистрированных посредников должен находиться после тех, которые модифицируют заголовок `Vary` (`django.contrib.sessions.middleware.SessionMiddleware` и `django.middleware.gzip.GZipMiddleware`).

  О кэшировании будет рассказано в *главе 25*.

Любой из этих посредников мы в случае необходимости можем вставить в список параметра `MIDDLEWARE` настроек проекта, не забывая при этом следовать указаниям касательно очередности их следования.

21.1.2. Порядок выполнения посредников

Посредники, зарегистрированные в проекте, при получении запроса и формировании ответа выполняются дважды.

Первый раз они выполняются при получении запроса — еще до того, как запрос достигнет контроллера. Выполняются они в том порядке, в котором записаны в списке параметра `MIDDLEWARE` настроек проекта.

Второй раз они выполняются непосредственно после того, как контроллер сгенерирует ответ, но еще до того, как тот будет отправлен клиенту. Если ответ представлен экземпляром класса `TemplateResponse`, посредники выполняются до непосредственного рендеринга шаблона (что позволяет изменить некоторые параметры запроса — например, добавить какие-либо данные в контекст шаблона). Порядок выполнения посредников на этот раз противоположен тому, в каком они записаны в списке параметра `MIDDLEWARE`.

Рассмотрим для примера посредники, зарегистрированные во вновь созданном проекте:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
```

```
    'django.contrib.sessions.middleware.SessionMiddleware',
    . . .
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

При получении запроса сначала будет выполнен самый первый в списке посредник `django.middleware.security.SecurityMiddleware`, далее — `django.contrib.sessions.middleware.SessionMiddleware` и т. д. После выполнения посредника `django.middleware.clickjacking.XFrameOptionsMiddleware`, последнего в списке, управление будет передано контроллеру.

После того как контроллер сгенерирует ответ, выполнится последний в списке посредник `django.middleware.clickjacking.XFrameOptionsMiddleware`, за ним — предпоследний `django.contrib.messages.middleware.MessageMiddleware` и т. д. После выполнения самого первого в списке посредника `django.middleware.security.SecurityMiddleware` ответ отправится клиенту.

21.1.3. Написание своих посредников

Разумеется, если нам не хватает стандартных посредников, мы можем написать свой собственный. Причем реализовать его мы можем как функцию или как класс.

21.1.3.1. Посредники-функции

Посредник-функцию написать проще всего. Однако и функциональных возможностей такие посредники предлагают не очень много.

Посредник такого рода реализуется как функция, которая в качестве результата возвращает другую функцию.

«Внешняя» функция в качестве единственного параметра должна принимать также функцию. Принимаемая функция представляет либо следующий в списке посредник (если это не последний посредник в списке), либо контроллер (если текущий посредник — последний в списке). Назовем ее *функцией-обработчиком*.

«Внутренняя» функция, которая возвращается «внешней» в качестве результата, должна в качестве единственного параметра принимать запрос, представленный в виде экземпляра класса `HttpResponse`. В процессе работы в ее теле будет вызвана функция-обработчик, которая была получена в качестве параметра «внешней» функцией и которой будет передан объект запроса. В качестве результата функция-обработчик вернет экземпляр класса `HttpResponse`, представляющий сгенерированный контроллером ответ. Этот ответ следует вернуть из «внутренней» функции в качестве результата.

Перед вызовом функции-обработчика можно разместить код, который выполняет обработку запроса и, возможно, добавляет в него какие-либо новые атрибуты или меняет его содержимое. После вызова функции-обработчика можно разместить код, обрабатывающий полученный от контроллера ответ.

Вот своеобразный шаблон, согласно которому пишутся посредники-функции:

```
def my_middleware(get_response):
    # Здесь можно выполнить какую-либо инициализацию

    def core_middleware(request):
        # Здесь выполняется обработка клиентского запроса

        response = get_response(request)

        # Здесь выполняется обработка ответа

        return response

    return core_middleware
```

21.1.3.2. Посредники-классы

Посредники-классы предлагают больше функциональных возможностей, но писать их несколько сложнее.

Посредник-класс должен объявлять, по меньшей мере, два метода:

- конструктор — должен принимать в качестве единственного параметра функцию-обработчик.

  Если в теле конструктора возбудить исключение `MiddlewareNotUsed` из модуля `django.core.exceptions`, посредник деактивируется и более не будет использоваться в дальнейшем;

- `__call__()` — должен принимать в качестве единственного параметра объект запроса и возвращать объект ответа. Тело этого метода пишется по тем же правилам, что и тело «внутренней» функции у посредника-функции.

Далее приведен аналогичный шаблон исходного кода, согласно которому пишутся посредники-классы:

```
class MyMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # Здесь можно выполнить какую-либо инициализацию

    def __call__(self, request):
        # Здесь выполняется обработка клиентского запроса

        response = self.get_response(request)

        # Здесь выполняется обработка ответа

        return response
```

Дополнительно в посреднике-классе можно объявить следующие методы:

- `process_view(<запрос>, <контроллер>, <позиционные параметры>, <именованные параметры>)` — выполняется непосредственно перед вызовом следующего в списке посредника или контроллера (если это последний посредник в списке).

  Первым параметром методу передается запрос в виде экземпляра класса `HttpRequest`, вторым — ссылка на функцию, реализующую контроллер. Это может быть контроллер-функция или функция, возвращенная методом `as_view()` контроллера-класса. Третьим параметром методу передается список позиционных параметров, а четвертым — словарь с позиционными параметрами, переданными контроллеру.

  Метод должен возвращать либо `None`, либо экземпляр класса `HttpResponse` (т. е. ответ). В первом случае обработка запроса продолжится: будет вызван следующий в списке посредник или контроллер. Во втором случае обработка запроса прервется, и возвращенный этим методом ответ будет отправлен клиенту;

- `process_exception(<запрос>, <исключение>)` — вызывается при возбуждении исключения в теле контроллера. Первым параметром методу передается запрос в виде экземпляра класса `HttpRequest`, вторым — само исключение в виде экземпляра класса `Exception`.

  Метод должен возвращать либо `None`, либо экземпляр класса `HttpResponse` (т. е. ответ). В первом случае будет выполнена обработка исключения по умолчанию. Во втором случае возвращенный этим методом ответ будет отправлен клиенту;

- `process_template_response(<запрос>, <ответ>)` — вызывается уже после того, как контроллер сгенерировал ответ, но перед рендерингом шаблона. *Запрос* представляется в виде экземпляра класса `HttpRequest`, а *ответ* — в виде экземпляра класса `TemplateResponse`.

  Метод должен возвращать ответ в виде экземпляра класса `TemplateResponse` — либо полученный со вторым параметром и измененный, либо новый, сгенерированный на основе полученного. Этот ответ и будет отправлен клиенту.

  Метод может заменить имя шаблона, занеся его в атрибут `template_name` ответа, или содержимое контекста данных, доступного из атрибута `context_data`.

  > ***Внимание!***
  >
  > Эффект от замены имени шаблона или изменения содержимого контекста в методе `process_template_response()` будет достигнут только в том случае, если ответ представлен экземпляром класса `TemplateResponse`.

В листинге 21.1 приведен код посредника `RubricsMiddleware`, который добавляет в контекст данных список рубрик, взятый из модели `Rubric`.

Листинг 21.1. Пример посредника, добавляющего в контекст шаблона дополнительные данные

```
from .models import Rubric

class RubricsMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        return self.get_response(request)

    def process_template_response(self, request, response):
        response.context_data['rubrics'] = Rubric.objects.all()
        return response
```

Не забудем зарегистрировать этот посредник в проекте (предполагается, что он сохранен в модуле `bboard.middlewares`):

```
MIDDLEWARE = [
    . . .
    'bboard.middlewares.RubricsMiddleware',
]
```

После чего можем удалить из контроллеров код, добавляющий в контекст шаблона список рубрик, разумеется, при условии, что ответ во всех этих контроллерах формируется в виде экземпляра класса `TemplateResponse`.

21.2. Обработчики контекста

Обработчик контекста — это программный модуль, добавляющий в контекст шаблона какие-либо дополнительные данные уже после его формирования контроллером.

Обработчики контекста удобно использовать, если нужно просто добавить в контекст шаблона какие-либо данные. Обработчики контекста реализуются проще, чем посредники, и работают в любом случае, независимо от того, представлен ответ экземпляром класса `TemplateResponse` или `HttpResponse`.

Обработчик контекста реализуется в виде обычной функции. Единственным параметром она должна принимать запрос в виде экземпляра класса `HttpRequest` и возвращать словарь с данными, которые нужно добавить в контекст шаблона.

Листинг 21.2 показывает код обработчика контекста `rubrics`, который добавляет в контекст шаблона список рубрик.

Листинг 21.2. Пример обработчика контекста

```
from .models import Rubric

def rubrics(request):
    return {'rubrics': Rubric.objects.all()}
```

Этот обработчик шаблона мы занесем в список параметра `context_processors`, что входит в состав дополнительных параметров используемого нами шаблонизатора:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            'context_processors': [
                . . .
                'bboard.middlewares.rubrics',
            ],
        . . .
        },
    },
]
```

ГЛАВА 22

Cookie, сессии, всплывающие сообщения и подписывание данных

Django поддерживает развитые средства для обработки cookie, хранения данных в сессиях, вывода всплывающих сообщений и защиты данных цифровой подписью. Эта глава будет посвящена им.

22.1. Cookie

Cookie — инструмент для сохранения произвольных данных на стороне клиента. Единственное требование: объем сохраняемых данных не должен превышать 4 Кбайт.

Все cookie, сохраненные на стороне клиента, относящиеся к текущему домену и еще не просроченные, доступны через атрибут `COOKIES` объекта запроса (экземпляра класса `HttpRequest`). Ключами элементов этого словаря выступают ключи всех доступных cookie, а значениями элементов — значения, сохраненные в этих cookie и представленные в виде строк. Значения cookie доступны только для чтения.

Вот пример извлечения из cookie текущего значения счетчика посещений страницы и увеличения его на единицу:

```
if 'counter' in request.COOKIES:
    cnt = int(request.COOKIES['counter']) + 1
else:
    cnt = 1
```

Для записи значения в cookie применяется метод `set_cookie()` класса `HttpResponse`, представляющего ответ. Вот формат вызова этого метода:

```
set_cookie(<ключ>[, value='']['', max_age=None][, expires=None][,
           path='/'][, domain=None][, secure=False][, httponly=False][,
           samesite=None])
```

Ключ записываемого значения указывается в виде строки. Само значение задается в параметре `value`, если он не указан, будет записана пустая строка.

Параметр `max_age` указывает время хранения cookie на стороне клиента в секундах. Параметр `expires` задает дату и время, после которых cookie станет недействительным и будет удален, в виде объекта типа `datetime` из модуля `datetime` Python. В вызове метода следует указать один из этих параметров, но не оба сразу. Если ни один из этих параметров не указан, cookie будет храниться лишь до тех пор, пока посетитель не уйдет с сайта.

Параметр `path` указывает путь, к которому будет относиться cookie, — в таком случае при запросе с другого пути сохраненное в cookie значение получить не удастся. Например, если задать значение `"/testapp/"`, cookie будет доступен только в контроллерах приложения `testapp`. Значение по умолчанию: `"/"` (путь к корневой папке) — в результате чего cookie станет доступным с любого пути.

Параметр `domain` указывает корневой домен, откуда должен быть доступен сохраняемый cookie, и применяется, если нужно создать cookie, доступный с другого домена. Так, если указать значение `"site.ru"`, cookie будет доступен с доменов **www.site.ru**, **support.site.ru**, **shop.site.ru** и др. Если параметр не указан, cookie будет доступен только в текущем домене.

Если параметру `secure` дать значение `True`, cookie будет доступен только при обращении к сайту по защищенному протоколу. Если параметр не указан (или если ему дано значение `False`), cookie будет доступен при обращении по любому протоколу.

Если параметру `httponly` дать значение `True`, cookie будет доступен только при обращении к сайту по протоколам HTTP и HTTPS. Если параметр не указан (или если ему дано значение `False`), cookie будет доступен также при обращении через AJAX-запрос.

Параметр `samesite` разрешает или запрещает отправку сохраненного cookie при выполнении запросов на другие сайты. Доступны три значения:

- `None` — разрешает отправку cookie (поведение по умолчанию);
- `"Lax"` — разрешает отправку cookie только при переходе на другие сайты по гиперссылкам;
- `"Strict"` — полностью запрещает отправку cookie другим сайтам.

Вот пример записи в cookie значения счетчика посещений страницы, полученного ранее:

```
response = HttpResponse(. . .)
response.set_cookie('counter', cnt)
```

Удалить cookie можно вызовом метода `delete_cookie()` класса `HttpResponse`:

```
delete_cookie(<ключ>[, path='/'][, domain=None])
```

Значения параметров `path` и `domain` должны быть теми же, что использовались в вызове метода `set_cookie()`, создавшего удаляемый cookie. Если cookie с заданным ключом не найден, метод ничего не делает.

Django также поддерживает создание и чтение *подписанных* cookie, в которых сохраненное значение дополнительно защищено цифровой подписью.

Сохранение значения в подписанном cookie выполняется вызовом метода `set_signed_cookie()` класса `HttpResponse`:

```
set_signed_cookie(<ключ>[, value=''][, salt=''][, max_age=None][,
                  expires=None][, path='/'][, domain=None][,
                  secure=False][, httponly=False][, samesite=None])
```

Здесь указываются те же самые параметры, что и у метода `set_cookie()`. Дополнительный параметр `salt` задает *соль* — особое значение, участвующее в генерировании цифровой подписи и служащее для повышения ее стойкости.

Если в параметре `max_age` или `expires` задано время существования подписанного cookie, сгенерированная цифровая подпись будет действительна в течение указанного времени.

Прочитать значение из подписанного cookie и удостовериться, что оно не скомпрометировано, позволяет метод `get_signed_cookie()` класса `HttpRequest`:

```
get_signed_cookie(<ключ>[, default=RAISE_ERROR][, salt=''][,
                                                max_age=None])
```

Значение соли, заданное в параметре `salt`, должно быть тем же, что использовалось в вызове метода `set_signed_cookie()`, создавшего этот cookie.

Если цифровая подпись у сохраненного значения не была скомпрометирована, метод вернет сохраненное значение. В противном случае будет возвращено значение, заданное в необязательном параметре `default`. То же самое случится, если cookie с заданным ключом не был найден.

Если в качестве значения параметра `default` указать значение переменной `RAISE_ERROR` из модуля `django.http.request`, будет возбуждено одно из двух исключений: `BadSignature` из модуля `django.core.signing`, если цифровая подпись скомпрометирована, или `KeyError`, если cookie с заданным ключом не найден.

Если в необязательном параметре `max_age` указано время существования подписанного cookie, дополнительно будет выполнена проверка, не устарела ли цифровая подпись. Если цифровая подпись устарела, метод возбудит исключение `SignatureExpired` из модуля `django.core.signing`.

Удалить подписанный cookie можно так же, как и cookie обычный, — вызовом метода `delete_cookie()` объекта ответа.

22.2. Сессии

Сессия — это промежуток времени, в течение которого посетитель пребывает на текущем сайте. Сессия начинается, как только посетитель заходит на сайт, и завершается после его ухода.

К сессии можно привязать произвольные данные и сохранить их в выбранном хранилище (базе данных, файле и др.). Эти данные будут храниться во время существования сессии и останутся в течение определенного времени после ее завершения,

пока не истечет указанный промежуток времени, и сессия не перестанет быть актуальной. Такие данные тоже называют *сессией*.

Для каждой сессии Django генерирует уникальный идентификатор, который затем сохраняется в подписанном cookie на стороне клиента (*cookie сессии*). Поскольку содержимое всех cookie, сохраненных для того или иного домена, автоматически отсылается веб-серверу в составе заголовка каждого запроса, Django впоследствии без проблем получит сохраненный на стороне клиента идентификатор и по нему найдет данные, записанные в соответствующей ему сессии.

Мы можем сохранить в сессии любые данные, какие нам нужны. В частности, подсистема разграничения доступа хранит в таких сессиях ключ пользователя, который выполнил вход на сайт.

Если cookie хранится на стороне клиента, то данные сессии, напротив, сохраняются на стороне сервера. Следовательно, в них можно хранить конфиденциальные данные, которые не должны быть доступны никому.

22.2.1. Настройка сессий

Чтобы успешно работать с сессиями, предварительно следует:

- проверить, присутствует ли приложение `django.contrib.sessions` в списке зарегистрированных в проекте (параметр `INSTALLED_APPS`);
- проверить, присутствует ли посредник `django.contrib.sessions.middleware.SessionMiddleware` в списке зарегистрированных в проекте (параметр `MIDDLEWARE`).

Впрочем, и приложение, и посредник присутствуют в списках изначально, поскольку активно используются другими стандартными приложениями Django.

А теперь рассмотрим параметры, влияющие на работу подсистемы сессий. Все эти параметры, как обычно, указываются в настройках проекта — в модуле settings.py пакета конфигурации:

- `SESSION_ENGINE` — имя класса, реализующего хранилище для сессий, в виде строки. Можно указать следующие классы:
  - `django.contrib.sessions.backends.db` — хранит сессии в базе данных. Имеет среднюю производительность, но гарантирует максимальную надежность хранения данных;
  - `django.contrib.sessions.backends.file` — хранит сессии в обычных файлах. По сравнению с предыдущим хранилищем имеет пониженную производительность, но создает меньшую нагрузку на базу данных;
  - `django.contrib.sessions.backends.cache` — хранит сессии в кэше стороны сервера. Обеспечивает высокую производительность, но требует наличия активной подсистемы кэширования;
  - `django.contrib.sessions.backends.cached_db` — хранит сессии в кэше стороны сервера, одновременно дублируя их в базе данных для надежности. По

сравнению с предыдущим хранилищем обеспечивает повышенную надежность, но увеличивает нагрузку на базу данных;

- `django.contrib.sessions.backends.signed_cookies` — хранит сессии непосредственно в cookie сессии. Обеспечивает максимальную производительность, но для каждой сессии позволяет сохранить не более 4 Кбайт данных.

Значение параметра по умолчанию: `"django.contrib.sessions.backends.db"`;

- `SESSION_SERIALIZER` — имя класса сериализатора, который будет использоваться для сериализации сохраняемых в сессиях данных, указанное в виде строки. В составе Django поставляются два сериализатора:
  - `django.contrib.sessions.serializers.JSONSerializer` — сериализует данные в формат JSON. Может обрабатывать только элементарные типы Python;
  - `django.contrib.sessions.serializers.PickleSerializer` — сериализует средствами модуля `pickle`. Способен обработать значение любого типа.

  Значение параметра по умолчанию: `"django.contrib.sessions.serializers.JSONSerializer"`;
- `SESSION_EXPIRE_AT_BROWSER_CLOSE` — если `True`, сессии со всеми сохраненными в них данными будут автоматически удаляться, как только посетитель закроет веб-обозреватель, если `False`, сессии будут сохраняться. Значение по умолчанию — `False`;
- `SESSION_SAVE_EVERY_REQUEST` — если `True`, сессии будут сохраняться в хранилище при выполнении каждого запроса, если `False` — только при изменении записанных в них данных. Значение по умолчанию — `False`;
- `SESSION_COOKIE_DOMAIN` — домен, к которому будут относиться cookie сессий. Значение по умолчанию — `None` (т. е. текущий домен);
- `SESSION_COOKIE_PATH` — путь, к которому будут относиться cookie сессий. Значение по умолчанию: `"/"`;
- `SESSION_COOKIE_AGE` — время существования cookie сессий, в виде целого числа в секундах. Значение по умолчанию: `1209600` (2 недели);
- `SESSION_COOKIE_NAME` — ключ, под которым в cookie будет сохранен идентификатор сессии. Значение по умолчанию: `"sessionid"`;
- `SESSION_COOKIE_HTTPONLY` — если `True`, cookie сессий будут доступны только при обращении к сайту по протоколам HTTP и HTTPS. Если `False`, cookie сессий также будут доступны при обращении через AJAX-запрос. Значение по умолчанию — `True`;
- `SESSION_COOKIE_SECURE` — если `True`, cookie сессий будут доступны только при обращении к сайту по защищенному протоколу, если `False` — при обращении по любому протоколу. Значение по умолчанию — `False`;
- `SESSION_COOKIE_SAMESITE` — признак, разрешающий или запрещающий отправку cookie сессий при переходе на другие сайты. Доступны три значения:

- `None` — разрешает отправку cookie сессий (поведение по умолчанию);
- `"Lax"` — разрешает отправку cookie сессий только при переходе на другие сайты по гиперссылкам;
- `"Strict"` — полностью запрещает отправку cookie сессий другим сайтам;

❐ `SESSION_FILE_PATH` — полный путь к папке, в которой будут храниться файлы с сессиями. Если указано значение `None`, Django использует системную папку для хранения временных файлов. Значение по умолчанию — `None`.

Этот параметр принимается во внимание только в том случае, если для хранения сессий были выбраны обычные файлы;

❐ `SESSION_CACHE_ALIAS` — название кэша, в котором будут храниться сессии. Значение по умолчанию: `"default"` (кэш по умолчанию).

Этот параметр принимается во внимание только в том случае, если для хранения сессий был выбран кэш стороны сервера без дублирования в базе данных или же с таковым.

Если в качестве хранилища сессий были выбраны база данных или кэш стороны сервера с дублированием в базе данных, перед использованием сессий следует выполнить миграции.

> ***На заметку***
>
> Если для хранения сессий были выбраны база данных или кэш стороны сервера с дублированием в базе данных, в базе данных будет создана таблица `django_session`.

22.2.2. Использование сессий

Посредник `django.contrib.sessions.middleware.SessionMiddleware` добавляет объекту запроса атрибут `sessions`. Он хранит объект, поддерживающий функциональность словаря и представляющий все значения, что были сохранены в текущей сессии.

Вот пример реализации счетчика посещений страницы, аналогичного представленному в *разд. 22.1*, но использующего для хранения текущего значения сессии:

```
if 'counter' in request.session:
    cnt = request.session['counter'] + 1
else:
    cnt = 1
. . .
request.session['counter'] = cnt
```

Помимо этого, объект, хранящийся в атрибуте `sessions` объекта запроса, поддерживает следующие методы:

❐ `flush()` — удаляет все данные, сохраненные в текущей сессии, наряду с cookie сессии (метод `clear()`, поддерживаемый тем же объектом, равно как и словарями Python, не удаляет cookie);

- `set_test_cookie()` — создает тестовый cookie, позволяющий удостовериться, что веб-обозреватель клиента поддерживает cookie;
- `test_cookie_worked()` — возвращает `True`, если веб-обозреватель клиента поддерживает cookie, и `False` — в противном случае. Проверка, поддерживает ли веб-обозреватель cookie, запускается вызовом метода `set_test_cookie()`;
- `delete_test_cookie()` — удаляет созданный ранее тестовый cookie.

Вот пример использования трех из только что описанных методов:

```
def test_cookie(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            # Веб-обозреватель поддерживает cookie
        else:
            # Веб-обозреватель не поддерживает cookie
    request.session.set_test_cookie()
    return render(request, 'testapp/test_cookie.html')
```

- `set_expiry(<время>)` — задает время устаревания текущей сессии. По достижении этого времени сессия будет удалена. В качестве значения *времени* можно указать:
  - целое число — задаст количество секунд, в течение которых сессия будет актуальна;
  - объект типа `datetime` или `timedelta` из модуля `datetime` — укажет дату и время устаревания сессии. Поддерживается только при использовании сериализатора `django.contrib.sessions.serializers.JSONSerializer`;
  - `0` — сессия перестанет быть актуальной и будет удалена, как только посетитель закроет веб-обозреватель;
  - `None` — будет использовано значение из параметра `SESSION_COOKIE_AGE` настроек проекта;
- `get_expiry_age([modification=datetime.datetime.today()][,][expiry=None])` — возвращает количество секунд, в течение которых сессия еще будет актуальна. Необязательный параметр `modification` указывает дату и время последнего изменения сессии (по умолчанию — текущие дата и время), а параметр `expiry` — дату и время ее устаревания (по умолчанию это дата и время, заданные вызовом метода `set_expiry()` или взятые из параметра `SESSION_COOKIE_AGE`);
- `get_expiry_date([modification=datetime.datetime.today()][,][expiry=None])` — возвращает дату устаревания текущей сессии. Необязательный параметр `modification` указывает дату и время последнего изменения сессии (по умолчанию — текущие дата и время), а параметр `expiry` — дату и время ее устаревания (по умолчанию это дата и время, заданные вызовом метода `set_expiry()` или взятые из параметра `SESSION_COOKIE_AGE`);

- `get_expire_at_browser_close()` — возвращает `True`, если текущая сессия устареет и будет удалена, как только посетитель закроет веб-обозреватель, и `False` — в противном случае;
- `clear_expired()` — удаляет устаревшие сессии;
- `cycle_key()` — создает новый идентификатор для текущей сессии, не теряя ее данных.

22.2.3. Дополнительная команда *clearsessions*

Для удаления всех устаревших сессий, которые по какой-то причине не были удалены автоматически, достаточно применить команду `clearsessions` утилиты manage.py. Формат ее вызова очень прост:

```
manage.py clearsessions
```

22.3. Всплывающие сообщения

Всплывающие сообщения существуют только во время выполнения текущего запроса. Они применяются для вывода на страницу какого-либо сообщения (например, об успешном добавлении новой записи), актуального только в данный момент.

22.3.1. Настройка всплывающих сообщений

Перед использованием подсистемы всплывающих сообщений необходимо:

- проверить, присутствует ли приложение `django.contrib.messages` в списке зарегистрированных в проекте (параметр `INSTALLED_APPS`);
- проверить, присутствуют ли посредники `django.contrib.sessions.middleware.SessionMiddleware` и `django.contrib.messages.middleware.MessageMiddleware` в списке зарегистрированных в проекте (параметр `MIDDLEWARE`);
- проверить, присутствует ли обработчик контекста `django.contrib.messages.context_processors.messages` в списке зарегистрированных для используемого шаблонизатора.

Впрочем, и приложение, и посредники, и обработчик контекста присутствуют в списках изначально.

Настроек, управляющих работой подсистемы всплывающих сообщений, очень немного. Все они записываются в модуле settings.py пакета конфигурации:

- `MESSAGE_STORAGE` — имя класса, реализующего хранилище всплывающих сообщений, представленное в виде строки. В составе Django поставляются три хранилища всплывающих сообщений:
  - `django.contrib.messages.storage.cookie.CookieStorage` — использует cookie;
  - `django.contrib.messages.storage.session.SessionStorage` — использует сессии;

- `django.contrib.messages.storage.fallback.FallbackStorage` — использует cookie, но сообщения, чей объем превышает 4 Кбайт, сохраняет в сессии.

Значение параметра по умолчанию: `"django.contrib.messages.storage.fallback.FallbackStorage"`;

- `MESSAGE_LEVEL` — минимальный уровень всплывающих сообщений, которые будут выводиться подсистемой. Указывается в виде целого числа. Значение по умолчанию: `20`;
- `MESSAGE_TAGS` — соответствия между уровнями сообщений и стилевыми классами. Более подробно будет рассмотрен позже.

22.3.2. Уровни всплывающих сообщений

Каждое всплывающее сообщение, обрабатываемое Django, помимо собственно текстового содержимого, имеет так называемый *уровень*, выраженный целым числом. Он указывает своего рода ранг всплывающего сообщения.

Для каждого уровня всплывающих сообщений существует свой стилевой класс. Он привязывается к тегу, в который помещается текст каждого сообщения при его выводе на экран.

Изначально в Django объявлено пять уровней всплывающих сообщений, каждый из которых имеет строго определенную область применения. Значение каждого из этих уровней занесено в отдельную переменную, и эти переменные, объявленные в модуле `django.contrib.messages`, можно использовать для указания уровня сообщений при их выводе.

Все пять уровней, вместе со связанными с ними стилевыми классами, приведены в табл. 22.1.

Таблица 22.1. *Уровни всплывающих сообщений, объявленные в Django*

Переменная	Значение	Описание	Стилевой класс
`DEBUG`	10	Отладочные сообщения, предназначенные только для разработчиков	`debug`
`INFO`	20	Информационные сообщения для посетителей	`info`
`SUCCESS`	25	Сообщения об успешном выполнении каких-либо действий	`success`
`WARNING`	30	Сообщения о возникновении каких-либо нештатных ситуаций, которые могут привести к сбою	`warning`
`ERROR`	40	Сообщения о неуспешном выполнении каких-либо действий	`error`

Параметр `MESSAGE_LEVEL` настроек проекта указывает минимальный уровень сообщений, которые будут обрабатываться. Если уровень создаваемого сообщения меньше указанной в нем величины, сообщение не будет обработано. По умолчанию

этот параметр имеет значение 20 (переменная INFO), следовательно, сообщения с меньшим уровнем, в частности отладочные (DEBUG), обрабатываться не будут. Если нужно сделать так, чтобы отладочные сообщения также выводились на экран, необходимо задать для этого параметра подходящее значение:

```
from django.contrib import messages
MESSAGE_LEVEL = messages.DEBUG
```

22.3.3. Создание всплывающих сообщений

Создать всплывающее сообщение для его последующего вывода можно вызовом различных функций, объявленных в модуле django.contrib.messages.

В первую очередь это, разумеется, «универсальная» функция add_message(), создающая и запускающая в обработку сообщение любого уровня. Вот формат ее вызова:

```
add_message(<запрос>, <уровень сообщения>, <текст сообщений>[,
                    extra_tags=''][, fail_silently=False])
```

Запрос представляется экземпляром класса HttpRequest, *уровень сообщения* — целым числом, а его *текст* — строкой.

Необязательный параметр extra_tags указывает перечень дополнительных стилевых классов, привязываемых к тегу, в который будет заключен текст этого всплывающего сообщения. Перечень стилевых классов должен представлять собой строку, а стилевые классы в нем должны отделяться друг от друга пробелами.

Если задать для необязательного параметра fail_silently значение True, то в случае невозможности создания нового всплывающего сообщения (например, если соответствующая подсистема отключена) ничего не произойдет. Используемое по умолчанию значение False указывает в таком случае возбудить исключение MessageFailure из того же модуля django.contrib.messages.

Вот пример создания нового всплывающего сообщения:

```
from django.contrib import messages
. . .
def edit(request, pk):
    . . .
    messages.add_message(request, messages.SUCCESS,
                                  'Объявление исправлено')
    . . .
```

А вот пример создания нового всплывающего сообщения с добавлением дополнительных стилевых классов first и second:

```
messages.add_message(request, messages.SUCCESS,
                     'Объявление исправлено', extra_tags='first second')
```

Также можно использовать одну из функций: debug(), info(), success(), warning() и error(), выводящих сообщение соответствующего уровня. Все они имеют одинаковый формат вызова:

```
debug|info|success|warning|error(<запрос>, <текст сообщения>[,
                                 extra_tags=''][, fail_silently=False])
```

Значения параметров здесь задаются в том же формате, что и у функции `add_message()`.

Пример:

```
messages.success(request, 'Объявление исправлено')
```

Добавить поддержку вывода всплывающих сообщений высокоуровневым контроллерам-классам можно, унаследовав их от класса-примеси `SuccessMessageMixin`, который объявлен в модуле `django.contrib.messages.views`. Класс `SuccessMessageMixin` поддерживает следующие атрибут и метод:

- `success_message` — текст сообщения об успешном выполнении операции в виде строки. В строке допускается применять специальные символы вида `%(`*<имя поля формы>*`)s`, вместо которых будут подставлены значения соответствующих *полей*;
- `get_success_message(`*<словарь с данными формы>*`)` — должен возвращать полностью сформированный текст всплывающего сообщения об успешном выполнении операции. *Словарь с данными формы* берется из атрибута `cleaned_data` и содержит полностью готовые к использованию значения полей формы.

  В изначальной реализации возвращает результат форматирования строки из атрибута `success_message` с применением полученного *словаря с данными формы*.

Листинг 22.1 показывает код контроллера, создающего новое объявление, который в случае успешного его создания отправляет посетителю всплывающее сообщение.

Листинг 22.1. Использование примеси `SuccessMessageMixin`

```
from django.views.generic.edit import CreateView
from django.contrib.messages.views import SuccessMessageMixin
from .models import Bb
from .forms import BbForm

class BbCreateView(SuccessMessageMixin, CreateView):
    template_name = 'bboard/create.html'
    form_class = BbForm
    success_url = '/{rubric_id}'
    success_message = 'Объявление о продаже товара "%(title)s" создано.'
```

22.3.4. Вывод всплывающих сообщений

Вывести всплывающие сообщения в шаблоне удобнее всего посредством обработчика контекста `django.contrib.messages.context_processors.messages`. Он добавляет в контекст шаблона переменную `messages`, хранящую последовательность всех всплывающих сообщений, созданных к настоящему времени в текущем запросе.

Каждый элемент этой последовательности представляет собой экземпляр класса `Message`. Все необходимые сведения о сообщении хранятся в его атрибутах:

- `message` — текст всплывающего сообщения;
- `level` — уровень всплывающего сообщения в виде целого числа;
- `level_tag` — имя основного стилевого класса, соответствующего уровню сообщения;
- `extra_tags` — строка с дополнительными стилевыми классами, указанными в параметре `extra_tags` при создании сообщения (см. *разд. 22.3.3*);
- `tags` — строка со всеми стилевыми классами — и основным, и дополнительными, — записанными через пробелы.

Вот пример кода шаблона, выполняющего вывод всплывающих сообщений:

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>
        {{ message }}
    </li>
    {% endfor %}
</ul>
{% endif %}
```

Еще обработчик контекста `django.contrib.messages.context_processors.messages` добавляет в контекст шаблона переменную `DEFAULT_MESSAGE_LEVELS`. Ее значением является словарь, в качестве ключей элементов которого выступают строковые названия уровней сообщений, а значений элементов — соответствующие им числа. Этот словарь можно использовать в операциях сравнения, подобных этой:

```
<li{% if message.tags %} class="{{ message.tags }}"{% endif %}>
    {% if message.level == DEFAULT_MESSAGE_LEVELS.ERROR %}
    Внимание!
    {% endif %}
    {{ message }}
</li>
```

Если же нужно получить доступ к сообщениям в контроллере, можно воспользоваться функцией `get_messages(`*<запрос>*`)` из модуля `django.contrib.messages`. *Запрос* представляется экземпляром класса `HttpRequest`, а результатом будет список сообщений, представленных экземплярами класса `Message`. Пример:

```
from django.contrib import messages
. . .
def edit(request, pk):
    . . .
    messages = messages.get_messages(request)
    first_message_text = messages[0].message
    . . .
```

22.3.5. Объявление своих уровней всплывающих сообщений

Никто и ничто не мешает нам при создании всплывающего сообщения указать произвольный уровень:

```
CRITICAL = 50
messages.add_message(request, CRITICAL,
                             'Случилось что-то очень нехорошее...')
```

Нужно только проследить за тем, чтобы выбранное нами значение уровня не совпало с каким-либо из объявленных в самом Django (см. *разд. 22.3.2*).

Если мы хотим, чтобы при выводе всплывающих сообщений на экран для объявленного нами уровня устанавливался основной стилевой класс, то должны выполнить дополнительные действия. Мы объявим словарь, добавим в него элемент, соответствующий объявленному нами уровню сообщений, установим в качестве ключа элемента значение уровня, а в качестве значения элемента — строку с именем стилевого класса, после чего присвоим этот словарь параметру `MESSAGE_TAGS` настроек проекта. Вот пример:

```
MESSAGE_TAGS = {
    CRITICAL: 'critical',
}
```

22.4. Подписывание данных

Разумеется, Django предоставляет средства для защиты произвольных значений цифровой подписью.

Для подписывания строковых значений обычной цифровой подписью применяется класс `Signer` из модуля `django.core.signing`. Конструктор этого класса вызывается в формате:

```
Signer([key=None][,][sep=':'][,][salt=None])
```

Параметр `key` указывает секретный ключ, на основе которого будет генерироваться цифровая подпись (по умолчанию используется секретный ключ из параметра `SECRET_KEY` настроек проекта). Параметр `sep` задает символ, которым будут отделяться друг от друга подписанное значение и сама подпись (по умолчанию — двоеточие). Наконец, параметр `salt` указывает соль (если он опущен, соль использована не будет).

Класс `Signer` поддерживает два метода:

- `sign(<подписываемое значение>)` — подписывает полученное *значение* и возвращает результат:

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> val = signer.sign('Django')
>>> val
'Django:-pY1RFu4UfVaZe2D5DGssA9-fCE'
```

- `unsign(<подписанное значение>)` — из полученного *подписанного значения* извлекает оригинальную величину, которую и возвращает в качестве результата:

```
>>> signer.unsign(val)
'Django'
```

Если подписанное значение скомпрометировано (не соответствует цифровой подписи), возбуждается исключение `BadSignature` из модуля `django.core.signing`:

```
>>> signer.unsign(val + 's')
Traceback (most recent call last):
. . .
django.core.signing.BadSignature: Signature
"-pY1RFu4UfVaZe2D5DGssA9-fCEs" does not match
```

Класс `TimestampSigner` из того же модуля `django.core.signing` подписывает значение цифровой подписью с ограниченным сроком действия. Формат вызова его конструктора такой же, как у конструктора класса `Signer`.

Класс `TimestampSigner` поддерживает два метода:

- `sign(<подписываемое значение>)` — подписывает полученное *значение* и возвращает результат:

```
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> val = signer.sign('Python')
>>> val
'Python:1fk3bI:8qkPUUQO_QoZi1-KYL4kAdbeIDM'
```

- `unsign(<подписанное значение>[, max_age=None])` — из полученного *подписанного значения* извлекает оригинальную величину, которую и возвращает в качестве результата. Параметр `max_age` задает промежуток времени, в течение которого актуальна цифровая подпись, — эта величина может быть указана в виде целого числа, в секундах, или в виде объекта типа `timedelta` из модуля `datetime`. Если подписанное значение скомпрометировано, возбуждается исключение `BadSignature` из модуля `django.core.signing`. Примеры:

```
>>> signer.unsign(val, max_age=3600)
'Python'
>>> from datetime import timedelta
>>> signer.unsign(val, max_age=timedelta(minutes=30))
'Python'
```

Если цифровая подпись уже не актуальна, возбуждается исключение `SignatureExpired` из модуля `django.core.signing`:

```
>>> signer.unsign(val, max_age=timedelta(seconds=30))
Traceback (most recent call last):
. . .
django.core.signing.SignatureExpired: Signature age
323.8853006362915 > 30.0 seconds
```

Если параметр `max_age` не указан, проверка на актуальность цифровой подписи не проводится, и метод `unsign()` работает так же, как его «тезка» у класса `Signer`:

```
>>> signer.unsign(val)
'Python'
```

Если нужно подписать значение, отличающееся от строки, следует воспользоваться двумя функциями из модуля `django.core.signing`:

- `dumps(<значение>[, key=None][, salt='django.core.signing'][, compress=False])` — подписывает указанное *значение* с применением класса `TimestampSigner` и возвращает результат в виде строки. Параметр `key` задает секретный ключ (по умолчанию — значение из параметра `SECRET_KEY` настроек проекта), а параметр `salt` — соль (по умолчанию — строка `"django.core.signing"`). Если параметру `compress` передать значение `True`, результат будет сформирован в сжатом виде (значение по умолчанию — `False`). Сжатие будет давать более заметный результат при подписывании данных большого объема. Примеры:

```
>>> from django.core.signing import dumps, loads
>>> s1 = dumps(123456789)
>>> s1
'MTIzNDU2Nzg5:1fk3pH:8RHbhlStnD1VfaFSBCbzEPiwJ4I'
>>> s2 = dumps([1, 2, 3, 4])
>>> s2
'WzEsMiwzLDRd:1fk3pf:OREHQtPp_R91A-4MQe3bg-Uw8Eg'
>>> s3 = dumps([1, 2, 3, 4], compress=True)
>>> s3
'WzEsMiwzLDRd:1fk3ps:rAFq6qUn3cvwIqlcoC5HIjCMJS4'
```

- `loads(<подписанное значение>[, key=None][, salt='django.core.signing'][, max_age=None])` — из полученного *подписанного значения* извлекает оригинальную величину и возвращает в качестве результата. Параметры `key` и `salt` указывают, соответственно, секретный ключ и соль — эти значения должны быть теми же, что использовались при подписывании значения вызовом функции `dumps()`. Параметр `max_age` задает промежуток времени, в течение которого цифровая подпись актуальна. Если он опущен, проверка на актуальность подписи не проводится. Примеры:

```
>>> loads(s1)
123456789
>>> loads(s2)
[1, 2, 3, 4]
>>> loads(s3)
[1, 2, 3, 4]
>>> loads(s3, max_age=10)
Traceback (most recent call last):
. . .
django.core.signing.SignatureExpired: Signature age
278.3182246685028 > 10 seconds
```

Если цифровая подпись скомпрометирована или потеряла актуальность, будут возбуждены исключения `BadSignature` или `SignatureExpired` соответственно.

ГЛАВА 23

Сигналы

Сигнал — это сущность, создаваемая Django при выполнении какого-либо действия: создании новой записи в модели, удалении записи, входе пользователя на сайт, выходе с него и пр. К сигналу можно привязать *обработчик* — функцию или метод, который будет вызываться при возникновении сигнала.

Сигналы предоставляют возможность вклиниться в процесс работы самого фреймворка или отдельных приложений — неважно, стандартных или написанных самим разработчиком сайта, — и произвести какие-либо дополнительные действия. Скажем, приложение `django-cleanup`, рассмотренное нами в *разд. 19.6* и удаляющее ненужные файлы, чтобы отследить момент правки или удаления записи, обрабатывает сигналы `post_init`, `pre_save`, `post_save` и `post_delete`.

23.1. Обработка сигналов

Все сигналы в Django представляются экземплярами класса `Signal` или его подклассов. Этот класс поддерживает два метода, предназначенные для привязки к сигналу обработчика или отмены его привязки.

Для привязки обработчика к сигналу применяется метод `connect()` класса `Signal`:

```
connect(<обработчик>[, sender=None][, weak=True][, dispatch_uid=None])
```

`обработчик` сигнала, как было сказано ранее, должен представлять собой функцию или метод. Формат написания этой функции (метода) мы рассмотрим позднее.

В необязательном параметре `sender` можно указать класс, из которого отправляется текущий сигнал (класс-*отправитель*). После чего *`обработчик`* будет обрабатывать сигналы исключительно от этого отправителя.

Если необязательному параметру `weak` присвоено значение `True` (а это его значение по умолчанию), обработчик может быть удален из памяти при выгрузке модуля, в котором он объявлен, и, соответственно, перестанет обрабатывать сигналы. Но если задать этому параметру значение `False`, обработчик никогда не будет выгружен. Задавать этот параметр имеет смысл только в том случае, если в качестве

обработчика выступает функция, вложенная в другую функцию, или метод какого-либо объекта, который существует ограниченное время.

Необязательный параметр `dispatch_uid` указывается, если к одному и тому же сигналу несколько раз привязывается один и тот же обработчик, и возникает необходимость как-то отличить одну такую привязку от другой. В этом случае в разных вызовах метода `connect()` нужно указать разные значения этого параметра, которые должны представлять собой строки.

К одному и тому же сигналу может быть привязано произвольное количество обработчиков, которые будут выполняться один за другим в той последовательности, в которой они были привязаны к сигналу.

Рассмотрим несколько примеров привязки обработчика к сигналу `post_save`, возникающему после сохранения записи модели:

```
from django.db.models.signals import post_save
# Простая привязка обработчика post_save_dispatcher() к сигналу
post_save.connect(post_save_dispatcher)
# Простая привязка обработчика к сигналу, возникающему в модели Bb
post_save.connect(post_save_dispatcher, sender=Bb)
# Двукратная привязка обработчиков к сигналу с указанием разных значений
# параметра dispatch_uid
post_save.connect(post_save_dispatcher,
                  dispatch_uid='post_save_dispatcher_1')
post_save.connect(post_save_dispatcher,
                  dispatch_uid='post_save_dispatcher_2')
```

Обработчик — функция или метод — должен принимать один позиционный параметр, с которым передается класс-отправитель сигнала. Помимо этого, обработчик может принимать произвольное количество именованных параметров, набор которых у каждого сигнала различается (стандартные сигналы Django и передаваемые ими параметры мы рассмотрим позже). Вот своего рода шаблоны для написания обработчиков разного типа:

```
def post_save_dispatcher(sender, **kwargs):
    # Тело функции-обработчика
    # Получаем класс-отправитель сигнала
    snd = sender
    # Получаем значение переданного обработчику именованного параметра
    # instance
    instance = kwargs['instance']
    . . .

class SomeClass:
    def post_save_dispatcher(self, sender, **kwargs):
        # Тело метода-обработчика
        . . .
```

Вместо метода `connect()` объекта сигнала можно использовать декоратор `receiver(<сигнал>)`, объявленный в модуле `django.dispatch`:

```
from django.dispatch import receiver
@receiver(post_save)
def post_save_dispatcher(sender, **kwargs):
    . . .
```

Код, выполняющий привязку обработчиков сигналов, которые должны действовать все время, пока работает сайт, как правило, записывается модуле models.py, в котором объявляются классы моделей.

Если же обработчик должен обрабатывать сигнал в течение какого-то определенного времени, его можно поместить в любой модуль. Обычно их записывают в модуле views.py, где находятся объявления контроллеров.

В последнем случае может возникнуть необходимость отменить привязку обработчика к сигналу. Для этого применяется метод `disconnect()` класса `Signal`:

```
disconnect([receiver=None][,][sender=None][,][dispatch_uid=None])
```

В параметре `receiver` указывается обработчик, ранее привязанный к сигналу. Если этот обработчик был привязан к сигналам, отправляемым конкретным отправителем, последний следует указать в параметре `sender`. Если в вызове метода `connect()`, выполнившем привязку обработчика, был задан параметр `dispatch_uid` с каким-либо значением, удалить привязку можно, записав в вызове метода `disconnect()` только параметр `dispatch_uid` и указав в нем то же значение. Примеры:

```
post_save.disconnect(receiver=post_save_dispatcher)
post_save.disconnect(receiver=post_save_dispatcher, sender=Bb)
post_save.disconnect(dispatch_uid='post_save_dispatcher_2')
```

23.2. Встроенные сигналы Django

Фреймворк предоставляет довольно много встроенных сигналов, отправляемых различными подсистемами. Давайте рассмотрим их, сгруппировав по отправителю.

Сигналы, отправляемые подсистемой доступа к базам данных и объявленные в модуле `django.db.models.signals`:

- `pre_init` — отправляется в самом начале создания новой записи модели, перед выполнением конструктора ее класса. Обработчику передаются следующие параметры:
  - `sender` — класс модели, запись которой создается;
  - `args` — список позиционных аргументов, переданных конструктору модели;
  - `kwargs` — словарь именованных аргументов, переданных конструктору модели.

  Например, при создании нового объявления выполнением выражения:

  ```
  Bb.objects.create(title='Дом', content='Трехэтажный, кирпич',
                    price=50000000)
  ```

обработчик с параметром sender получит ссылку на класс модели Bb, с параметром args — «пустой» список, а с параметром kwargs — словарь {'title': 'Дом', 'content': 'Трехэтажный, кирпич', 'price': 50000000};

- post_init — отправляется в конце создания новой записи модели, после выполнения конструктора ее класса. Обработчику передаются следующие параметры:
  - sender — класс модели, запись которой была создана;
  - instance — объект созданной записи;
- pre_save — отправляется перед сохранением записи модели, перед выполнением ее метода save(). Обработчику передаются параметры:
  - sender — класс модели, запись которой сохраняется;
  - instance — объект сохраняемой записи;
  - raw — True, если запись будет сохранена как есть, без обращения к другим записям за дополнительными данными и без исправления других записей, и False в противном случае;
  - update_fields — множество имен полей, заданных в параметре update_fields метода save(), или None, если этот параметр не был указан;
- post_save — отправляется после сохранения записи модели, после выполнения ее метода save(). Обработчику передаются такие параметры:
  - sender — класс модели, запись которой была сохранена;
  - instance — объект сохраненной записи;
  - created — True, если это вновь созданная запись, и False в противном случае;
  - raw — True, если запись была сохранена как есть, без обращения к другим записям за дополнительными данными и без исправления других записей, и False в противном случае;
  - update_fields — множество имен полей, заданных в параметре update_fields метода save(), или None, если этот параметр не был указан.

Вероятно, это один из наиболее часто обрабатываемых сигналов. Вот пример его обработки с целью вывести в консоли Django сообщение о добавлении объявления:

```
from django.db.models.signals import post_save

def post_save_dispatcher(sender, **kwargs):
    if kwargs['created']:
        print('Объявление в рубрике "%s" создано' % \
              kwargs['instance'].rubric.name)

post_save.connect(post_save_dispatcher, sender=Bb)
```

- pre_delete — отправляется перед удалением записи, перед выполнением ее метода delete(). Параметры, передаваемые обработчику:

- sender — класс модели, запись которой удаляется;
- instance — объект удаляемой записи;

- post_delete — отправляется после удаления записи, после выполнения ее метода delete(). Обработчик этого сигнала получит следующие параметры:
  - sender — класс модели, запись которой была удалена;
  - instance — объект удаленной записи. Отметим, что эта запись более не существует в базе данных;
- m2m_changed — отправляется при изменении состава записей, связанных с обрабатываемой записью посредством связи «многие-со-многими» (см. *разд. 4.4.3*). Обработчик этого сигнала примет весьма много параметров:
  - sender — класс связующей модели. Это может быть как модель, объявленная явно и заданная параметром through конструктора класса поля ManyToManyField, так и модель, что создается фреймворком неявно;
  - instance — объект записи, с которым выполняются манипуляции по изменению состава связанных записей;
  - action — строковое обозначение выполняемого действия:
    - "pre_add" — начало добавления новой записи в состав связанных;
    - "post_add" — окончание добавления новой связанной записи в состав связываемых;
    - "pre_remove" — начало удаления записи из состава связанных;
    - "post_remove" — окончание удаления записи из состава связанных;
    - "pre_clear" — начало удаления всех записей из состава связанных;
    - "post_clear" — окончание удаления всех записей из состава связанных;
  - reverse — False, если изменение состава связанных записей выполняется в записи ведущей модели, и True, если в записи ведомой модели;
  - model — класс модели, к которой принадлежит запись, что добавляется в состав связанных или удаляется оттуда;
  - pk_set — множество ключей записей, что добавляются в состав связанных или удаляются оттуда. Для действий "pre_clear" и "post_clear" всегда None.

В качестве примера рассмотрим модели Machine и Spare, чей код приведен в листинге 4.2. При выполнении операций:

```
m = Machine.objects.create(name='Самосвал')
s = Spare.objects.create(name='Болт')
m.spares.add(s)
```

обработчик сигнала m2m_changed с параметром sender получит ссылку на класс промежуточной модели, неявно созданной самим фреймворком, с параметром instance — запись m (т. к. действия по изменению состава связанных записей

выполняются в ней), с параметром `action` — строку `"pre_add"`, с параметром `reverse` — `False` (действия по изменению состава связанных записей выполняются в записи ведущей модели), с параметром `model` — запись `s`, а с параметром `pk_set` — множество из единственного элемента — ключа записи `s`. Впоследствии тот же самый сигнал будет отправлен еще раз, и его обработчик получит с параметрами те же значения, за исключением параметра `action`, который будет иметь значение `"post_add"`.

А после выполнения действия:

```
s.machine_set.remove(m)
```

обработчик сигнала `m2m_changed` с параметром `sender` получит ссылку на класс промежуточной модели, с параметром `instance` — запись `s`, с параметром `action` — строку `"pre_remove"`, с параметром `reverse` — `True` (поскольку теперь действия по изменению состава связанных записей выполняются в записи ведомой модели), с параметром `model` — запись `m`, а с параметром `pk_set` — множество из единственного элемента — ключа записи `m`. Далее тот ж самый сигнал будет отправлен еще раз, и его обработчик получит с параметрами те же значения, за исключением параметра `action`, который будет иметь значение `"post_remove"`.

Сигналы, отправляемые подсистемой обработки запросов и объявленные в модуле `django.core.signals`:

- `request_started` — отправляется в самом начале обработки запроса. Обработчик получит параметры:
  - `sender` — класс `django.core.handlers.wsgi.WsgiHandler`, обрабатывающий все полученные запросы;
  - `environ` — словарь, содержащий переменные окружения;
- `request_finished` — отправляется после пересылки ответа клиенту. Обработчик с параметром `sender` получит класс `django.core.handlers.wsgi.WsgiHandler`, обрабатывающий все полученные запросы;
- `got_request_exception` — отправляется при возбуждении исключения в процессе обработки запроса. Вот параметры, передаваемые обработчику:
  - `sender` — класс `django.core.handlers.wsgi.WsgiHandler`, обрабатывающий все полученные запросы;
  - `request` — сам запрос, представленный экземпляром класса `HttpRequest`.

Сигналы, отправляемые подсистемой разграничения доступа и объявленные в модуле `django.contrib.auth.signals`:

- `user_logged_in` — отправляется сразу после удачно выполненного пользователем входа на сайт. Параметры, передаваемые обработчику:
  - `sender` — класс модели пользователя (обычно `User`);
  - `request` — текущий запрос, представленный экземпляром класса `HttpRequest`;

- `user` — запись модели пользователя, представляющая пользователя, который вошел на сайт;

- `user_logged_out` — отправляется сразу после удачно выполненного пользователем выхода с сайта. Вот параметры, которые получит обработчик:
  - `sender` — класс модели пользователя (обычно `User`) или `None`, если пользователь ранее не выполнил вход на сайт;
  - `request` — текущий запрос, представленный экземпляром класса `HttpRequest`;
  - `user` — запись модели пользователя, представляющая пользователя, который вышел с сайта, или `None`, если пользователь ранее не выполнил вход на сайт;

- `user_login_failed` — отправляется, если посетитель не смог войти на сайт. Параметры, передаваемые обработчику:
  - `sender` — строка с именем модуля, выполнявшего аутентификацию;
  - `credentials` — словарь со сведениями, занесенными посетителем в форму входа и переданными впоследствии функции `authenticate()`. Вместо пароля будет подставлена последовательность звездочек;
  - `request` — текущий запрос, представленный экземпляром класса `HttpRequest`, если таковой был передан функции `authenticate()`, в противном случае `None`.

На заметку

Некоторые специфические сигналы, используемые внутренними механизмами Django или подсистемами, не рассматриваемыми в этой книге, здесь не описаны. Их описание можно найти на странице https://docs.djangoproject.com/en/2.1/ref/signals/.

23.3. Объявление своих сигналов

Для своих собственных нужд мы можем объявить дополнительные сигналы. Делается это очень просто.

Сначала нужно объявить сигнал, создав экземпляр класса `Signal` из модуля `django.dispatch`. Конструктор этого класса вызывается согласно формату:

```
Signal(providing_args=<список имен параметров, передаваемых обработчику>)
```

Имена параметров в передаваемом *списке* должны быть представлены в виде строк.

Пример объявления сигнала `add_bb`, который будет передавать обработчику параметры `instance` и `rubric`:

```
from django.dispatch import Signal
add_bb = Signal(providing_args=['instance', 'rubric'])
```

Теперь необходимо в нужном месте кода инициировать отправку объявленного сигнала. Для этого применяются два следующих метода класса `Signal`:

- `send(<отправитель>[, <именованные параметры, указанные при объявлении сигнала>])` — выполняет отправку текущего сигнала от имени указанного *отправите-*

ля, возможно, с *именованными параметрами*, которые были указаны при объявлении сигнала и будут отправлены прямиком его обработчику.

В качестве результата метод возвращает список, каждый из элементов которого представляет один из привязанных к текущему сигналу обработчиков. Элемент этого списка представляет собой кортеж из двух элементов: ссылки на обработчик и возвращенный им результат. Если обработчик не возвращает результата, вторым элементом станет значение `None`.

Пример:

```
add_bb.send(Bb, instance=bb, rubric=bb.rubric)
```

Если к сигналу привязано несколько обработчиков, и в одном из них было возбуждено исключение, последующие обработчики выполнены не будут;

- `send_robust(<отправитель>[, <именованные параметры, указанные при объявлении сигнала>])` — то же самое, что `send()`, но обрабатывает все исключения, что могут быть возбуждены в обработчиках. Объекты исключений будут присутствовать в результате, возвращенном методом, во вторых элементах соответствующих вложенных кортежей.

  Поскольку исключения обрабатываются внутри метода, то, если к сигналу привязано несколько обработчиков, и в одном из них было возбуждено исключение, последующие обработчики все же будут выполнены.

Объявленный нами сигнал может быть обработан точно так же, как и любой из встроенных в Django:

```
def add_bb_dispatcher(sender, **kwargs):
    print('Объявление в рубрике "%s" с ценой %.2f создано' % \
          (kwargs['rubric'].name, kwargs['instance'].price))

add_bb.connect(add_bb_dispatcher)
```

ГЛАВА 24

Отправка электронных писем

В настоящее время очень многие сайты отправляют зарегистрированным на них пользователям электронные письма: уведомления о полученных письмах, сообщениях, комментариях, изменении в статусе купленного товара и др. Неудивительно, что Django предоставляет соответствующие средства, описанию которых отведена вся текущая глава.

24.1. Настройка подсистемы отправки электронных писем

На работу подсистемы рассылки электронных писем влияет весьма много настроек. Все они записываются в модуле settings.py пакета конфигурации:

- `EMAIL_BACKEND` — строка с именем класса, который реализует отправку писем. В составе Django поставляются следующие классы-отправители писем:
  - `django.core.mail.backends.smtp.EmailBackend` — отправляет письма на почтовый сервер по протоколу SMTP. Может использоваться как при разработке сайта, так и при его эксплуатации.

    Следующие классы используются исключительно при разработке и отладке сайта:
  - `django.core.mail.backends.filebased.EmailBackend` — сохраняет письма в файлах;
  - `django.core.mail.backends.console.EmailBackend` — выводит письма в командной строке;
  - `django.core.mail.backends.locmem.EmailBackend` — сохраняет письма в оперативной памяти. В модуле `django.core.mail` создается переменная `outbox`, и все отправленные письма записываются в нее в виде списка;
  - `django.core.mail.backends.dummy.EmailBackend` — никуда не отправляет, нигде не сохраняет и не выводит письма.

  Значение параметра по умолчанию: `"django.core.mail.backends.smtp.EmailBackend"`;

- DEFAULT_FROM_EMAIL — адрес электронной почты отправителя, по умолчанию указываемый в отправляемых письмах. Значение по умолчанию: "webmaster@localhost".

  Следующие параметры принимаются во внимание, только если в качестве класса-отправителя писем был выбран django.core.mail.backends.smtp.EmailBackend:

  - EMAIL_HOST — интернет-адрес SMTP-сервера, которому будут отправляться письма. Значение по умолчанию: "localhost";
  - EMAIL_PORT — номер TCP-порта, через который работает SMTP-сервер, в виде числа. Значение по умолчанию: 25;
  - EMAIL_HOST_USER — имя пользователя для аутентификации на SMTP-сервере. Значение по умолчанию — «пустая» строка.
  - EMAIL_HOST_PASSWORD — пароль для аутентификации на SMTP-сервере. Значение по умолчанию — «пустая» строка;

    Если значение хотя бы одного из параметров EMAIL_HOST_USER и EMAIL_HOST_PASSWORD равно «пустой» строке, аутентификация на SMTP-сервере выполняться не будет;
  - EMAIL_USE_SSL — если True, для взаимодействия с SMTP-сервером будет использоваться протокол SSL (Secure Sockets Layer, уровень защищенных сокетов), если False, таковой применяться не будет. Протокол SSL работает через TCP-порт 465. Значение по умолчанию — False;
  - EMAIL_USE_TLS — если True, для взаимодействия с SMTP-сервером будет использоваться протокол TLS (Transport Layer Security, протокол защиты транспортного уровня), если False, таковой применяться не будет. Протокол TLS работает через TCP-порт 587. Значение по умолчанию — False;

    Внимание!

    Можно указать значение True только для одного из параметров: EMAIL_USE_SSL или EMAIL_USE_TLS.
  - EMAIL_SSL_CERTFILE — строка с путем к файлу сертификата. Принимается во внимание, только если для одного из параметров: EMAIL_USE_SSL или EMAIL_USE_TLS — было указано значение True. Значение по умолчанию — None;
  - EMAIL_SSL_KEYFILE — строка с путем к файлу с закрытым ключом. Принимается во внимание, только если для одного из параметров: EMAIL_USE_SSL или EMAIL_USE_TLS — было указано значение True. Значение по умолчанию — None;
  - EMAIL_TIMEOUT — промежуток времени, в течение которого класс-отправитель будет пытаться установить соединение с SMTP-сервером, в виде целого числа в секундах. Если соединение установить не удастся, будет возбуждено исключение timeout из модуля socket. Если для параметра указать значение

None, будет использовано значение промежутка времени по умолчанию. Значение по умолчанию — None;

- EMAIL_USE_LOCALTIME — если True, в заголовках отправляемых писем будет указано локальное время, если False — всемирное координированное время (UTC). Значение по умолчанию — False;
- EMAIL_FILE_PATH — полный путь к папке, в которой будут сохраняться файлы с письмами. Значение по умолчанию — «пустая» строка. Этот параметр принимается во внимание, только если в качестве класса-отправителя писем был выбран django.core.mail.backends.filebased.EmailBackend.

24.2. Низкоуровневые инструменты для отправки писем

Инструменты низкого уровня для отправки электронных писем имеет смысл применять лишь в том случае, если нужно отправить письмо с вложениями.

24.2.1. Класс *EmailMessage*: обычное электронное письмо

Класс EmailMessage из модуля django.core.mail позволяет отправить обычное текстовое электронное письмо, возможно, включающее какие-либо вложения.

Конструктор этого класса принимает весьма большое количество параметров. Все они являются именованными и необязательными:

- subject — тема письма;
- body — тело письма;
- from_email — адрес отправителя в виде строки. Может быть также записан в формате *<имя отправителя> <<адрес электронной почты>>*, например: "Admin <admin@supersite.ru>". Если не указан, будет использован адрес из параметра DEFAULT_FROM_EMAIL настроек сайта;
- to — список или кортеж адресов получателей письма;
- cc — список или кортеж адресов получателей копии письма;
- bcc — список или кортеж адресов получателей скрытой копии письма;
- reply_to — список или кортеж адресов для отправки ответа на письмо;
- attachments — список вложений, которые нужно добавить в письмо. Каждое вложение может быть задано в виде:
  - экземпляра класса MIMEBase из модуля email.mime.base Python или одного из его подклассов;
  - кортежа из трех элементов: строки с именем файла, строки или объекта bytes с содержимым файла и строки с MIME-типом файла;

- `headers` — словарь с дополнительными заголовками, которые нужно добавить в письмо. Ключи элементов этого словаря задают имена заголовков, а значения элементов — значения заголовков;
- `connection` — объект соединения, используемого для отправки письма (его использование будет описано позже). Если не указан, для отправки письма будет установлено отдельное соединение.

Для работы с письмами класс `EmailMessage` предоставляет ряд методов:

- `attach()` — добавляет вложение к письму. Форматы вызова метода:

  `attach(`*<объект вложения>*`)`
  `attach(`*<имя файла>*`, `*<содержимое файла>*`[, `*<MIME-тип содержимого>*`])`

  Первый формат принимает в качестве параметра *объект вложения*, представленный экземпляром класса `MIMEBase` или одного из его подклассов.

  Второй формат принимает *имя файла*, который будет сформирован в письме в качестве вложения, *содержимое* этого файла в виде строки или объекта `bytes` и строку с *MIME-типом содержимого* файла, которую можно не указывать. Если MIME-тип не указан, Django определит его по расширению из *имени файла*;

- `attach_file(`*<путь к файлу>*`[, `*<MIME-тип файла>*`])` — добавляет файл к письму в качестве вложения. Если *MIME-тип* не указан, Django определит его по расширению файла;
- `send([fail_silently=False])` — выполняет отправку письма. Если параметру `fail_silently` дать значение `True`, в случае возникновения нештатной ситуации при отправке письма никаких исключений возбуждено не будет (по умолчанию в таких случаях возбуждается исключение `SMTPException` из модуля `smtplib`);
- `message()` — возвращает экземпляр класса `SafeMIMEText`, объявленного в модуле `django.core.mail`, представляющего текущее письмо. Этот класс является производным от класса `MIMEBase`, следовательно, может быть указан в списке вложений, задаваемых параметром `attachments` конструктора класса или вызове метода `attach()`;
- `recipients()` — возвращает список адресов всех получателей письма и его копий, которые указаны в параметрах `to`, `cc` и `bcc` конструктора класса.

Вот примеры, демонстрирующие наиболее часто встречающиеся на практике случаи отправки электронных писем:

```
from django.core.mail import EmailMessage
# Отправка обычного письма
em = EmailMessage(subject='Test', body='Test', to=['user@supersite.ru'])
em.send()
# Отправка письма с вложением, заданным через параметр attachments
# конструктора. Отметим, что файл password.txt в реальности не существует
# и формируется в письме программно
em = EmailMessage(subject='Ваш новый пароль',
          body='Ваш новый пароль находится во вложении',
```

```
        attachments=[('password.txt', '123456789', 'text/plain')],
        to=['user@supersite.ru'])
em.send()
# Отправка письма с вложением, созданным на основе файла, который
# хранится на локальном диске
em = EmailMessage(subject='Запрошенный вами файл',
                  body='Получите запрошенный вами файл',
                  to=['user@supersite.ru'])
em.attach_file(r'C:\work\file.txt')
em.send()
```

24.2.2. Формирование писем на основе шаблонов

Мы можем формировать предназначенные для отправки электронные письма на основе обычных шаблонов Django (см. *главу 11*). Для этого могут быть использованы уже знакомые нам инструменты фреймворка, в частности, функция `render_to_string()` из модуля `django.template.loader`.

Вот пример отправки электронного письма, которое формируется на основе шаблона email\letter.txt:

```
from django.core.mail import EmailMessage
from django.template.loader import render_to_string
context = {'user': 'Вася Пупкин'}
s = render_to_string('email/letter.txt', context)
em = EmailMessage(subject='Оповещение', body=s,
                                        to=['vpupkin@othersite.ru'])
em.send()
```

Код шаблона email\letter.txt может выглядеть так:

```
Уважаемый {{ user }}, вам пришло сообщение!
```

24.2.3. Использование соединений. Массовая рассылка писем

При отправке каждого письма, представленного экземпляром класса `EmailMessage`, каждый раз создается соединение с SMTP-сервером. Установление этого соединения отнимает заметное время, которое может достигать нескольких секунд. И, если отправляется только одно письмо, с такой задержкой еще можно смириться, но при отправке большого количества писем — при массовой рассылке — это совершенно неприемлемо.

Параметр `connection` конструктора класса `EmailMessage` позволяет указать объект соединения, которое будет использоваться для отправки текущего письма. Мы можем использовать одно и то же соединение для отправки произвольного количества писем, тем самым устранив задержку на установление соединения при отправке каждого письма.

Получить соединение мы можем вызовом функции `get_connection()` из модуля `django.core.mail`. Функция в качестве результата вернет то, что нам нужно, — объект соединения.

Для открытия соединения мы вызовем у полученного объекта метод `open()`, а для закрытия — метод `close()`.

Вот пример отправки трех писем с применением одного и того же соединения:

```
from django.core.mail import EmailMessage, get_connection
con = get_connection()
con.open()
email1 = EmailMessage( . . . connection=con)
email1.send()
email2 = EmailMessage( . . . connection=con)
email2.send()
email3 = EmailMessage( . . . connection=con)
email3.send()
con.close()
```

Недостатком здесь может стать то, что соединение придется указывать при создании каждого сообщения — в параметре `connection` конструктора класса `EmailMessage`. Но существует удобная альтернатива — использование метода `send_messages(`*<отправляемые письма>*`)` объекта соединения. В его вызове *отправляемые письма* задаются в виде списка из экземпляров класса `EmailMessage`. Пример:

```
from django.core.mail import EmailMessage, get_connection
con = get_connection()
con.open()
email1 = EmailMessage( . . . )
email2 = EmailMessage( . . . )
email3 = EmailMessage( . . . )
con.send_messages([email1, email2, email3])
con.close()
```

24.2.4. Класс *EmailMultiAlternatives*: электронное письмо, состоящее из нескольких частей

Класс `EmailMultiAlternatives` из того же модуля `django.core.mail` представляет письмо, которое состоит из нескольких частей, записанных в разных форматах. Обычное такое письмо содержит основную часть, представляющую собой обычный текст, и вторую часть, которая написана на языке HTML.

Класс `EmailMultiAlternatives` является производным от класса `EmailMessage`, имеет тот же формат вызова конструктора и поддерживает те же методы. Он добавляет лишь поддержку метода `attach_alternative(`*<содержимое части>*`, `*<MIME-тип части>*`)`, который добавляет к письму новую часть с указанным в виде строки *содержимым*.

Вот пример отправки письма, которое, помимо текстовой части, содержит еще и часть, написанную на HTML:

```
from django.core.mail import EmailMultiAlternatives
em = EmailMultiAlternatives(subject='Test', body='Test',
                                                to=['user@supersite.ru'])
em.attach_alternative('<h1>Test</h1>', 'text/html')
em.send()
```

Разумеется, для формирования дополнительных частей в таких письмах можно использовать шаблоны.

24.3. Высокоуровневые инструменты для отправки писем

Если нет необходимости создавать письма с вложениями, можно воспользоваться высокоуровневыми средствами отправки писем.

24.3.1. Отправка писем по произвольным адресам

Начнем мы со средств, предназначенных для отправки писем по произвольным адресам. Эти средства реализуются следующими двумя функциями, объявленными в модуле `django.core.mail`:

- `send_mail()` — отправляет одно письмо, возможно, включающее HTML-часть, по указанным адресам. Формат вызова функции:

```
send_mail(<тема>, <тело>, <адрес отправителя>, <адреса получателей>[,
          fail_silently=False][, auth_user=None][,
          auth_password=None][, connection=None][, html_message=None])
```

Адрес отправителя указывается в виде строки. Его также можно записать в формате *<имя отправителя> <<адрес электронной почты>>*, например: "Admin <admin@supersite.ru>". *Адреса получателей* указываются в виде списка или кортежа.

Если параметру `fail_silently` дать значение `True`, в случае возникновения при отправке письма нештатной ситуации никаких исключений возбуждено не будет. Значение `False` указывает возбудить в таких случаях исключение `SMTPException` из модуля `smtplib`.

Параметры `auth_user` и `auth_password` задают, соответственно, имя пользователя и пароль для подключения к SMTP-серверу. Если эти параметры не заданы, их значения будут взяты из параметров `EMAIL_HOST_USER` и `EMAIL_HOST_PASSWORD` настроек проекта (см. *разд. 24.1*).

Параметр `connection` указывает объект соединения, применяемого для отправки письма. Если он не задан, для отправки письма будет установлено отдельное соединение.

Параметр `html_message` задает строку с HTML-кодом второй части. Если он отсутствует, письмо будет содержать всего одну часть — текстовую.

Функция возвращает 1 в случае успешной отправки письма и 0, если письмо по какой-то причине отправить не удалось.

Пример:

```
from django.core.mail import send_mail
send_mail('Test', 'Test!!!', 'webmaster@supersite.ru',
          ['user@othersite.ru'], html_message='<h1>Test!!!</h1>')
```

- `send_mass_mail()` — выполняет отправку писем из указанного *перечня*. Формат вызова:

```
send_mass_mail(<перечень писем>[, fail_silently=False][,
               auth_user=None][, auth_password=None][,
               connection=None])
```

Перечень писем должен представлять собой кортеж, каждый элемент которого описывает одно из отправляемых писем и также представляет собой кортеж из четырех элементов: темы письма, тела письма, адреса отправителя и списка или кортежа адресов получателей.

Назначение остальных параметров этого метода было описано ранее, в описании метода `send_mail()`.

Для отправки всех писем из *перечня* используется всего одно соединение с SMTP-сервером.

В качестве результата метод `send_mass_mail()` возвращает количество успешно отправленных писем.

Пример:

```
from django.core.mail import send_mass_mail
msg1 = ('Подписка', 'Подтвердите, пожалуйста, подписку',
        'subscribe@supersite.ru',
        ['user@othersite.ru', 'user2@thirdsite.ru'])
msg2 = ('Подписка', 'Ваша подписка подтверждена',
        'subscribe@supersite.ru', ['megauser@megasite.ru'])
send_mass_mail((msg1, msg2))
```

24.3.2. Отправка писем зарегистрированным пользователям

Мы можем отправить письмо любому зарегистрированному на сайте пользователю. Для этого класс `User`, реализующий модель пользователя, предлагает метод `email_user()`:

```
email_user(<тема>, <тело>[, from_email=None][,
                          <дополнительные параметры>])
```

Параметр `from_email` указывает адрес отправителя. Если он опущен, адрес отправителя будет взят из параметра `DEFAULT_FROM_EMAIL` настроек проекта.

Все *дополнительные параметры*, указанные в вызове метода, будут без изменений переданы функции `send_mail()` (см. *разд. 24.3.1*), используемой методом для выполнения отправки.

Пример:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(username='admin')
>>> user.email_user('Подъем!', 'Admin, не спи!', fail_silently=True)
```

В этом примере параметр `fail_silently` будет передан функции `send_mail()`, лежащей в основе метода `email_user()`.

24.3.3. Отправка писем администраторам и редакторам сайта

Еще Django поддерживает рассылку писем по адресам, указанным в двух параметрах настроек сайта. Один из этих параметров задает список адресов администраторов, получающих сообщения об ошибках в программном коде сайта, а второй — список адресов редакторов, которым будут отсылаться уведомления об отсутствующих страницах. Обычно инструменты, осуществляющие отправку писем по этим адресам, применяются в подсистеме журналирования (речь о которой пойдет в *главе 28*).

Рассмотрение этих инструментов мы начнем с параметров, записываемых в модуле settings.py пакета конфигурации, — т. е. там, где задаются все настройки проекта:

- `ADMINS` — список администраторов. Каждый элемент этого списка обозначает одного из администраторов и должен представлять собой кортеж из двух элементов: имени администратора и его адреса электронной почты. Пример:

```
ADMIN = [
    ('Admin1', 'admin1@supersite.ru'),
    ('Admin2', 'admin2@othersite.ru'),
    ('MegaAdmin', 'megaadmin@megasite.ru')
]
```

  Значение параметра по умолчанию — «пустой» список;

- `MANAGERS` — список редакторов. Задается в том же формате, что и список администраторов из параметра `ADMIN`. Значение по умолчанию — «пустой» список;

- `SERVER_EMAIL` — адрес электронной почты отправителя, указываемый в письмах, что отправляются администраторам и редакторам. Значение по умолчанию: `"root@localhost"`;

- `EMAIL_SUBJECT_PREFIX` — префикс, который добавляется к теме каждого письма, отправляемого администраторам и редакторам. Значение по умолчанию: `"[Django]"`.

Для отправки писем администраторам применяется функция `mail_admins()`, для отправки писем редакторам — функция `mail_managers()`. Обе функции объявлены в модуле `django.core.mail` и имеют одинаковый формат вызова:

```
mail_admins|mail_managers(<тема>, <тело>[, fail_silently=False][,
                                  connection=None][, html_message=None])
```

Если параметру `fail_silently` дать значение `True`, то в случае возникновения при отправке письма нештатной ситуации никаких исключений возбуждено не будет. Значение `False` указывает возбудить в таких случаях исключение `SMTPException` из модуля `smtplib`.

Параметр `connection` указывает объект соединения, применяемого для отправки письма. Если он не задан, для отправки письма будет установлено отдельное соединение.

Параметр `html_message` задает строку с HTML-кодом второй части. Если он отсутствует, письмо будет содержать всего одну часть — текстовую.

Пример:

```
from django.core.mail import mail_managers
mail_managers('Подъем!', 'Редакторы, не спите!',
              html_message='<strong>Редакторы, не спите!</strong>')
```

ГЛАВА 25

Кэширование

Когда веб-обозреватель получает от веб-сервера какой-либо файл, он сохраняет его на локальном диске — выполняет его *кэширование*. Впоследствии, если этот файл не изменился, веб-обозреватель использует его кэшированную копию вместо того, чтобы вновь загружать с сервера, — это заметно увеличивает производительность. Так работает *кэширование на стороне клиента*.

Django предоставляет ряд инструментов для управления процессом кэширования на стороне клиента. Мы можем отправлять в заголовке запроса дату и время последнего изменения страницы или какой-либо признак, указывающий, изменилась ли страница с момента последнего доступа к ней. Также мы можем отключать кэширование на стороне клиента для каких-либо страниц, если хотим, чтобы они всегда отображали актуальную информацию.

Помимо этого, Django может выполнять *кэширование на стороне сервера*, сохраняя какие-либо данные, фрагменты страниц или даже целые страницы в особом хранилище — *кэше сервера*. Благодаря этому можно увеличить производительность работы сайтов с высокой нагрузкой.

25.1. Кэширование на стороне сервера

Знакомство со средствами кэширования, встроенными в Django, мы начнем с теми из них, что реализуют кэширование на стороне сервера.

Кэш стороны сервера в Django организован по принципу словаря Python. Любая кэшируемая величина сохраняется в отдельном элементе, имеющем ключ, который однозначно идентифицирует кэшированное значение. Ключ этот может как генерироваться Django на основе каких-либо сведений (например, интернет-адреса страницы), так и задаваться произвольно.

25.1.1. Подготовка подсистемы кэширования на стороне сервера

Прежде чем задействовать кэширование на стороне сервера, необходимо выполнить подготовительные действия.

25.1.1.1. Настройка подсистемы кэширования на стороне сервера

Все настройки этой подсистемы указываются в параметре CACHES модуля settings.py пакета конфигурации.

Значением этого параметра должен быть словарь. Ключи его элементов указывают псевдонимы кэшей, созданных в Django-сайте. Можно указать произвольное количество кэшей, принадлежащих к разным типам. Кэш с псевдонимом `default` будет использоваться, если при выполнении операций с кэшем таковой не был указан явно.

В качестве значений элементов словаря также указываются словари, хранящие, собственно, параметры соответствующего кэша. Каждый элемент вложенного словаря указывает отдельный параметр.

Вот значение параметра CACHES по умолчанию:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
    }
}
```

Оно задает единственный кэш, используемый по умолчанию и сохраняющий данные в оперативной памяти.

Теперь рассмотрим доступные для указания параметры кэша:

- BACKEND — строка с именем класса, выполняющего сохранение кэшируемых данных. В составе Django поставляются следующие классы:
  - `django.core.cache.backends.db.DatabaseCache` — сохраняет данные в указанной таблице базы данных. Обеспечивает среднюю производительность и высокую надежность, но дополнительно нагружает базу данных;
  - `django.core.cache.backends.filebased.FileBasedCache` — сохраняет данные в файлах, находящихся в заданной папке. Обеспечивает немного более низкую производительность, нежели предыдущий класс, но не обращается к базе данных и может быть использован в случаях, когда необходимо уменьшить нагрузку на базу;
  - `django.core.cache.backends.locmem.LocMemCache` — сохраняет данные в оперативной памяти. Дает наивысшую производительность, но при отключении компьютера содержимое кэша будет потеряно;
  - `django.core.cache.backends.memcached.MemcachedCache` и `django.core.cache.backends.memcached.PyLibMCCache` — используют для хранения данных популярную программу Memcached.

> ***На заметку***
>
> Применение Memcached для кэширования данных не описывается в этой книге. Документацию по самой программе можно найти на сайте **https://memcached.org/**,

а инструкции по настройке Django-проекта для работы с ней — на странице **https://docs.djangoproject.com/en/2.1/topics/cache/#memcached**.

- `django.core.cache.backends.dummy.DummyCache` — вообще не сохраняет кэшируемые данные. Используется исключительно при отладке;

- `LOCATION` — назначение параметра зависит от выбранного класса, сохраняющего данные:
  - имя таблицы — если выбран класс, хранящий кэш в таблице базы данных;
  - полный путь к папке — если выбран класс, хранящий данные в файлах по указанному пути;
  - псевдоним хранилища — если выбран класс, хранящий данные в оперативной памяти. Указывается только в том случае, если используются несколько кэшей такого типа. Пример:

```
CACHES = {
    'default': {
        'BACKEND':
            'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'cache1',
    }
    'special': {
        'BACKEND':
            'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'cache2',
    }
}
```

  Значение параметра по умолчанию — «пустая» строка;

- `TIMEOUT` — время, в течение которого значение, помещенное в кэш, будет считаться актуальным, в виде целого числа в секундах. Устаревшие значения впоследствии будут удалены. Значение параметра по умолчанию: `300`;

- `OPTIONS` — дополнительные параметры кэша. Значение указывается в виде словаря, каждый элемент которого задает отдельный параметр. Поддерживаются два универсальных параметра, обрабатываемые всеми классами-хранилищами данных:
  - `MAX_ENTRIES` — количество значений, которые могут храниться в кэше, в виде целого числа. Значение по умолчанию: `300`;
  - `CULL_FREQUENCY` — часть кэша, которая будет очищена, если количество значений в кэше превысит величину, заданную параметром `MAX_ENTRIES`, указанная в виде целого числа. Так, если дать параметру `CULL_FREQUENCY` значение `2`, то при заполнении кэша будет удалена половина хранящихся в нем значений. Если задать значение `0`, при заполнении кэша из него будут удалены все значения. Значение по умолчанию: `3`.

Помимо этого, можно задать значения, специфичные для отдельных классов, которые реализуют сохранение данных. Такие параметры характерны для классов

из сторонних библиотек. Классы, поставляемые в составе Django, не поддерживают дополнительных параметров;

- `KEY_PREFIX` — используемый по умолчанию префикс, который участвует в формировании конечного ключа, записываемого в кэше. Значение по умолчанию — «пустая» строка;
- `VERSION` — используемая по умолчанию версия кэша, которая участвует в формировании конечного ключа, записываемого в кэше. Значение по умолчанию: `1`;
- `KEY_FUNCTION` — строка с именем функции, которая формирует конечный ключ, записываемый в кэше, из префикса, номера версии и ключа кэшируемого значения. По умолчанию используется функция, которая формирует конечный ключ из префикса, номера версии и ключа, разделяя их символами двоеточия, и имеет следующий вид:

```
def make_key(key, key_prefix, version):
    return ':'.join([key_prefix, str(version), key])
```

Пример указания параметров кэша:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'cache_table',
        'TIMEOUT': 120,
        'OPTIONS': {
            'MAX_ENTRIES': 200,
        }
    }
}
```

25.1.1.2. Создание таблицы для хранения кэша

Если был выбран класс, сохраняющий кэшированные данные в таблице базы данных, эту таблицу необходимо создать. В этом нам поможет команда `createcachetable` утилиты manage.py:

```
manage.py createcachetable [--dry-run]
```

Дополнительный ключ `--dry-run` выводит на экран сведения о создаваемой таблице, но не создает ее.

25.1.2. Высокоуровневые средства кэширования

Высокоуровневые средства выполняют кэширование либо всего сайта (всех его страниц), либо страниц, сгенерированных отдельными контроллерами.

Как мы уже знаем, кэш в Django организован в виде словаря: каждое кэшированное значение сохраняется в отдельном элементе с ключом, однозначно идентифицирующим это значение. В случае кэшированной страницы этот ключ формируется

самим Django на основе интернет-адреса страницы (точнее, пути и набора GET-параметров).

25.1.2.1. Кэширование всего веб-сайта

Проще всего реализовать кэширование всего сайта. При этом подсистема кэширования Django работает согласно следующим принципам:

- кэшируются все страницы, сгенерированные контроллерами в ответ на получение GET- и HEAD-запросов, с кодом статуса 200 (т. е. запрос был обработан успешно);
- страницы с одинаковыми интернет-адресами, но разным набором GET-параметров, считаются разными, и, соответственно, для каждой из них в кэше создается отдельная копия.

Чтобы запустить кэширование всего сайта, прежде всего необходимо добавить посредники `django.middleware.cache.UpdateCacheMiddleware` и `django.middleware.cache.FetchFromCacheMiddleware` в список зарегистрированных в проекте (параметр `MIDDLEWARE`):

```
MIDDLEWARE = [
    . . .
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    . . .
]
```

Еще можно указать дополнительные параметры, влияющие на кэширование всего сайта:

- `CACHE_MIDDLEWARE_ALIAS` — псевдоним кэша, в котором будут сохраняться страницы. Значение по умолчанию: `"default"`;
- `CACHE_MIDDLEWARE_SECONDS` — время, в течение которого страница, помещенная в кэш, будет считаться актуальной, в виде целого числа в секундах. Значение параметра по умолчанию: `600`;
- `CACHE_MIDDLEWARE_KEY_PREFIX` — префикс конечного ключа, применяемый только при кэшировании всего сайта. Значение по умолчанию — «пустая» строка.

В ответы, отсылаемые клиентам и содержащие страницы, добавляется два следующих заголовка:

- `Expires` — в качестве значения задается дата и время устаревания кэшированной страницы, полученное сложением текущих даты и времени и значения из параметра `CACHE_MIDDLEWARE_SECONDS`;
- `Cache-Control` — добавляется параметр `max-age` со значением, взятым из параметра `CACHE_MIDDLEWARE_SECONDS`.

Во многих случаях кэширование всего сайта — наилучший вариант повышения его производительности. Такой вариант кэширования быстро реализуется, не требует ни сложного программирования, ни переписывания кода контроллеров, чьи ответы нужно кэшировать.

Однако у этого варианта кэширования есть огромный недостаток: он не учитывает, что какая-либо страница может формироваться в разных редакциях в зависимости от некоторых условий — например, выполнил пользователь вход на сайт или нет. В самом деле, если пользователь, не выполнив вход, запросит такую страницу, она будет сгенерирована и сохранена в кэше. Далее, когда после выполнения входа пользователь снова запросит эту страницу, он получит вариант, взятый из кэша и выглядящий так, как будто пользователь и не входил на сайт.

25.1.2.2. Кэширование на уровне отдельных контроллеров

Избежать проблемы, описанной в конце *разд. 25.1.2.1*, можно применением другого варианта кэширования — на уровне не всего сайта, а отдельных контроллеров. Точнее, тех контроллеров, что выдают страницы, не меняющиеся в зависимости от каких бы то ни было «внутренних» условий.

Для реализации кэширования на уровне контроллера применяется декоратор `cache_page()` из модуля `django.views.decorators.cache`:

```
cache_page(<время хранения страницы>[, cache=None][, key_prefix=None])
```

Время хранения сгенерированной страницы в кэше задается в виде целого числа в секундах.

Необязательный параметр `cache` указывает псевдоним кэша, в котором будет сохранена страница. Если он не указан, будет использован кэш по умолчанию с псевдонимом `default`.

В необязательном параметре `key_prefix` можно указать другой префикс конечного ключа. Если он не указан, применяется префикс из параметра `KEY_PREFIX` настроек текущего кэша.

Декоратор `cache_page()` указывается непосредственно у контроллера-функции, результаты работы которого необходимо кэшировать:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 5)
def by_rubric(request, pk):
    . . .
```

Также этот декоратор можно указать в объявлении маршрута, ведущего на нужный контроллер. Это можно использовать для кэширования страниц, генерируемых контроллерами-классами. Пример:

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    . . .
```

```
    path('<int:rubric_id>/', cache_page(60 * 5)(by_rubric)),
    path('', cache_page(60 * 5)(BbIndexView.as_view())),
]
```

Выбрав кэширование на уровне контроллеров, мы сможем указать, какие страницы следует кэшировать, а какие — нет. Понятно, что в первую очередь кэшировать стоит страницы, которые просматриваются большей частью посетителей и не изменяются в зависимости от каких-либо «внутренних» условий. Страницы же, меняющиеся в зависимости от таких условий, лучше не кэшировать (или, как вариант, применять средства управления кэшированием, что будут рассмотрены прямо сейчас).

25.1.2.3. Управление кэшированием

Давайте вспомним принципы, согласно которым работают высокоуровневые средства Django для кэширования страниц (они были приведены в *разд. 25.1.2.1*). Для каждого из интернет-адресов, что получает от клиентов Django-сайт, в кэше создается отдельная страница, которая потом и отправляется клиентам. При этом интернет-адреса с одинаковым путем, но разным набором GET-параметров считаются разными, и, соответственно, для них создаются отдельные кэшированные страницы.

Этот подход работает только в тех случаях, когда генерируемые сайтом страницы не меняются в зависимости от каких-либо «внутренних» условий (например, признака, выполнил ли пользователь вход на сайт). Это и понятно — ведь кэшированные страницы с точки зрения подсистемы кэширования отличаются лишь их интернет-адресами. А у разных редакций одной страницы он всегда один и тот же.

Если же мы хотим создавать в кэше по отдельной редакции кэшированной страницы для разных значений «внутреннего» признака, нам нужно как-то указать этот признак подсистеме кэширования. Сделать это можно посредством декоратора `vary_on_headers()` из модуля `django.views.decorators.vary`:

```
vary_on_headers(<заголовок 1>, <заголовок 2> . . . <заголовок n>)
```

В качестве признаков, на основе значений которых в кэше будут создаваться отдельные редакции страницы, здесь указываются *заголовки* запроса, записанные в виде строк. Можно указать произвольное количество *заголовков*.

Вот как можно задать создание в кэше отдельных копий одной и той же страницы для разных значений заголовка `User-Agent` (он обозначает программу веб-обозревателя, используемую клиентом):

```
from django.views.decorators.vary import vary_on_headers

@vary_on_headers('User-Agent')
def index(request):
    . . .
```

Теперь при запросе одной и той же страницы разными веб-обозревателями в кэше будет создаваться отдельная копия страницы на каждый веб-обозреватель, что ее запрашивал.

Если нам нужно создавать отдельные копии страницы для гостя и пользователя, выполнившего вход, мы можем воспользоваться тем, что при создании сессии (в которой сохраняется ключ вошедшего на сайт пользователя) веб-обозревателю посылается cookie, который хранит идентификатор сессии. А при отправке запроса на сервер все сохраненные для домена, по которому отправляется запрос, cookie посылаются серверу в заголовке `Cookie`. В таком случае мы можем записать код:

```
@vary_on_headers('Cookie')
def user_profile(request):
    . . .
```

Мы можем указать в декораторе `vary_on_headers()` несколько заголовков:

```
@vary_on_headers('User-Agent', 'Cookie')
def user_account(request):
    . . .
```

Поскольку создание в кэше разных копий страницы для разных значений заголовка `Cookie` является распространенной практикой, Django предоставляет декоратор `vary_on_cookie()` из того же модуля `django.views.decorators.vary`. Пример:

```
from django.views.decorators.vary import vary_on_cookie

@vary_on_cookie
def user_profile(request):
    . . .
```

25.1.3. Низкоуровневые средства кэширования

Низкоуровневые средства кэширования можно применять для сохранения в кэше фрагментов страниц или произвольных значений. С их помощью кэшируют сложные части страниц и значения, получение которых сопряжено с интенсивными вычислениями.

При кэшировании на низком уровне ключ, под которым должна сохраняться кэшируемая величина, указывается самим разработчиком. Этот ключ объединяется с префиксом и номером версии для получения *конечного ключа*, под которым значение и будет сохранено в кэше.

25.1.3.1. Кэширование фрагментов веб-страниц

Для кэширования фрагментов страниц нам понадобится библиотека тегов с псевдонимом `cache`:

```
{% load cache %}
```

Собственно кэширование выполняет парный тег `cache . . . endcache`:

```
cache <время хранения фрагмента> <ключ фрагмента> [<набор параметров>] ↪
[using="template_fragments"]
```

Время хранения фрагмента в кэше указывается в секундах. Если задать значение `None`, фрагмент будет храниться вечно (пока не будет явно удален).

Ключ фрагмента используется для формирования конечного ключа, под которым фрагмент будет сохранен в кэше. *Ключ* должен быть уникальным в пределах всех шаблонов текущего проекта.

Набор параметров указывается в случаях, если требуется сохранять разные копии фрагмента в зависимости от выполнения каких-либо условий. В качестве таких условий выступают комбинации значений заданных в *наборе* параметров — для каждой их комбинации в кэше будет создана отдельная копия фрагмента.

Необязательный параметр `using` указывает псевдоним кэша, используемого для хранения фрагментов страниц. По умолчанию для этого будет применяться кэш с псевдонимом `template_fragments` или, если таковой отсутствует, кэш по умолчанию.

Вот так мы можем сохранить в кэше панель навигации нашего сайта на 300 секунд:

```
{% cache 300 navbar %}
<nav>
    <a href="{% url 'bboard:index' %}">Главная</a>
    {% for rubric in rubrics %}
    <a href="{% url 'bboard:by_rubric' rubric_id=rubric.pk %}">
    {{ rubric }}</a>
    {% endfor %}
</nav>
{% endcache %}
```

Если нам нужно хранить в кэше две копии фрагмента, одна из которых должна выдаваться гостям, а другая — пользователям, выполнившим вход, мы используем код такого вида:

```
{% cache 300 navbar request.user.is_authenticated %}
<nav>
    <a href="{% url 'bboard:index' %}">Главная</a>
    {% if user.is_authenticated %}
    <a href="{% url 'logout' %}">Выйти</a>
    {% else %}
    <a href="{% url 'login' %}">Войти</a>
    {% endif %}
</nav>
{% endcache %}
```

А если нужно хранить по отдельной копии фрагмента еще и для каждого из зарегистрированных пользователей сайта, мы напишем код:

```
{% cache 300 greeting request.user.is_authenticated ↳
request.user.username %}
{% if user.is_authenticated %}
<p>Добро пожаловать, {{ user.username }}!</p>
{% endif %}
{% endcache %}
```

25.1.3.2. Кэширование произвольных значений

Наконец, есть возможность сохранить в кэше какое-либо произвольное значение: строку, число, список, набор записей и т. п. Как уже говорилось ранее, таким образом кэшируются данные, получаемые в результате интенсивных вычислений.

Словарь со всеми кэшами, объявленными в настройках проекта, хранится в переменной `caches` из модуля `django.core.cache`. Ключи элементов этого словаря совпадают с псевдонимами кэшей, а значениями элементов являются сами кэши, представленные особыми объектами. Пример:

```
from django.core.cache import caches
default_cache = caches['default']
special_cache = caches['special']
```

Если кэша с указанным псевдонимом не существует, будет возбуждено исключение `InvalidCacheBackendError` из модуля `django.core.cache.backends.base`.

Переменная `cache`, также объявленная в модуле `django.core.cache`, хранит ссылку на объект кэша по умолчанию.

Любой объект кэша поддерживает следующие методы:

- `set(<ключ>, <значение>[, timeout=<время хранения>][, version=None])` — заносит в кэш *значение* под заданным *ключом*. Если заданный *ключ* уже существует в кэше, перезаписывает сохраненное под ним значение. Параметр `timeout` указывает время хранения значения в виде целого числа в секундах. Если он не задан, используется значение параметра `TIMEOUT` соответствующего кэша (см. *разд. 25.1.1.1*).

  В необязательном параметре `version` можно задать версию, на основе которой будет формироваться конечный ключ для сохраняемого в кэше значения. Если версия не задана, будет использовано значение параметра `VERSION` соответствующего кэша.

  Примеры:

  ```
  from django.core.cache import cache
  cache.set('rubrics', Rubric.objects.get())
  cache.set('rubrics_sorted', Rubric.objects.order_by('name'), 240)
  ```

  Имеется возможность сохранить произвольное количество значений под одним ключом, просто указав для этих значений разные номера версий. В некоторых случаях это может оказаться полезным. Пример:

  ```
  # Если номер версии не указан, для сохраняемого значения будет
  # установлен номер версии из параметра VERSION (по умолчанию — 1)
  cache('val', 10)
  cache('val', 100, version=2)
  cache('val', 1000, version=3)
  ```

- `add(<ключ>, <значение>[, timeout=<время хранения>][, version=None])` — то же самое, что `set()`, но сохраняет *значение* только в том случае, если указанный

ключ не существует в кэше. В качестве результата возвращает `True`, если значение было сохранено в кэше, и `False` — в противном случае;

- `get(<ключ>[, <значение по умолчанию>][, version=None])` — извлекает из кэша значение, ранее сохраненное под заданным *ключом*, и возвращает его в качестве результата. Если указанного *ключа* в кэше нет (например, значение еще не создано или уже удалено по причине устаревания), возвращает *значение по умолчанию* или `None`, если оно не указано. Параметр `version` указывает номер версии. Пример:

```
rubrics_sorted = cache.get('rubrics_sorted')
rubrics = cache.get('rubrics', Rubric.objects.get())
```

Сохранив ранее набор значений под одним ключом, мы можем извлекать эти значения, задавая для них номера версий, под которыми они были сохранены:

```
# Извлекаем значение, сохраненное под версией 1
val1 = cache.get('val')
val2 = cache.get('val', version=2)
val3 = cache.get('val', version=3)
```

- `get_or_set(<ключ>, <значение>[, timeout=<время хранения>][, version=None])` — если указанный *ключ* существует в кэше, извлекает хранящееся под ним значение и возвращает в качестве результата. В противном случае заносит в кэш *значение* под этим *ключом* и также возвращает это значение в качестве результата. Параметр `timeout` задает время хранения, а параметр `version` — номер версии. Пример:

```
rubrics = cache.get_or_set('rubrics', Rubric.objects.get(), 240)
```

- `incr(<ключ>[, delta=1][, version=None])` — увеличивает значение, хранящееся в кэше под заданным *ключом*, на величину, которая указана параметром `delta` (по умолчанию: `1`). В качестве результата возвращает новое значение. Параметр `version` задает номер версии. Пример:

```
cache.set('counter', 0)
cache.incr('counter')
cache.incr('counter', delta=2)
```

- `decr(<ключ>[, delta=1][, version=None])` — уменьшает значение, хранящееся в кэше под заданным *ключом*, на величину, которая указана параметром `delta` (по умолчанию: `1`). В качестве результата возвращает новое значение. Параметр `version` задает номер версии;

- `has_key(<ключ>[, version=None])` — возвращает `True`, если в кэше существует значение с указанным ключом, и `False` — в противном случае. Параметр `version` задает номер версии. Пример:

```
if cache.has_key('counter'):
    # Значение с ключом counter в кэше существует
```

- `delete(<ключ>[, version=None])` — удаляет из кэша значение под указанным *ключом*. Параметр `version` задает номер версии. Пример:

```
cache.delete('rubrics')
cache.delete('rubrics_sorted')
```

- `set_many(<данные>[, timeout=<время хранения>][, version=None])` — заносит в кэш заданные *данные*, представленные в виде словаря. Параметр `timeout` указывает время хранения данных, а параметр `version` — номер версии. Пример:

```
data = {
    'rubrics': Rubric.objects.get(),
    'rubrics_sorted': Rubric.objects.order_by('name')
}
cache.set_many(data, timeout=600)
```

- `get_many(<ключи>[, version=None])` — извлекает из кэша значения, ранее сохраненные под заданными *ключами*, и возвращает их в качестве результата. *Ключи* должны быть указаны в виде списка или кортежа, а возвращаемый результат представляется словарем. Параметр `version` указывает номер версии. Пример:

```
data = cache.get_many(['rubrics', 'counter'])
rubrics = data['rubrics']
counter = data['counter']
```

- `delete_many(<ключи>[, version=None])` — удаляет из кэша значения под указанными *ключами*, которые должны быть представлены в виде списка или кортежа. Параметр `version` задает номер версии. Пример:

```
cache.delete_many(['rubrics_sorted', 'counter'])
```

- `touch(<ключ>[, timeout=<время хранения>])` — задает для значения с заданным *ключом* новое время хранения, указанное в параметре `timeout`. Если этот параметр опущен, задается время кэширования из параметра `TIMEOUT` настроек кэша;
- `incr_version(<ключ>[, delta=1][, version=None])` — увеличивает версию значения, хранящегося в кэше под заданными *ключом* и версией, которая указана в параметре `version`, на величину, заданную в параметре `delta`. Новый номер версии возвращается в качестве результата;
- `decr_version(<ключ>[, delta=1][, version=None])` — уменьшает версию значения, хранящегося в кэше под заданными *ключом* и версией, которая указана в параметре `version`, на величину, заданную в параметре `delta`. Новый номер версии возвращается в качестве результата;
- `clear()` — полностью очищает кэш;

> ***Внимание!***
>
> Вызов метода `clear()` выполняет полную очистку кэша, при которой из него удаляются абсолютно все данные.

- `close()` — закрывает соединение с хранилищем, используемым в качестве кэша.

25.2. Кэширование на стороне клиента

Клиент, т. е. веб-обозреватель, тоже кэширует загружаемые с веб-сервера файлы, в частности страницы, на диске локального компьютера. Впоследствии, если запрошенный ранее файл не изменился, он будет извлечен из локального кэша, что существенно повысит производительность и уменьшит объем информации, пересылаемой по сети.

Для управления кэшированием на стороне клиента (в частности, для уменьшения объема пересылаемых по сети данных) Django предоставляет один посредник и несколько декораторов.

25.2.1. Автоматическая обработка заголовков

Посредник `django.middleware.http.ConditionalGetMiddleware` применяется при использовании кэширования на стороне сервера и выполняет автоматическую обработку заголовков, управляющих кэшированием на стороне клиента.

Как только клиенту отправляется запрошенная им страница, в состав ответа добавляется заголовок `E-Tag`, хранящий конечный ключ, под которым эта страница была сохранена в кэше, или `Last-Modified`, содержащий дату и время последнего изменения страницы. Эти сведения веб-обозреватель сохраняет в своем кэше.

Как только посетитель снова запрашивает загруженную ранее и сохраненную в локальном кэше страницу, веб-обозреватель посылает в составе запроса такие заголовки:

- `If-Match` или `If-None-Match` со значением полученного с ответом заголовка `E-Tag` — если ранее с ответом пришел заголовок `E-Tag`;
- `If-Modified-Since` или `If-Unmodified-Since` с датой и временем последнего изменения страницы, полученным с заголовком `Last-Modified` — если ранее в составе ответа был получен заголовок `Last-Modified`.

Получив запрос, посредник сравнивает значения:

- конечного ключа — полученное от клиента в заголовке `If-Match` или `If-None-Match` и находящееся в кэше;
- даты и времени последнего изменения страницы — полученное от клиента в заголовке `If-Modified-Since` или `If-Unmodified-Since` и хранящееся в кэше вместе с самой страницей.

Если эти значения равны, Django предполагает, что кэшированная страница еще не устарела, и отправляет клиенту ответ с кодом статуса 304 (запрошенная страница не изменилась). Веб-обозреватель вместо того, чтобы повторно загружать страницу по сети, извлекает ее из локального кэша, что выполняется гораздо быстрее.

Если же эти значения не равны, страница либо устарела и была удалена из кэша, либо еще не кэшировалась. Тогда веб-обозреватель получит полноценный ответ с кодом статуса 200, содержащий запрошенную страницу.

Посредник `django.middleware.http.ConditionalGetMiddleware` в списке зарегистрированных в проекте (параметр `MIDDLEWARE` настроек сайта) должен помещаться перед посредником `django.middleware.common.CommonMiddleware`. Автор книги обычно помещает его в самом начале списка зарегистрированных посредников:

```
MIDDLEWARE = [
    'django.middleware.http.ConditionalGetMiddleware',
    'django.middleware.security.SecurityMiddleware',
    . . .
]
```

Если кэширование на стороне сервера не задействовано, посредник `django.middleware.http.ConditionalGetMiddleware` не дает никакого эффекта. Хотя описанные ранее заголовки добавляются в ответ, получаются из запроса и обрабатываются посредником, но, поскольку страница не сохранена в кэше, она при каждом запросе генерируется заново.

25.2.2. Условная обработка запросов

Если кэширование на стороне сервера не используется, для управления кэшированием на стороне клиента нам следует применить другие средства. Это три декоратора, объявленные в модуле `django.views.decorators.http`.

- `condition([etag_func=None][,][last_modified_func=None])` — выполняет обработку заголовков `E-Tag` и `Last-Modified`.

  В параметре `etag_func` указывается ссылка на функцию, которая будет вычислять значение заголовка `E-Tag`. Эта функция должна принимать те же параметры, что и контроллер-функция, у которого указывается этот декоратор, и возвращать в качестве результата значение для упомянутого ранее заголовка, представленное в виде строки.

  В параметре `last_modififed_func` указывается ссылка на аналогичную функцию, вычисляющую значение для заголовка `Last-Modified`. Эта функция должна возвращать значение заголовка в виде даты и времени.

  Можно указать либо оба параметра сразу, либо, как поступают наиболее часто, лишь один из них.

  Декоратор указывается у контроллера-функции и в дальнейшем управляет кэшированием на уровне клиента страницы, которую формирует этот контроллер.

  При формировании страницы функции, записанные в вызове декоратора, вычисляют значения заголовков `E-Tag` и (или) `Last-Modified` в зависимости от того, какие функции были указаны в вызове. Эти заголовки отсылаются клиенту в составе ответа вместе с готовой страницей.

  Как только от того же клиента будет получен запрос на ту же страницу, еще до выполнения контроллера, что генерирует эту страницу, декоратор извлечет из запроса значения заголовков `E-Tag` и (или) `Last-Modified`. Далее он сравнит эти

значения с величинами, возвращенными функциями, что указаны в его вызове. Если значения `E-Tag` не совпадают, или если значение `Last-Modified`, вычисленное функцией, больше значения того же заголовка, полученного в составе клиентского запроса, будет выполнен контроллер, который сгенерирует страницу. В противном случае клиенту отправится ответ с кодом статуса 304 (запрошенная страница не изменилась).

Пример:

```
from django.views.decorators.http import condition
from .models import Bb

def bb_lmf(request, pk):
    return Bb.objects.get(pk=pk).published

@condition(last_modified_func=bb_lmf)
def detail(request, pk):
    . . .
```

Декоратор `condition()` можно указать и у контроллера-класса:

```
urlpatterns = [
    . . .
    path('detail/<int:pk>/',
        condition(last_modified_func=bb_lmf)(BbDetailView.as_view())),
    . . .
]
```

- `etag(<функция E-Tag>)` — применяется в качестве сокращения, если необходимо обрабатывать только заголовок `E-Tag`;
- `last_modified(<функция Last-Modified>)` — применяется в качестве сокращения, если необходимо обрабатывать только заголовок `Last_Modified`. Пример:

```
from django.views.decorators.http import last_modified

@last_modified(bb_lmf)
def detail(request, pk):
    . . .

urlpatterns = [
    . . .
    path('detail/<int:pk>/',
         last_modified(bb_lmf)(BbDetailView.as_view())),
    . . .
]
```

25.2.3. Прямое указание параметров кэширования

Параметры кэширования страницы на стороне клиента записываются в составе значения заголовка `Cache-Control`, который отсылается клиенту в составе ответа, включающего эту страницу. Указать эти параметры напрямую мы можем, вос-

пользовавшись декоратором `cache_control(<параметры>)` из модуля `django.views.decorators.cache`. В его вызове указываются именованные *параметры*, которые и зададут настройки кэширования. Примеры:

```
from django.views.decorators.cache import cache_control

# Параметр max_age указывает промежуток времени, в течение которого
# страница будет сохранена в клиентском кэше, в секундах
@cache_control(max_age=3600)
def detail(request, pk):
    ...

# Значение True, данное параметру private, указывает, что страница
# содержит конфиденциальные данные и может быть сохранена только в кэше
# веб-обозревателя. Промежуточные программы, наподобие прокси-серверов,
# кэшировать ее не будут
@cache_control(private=True)
def account(request, user_pk):
    ...
```

25.2.4. Запрет кэширования

Если необходимо запретить кэширование какой-либо страницы на уровне клиента (например, если страница содержит часто обновляющиеся или конфиденциальные данные), достаточно использовать декоратор `never_cache()` из модуля `django.views.decorators.cache`. Пример:

```
from django.views.decorators.cache import never_cache

@never_cache
def fresh_news(request):
    ...
```

ГЛАВА 26

Административный веб-сайт Django

Любой сайт, хранящий свои внутренние данные в информационной базе, должен предусматривать инструменты для работы с этими данными: их пополнения, правки и удаления. Эти инструменты составляют так называемый *административный раздел*. Понятно, что он должен быть закрыт от посторонних и доступен только зарегистрированным пользователям, которые обладают особыми правами.

Разработка такого административного раздела в большинстве случаев представляет собой задачу, вполне сравнимую по сложности с программированием общедоступных страниц сайта. Но Django избавляет нас от этой трудоемкой работы, предоставляя административный веб-сайт, полностью готовый к работе, открывающий доступ к любым внутренним данным, гибко настраиваемый и исключительно удобный в использовании.

Доступ к административному сайту Django имеют только суперпользователь и пользователи со статусом персонала.

26.1. Подготовка административного веб-сайта к работе

Прежде чем работать с административным сайтом, необходимо осуществить некоторые подготовительные операции. Для этого нужно открыть модуль настроек проекта settings.py, находящийся в пакете конфигурации, и выполнить следующие действия:

- проверить, присутствуют ли в списке зарегистрированных в проекте приложений (параметр `INSTALLED_APPS`) приложения `django.contrib.admin`, `django.contrib.auth`, `django.contrib.contenttypes`, `django.contrib.messages` и `django.contrib.sessions`;
- проверить, присутствуют ли в списке зарегистрированных в проекте посредников (параметр `MIDDLEWARE`) посредники `django.contrib.auth.middleware.AuthenticationMiddleware`, `django.contrib.sessions.middleware.SessionMiddleware` и `django.contrib.messages.middleware.MessageMiddleware`;

- проверить, присутствуют ли в списке зарегистрированных для используемого шаблонизатора обработчиков контекста (вложенный параметр `context_processors` параметра `OPTIONS`) обработчики `django.contrib.auth.context_processors.auth` и `django.contrib.messages.context_processors.messages`;
- добавить в список маршрутов уровня проекта (модуль urls.py пакета конфигурации) маршрут, который свяжет какой-либо шаблонный путь (обычно используется **admin/**) со списком маршрутов, записанным в атрибуте `urls` объекта административного сайта, который хранится в переменной `site` из модуля `django.contrib.admin`:

```
from django.contrib import admin
. . .
urlpatterns = [
    . . .
    path('admin/', admin.site.urls),
]
```

Впрочем, во вновь созданном проекте все эти действия выполнены изначально;

- выполнить миграции;
- создать суперпользователя (см. *разд. 15.2.2*).

26.2. Регистрация моделей на административном веб-сайте

Чтобы с данными, хранящимися в определенной модели, можно было работать посредством административного сайта, модель надо зарегистрировать на сайте. Делается это вызовом метода `register(<модель>)` объекта административного сайта. Пример:

```
from django.contrib import admin
from .models import Bb, Rubric

admin.site.register(Rubric)
```

Код, выполняющий регистрацию моделей на административном сайте, следует помещать в модуль admin.py пакета приложения.

Это самый простой способ сделать модель доступной через административный сайт. Однако в этом случае записи, хранящиеся в модели, будут иметь представление по умолчанию: отображаться в виде их строкового представления (которое формируется методом `__str__()`), выводиться в порядке их добавления, не поддерживать специфических средств поиска и пр.

Если мы хотим, чтобы список хранящихся в модели записей представлялся в виде таблицы с колонками, в которых выводятся значения их отдельных полей, а также получить доступ к специфическим средствам поиска и настроить внешний вид страниц добавления и правки записей, нам придется создать для этой модели класс редактора.

26.3. Редакторы моделей

Редактор модели — это класс, указывающий параметры представления модели на административном сайте. Редактор позволит нам задать список полей, которые должны выводиться на страницах списков записей, порядок сортировки записей, специальные средства их фильтрации, элементы управления, посредством которых пользователь будет заносить значения в поля записей, и пр.

Класс редактора должен быть производным от класса `ModelAdmin`, объявленного в модуле `django.contrib.admin`. Его объявление, равно как и регистрирующий его код, следует записывать в модуле admin.py пакета приложения. Пример объявления класса-редактора можно увидеть в листинге 1.14.

Различные параметры представления модели записываются в редакторе в виде атрибутов класса либо методов. Сейчас мы их рассмотрим.

26.3.1. Параметры списка записей

Страница списка записей, как понятно из ее названия, выводит все записи, что сохранены на данный момент в модели. Также она позволяет производить фильтрацию, сортировку записей, выполнять различные действия над группой выбранных записей (в частности, их удаление) и даже, при указании соответствующих настроек, править записи, не заходя на страницу правки.

26.3.1.1. Параметры списка записей: состав выводимого списка

Параметры из этого раздела управляют набором полей, которые будут выводиться в списке записей, и возможностью правки записей непосредственно на странице списка:

- `list_display` — атрибут, задает набор выводимых в списке полей. Его значением должен быть список или кортеж, элементом которого может быть:
  - имя поля модели в виде строки:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'price', 'published',
                    'rubric')
```

  - имя функционального поля, объявленного в модели, в виде строки (о функциональных полях рассказывалось в *разд. 4.7*). Вот пример указания функционального поля `title_and_price` модели `Bb`:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title_and_price', 'content', 'published',
                    'rubric')
```

    Также можно указать имя метода `__str__()`, который формирует строковое представление модели:

```
class RubricAdmin(admin.ModelAdmin):
    list_display = ('__str__', 'order')
```

- имя функционального поля, объявленного непосредственно в классе редактора, также в виде строки. Такое поле реализуется методом, принимающим в качестве единственного параметра объект записи и возвращающим результат, который и будет выведен на экран. Вот пример указания функционального поля `title_and_rubric`, объявленного в классе редактора `BbAdmin`:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title_and_rubric', 'content', 'price',
                    'published')

    def title_and_rubric(self, rec):
        return '%s (%s)' % (rec.title, rec.rubric.name)
```

Можно указать название для созданного таким образом функционального поля, присвоив его атрибуту `short_description` объекта метода, реализующего поле:

```
class BbAdmin(admin.ModelAdmin):
    . . .

    def title_and_rubric(self, rec):
        . . .
    title_and_rubric.short_description = 'Название и рубрика'
```

- ссылка на функциональное поле, объявленное в виде обычной функции. Такая функция должна принимать с единственным параметром объект записи и возвращать результат, который и будет выведен на экран. Для этого поля мы можем указать название, присвоив его атрибуту `short_description` объекта функции. Вот пример указания функционального поля `title_and_rubric`, реализованного в виде обычной функции:

```
def title_and_rubric(rec):
    return '%s (%s)' % (rec.title, rec.rubric.name)
title_and_rubric.short_description = 'Название и рубрика'

class BbAdmin(admin.ModelAdmin):
    list_display = [title_and_rubric, 'content', 'price',
                    'published']
```

Поля типа `ManyToManyField` не поддерживаются, и их содержимое не будет выводиться на экран.

По значениям функциональных полей невозможно будет выполнять сортировку, поскольку Django сортирует записи на уровне СУБД, а значения функционального поля не существуют на уровне базы данных. Однако можно связать функциональное поле с обычным полем модели, и тогда щелчок на заголовке функционального поля на странице списка записей приведет к сортировке записей по

значениям указанного обычного поля. Для этого достаточно присвоить строку с именем связываемого обычного поля модели атрибуту admin_order_field объекта метода (функции), реализующего функциональное поле. Вот пара примеров:

```
def title_and_rubric(rec):
    . . .
# Связываем с функциональным полем title_and_rubric обычное поле
# title модели
title_and_rubric.admin_order_field = 'title'
# А теперь связываем с тем же полем поле name связанной модели Rubric
title_and_rubric.admin_order_field = 'rubric__name'
```

- get_list_display(*<запрос>*) — метод, должен возвращать набор выводимых в списке полей. Полученный с параметром *запрос* можно использовать при формировании этого набора.

  В этом примере обычным пользователям в списке объявлений показываются только название, описание и цена товара, а суперпользователь может видеть также рубрику и дату публикации:

```
class BbAdmin(admin.ModelAdmin):
    def get_list_display(self, request):
        ld = ['title', 'content', 'price']
        if request.user.is_superuser:
            ld += ['published', 'rubric']
        return ld
```

  В изначальной реализации метод возвращает значение атрибута list_display;

- list_display_links — атрибут, задает перечень полей, чьи значения будут превращены в гиперссылки, указывающие на страницы правки соответствующих записей. В качестве значения указывается список или кортеж, элементами которого являются строки с именами полей. Эти поля должны присутствовать в перечне, заданном атрибутом list_display. Пример:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'price', 'published',
                    'rubric')
    list_display_links = ('title', 'content')
```

  Если в качестве значения атрибута задать «пустой» список или кортеж, в гиперссылку будет превращено значение самого первого поля, выводящегося в списке, а если указать None, такие гиперссылки вообще не будут создаваться. Значение по умолчанию — «пустой» список;

- get_list_display_links(*<запрос>*, *<набор выводимых полей>*) — метод, должен возвращать перечень полей, чьи значения будут превращены в гиперссылки, указывающие на страницы правки соответствующих записей. Вторым параметром передается *набор выводимых полей*, который может быть использован для создания перечня полей-гиперссылок.

Вот пример преобразования в гиперссылки всех полей, что выводятся на экран:

```
class BbAdmin(admin.ModelAdmin):
    def get_list_display_links(self, request, list_display):
        return list_display
```

В изначальной реализации метод возвращает значение атрибута `list_display_links`, если его значение не равно «пустому» списку или кортежу. В противном случае возвращается список, содержащий первое поле из состава выводимых на экран;

- `list_editable` — атрибут, задает перечень полей, которые можно будет править непосредственно на странице списка записей, не переходя на страницу правки.

  После задания перечня полей в соответствующих столбцах списка записей появятся элементы управления, с помощью которых пользователь сможет исправить значения полей, указанных в перечне. После правки необходимо нажать расположенную в нижней части страницы кнопку **Сохранить**, чтобы внесенные правки были сохранены.

  В качестве значения атрибута указывается список или кортеж, элементами которого должны быть имена полей, представленные в виде строк. Если указать «пустой» список или кортеж, ни одно поле модели не может быть исправлено на странице списка.

  > ***Внимание!***
  >
  > **Поля, указанные в перечне, который задан для атрибута `list_editable`, не должны присутствовать в перечне из атрибута `list_display_links`.**

  Пример:

  ```
  class BbAdmin(admin.ModelAdmin):
      list_display = ('title', 'content', 'price', 'published',
                      'rubric')
      list_display_links = None
      list_editable = ('title', 'content', 'price', 'rubric')
  ```

  Значение атрибута по умолчанию — «пустой» кортеж;

- `list_select_related` — атрибут, устанавливает набор первичных моделей, чьи связанные записи будут извлекаться одновременно с записями текущей модели посредством вызова метода `select_related()` (см. *разд. 16.1*). В качестве значения можно указать:

  - `True` — извлекать связанные записи всех первичных моделей, связанных с текущей моделью;
  - `False` — извлекать связанные записи только тех первичных моделей, что соответствуют полям типа `ForeignKey`, присутствующим в перечне из атрибута `list_display`;

- список или кортеж, содержащий строки с именами полей типа `ForeignKey`, — извлекать связанные записи только тех первичных моделей, что соответствуют приведенным в этом списке (кортеже) полям;
- «пустой» список или кортеж — вообще не извлекать связанные записи.

Извлечение из базы связанных записей первичных моделей одновременно с записями текущей модели позволяет повысить производительность, т. к. впоследствии Django не придется для получения связанных записей обращаться к базе еще раз.

Значение атрибута по умолчанию — `False`;

- `get_list_select_related(<запрос>)` — метод, должен возвращать набор первичных моделей, чьи связанные записи будут извлекаться одновременно с записями текущей модели посредством вызова метода `select_related()`. В изначальной реализации возвращает значение атрибута `list_select_related`;
- `get_queryset(<запрос>)` — метод, должен возвращать набор записей, который и станет выводиться на странице списка. Переопределив этот метод, можно установить какую-либо дополнительную фильтрацию записей, создать вычисляемые поля или изменить сортировку.

  Вот пример переопределения этого метода с целью дать возможность суперпользователю просматривать все объявления, а остальным пользователям — только объявления, не помеченные как скрытые:

```
class BbAdmin(admin.ModelAdmin):
    def get_queryset(self, request):
        qs = super().get_queryset(request)
        if request.user.is_superuser:
            return qs
        else:
            return qs.filter(is_hidden=False)
```

26.3.1.2. Параметры списка записей: фильтрация и сортировка

В этом разделе мы познакомимся с параметрами, настраивающими средства для фильтрации и сортировки записей:

- `ordering` — атрибут, задает порядок сортировки записей. Значение указывается в том же формате, что и значение параметра модели `ordering` (см. *разд. 4.5*). Если указать значение `None`, будет использована сортировка, заданная для модели. Значение по умолчанию — `None`;
- `get_ordering(<запрос>)` — метод, должен возвращать параметры сортировки. В изначальной реализации возвращает значение атрибута `ordering`;
- `sortable_by` — атрибут, задает перечень полей модели, по которым пользователь сможет выполнять сортировку записей. Перечень полей задается в виде списка, кортежа или множества. Если задать «пустую» последовательность, пользова-

тель не сможет сортировать записи по своему усмотрению. Значение по умолчанию — None (разрешена сортировка записей по всем полям);

- get_sortable_by(*<запрос>*) — метод, должен возвращать перечень полей модели, по которым пользователь сможет выполнять сортировку записей. В изначальной реализации возвращает значение атрибута sortable_by;
- search_fields — атрибут, указывает перечень полей модели, по которым будет выполняться фильтрация записей. Значением должен быть список или кортеж имен полей, каждое из которых представляет собой строку. Пример:

```
class BbAdmin(admin.ModelAdmin):
    search_fields = ('title', 'content')
```

Фильтрация по указанным в этом атрибуте полям производится занесением в расположенное вверху страницы поле ввода искомого слова или фразы и нажатием кнопки **Найти**. Чтобы отменить фильтрацию и вывести все записи, достаточно очистить поле ввода и снова нажать кнопку **Найти**.

По умолчанию Django выбирает те записи, что содержат все занесенные в поле ввода слова, независимо от того, в каком месте поля встретилось это слово и является оно отдельным словом или частью другого, более длинного. Фильтрация выполняется без учета регистра. Например, если указать для фильтрации строку «газ кирпич», будут отобраны записи, которые в указанных полях хранят слова «Газ» и «Кирпич», «керогаз» и «кирпичный» и т. п., но не «газ» и «дерево».

Мы можем изменить поведение фреймворка при фильтрации, предварив имя поля одним из поддерживаемых префиксов:

- ^ — искомое слово должно присутствовать в начале поля:

```
search_fields = ('^content',)
```

При задании слова «газ» будут отобраны записи со словами «газ», «газовый», но не «керогаз»;

- = — точное совпадение, т. е. указанное слово должно полностью совпадать со значением поля. Однако регистр и в этом случае не учитывается. Пример:

```
search_fields = ('=content',)
```

Будут отобраны записи со словами «газ», но не «газовый» или «керогаз»;

- @ — будет выполнен полнотекстовый поиск. Поддерживается только базами данных MySQL.

Значение атрибута по умолчанию — «пустой» кортеж;

- get_search_fields(*<запрос>*) — метод, должен возвращать перечень полей, по которым будет выполняться фильтрация записей. В изначальной реализации возвращает значение атрибута search_fields;
- show_full_result_count — атрибут. Если True, после выполнения фильтрации в полях, заданных атрибутом search_fields, правее кнопки **Найти** будет выведено количество отфильтрованных записей и общее число записей в модели.

Если `False`, вместо общего числа записей в модели будет выведена гиперссылка **Показать все**. Значение по умолчанию — `True`.

Для вывода общего количества записей Django выполняет дополнительный запрос к базе данных. Поэтому, если стоит задача всемерно уменьшить нагрузку на базу, этому атрибуту имеет смысл дать значение `False`;

- `list_filter` — атрибут, указывает перечень полей, по которым можно будет выполнять быструю фильтрацию. После указания такого перечня в правой части страницы списка записей появятся списки всех значений, занесенных в заданные поля, и при щелчке на таком значении на странице будут выведены только те записи, у которых указанное поле хранит выбранное значение. Чтобы вновь вывести в списке все записи, следует выбрать пункт **Все**.

  Значением поля должен быть список или кортеж, каждый элемент которого представляет собой:

  - имя поля, записанное в виде строки:

    ```
    class BbAdmin(admin.ModelAdmin):
        list_display = ['title', 'content', 'price', 'published',
                        'rubric']
        list_filter = ('title', 'rubric__name',)
    ```

    Как видно из примера, в перечне можно указывать поля связанных моделей;

  - ссылку на подкласс класса `SimpleListFilter` из модуля `django.contrib.admin`, реализующий более сложную фильтрацию. В этом подклассе следует объявить:

    - `title` — атрибут, указывает заголовок, что будет выведен над перечнем доступных для выбора значений, по которым, собственно, и будет выполняться фильтрация. Значение задается в виде строки;
    - `parameter_name` — атрибут, указывает имя GET-параметра, посредством которого будет пересылаться внутренний идентификатор выбранного пользователем значения для фильтрации. Это имя также должно представлять собой строку;
    - `lookups(`*<запрос>*`,` *<редактор>*`)` — метод, должен возвращать перечень доступных для выбора значений, по которым будет выполняться фильтрация. В качестве параметров принимает ссылки на объекты *запроса* и текущего *редактора*.

      Возвращаемое методом значение должно представлять собой кортеж, каждый элемент которого задаст отдельное значение для фильтрации. Этот элемент должен представлять собой кортеж из двух строковых элементов: внутреннего идентификатора значения (того самого, что пересылается через GET-параметр, заданный атрибутом `parameter_name`) и названия, выводящегося на экран;
    - `queryset(`*<запрос>*`,` *<набор записей>*`)` — метод, вызывается после щелчка пользователя на одном из значений, и должен возвращать соответствую-

щим образом отфильтрованный набор записей. Изначальный *набор записей*, равно как и объект *запроса*, принимается им в качестве параметров.

Чтобы получить внутренний идентификатор выбранного пользователем для фильтрации записей значения, нужно обратиться к методу `value()`, унаследованному классом редактора от суперкласса.

Далее приведен код класса, реализующего фильтрацию объявлений по цене: является она низкой (менее 500 руб.), средней (от 500 до 5000 руб.) или высокой (более 5000 руб.).

```
class PriceListFilter(admin.SimpleListFilter):
    title = 'Категория цен'
    parameter_name = 'price'

    def lookups(self, request, model_admin):
        return (
            ('low', 'Низкая цена'),
            ('medium', 'Средняя цена'),
            ('high', 'Высокая цена'),
        )

    def queryset(self, request, queryset):
        if self.value() == 'low':
            return queryset.filter(price__lt=500)
        elif self.value() == 'medium':
            return queryset.filter(price__gte=500,
                                   price__lte=5000)
        elif self.value() == 'high':
            return queryset.filter(price__gt=5000)
```

Теперь мы можем использовать этот класс для быстрой фильтрации объявлений по категории цен:

```
class BbAdmin(admin.ModelAdmin):
    . . .
    list_filter = (PriceListFilter,)
```

Если в качестве значения атрибута указать «пустой» список или кортеж, быстрая сортировка будет отключена. Значение по умолчанию — «пустой» кортеж;

- `get_list_filter(<запрос>)` — метод, должен возвращать перечень полей, по которым можно будет выполнять быструю фильтрацию. В изначальной реализации возвращает значение атрибута `list_filter`;
- `date_hierarchy` — атрибут, включающий быструю фильтрацию по величинам даты. Если указать в качестве его значения строку с именем поля типа `DateField` или `DateTimeField`, над списком записей будет выведен набор гиперссылок, представляющих собой значения, сохраненные в этом поле. Если щелкнуть на такой гиперссылке, в списке будут выведены только те записи, в указанном поле

которых хранится выбранное значение. Значение атрибута по умолчанию — None (быстрая фильтрация по дате не выполняется). Пример:

```
class BbAdmin(admin.ModelAdmin):
    . . .
    date_hierarchy = 'published'
```

- preserve_filters — атрибут. Если True, после сохранения добавленной или исправленной записи все заданные пользователем условия фильтрации продолжают действовать. Если False, фильтрация записей в списке после этого будет отменена. Значение по умолчанию — True.

26.3.1.3. Параметры списка записей: прочие

Здесь мы рассмотрим остальные параметры, затрагивающие, по большей части, внешний вид списка записей:

- list_per_page — атрибут, задает количество записей в части пагинатора, что применяется при выводе списка записей. Количество указывается в виде целого числа. Значение по умолчанию: 100;
- list_max_show_all — атрибут, задает количество записей, которые появятся на странице списка после щелчка на гиперссылке **Показать все** (она находится под списком), если общее количество записей меньше или равно количеству, заданному этим атрибутом. Значение атрибута указывается в виде целого числа. Значение по умолчанию: 200;
- actions_on_top — атрибут. Если True, над списком записей будет выведен раскрывающийся список всех действий, доступных для выполнения над группой выбранных записей. Если False, список действий над списком записей выведен не будет. Значение по умолчанию — True;
- actions_on_bottom — атрибут. Если True, под списком записей будет выведен раскрывающийся список всех действий, доступных для выполнения над группой выбранных записей. Если False, список действий под списком записей выведен не будет. Значение по умолчанию — False;
- actions_selection_counter — атрибут. Если True, возле раскрывающегося списка доступных действий будут выведены количество выбранных записей и общее количество записей, присутствующих на текущей странице списка. Если False, эти сведения не будут выводиться. Значение по умолчанию — True;
- empty_value_display — атрибут, указывает символ или строку, которая будет выводиться вместо пустого значения поля. Значение по умолчанию — дефис. Пример:

```
class BbAdmin(admin.ModelAdmin):
    . . .
    empty_value_display = '---'
```

Если в редакторе объявлено функциональное поле, можно указать подобного рода значение только для этого поля, присвоив его атрибуту empty_value_display объекта метода, который реализует это поле:

```
class RubricAdmin(admin.ModelAdmin):
    fields = ('name', 'super_rubric')

    def super_rubric(self, rec):
        return rec.super_rubric.name
    super_rubric.empty_value_display = '[нет]'
```

- `paginator` — атрибут, задает ссылку на класс пагинатора, используемого для вывода списка записей. Значение по умолчанию — ссылка на класс `Paginator` из модуля `django.core.paginator`;
- `get_paginator()` — метод, должен возвращать объект пагинатора, используемого для вывода списка записей. Формат вызова:

```
get_paginator(<запрос>, <набор записей>,
              <количество записей в части>[, orphans=0][,
              allow_empty_first_page=True])
```

Параметры, получаемые этим методом, кроме первого, полностью аналогичны тем, что передаются конструктору класса пагинатора (см. *главу 12*).

В реализации по умолчанию возвращает экземпляр класса пагинатора, заданного в атрибуте `paginator`.

26.3.2. Параметры страниц добавления и правки записей

Страницы добавления и правки записей выводят формы, посредством которых пользователь сможет добавить в модель новую запись или же исправить имеющуюся. Здесь мы познакомимся с параметрами редактора, которые позволят нам настроить эти страницы.

26.3.2.1. Параметры страниц добавления и правки записей: набор выводимых полей

Мы можем указать административному сайту вывести на этих страницах не все поля модели, а только выбранные нами:

- `fields` — атрибут, задает последовательность имен полей модели, которые должны присутствовать в форме (не указанные поля выведены не будут). Поля будут выведены в том порядке, в котором они указаны в последовательности. Вот пример вывода в форме объявления только полей названия, описания и цены товара:

```
class BbAdmin(admin.ModelAdmin):
    fields = ('title', 'content', 'price')
```

В последовательности можно указать поля, доступные для чтения, которые указаны в атрибуте `readonly_fields` (он будет описан далее). Понятно, что значения таких полей править не получится.

По умолчанию поля в форме выводятся по вертикали сверху вниз. Чтобы вывести какие-либо поля в одной строчке по горизонтали, нужно заключить их имена во вложенный кортеж. Вот пример вывода полей названия и цены товара в одной строчке:

```
class BbAdmin(admin.ModelAdmin):
    fields = (('title', 'price'), 'content')
```

Если указать для атрибута значение `None`, в форме будут выведены все поля, кроме имеющих типы `AutoField`, `BigAutoField` и тех, у которых для параметра `editable` конструктора класса было явно или неявно задано значение `True`. Значение по умолчанию — `None`;

- `get_fields(<запрос>[, obj=None])` — метод, должен возвращать последовательность имен полей модели, которые следует вывести в форме. В необязательном параметре `obj` может быть получен объект исправляемой записи или `None`, если запись в данный момент добавляется.

  Для примера сделаем так, чтобы у создаваемого объявления можно было указать рубрику, а у исправляемого — уже нет:

  ```
  class BbAdmin(admin.ModelAdmin):
      def get_fields(self, request, obj=None):
          f = ['title', 'content', 'price']
          if not obj:
              f.append('rubric')
          return f
  ```

  В изначальной реализации метод возвращает значение атрибута `fields`, если его значение отлично от `None`. В противном случае возвращается список из всех полей, объявленных в модели, и всех полей, заявленных в перечне доступных только для чтения (атрибут `readonly_fields`, описанный далее);

- `exclude` — атрибут, задает последовательность имен полей модели, которые следует исключить из числа выводимых в форме (все остальные поля модели будут выведены). Вот пример вывода в форме объявления всех полей, кроме рубрики и типа объявления:

  ```
  class BbAdmin(admin.ModelAdmin):
      exclude = ('rubric', 'kind')
  ```

  Если задать значение `None`, ни одно поле модели не будет исключено из числа выводимых в форме. Значение по умолчанию — `None`;

- `get_exclude(<запрос>[, obj=None])` — метод, должен возвращать последовательность имен полей модели, которые следует исключить из числа выводимых в форме. В необязательном параметре `obj` может быть получен объект исправляемой записи или `None`, если запись в данный момент добавляется. В изначальной реализации возвращает значение атрибута `exclude`;

- `readonly_fields` — атрибут, задает перечень полей, которые должны быть доступны только для чтения. Значения таких полей будут выведены в виде обычно-

го текста. Значение атрибута должно представлять собой список или кортеж с именами полей, представленных строками. Пример:

```
class BbAdmin(admin.ModelAdmin):
    fields = ('title', 'content', 'price', 'published')
    readonly_fields = ('published',)
```

Вероятно, указание в списке атрибута `readonly_fields` — единственный способ вывести на странице правки записи значение поля, у которого при создании параметр `editable` конструктора был установлен в `False`.

Также в этом атрибуте можно указать функциональные поля, объявленные в классе редактора.

Значение атрибута по умолчанию — «пустой» кортеж;

- `get_readonly_fields(<запрос>[, obj=None])` — метод, должен возвращать перечень полей, которые должны быть доступны только для чтения. В необязательном параметре `obj` может быть получен объект исправляемой записи или `None`, если запись в данный момент добавляется. В изначальной реализации возвращает значение атрибута `readonly_fields`;
- `inlines` — атрибут, задает список встроенных редакторов, присутствующих в текущем редакторе (будут рассмотрены позже). Значение по умолчанию — «пустой» список;
- `fieldsets` — атрибут, задает все наборы полей, которые будут созданы в форме.

  Набор полей, как и следует из его названия, объединяет указанные поля формы. Он может вообще никак не выделяться на экране (*основной набор полей*), а может представляться в виде спойлера, изначально свернутого или развернутого.

  Перечень наборов полей записывается в виде кортежа, в котором каждый элемент представляет один набор и также должен являться кортежем из двух элементов: заголовка для набора полей (если задать `None`, будет создан основной набор полей) и словаря с дополнительными параметрами. Поддерживаемые Django дополнительные параметры наборов полей таковы:

  - `fields` — кортеж из строк с именами полей модели, которые должны выводиться в наборе. Этот параметр обязателен для указания.

    Здесь можно указать доступные для чтения поля, которые приведены в атрибуте `readonly_fields`, том числе и функциональные поля.

    По умолчанию поля в наборе выводятся по вертикали сверху вниз. Чтобы вывести какие-либо поля в одной строчке по горизонтали, нужно заключить их имена во вложенный кортеж;
  - `classes` — список или кортеж с именами стилевых классов, которые будут привязаны к набору, представленными строками. Поддерживаются два стилевых класса: `collapse` (набор полей, представленный в виде спойлера, будет изначально свернут) и `wide` (поля в наборе займут все пространство окна по ширине);

- description — строка с поясняющим текстом, который будет выведен в верху набора форм, непосредственно под его названием. Вместо обычного текста можно указать HTML-код.

Пример указания набора полей:

```
class BbAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': (('title', 'rubric'), 'content'),
            'classes': ('wide',),
        }),
        ('Дополнительные сведения', {
            'fields': ('price',),
            'description': 'Параметры, необязательные для указания.',
        })
    )
```

- get_fieldsets(*<запрос>*[, obj=None]) — метод, должен возвращать перечень наборов полей, что будут выведены в форме. В необязательном параметре obj может быть получен объект исправляемой записи или None, если запись в данный момент добавляется.

  В изначальной реализации метод возвращает значение атрибута fieldsets, если его значение отлично от «пустого» словаря. В противном случае возвращается перечень из одного основного набора, содержащего все имеющиеся в форме поля;

- form — атрибут, задает класс связанной с моделью формы, на основе которой будет создана окончательная форма, применяемая для работы с записью. Значение по умолчанию — ссылка на класс ModelForm;

- get_form(*<запрос>*[, obj=None][, *<параметры формы>*]) — метод, должен возвращать класс формы, которая будет использоваться для занесения данных в создаваемую или исправляемую запись.

  В необязательном параметре obj может быть получен объект исправляемой записи или None, если запись в данный момент добавляется. А *параметры формы* будут переданы функции modelform_factory(), которая применяется методом для создания класса формы.

  Вот пример использования для добавления записи формы BbAddModelForm, а для правки существующей записи — формы BbModelForm:

```
class BbAdmin(admin.ModelAdmin):
    def get_form(self, request, obj=None, **kwargs):
        if obj:
            return BbModelForm
        else:
            return BbAddModelForm
```

В изначальной реализации метод возвращает класс формы, сгенерированный вызовом функции `modelform_factory()`.

26.3.2.2. Параметры страниц добавления и правки записей: элементы управления

Эти параметры настраивают внешнее представление отдельных полей в форме, в частности, задают для них элементы управления:

- `radio_fields` — атрибут, задает перечень полей типа `ForeignKey` и полей со списком, для отображения которых вместо раскрывающегося списка будет использован набор переключателей. В качестве значения задается словарь, ключами элементов которого должны быть имена полей, а значениями — одна из переменных, объявленных в модуле `django.contrib.admin`: `HORIZONTAL` (переключатели в наборе располагаются по горизонтали) или `VERTICAL` (по вертикали). Пример:

```
class BbAdmin(admin.ModelAdmin):
    radio_fields = {'rubric': admin.VERTICAL}
```

  Значение атрибута по умолчанию — «пустой» словарь;

- `autocomplete_fields` — атрибут, задает перечень полей типов `ForeignKey` и `ManyToManyField`, для отображения которых вместо обычного списка должен быть использован список с возможностью поиска. Такой список будет включать в свой состав поле для указания текста искомого пункта или его части. Значение атрибута указывается в виде обычного списка или кортежа, включающего строки с именами полей.

  > ***Внимание!***
  > Чтобы список с возможностью поиска успешно работал, необходимо в классе редактора, представляющем связанную модель, задать перечень полей, по которым можно выполнять поиск записей (атрибут `search_fields`).

  Пример:

```
class RubricAdmin(admin.ModelAdmin):
    search_fields = ('name',)

class BbAdmin(admin.ModelAdmin):
    autocomplete_fields = ('rubric',)
```

  Значение атрибута по умолчанию — «пустой» кортеж;

- `get_autocomplete_fields(<запрос>)` — метод, должен возвращать перечень полей, для отображения которых вместо обычного списка нужно использовать список с возможностью поиска. В изначальной реализации возвращает значение атрибута `autocomplete_fields`;

- `filter_horizontal` — атрибут, указывает кортеж строк с именами полей типа `ManyToManyField`, которые должны быть отображены в виде пары расположенных по горизонтали списков с возможностью поиска пунктов.

Пример такого элемента управления, надо сказать, исключительно удобного в использовании, можно увидеть на рис. 15.1. В левом списке выводятся только записи ведомой модели, не связанные с текущей записью ведущей модели, а в правом — только записи, связанные с ней. Связывание записей ведомой модели с текущей записью ведущей модели выполняется переносом их из левого списка в правый щелчком на кнопке со стрелкой, направленной вправо. Аналогично удаление записей из числа связанных с текущей записью производится переносом из правого списка в левый, для чего нужно щелкнуть на кнопке со стрелкой влево.

Вот пример вывода поля `spares` модели `Machine` (см. листинг 4.2) в виде подобного рода элемента управления:

```
class MachineAdmin(admin.ModelAdmin):
    filter_horizontal = ('spares',)
```

Поля типа `ManyToManyField`, не указанные в этом атрибуте, будут выводиться в виде обычного списка с возможностью выбора произвольного количества пунктов. Такой элемент управления не очень удобен в использовании, особенно если записей в ведомой модели достаточно много.

Значение параметра по умолчанию — «пустой» кортеж;

- `filter_vertical` — то же самое, что `filter_horizontal`, только списки выводятся не по горизонтали, а по вертикали, друг над другом: сверху — список несвязанных записей, а под ним — список связанных записей;
- `formfield_overrides` — атрибут, позволяет переопределить параметры полей формы, которая будет выводиться на страницах добавления и правки. В качестве значения указывается словарь, ключами элементов которого выступают ссылки на классы полей модели, а значения указывают параметры соответствующих им полей формы и также записываются в виде словарей.

  Чаще всего этот атрибут применяется для указания элементов управления, которыми будут представляться в форме поля модели заданного типа. Вот пример указания для поля типа `ForeignKey` в качестве элемента управления обычного списка вместо применяемого по умолчанию раскрывающегося:

```
from django import forms
from django.db import models

class BbAdmin(admin.ModelAdmin):
    formfield_overrides = {
        models.ForeignKey: {'widget': forms.widgets.Select(
                                       attrs={'size': 8})},
    }
```

  Значение атрибута по умолчанию — «пустой» словарь;

- `prepopulated_fields` — атрибут, устанавливает набор полей, чьи значения должны формироваться на основе значений из других полей. В качестве значения указывается словарь, ключи элементов которого должны соответствовать полям,

чьи значения будут формироваться описанным ранее образом, а значениями станут кортежи имен полей, откуда будут браться данные для формирования значений.

Основное назначение этого атрибута — указание сведений для формирования слагов. Обычно слаг создается из названия какой-либо позиции путем преобразования букв кириллицы в символы латиницы, удаления знаков препинания и замены пробелов на дефисы. Также можно формировать слаги на основе нескольких значений (например, названия и рубрики) — в этом случае отдельные значения объединяются.

Пример:

```
class BbAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
```

Значение атрибута по умолчанию — «пустой» словарь;

- `get_prepopulated_fields(<запрос>[, obj=None])` — метод, должен возвращать набор полей, чьи значения должны формироваться на основе значений из других полей. В необязательном параметре `obj` может быть получен объект исправляемой записи или `None`, если запись в данный момент добавляется. В изначальной реализации возвращает значение атрибута `prepopulated_fields`;
- `raw_id_fields` — атрибут, задает перечень полей типа `ForeignKey` или `ManyToManyField`, для отображения которых вместо списка нужно использовать обычное поле ввода. В такое поле вводится ключ связанной записи или сразу несколько ключей, приведенных через запятую. Значение атрибута указывается в виде списка или кортежа, в котором приводятся имена полей, представленные строками. Значение по умолчанию — «пустой» кортеж.

26.3.2.3. Параметры страниц добавления и правки записей: прочие

Осталось рассмотреть весьма немногочисленные прочие параметры, управляющие внешним видом и поведением страниц добавления и правки записей:

- `view_on_site` — атрибут, указывает, будет ли на странице правки записи выводиться гиперссылка **Смотреть на сайте**, при щелчке на которой осуществляется переход по интернет-адресу модели (см. *разд. 4.6*). В качестве значения атрибута можно указать:
  - `True` — выводить гиперссылку, если в модели объявлен метод `get_absolute_url()`, формирующий интернет-адрес модели. Если такого метода нет, гиперссылка выводиться не будет;
  - `False` — не выводить гиперссылку в любом случае.

  Значение по умолчанию — `True`.

  Аналогичного результата можно достичь, объявив непосредственно в классе редактора метод `view_on_site()`, который в качестве единственного параметра будет получать объект записи и возвращать строку с интернет-адресом модели. Вот пример:

```
from django.urls import reverse

class BbAdmin(admin.ModelAdmin):
    def view_on_site(self, rec):
        return reverse('bboard:detail', kwargs={'pk': rec.pk})
```

- `save_as` — атрибут. Если `True`, на странице будет выведена кнопка **Сохранить как новый объект**, если `False`, вместо нее будет присутствовать кнопка **Сохранить и добавить другой объект**. Значение по умолчанию — `False`;
- `save_as_continue` — атрибут. Принимается во внимание только в том случае, если для атрибута `save_as` задано значение `True`. Если `True`, после сохранения новой записи выполняется перенаправление на страницу правки той же записи, если `False` — возврат на страницу списка записей. Значение по умолчанию — `True`;
- `save_on_top` — атрибут. Если `True`, кнопки сохранения записи будут присутствовать и вверху, и внизу страницы, если `False` — только внизу. Значение по умолчанию — `False`.

> ***На заметку***
>
> Помимо рассмотренных здесь, редакторы Django поддерживают ряд более развитых инструментов: обработку сохранения записей и наборов записей, удаления записей, указание элементов управления для полей в зависимости от различных условий (например, является ли текущий пользователь суперпользователем), замену шаблонов страниц и др. Эти инструменты достаточно сложны в использовании и применяются относительно редко, поэтому и не рассматриваются в этой книге. За инструкциями по их применению обращайтесь на страницу **https://docs.djangoproject.com/en/2.1/ref/contrib/admin/**.

26.3.3. Регистрация редакторов на административном веб-сайте

Чтобы административный сайт смог использовать в работе написанный нами редактор, последний должен быть соответствующим образом зарегистрирован. Сделать это можно двумя способами:

- применить расширенный формат вызова метода `register()` объекта административного сайта:

```
register(<модель>, <редактор>)
```

Пример:

```
from django.contrib import admin
from .models import Bb, Rubric

class BbAdmin(admin.ModelAdmin):
    . . .

admin.site.register(Bb, BbAdmin)
```

- применить декоратор `register()`, объявленный в модуле `django.contrib.admin`, формат вызова которого следующий:

```
register(<модель 1>, <модель 2> . . . <модель n>)
```

Этот декоратор указывается непосредственно у объявления класса редактора.

Пример:

```
from django.contrib import admin
from .models import Bb, Rubric

@admin.register(Bb)
class BbAdmin(admin.ModelAdmin):
    . . .
```

Применив декоратор, один и тот же класс редактора можно указать сразу для нескольких моделей:

```
@admin.register(Bb, Rubric, Machine, Spare)
class UniversalAdmin(admin.ModelAdmin)
    . . .
```

26.4. Встроенные редакторы

Встроенный редактор по назначению аналогичен встроенному набору форм (см. *разд. 14.5*). Он позволяет работать с набором записей вторичной модели, связанным с записью первичной модели, которая выводится основным редактором.

26.4.1. Объявление встроенного редактора

Класс встроенного редактора должен быть производным от одного из следующих классов, объявленных в модуле `django.contrib.admin`:

- `StackedInline` — элементы управления в отдельных формах такого набора располагаются по вертикали;
- `TabularInline` — элементы управления в отдельных формах такого набора располагаются по горизонтали, в виде строки таблицы. Для формирования всего набора форм применяется таблица HTML.

При объявлении встроенного редактора в нем указывается модель, которую он должен обслуживать. После объявления он связывается с классом основного редактора.

Встроенный редактор создает на странице добавления или правки записи первичной модели набор форм, с помощью которого и будет производиться работа со связанными записями вторичной модели.

Листинг 26.1 показывает код, объявляющий и регистрирующий редактор `RubricAdmin`, который предназначен для работы с моделью рубрик `Rubric`. Этот редактор связан со встроенным редактором `BbInline`, обслуживающим модель объявлений `Bb` и выводящим объявления, занесенные в текущую рубрику.

Листинг 26.1. Пример использования встроенного редактора

```
from django.contrib import admin
from .models import Bb, Rubric

class BbInline(admin.StackedInline):
    model = Bb

class RubricAdmin(admin.ModelAdmin):
    inlines = [BbInline]

admin.site.register(Rubric, RubricAdmin)
```

26.4.2. Параметры встроенного редактора

Оба класса встроенных редакторов наследуют от класса `ModelAdmin` атрибуты `ordering`, `fields`, `exclude`, `fieldsets`, `radio_fields`, `filter_horizontal`, `filter_vertical`, `formfield_overrides`, `readonly_fields`, `prepopulated_fields`, `raw_id_fields` и методы `get_ordering()`, `get_queryset()`, `get_fields()`, `get_exclude()`, `get_fieldsets()`, `get_readonly_fields()`, `get_prepopulated_fields()`, описанные ранее.

Кроме того, встроенные редакторы поддерживают следующие дополнительные атрибуты и методы:

- `model` — атрибут, указывает ссылку на класс вторичной модели, которая будет обслуживаться встроенным редактором. Единственный обязательный для указания атрибут. Значение по умолчанию — `None`;
- `fk_name` — атрибут, задает имя поля внешнего ключа у вторичной модели в виде строки. Обязателен для указания, если вторичная модель связана с разными первичными моделями и, соответственно, включает несколько полей внешнего ключа. Если задать значение `None`, Django использует самое первое из объявленных в модели поле внешнего ключа. Значение по умолчанию — `None`;
- `extra` — атрибут, указывает количество пустых форм, предназначенных для создания новых записей, которые будут присутствовать в редакторе. Количество форм должно быть задано в виде целого числа. Значение по умолчанию: `3`;
- `get_extra(<запрос>[, obj=None][, <дополнительные параметры>])` — метод, должен возвращать количество пустых форм, предназначенных для создания новых записей.

  В необязательном параметре `obj` может быть получен объект исправляемой записи первичной модели или `None`, если такая запись еще не создана. Назначение *дополнительных параметров* этого метода автором не установлено — вероятно, они оставлены на будущее.

  Вот пример указания десяти «пустых» форм при создании записи первичной модели и трех при ее правке:

```
class BbInline(admin.StackedInline):
    model = Bb

    def get_extra(self, request, obj=None, **kwargs):
        if obj:
            return 3
        else:
            return 10
```

В изначальной реализации метод возвращает значение атрибута `extra`;

- `can_delete` — атрибут. Если `True`, редактор разрешит пользователю удалять записи, если `False` — не разрешит. Значение по умолчанию — `True`;
- `show_change_link` — атрибут. Если `True`, в каждой из форм встроенного редактора будет выведена гиперссылка **Изменить**, ведущая на страницу правки соответствующей записи. Если `False`, такая гиперссылка выводиться не будет. Значение по умолчанию — `False`;
- `min_num` — атрибут, указывает минимальное допустимое количество форм в редакторе в виде целого числа. Если задать `None`, редактор может включать сколько угодно форм. Значение по умолчанию — `None`;
- `get_min_num(`*<запрос>*`[, obj=None][, `*<дополнительные параметры>*`])` — метод, должен возвращать минимальное допустимое количество форм в редакторе. В необязательном параметре `obj` может быть получен объект исправляемой записи первичной модели или `None`, если такой записи еще нет. Назначение *дополнительных параметров* этого метода автором не установлено — вероятно, они оставлены на будущее. В изначальной реализации метод возвращает значение атрибута `min_num`;
- `max_num` — атрибут, указывает максимальное допустимое количество форм в редакторе в виде целого числа. Если задать `None`, редактор может включать сколько угодно форм. Значение по умолчанию — `None`;
- `get_max_num(`*<запрос>*`[, obj=None][, `*<дополнительные параметры>*`])` — метод, должен возвращать максимальное допустимое количество форм в редакторе. В необязательном параметре `obj` может быть получен объект исправляемой записи первичной модели или `None`, если такой записи еще нет. Назначение *дополнительных параметров* этого метода автором не установлено — вероятно, они оставлены на будущее. В изначальной реализации метод возвращает значение атрибута `max_num`;
- `classes` — атрибут, задает набор стилевых классов, которые будут привязаны к встроенному редактору. Значение указывается в виде списка или кортежа, содержащего строки с именами стилевых классов. Поддерживаются два стилевых класса: `collapse` (редактор, представленный в виде спойлера, будет изначально свернут) и `wide` (редактор займет все пространство окна по ширине). Значение по умолчанию — `None`;
- `verbose_name` — атрибут, задает название сущности, хранящейся в записи вторичной модели, в виде строки. Если задать значение `None`, будет использовано

название, записанное в одноименном параметре обслуживаемой редактором модели. Значение по умолчанию — None;

- verbose_name_plural — атрибут, задает название набора сущностей, хранящихся во вторичной модели, в виде строки. Если задать значение None, будет использовано название, записанное в одноименном параметре обслуживаемой редактором модели. Значение по умолчанию — None;
- formset — атрибут, указывает набор форм, связанный с моделью, на основе которого будет создан окончательный набор форм, выводимый на страницу. Значение по умолчанию — ссылка на класс BaseInlineFormSet;
- get_formset(*<запрос>*[, obj=None][, *<параметры набора форм>*]) — метод, должен возвращать класс встроенного набора форм, который будет использован в редакторе.

  В необязательном параметре obj может быть получен объект исправляемой записи первичной модели или None, если такой записи еще нет. А *параметры набора форм* будут переданы функции inlineformset_factory(), которая применяется методом для создания класса набора форм.

  В изначальной реализации метод возвращает класс встроенного набора форм, сгенерированный вызовом функции inlineformset_factory();
- form — атрибут, задает класс связанной с моделью формы, которая будет применяться в создаваемом наборе форм. Значение по умолчанию — ссылка на класс ModelForm.

26.4.3. Регистрация встроенного редактора

Встроенный редактор регистрируется в редакторе, обслуживающем первичную модель. Для этого служит атрибут inlines класса ModelAdmin. В качестве значения этого атрибута указывается список или кортеж, содержащий ссылки на классы всех встроенных редакторов, которые необходимо зарегистрировать в текущем редакторе. Значение атрибута inlines по умолчанию — «пустой» список.

Вот таким образом встроенный редактор BbInline, обслуживающий вторичную модель Bb, регистрируется в редакторе RubricAdmin, который обслуживает первичную модель:

```
class RubricAdmin(admin.ModelAdmin):
    . . .
    inlines = (BbInline,)
```

26.5. Действия

Действие в терминологии административного сайта Django — это операция, выбираемая из раскрывающегося списка **Действие** и выполняемая применительно к выбранным в списке записям. Список **Действие** находится над списком записей, а увидеть его можно на рис. 1.8 и 1.10.

Изначально в этом списке присутствует лишь действие **Удалить выбранные *<название набора сущностей>***, однако мы можем добавить туда свои собственные действия. Сделать это несложно.

Сначала необходимо объявить функцию, которая, собственно, и реализует действие. В качестве параметров она должна принимать:

- экземпляр класса, представляющий редактор, к которому будет привязано действие;
- запрос;
- набор записей, содержащий записи, которые были выбраны пользователем.

Никакого результата она возвращать не должна.

Далее мы укажем для действия название, которое будет выводиться в списке **Действие**. Строку с названием мы присвоим атрибуту `short_description` объекта функции, реализующей это действие.

По завершении выполнения действия, равно как и при возникновении ошибки, правилом хорошего тона считается отправка пользователю всплывающего сообщения (см. *разд. 22.3*). Сделать это можно вызовом метода `message_user()` класса `ModelAdmin`:

```
message_user(<запрос>, <текст сообщения>[, level=messages.INFO][,
                        extra_tags=''][, fail_silently=False])
```

Запрос представляется экземпляром класса `HttpRequest`, *текст сообщения* — строкой. Необязательный параметр `level` указывает уровень сообщения.

Необязательный параметр `extra_tags` указывает перечень дополнительных стилевых классов, привязываемых к тегу, в который будет заключен текст выводимого всплывающего сообщения. Перечень стилевых классов должен представлять собой строку, а стилевые классы в нем должны отделяться друг от друга пробелами.

Если задать для необязательного параметра `fail_silently` значение `True`, то в случае невозможности создания нового всплывающего сообщения (например, если соответствующая подсистема отключена) ничего не произойдет. Используемое по умолчанию значение `False` указывает в таком случае возбудить исключение `MessageFailure` из модуля `django.contrib.messages`.

В листинге 26.2 приведен код функции `discount()`, реализующей действие, которое уменьшит цены в выбранных объявлениях вдвое и уведомит о завершении всплывающим сообщением.

Листинг 26.2. Пример создания действия в виде функции

```
from django.db.models import F

def discount(modeladmin, request, queryset):
    f = F('price')
```

```
    for rec in queryset:
        rec.price = f / 2
        rec.save()
    modeladmin.message_user(request, 'Действие выполнено')
discount.short_description = 'Уменьшить цену вдвое'
```

Для регистрации действий в редакторе используется атрибут `actions`, поддерживаемый классом `ModelAdmin`. В качестве значения атрибута указывается список или кортеж, содержащий ссылки на функции, что реализуют регистрируемые в редакторе действия. Значение атрибута `actions` по умолчанию — «пустой» список.

Вот так мы можем зарегистрировать в редакторе `BbAdmin` созданное нами действие `discount()`:

```
class BbAdmin(admin.ModelAdmin):
    . . .
    actions = (discount,)
```

Мы можем реализовать действие в виде метода того же класса редактора, в котором хотим зарегистрировать это действие. Подобное действие объявляется и регистрируется так же, как действие-функция, но только имя метода в списке атрибута `actions` следует указать в виде строки.

Далее показан код действия, реализованного в виде метода `discount()` класса редактора `BbAdmin`.

```
class BbAdmin(admin.ModelAdmin):
    . . .
    actions = ('discount',)

    def discount(self, request, queryset):
        f = F('price')
        for rec in queryset:
            rec.price = f / 2
            rec.save()
        self.message_user(request, 'Действие выполнено')
    discount.short_description = 'Уменьшить цену вдвое'
```

На заметку

Django поддерживает создание более сложных действий, выводящих какие-либо промежуточные страницы (наподобие страницы подтверждения), доступных во всех редакторах, а также временное отключение действий и повторное их включение. Руководство по написанию таких действий находится здесь: **https://docs.djangoproject.com/en/2.1/ref/contrib/admin/actions/**.

ГЛАВА 27

Разработка веб-служб REST. Библиотека Django REST framework

Многие современные веб-сайты предоставляют программные интерфейсы, предназначенные для использования сторонними программами. Такими программами могут быть как обычные настольные или мобильные приложения, так и другие веб-сайты. С помощью подобных интерфейсов, называемых также *веб-службами*, они могут получать информацию или, наоборот, заносить ее на сайт.

Интерфейсы подобного рода строятся согласно принципам архитектуры *REST* (Representational State Transfer, репрезентативная передача состояния). К числу этих принципов относится, в частности, идентификация запрашиваемых ресурсов посредством обычных интернет-адресов. То есть для того чтобы получить список рубрик, сторонней программе достаточно лишь обратиться по интернет-адресу, скажем, **http://www.bboard.ru/api/rubrics/**.

Веб-сайт, предоставляющий программные интерфейсы, выступает в качестве сервера. Он отсылает клиентам данные, закодированные в каком-либо компактном формате, как правило, JSON. Клиенты получают эти данные и обрабатывают нужным им образом: выводят на экран, сохраняют в локальной базе данных и др.

Такой программный интерфейс можно реализовать исключительно средствами Django. Еще в *разд. 9.8.3* мы познакомились с классом `JsonResponse`, представляющим отправляемые клиенту данные в формате JSON. Однако для этих целей удобнее применить библиотеку Django REST framework. Она возьмет на себя львиную долю работы по извлечению данных из базы, кодированию данных в JSON, формированию ответа, приему данных, отправленных приложением, их валидации, занесению в базу и даже реализует разграничение доступа.

> ***Внимание!***
>
> Django REST framework — это даже не дополнительная библиотека, а полноценный фреймворк, предназначенный для разработки веб-служб REST, хоть и базирующийся на Django. Этому фреймворку впору посвящать отдельную книгу. Поэтому здесь будет приведен только вводный курс, описывающий лишь наиболее типичные возможности и способы применения Django REST framework.
>
> Полная документация по Django REST framework приведена на домашнем сайте этой библиотеки, находящемся по интернет-адресу **http://www.django-rest-framework.org/**.

Помимо всего прочего, там можно найти список дополнительных библиотек, расширяющих ее функциональность.

27.1. Установка и подготовка к работе Django REST framework

Установка этой библиотеки выполняется отдачей команды:

```
pip install djangorestframework
```

Изначально Django обрабатывает только те запросы, что пришли с того же домена, на котором располагается веб-служба. Однако, если мы хотим предоставить наш программный интерфейс любым сайтам и приложениям, нам понадобится разрешить фреймворку обрабатывать запросы, пришедшие с любых доменов. Проще всего сделать это, установив дополнительную библиотеку django-cors-headers, что выполняется командой:

```
pip install django-cors-headers
```

Программными ядрами библиотек Django REST framework и django-cors-headers являются приложения `rest_framework` и `corsheaders` соответственно. Нам необходимо добавить их в список зарегистрированных в проекте (параметр `INSTALLED_APPS` модуля settings.py из пакета конфигурации):

```
INSTALLED_APPS = [
    . . .
    'rest_framework',
    'corsheaders',
]
```

Кроме того, в список зарегистрированных в проекте (параметр `MIDDLEWARE`) нужно добавить посредник `corsheaders.middleware.CorsMiddleware`, расположив его перед посредником `django.middleware.common.CommonMiddleware`:

```
MIDDLEWARE = [
    . . .
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    . . .
]
```

Библиотека Django REST framework для успешной работы не требует обязательного указания каких-либо настроек. Настройки библиотеки django-cors-headers весьма немногочисленны:

- `CORS_ORIGIN_ALLOW_ALL` — если `True`, Django будет обрабатывать запросы, приходящие с любого домена. Если `False`, будут обрабатываться только запросы, пришедшие с текущего домена и с разрешенных доменов, указанных в параметрах `CORS_ORIGIN_WHITELIST` и `CORS_ORIGIN_REGEX_WHITELIST`. Значение параметра по умолчанию — `False`.

Параметр CORS_ORIGIN_WHITELIST задает перечень доменов, запросы с которых разрешено обрабатывать в виде кортежа. Параметр CORS_ORIGIN_REGEX_WHITELIST указывает регулярное выражение, с которым должен совпадать домен, чтобы пришедший с него запрос был обработан;

- CORS_URLS_REGEX — регулярное выражение, с которым должен совпадать путь, запрос по которому будет допущен к обработке. Значение по умолчанию — строка "^.*$" (регулярное выражение, совпадающее с любым путем).

Чтобы разрешить обработку запросов, приходящих с любых доменов, но только к тем путям, что включают префикс api, следует добавить в модуль settings.py пакета конфигурации такие выражения:

```
CORS_ORIGIN_ALLOW_ALL = True
CORS_URLS_REGEX = r'^/api/.*$'
```

> ***На заметку***
>
> Полное руководство по библиотеке django-cors-headers находится здесь: **https://github.com/ottoyiu/django-cors-headers/**.

27.2. Введение в Django REST framework. Вывод данных

Сначала мы познакомимся с самой этой библиотекой, узнаем принципы, на которых она основана, и напишем для нашей электронной доски объявлений простенький программный интерфейс, выдающий список всех рубрик.

27.2.1. Сериализаторы

Первое, что нам необходимо сделать для реализации какой-либо функции разрабатываемого программного интерфейса REST, — объявить сериализатор.

Сериализатор в Django REST framework выступает в той же роли, что и форма в Django. Он представляет извлеченные из базы сайта данные стороннему приложению, которое запрашивает их посредством обращения к программному интерфейсу. Сериализатор, как и форма, содержит объявления полей, либо записанные явно, либо созданные на основе полей связанной с ним модели.

Как и формы Django, сериализаторы могут быть как связанными с моделями, так и не связанными с ними. Первые самостоятельно извлекают данные из модели, а также «умеют» сохранять в модели данные, полученные от посетителя. Вторые не обладают такими возможностями, зато с их помощью можно обрабатывать произвольные данные.

Сериализатор — это обычный класс Python. Сериализатор, связанный с моделью, должен быть производным от класса ModelSerializer, объявленного в модуле rest_framework.serializers.

Код сериализаторов, по соглашению, обычно записывается в модуле serializers.py пакета приложения. Этот модуль изначально отсутствует, и нам придется создать его.

Давайте рассмотрим код из листинга 27.1, который объявляет сериализатор RubricSerializer, связанный с моделью Rubric и обрабатывающий рубрики. Он похож на код формы, связанной с моделью (более подробно о них рассказывалось в *главе 13*). Мы видим вложенный класс Meta, в котором с помощью знакомых нам атрибутов записаны параметры самого сериализатора: связанная с ним модель и набор полей модели, которые должны присутствовать в сериализаторе.

Листинг 27.1. Пример сериализатора

```
from rest_framework import serializers
from .models import Rubric

class RubricSerializer(serializers.ModelSerializer):
    class Meta:
        model = Rubric
        fields = ('id', 'name', 'order')
```

Давайте проверим наш первый сериализатор в действии. Добавим в модуль views.py код, объявляющий контроллер-функцию api_rubrics(), который будет выдавать клиентам список рубрик, закодированный в формат JSON. Код контроллера приведен в листинге 27.2.

Листинг 27.2. Пример контроллера, использующего сериализатор для вывода простого набора записей

```
from django.http import JsonResponse
from .models import Rubric
from .serializers import RubricSerializer

def api_rubrics(request):
    if request.method == 'GET':
        rubrics = Rubric.objects.all()
        serializer = RubricSerializer(rubrics, many=True)
        return JsonResponse(serializer.data, safe=False)
```

Конструктору класса сериализатора мы передали набор записей, который следует сериализовать, и параметр many со значением True, говоря тем самым, что сериализовать нужно именно набор записей, а не единичную запись. А для формирования ответа мы использовали класс JsonResponse.

Наконец, добавим в список маршрутов уровня приложения (модуль urls.py пакета приложения) маршрут, указывающий на только что написанный нами контроллер:

```
from .views import api_rubrics
. . .
urlpatterns = [
    . . .
    path('api/rubrics/', api_rubrics),
    . . .
]
```

Запустим отладочный веб-сервер Django, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost:8000/api/rubrics/**. Мы увидим на экране вот такие данные, представленные в формате JSON:

```
[{"id": 1, "name": "\u041d\u0435\u0434\u0432\u0438\u0436\u0438\u043c
\u043e\u0441\u0442\u044c", "order": 1},
{"id": 2, "name": "\u0422\u0440\u0430\u043d\u0441\u043f\u043e\u0440
\u0442", "order": 2},
. . . Остальной вывод пропущен . . .
]
```

Надо полагать, что это и есть список рубрик. Хотя проверить это крайне сложно из-за того, что класс `JsonResponse` представляет все символы кириллицы их Unicode-кодами.

27.2.2. Веб-представление JSON

Давайте дадим Django REST framework указание показывать нам JSON-данные в удобном для чтения виде. Иначе говоря, использовать для вывода таких данных *веб-представление JSON*.

Если задействовано веб-представление, данные JSON будут выводиться в виде обычной веб-страницы, причем отформатированными для удобства чтения и с некоторыми дополнительными сведениями. Если же нам зачем-то понадобится просмотреть «сырой» JSON-код, мы без проблем сможем добраться до него.

Задействовать веб-представление очень просто — для этого нужно всего лишь:

- указать у контроллера-функции декоратор `api_view(<допустимые HTTP-методы>)` из модуля `rest_framework.decorators`. *Допустимые HTTP-методы* записываются в виде списка, в котором приводятся наименования этих методов в виде строк;
- для формирования ответа вместо класса `JsonResponse` использовать класс `Response` из модуля `rest_framework.response`. Конструктор этого класса вызывается в формате: `Response(<отправляемые данные>)`.

Листинг 27.3 показывает полный код обновленной версии контроллера-функции `api_rubrics()`, которая реализует веб-представление.

Листинг 27.3. Пример контроллера, реализующего веб-представление

```
from rest_framework.response import Response
from rest_framework.decorators import api_view
```

```
from .models import Rubric
from .serializers import RubricSerializer

@api_view(['GET'])
def api_rubrics(request):
    if request.method == 'GET':
        rubrics = Rubric.objects.all()
        serializer = RubricSerializer(rubrics, many=True)
        return Response(serializer.data)
```

Сохраним исправленный код и попробуем наведаться по тому же интернет-адресу **http://localhost:8000/api/rubrics/**. На этот раз веб-обозреватель покажет нам веб-представление JSON (рис. 27.1).

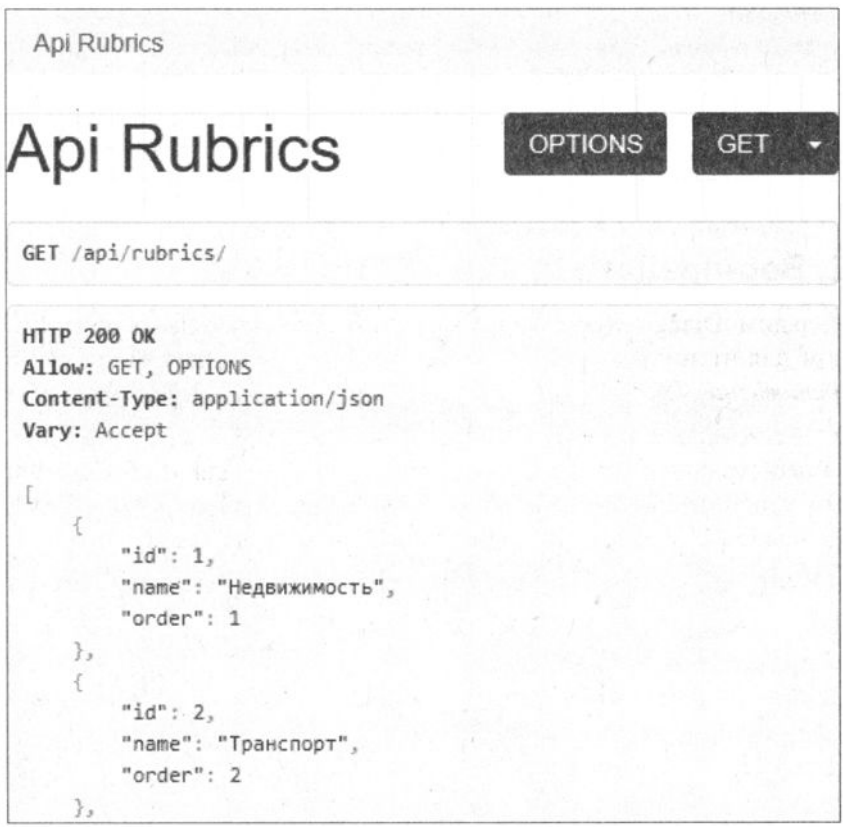


Рис. 27.1. Веб-представление JSON для модели рубрик `Rubric`

Здесь мы видим, прежде всего, сами JSON-данные, представленные в удобном для изучения виде, и сведения о полученном ответе (код статуса, MIME-тип содержимого и пр.). Кнопка **GET** позволит нам вывести обычный JSON-код — для этого достаточно щелкнуть на расположенной в ее правой части стрелке, направленной вниз, и выбрать в появившемся на экране меню пункт **json** (вернуть веб-пред-

ставление можно выбором в том же меню пункта **api** или нажатием непосредственно кнопки **GET**, не затрагивая стрелки). Кнопка **OPTIONS**, выводящая сведения о созданном нами программном интерфейсе, для нас не очень полезна.

27.2.3. Вывод данных на стороне клиента

Клиент, будь то обычное приложение или, как в нашем случае, сторонний веб-сайт, получает запрошенные данные в формате JSON. Эти данные нужно как-то декодировать и вывести на экран. К счастью, все современные веб-обозреватели поддерживают единообразный набор инструментов для загрузки с веб-сервера произвольных данных по технологии AJAX и их обработки, и эти инструменты просты в использовании.

> ***Внимание!***
>
> Веб-обозреватель может загружать произвольные данные по технологии AJAX только с веб-сервера. Загрузка данных с локального диска невозможна.
>
> Также нужно иметь в виду, что не все веб-обозреватели поддерживают загрузку данных по технологии AJAX страницей, открытой с локального диска. В частности, Google Chrome поддерживает это, чего нельзя сказать о Microsoft Edge. Поэтому, если загрузка данных почему-то не выполняется, имеет смысл попытаться установить программу веб-сервера и открыть страницу с него.

Сначала создадим обычную веб-страницу, чей код приведен в листинге 27.4, и сохраним ее под именем, скажем, rubrics.html.

Листинг 27.4. Веб-страница, получающая с сайта список рубрик

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8">
        <script src="rubrics.js"></script>
        <title>Список рубрик</title>
    </head>
    <body>
        <div id="list"></div>
    </body>
</html>
```

Блок (блочный контейнер, тег `<div>`) с именем `list` мы используем для вывода маркированного списка с перечнем рубрик.

После чего создадим файл rubrics.js с веб-сценарием из листинга 27.5. Сохраним этот файл в той же папке, что и страницу.

Листинг 27.5. Веб-сценарий, реализующий загрузку списка рубрик

```
var domain = 'http://localhost:8000/'

window.onload = function() {
    var list = document.getElementById('list');

    var rubricListLoader = new XMLHttpRequest()
    rubricListLoader.onreadystatechange = function() {
        if (rubricListLoader.readyState == 4) {
            if (rubricListLoader.status == 200) {
                var data = JSON.parse(rubricListLoader.responseText);
                var s = '<ul>';
                for (i = 0; i < data.length; i++) {
                    s += '<li>' + data[i].name + '</li>';
                }
                s += '</ul>'
                list.innerHTML = s;
            }
        }
    }

    function rubricListLoad() {
        rubricListLoader.open('GET', domain + 'api/rubrics/', true);
        rubricListLoader.send();
    }

    rubricListLoad();
}
```

Здесь нет ничего особо сложного. Мы создаем экземпляр объекта `XMLHttpRequest`, реализующий загрузку данных по технологии AJAX. С его помощью мы обращаемся методом GET к интернет-адресу **http://localhost:8000/api/rubrics/**, чтобы получить закодированный в формат JSON список рубрик. И не забываем привязать обработчик к событию `readystatechange`, возникающему при изменении состояния выполненного нами запроса.

В теле этого обработчика мы проверяем свойство `readyState`, где хранится состояние запроса. Если значение этого свойства равно 4 (т. е. ответ получен и обработан), мы проверяем код статуса ответа и, если он равен 200 (запрос успешно обработан), приступаем к обработке полученных данных.

Принятый список рубрик мы преобразуем (десериализуем) в соответствующий тип языка JavaScript — массив экземпляров объекта `Object` — вызовом статического метода `parse()` объекта `JSON`, встроенного в JavaScript. После чего нам останется только перебрать этот массив, сформировать на его основе маркированный список и вывести его на страницу в блоке `list`.

Отметим, что код, загружающий список рубрик, мы оформили в виде функции `rubricListLoad()`. Это позволит нам впоследствии, после добавления, правки или удаления рубрики, выполнить обновление списка простым вызовом этой функции.

Запустим отладочный веб-сервер Django и откроем нашу страничку rubrics.html в веб-обозревателе. На странице мы увидим перечень рубрик наподобие того, что показан на рис. 27.2.

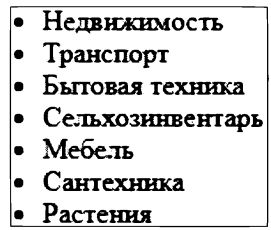


Рис. 27.2. Список рубрик, загруженный с сайта

27.2.4. Первый принцип REST: идентификация ресурса по интернет-адресу

Ранее говорилось о том, что, согласно принципам REST, идентификация ресурса, к которому обращается сторонний сайт или приложение, выполняется посредством указания его интернет-адреса. Как видим, здесь реализуется тот же самый принцип, что лежит в основе традиционных сайтов, в которых каждая страница также идентифицируется ее интернет-адресом.

В нашем случае для получения списка рубрик с нашего сайта стороннему приложению достаточно обратиться по интернет-адресу **/api/rubrics/**. Если же нужно получить сведения о выбранной рубрике, будет логично отправить запрос на интернет-адрес вида **/api/rubrics/*<ключ рубрики>*/**.

Сначала напишем контроллер, который будет отправлять клиенту сведения о рубрике с указанным ключом. Его код приведен в листинге 27.6.

Листинг 27.6. Контроллер, выводящий сведения об отдельной записи

```
@api_view(['GET'])
def api_rubric_detail(request, pk):
    if request.method == 'GET':
        rubric = Rubric.objects.get(pk=pk)
        serializer = RubricSerializer(rubric)
        return Response(serializer.data)
```

Здесь все нам уже знакомо по ранее написанным контроллерам. Единственное: в вызове конструктора класса сериализатора `RubricSerializer` мы уже не указываем параметр `many` со значением `True`, т. к. сериализовать нужно лишь одну запись.

Добавим в список маршрутов уровня приложения маршрут, который укажет на новый контроллер (добавленный код выделен полужирным шрифтом):

```
from .views import api_rubric_detail
. . .
urlpatterns = [
    . . .
    path('api/rubrics/<int:pk>/', api_rubric_detail),
    path('api/rubrics/', api_rubrics),
    . . .
]
```

На страницу rubrics.html, непосредственно под блоком `list`, поместим веб-форму, в которой будут выводиться сведения о выбранной посетителем рубрике. Эту форму мы потом используем для добавления и правки рубрик. Вот HTML-код, создающий ее:

```
<form id="rubric_form" method="post">
    <input type="hidden" name="id" id="id">
    <p>Название: <input name="name" id="name"></p>
    <p>Порядок:
    <input type="number" name="order" id="order" value="0"></p>
    <p><input type="reset" value="Очистить">
    <input type="submit" value="Сохранить"></p>
</form>
```

Обратим внимание, что мы предусмотрели скрытое поле `id` для хранения ключа исправляемой рубрики. Он нам понадобится впоследствии.

Теперь исправим веб-сценарий, хранящийся в файле rubrics.js. Сначала сделаем так, чтобы рядом с названием каждой рубрики выводилось значение ее поля `order` и гиперссылка, запускающая загрузку и вывод сведений о рубрике в только что созданной веб-форме. Вот правки, которые нам нужно внести в код (добавленный и исправленный код выделен полужирным шрифтом):

```
rubricListLoader.onreadystatechange = function() {
    . . .
        if (rubricListLoader.status == 200) {
            . . .
            for (i = 0; i < data.length; i++) {
                d = data[i];
                detail_url = '<a href="' + domain + 'api/rubrics/' +
                d.id + '/" class="detail">Вывести</a>';
                s += '<li>' + d.name + ' (' + d.order + ') [' +
                detail_url + ']</li>';
            }
            s += '</ul>'
            list.innerHTML = s;
            links = list.querySelectorAll('ul li a.detail');
```

```
            for (var i = 0; i < links.length; i++) {
                links[i].addEventListener('click', rubricLoad);
            }
        } else {
            . . .
        }
        . . .
}
```

Приведенный здесь код не требует особых пояснений, за исключением трех моментов. Во-первых, мы привязали к каждой из созданных гиперссылок стилевой класс `detail` — это упростит нам задачу привязки к гиперссылкам обработчика события `click`. Во-вторых, мы записали интернет-адреса для загрузки рубрик непосредственно в тегах `<a>`, создающих гиперссылки, — это также упростит нам дальнейшее программирование. В-третьих, мы привязали к созданным гиперссылкам обработчик события `click`, который загрузит с сайта сведения о гиперссылке.

Рассмотрим сам этот обработчик события (добавленный код выделен полужирным шрифтом):

```
window.onload = function() {
    var id = document.getElementById('id');
    var name = document.getElementById('name');
    var order = document.getElementById('order');

    var rubricLoader = new XMLHttpRequest()
    rubricLoader.onreadystatechange = function () {
        if (rubricLoader.readyState == 4) {
            if (rubricLoader.status == 200) {
                var data = JSON.parse(rubricLoader.responseText);
                id.value = data.id;
                name.value = data.name;
                order.value = data.order;
            } else {
                window.alert(rubricLoader.statusText);
            }
        }
    }

    function rubricLoad(evt) {
        evt.preventDefault();
        var url = evt.target.href;
        rubricLoader.open('GET', url, true);
        rubricLoader.send();
    }
    . . .
}
```

В качестве обработчика события выступает функция `rubricLoad()`. Она извлекает из атрибута `href` тега `<a>` гиперссылку, на которой был выполнен щелчок мышью,

интернет-адрес и запускает процесс загрузки с этого адреса сведений о рубрике. Также она вызовом метода `preventDefault()` события подавляет выполнение обработчика по умолчанию, чтобы веб-обозреватель не выполнил незапланированный переход по гиперссылке.

Загруженные с сайта сведения о выбранной посетителем рубрике выводятся в веб-форме функцией-обработчиком события `readystatechange`. В ее коде все нам уже знакомо.

Запустим отладочный веб-сервер Django и откроем страницу rubrics.html. Дождемся, когда на ней появится список рубрик, и щелкнем на гиперссылке **Вывести** любой рубрики. Если мы не допустили ошибок в коде, в веб-форме должны появиться сведения об этой рубрике.

27.3. Ввод и правка данных

Что ж, реализовывать отправку данных клиенту, пользуясь средствами библиотеки Django REST framework, мы научились (как и выводить эти данные на экран в клиентских приложениях, для чего нам пришлось оперативно освежать знания о JavaScript). И выяснили, что с применением этой библиотеки такие действия выполняются буквально несколькими строчками программного кода.

А что же ввод и правка данных? Поможет ли Django REST framework нам и в этом случае? Безусловно!

27.3.1. Второй принцип REST: идентификация действия по HTTP-методу

Мы уже знаем, что ресурс, который обслуживается веб-службой REST и к которому обращается клиентское приложение, однозначно идентифицируется его интернет-адресом. Так, для получения списка рубрик мы обратимся к интернет-адресу **/api/rubrics/**, а для получения рубрики с ключом 4 — к интернет-адресу **/api/rubrics/4/**.

Но как идентифицировать действие, которое клиентское приложение собирается произвести с ресурсом? Как указать, желает оно загрузить список каких-либо позиций или добавить новую позицию?

А делается это посредством указания соответствующего HTTP-метода. Для взаимодействия между клиентскими приложениями и веб-службами REST применяются следующие методы:

- GET — загрузка либо списка сущностей, либо сведений об отдельной сущности (в зависимости от интернет-адреса, по которому выполняется обращение);
- POST — создание новой сущности;
- PUT — исправление значений всех полей у имеющейся сущности. Отметим, что при использовании этого метода клиентское приложение должно отправить веб-службе значения всех полей, в противном случае возникнет ошибка;

- PATCH — исправление отдельных полей у имеющейся сущности. В этом случае клиентское приложение может отправить веб-службе значения только тех полей, которые нужно исправить.

  Часто в веб-службе реализуют только поддержку метода PUT. При получении запроса, отправленного методом PATCH, выполняется тот же код, что реализует поддержку метода PUT;
- DELETE — удаление сущности.

Обычно создание новой сущности реализует тот же контроллер, что выдает список сущностей, а исправление и удаление — тот же контроллер, что выдает сведения об отдельной сущности. При этом отпадает необходимость в создании отдельного контроллера для обработки каждой операции над хранящимися в базе сущностями (соответственно, каждого из приведенных HTTP-методов) и отдельного маршрута для каждого из этих контроллеров.

В листинге 27.7 приведен исправленный код контроллеров `api_rubrics()` и `api_rubric_detail()`, которые получили поддержку добавления, правки и удаления рубрик.

Листинг 27.7. Контроллеры, поддерживающие добавление, правку и удаление записей

```
from rest_framework import status

@api_view(['GET', 'POST'])
def api_rubrics(request):
    if request.method == 'GET':
        rubrics = Rubric.objects.all()
        serializer = RubricSerializer(rubrics, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = RubricSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
                            status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)

@api_view(['GET', 'PUT', 'PATCH', 'DELETE'])
def api_rubric_detail(request, pk):
    rubric = Rubric.objects.get(pk=pk)
    if request.method == 'GET':
        serializer = RubricSerializer(rubric)
        return Response(serializer.data)
    elif request.method == 'PUT' or request.method == 'PATCH':
        serializer = RubricSerializer(rubric, data=request.data)
```

```
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'DELETE':
        rubric.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

Как видим, сохранение и удаление записей с помощью сериализаторов, связанных с моделями, выполняется точно так же, как и в случае применения связанных с моделями форм. Мы вызываем методы `is_valid()`, `save()` и `delete()`, а все остальное выполняют фреймворк Django и библиотека Django REST framework.

Теперь отметим три важных момента. Прежде всего, мы добавили в вызовы декоратора `api_view()` обозначения HTTP-методов POST, PUT, PATCH и DELETE, указав тем самым, что контроллеры поддерживают запросы, выполненные с применением этих методов. Если этого не сделать, мы получим ошибку.

Далее, чтобы занести в сериализатор данные, полученные от клиента, мы присваиваем их именованному параметру `data` конструктора класса сериализатора. Сами эти данные можно извлечь из атрибута `data` объекта запроса. Вот пример:

```
serializer = RubricSerializer(data=request.data)
```

Наконец, после успешного добавления рубрики в базу мы отсылаем клиенту ответ с кодом `201` (сущность успешно добавлена). Это выполняется передачей конструктору класса `Response` числового кода статуса посредством именованного параметра `status`:

```
return Response(serializer.data, status=status.HTTP_201_CREATED)
```

При успешном исправлении рубрики отправленный клиенту ответ будет иметь код статуса по умолчанию: `200`. После успешного удаления рубрики клиент получит ответ с кодом статуса `204` (сущность отсутствует, т. е. была благополучно удалена). Если же занесенные в форму посетителем данные некорректны, клиентское приложение получит ответ с числовым кодом `400` (некорректно сформированный запрос).

Указанные коды статуса хранятся в переменных `HTTP_201_CREATED`, `HTTP_204_NO_CONTENT` и `HTTP_400_BAD_REQUEST`, объявленных в модуле `rest_framework.status`.

Настало время написать код, выполняющий добавление и исправление записи на стороне клиента. Откроем файл rubrics.js и вставим в него следующий код (выделен полужирным шрифтом):

```
window.onload = function() {
    . . .
    var rubricUpdater = new XMLHttpRequest()
    rubricUpdater.onreadystatechange = function() {
        if (rubricUpdater.readyState == 4) {
            if ((rubricUpdater.status == 200) ||
```

```
                (rubricUpdater.status == 201)) {
                rubricListLoad();
                rubricForm.reset();
                id.value = '';
            } else {
                window.alert(rubricUpdater.statusText);
            }
        }
    }

    var rubricForm = document.getElementById('rubric_form');
    rubricForm.addEventListener('submit', function (evt) {
        evt.preventDefault();
        var vid = id.value;
        if (vid) {
            var url = 'api/rubrics/' + vid + '/';
            var method = 'PUT';
        } else {
            var url = 'api/rubrics/';
            var method = 'POST';
        }
        data = JSON.stringify({id: vid, name: name.value,
                               order: order.value});
        rubricUpdater.open(method, domain + url, true);
        rubricUpdater.setRequestHeader('Content-Type',
                                       'application/json');
        rubricUpdater.send(data);
    });
}
```

Сначала рассмотрим обработчик события `submit` веб-формы. Он проверяет, хранится ли в скрытом поле `id` ключ рубрики. Если скрытое поле хранит ключ, значит, выполняется правка уже имеющей рубрики, в противном случае в список добавляется новая рубрика. Исходя из этого, вычисляется интернет-адрес, по которому следует отправить введенные в форму данные, и выбирается HTTP-метод, применяемый для отсылки запроса. После этого введенные данные кодируются в формат JSON (для чего они сначала представляются в виде экземпляра объекта `Object`, а потом «пропускаются» через статический метод `stringify()` объекта `JSON`) и отсылаются веб-службе.

Но перед отправкой мы в обязательном порядке указываем тип отсылаемых данных: `application/json` — записав его в заголовке `Content-Type` запроса (для чего используем метод `setRequestHeader()` объекта `XMLHttpRequest`). Это нужно для того, чтобы Django REST framework выяснила, какой парсер следует применить для разбора этих данных (о парсерах мы поговорим позже).

В обработчике события `readystatechange` мы после ответа с кодом `200` или `201` (т. е. после успешного исправления или добавления рубрики) обновляем список рубрик, очищаем форму и заносим в скрытое поле `id` веб-формы «пустую» строку. Так мы,

с одной стороны, сообщим пользователю, что все прошло нормально, а с другой, подготовим форму для ввода новой рубрики.

Пока код из файла rubrics.js у нас перед глазами, внесем в него правки, реализующие удаление рубрик. Сначала добавим в код, выводящий список рубрик, фрагмент, который создаст гиперссылки для удаления рубрик (добавленный и исправленный код выделен полужирным шрифтом):

```
rubricListLoader.onreadystatechange = function() {
    . . .
        if (rubricListLoader.status == 200) {
            . . .
            for (i = 0; i < data.length; i++) {
                d = data[i];
                detail_url = '<a href="' + domain + 'api/rubrics/' +
                d.id + '/" class="detail">Вывести</a>';
                delete_url = '<a href="' + domain + 'api/rubrics/' +
                d.id + '/" class="delete">Удалить</a>';
                s += '<li>' + d.name + ' (' + d.order + ') [' +
                detail_url + ' ' + delete_url + ']</li>';
            }
            . . .
            links = list.querySelectorAll('ul li a.delete');
            for (var i = 0; i < links.length; i++) {
                links[i].addEventListener('click', rubricDelete);
            }
        } else {
            . . .
        }
    . . .
}
```

Он аналогичен тому, что мы написали в *разд. 27.2.4*. К созданным гиперссылкам привязывается обработчик события `click` — функция `rubricDelete()`.

Объявим эту функцию и напишем сопутствующий код (выделен полужирным шрифтом):

```
window.onload = function() {
    . . .
    var rubricDeleter = new XMLHttpRequest()
    rubricDeleter.onreadystatechange = function () {
        if (rubricDeleter.readyState == 4) {
            if (rubricDeleter.status == 204) {
                rubricListLoad();
            } else {
                window.alert(rubricDeleter.statusText);
            }
        }
    }
```

```
    function rubricDelete(evt) {
        evt.preventDefault();
        var url = evt.target.href;
        rubricDeleter.open('DELETE', url, true);
        rubricDeleter.send();
    }
    . . .
}
```

Этот код также аналогичен написанному нами ранее. Единственное исключение: мы ожидаем ответа с кодом `204`, дабы удостовериться, что рубрика была успешно удалена.

Перезапустим отладочный веб-сервер, обновим страницу rubrics.html в веб-обозревателе и проверим, как все работает.

27.3.2. Парсеры веб-форм

Получив от клиента данные, занесенные посетителем в форму или сформированные программно, библиотека Django REST framework пытается разобрать их. Для этого она использует наиболее подходящий для этого парсер.

Парсер — это класс, выполняющий разбор переданных клиентом данных и их преобразование в объекты языка Python, пригодные для дальнейшей обработки. Парсер задействуется программным ядром библиотеки перед вызовом контроллера — таким образом, последний получит уже обработанные данные.

В составе Django REST framework поставляются три наиболее интересных для нас класса парсеров, каждый из которых обрабатывает данные, представленные в определенном формате:

- `JSONParser` — обрабатывает данные, представленные в формате JSON (MIME-тип `application/json`);
- `FormParser` — обрабатывает данные из обычных веб-форм (MIME-тип `application/x-www-form-urlencoded`);
- `MultiPartParser` — обрабатывает данные из веб-форм, выгружающих файлы (MIME-тип `multipart/form-data`).

По умолчанию активны все эти три класса парсеров. Библиотека выбирает нужный парсер, основываясь на MIME-типе переданных клиентом данных, который записывается в заголовке `Content-Type` запроса. Именно поэтому на стороне клиента перед отправкой данных необходимо указать их MIME-тип.

Ранее мы пересылали от клиента данные, закодированные в формате JSON, и обрабатывались они парсером `JSONParser`. Однако мы можем переслать данные в формате обычных форм:

```
data = 'id=' + encodeURIComponent(vid);
data += '&name=' + encodeURIComponent(name.value);
```

```
data += '&order=' + encodeURIComponent(order.value);
rubricUpdater.setRequestHeader('Content-Type',
                               'application/x-www-form-urlencoded');
```

Такие данные будут обработаны парсером `FormParser`.

27.4. Контроллеры-классы Django REST framework

Все написанные нами к настоящему моменту контроллеры были реализованы в виде функций, т. е. представляли собой контроллеры-функции. Однако библиотека Django REST framework позволяет использовать и контроллеры-классы различного уровня сложности.

27.4.1. Контроллер-класс низкого уровня

Контроллер-класс `APIView` из модуля `rest_framework.views` работает на самом низком уровне. Он всего лишь, как и аналогичный ему класс `View` (см. *разд. 10.2.1*) — кстати, его суперкласс, — осуществляет диспетчеризацию по HTTP-методу: при получении запроса по методу GET вызывает метод `get()`, при получении запроса по методу POST — метод `post()` и т. д. Если был получен запрос по методу PATCH, а метод `patch()` отсутствует, будет выполнен метод `put()`.

Класс `APIView` реализует веб-представление (см. *разд. 27.2.2*) самостоятельно, так что декоратор `api_view()` указывать не нужно.

Листинг 27.8 показывает код контроллера-класса `APIRubrics`, являющегося производным от `APIView` и выполняющего вывод списка рубрик и добавление новой рубрики.

Листинг 27.8. Пример использования класса `APIView`

```
from rest_framework.views import APIView

class APIRubrics(APIView):
    def get(self, request):
        rubrics = Rubric.objects.all()
        serializer = RubricSerializer(rubrics, many=True)
        return Response(serializer.data)
    def post(self, request):
        serializer = RubricSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
                            status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)
```

Как видим, код этого класса очень похож на код контроллеров-функций из листинга 27.7, выполняющих те же задачи. Так что контроллер-класс `APIRubricDetail`, который реализует вывод сведений о выбранной рубрике, правку и удаление рубрик, вы, уважаемые читатели, можете написать самостоятельно.

27.4.2. Контроллеры-классы высокого уровня: комбинированные и простые

При выполнении специфических задач не обойтись без низкоуровневых классов, поскольку они позволяют нам реализовать все, что угодно. Но если нужно выполнять какие-либо типовые действия (выводить список сущностей, сведения об отдельной сущности, добавлять, править и удалять сущности), удобнее применить классы высокого уровня — они все делают сами.

Django REST framework предоставляет нам целый набор таких высокоуровневых классов, объявленных в модуле `rest_framework.generics`. Прежде всего, это четыре комбинированных контроллера-класса, которые могут выполнять сразу два или три действия, в зависимости от HTTP-метода, которым был отправлен запрос:

- `ListCreateAPIView` — выполняет вывод перечня сущностей и создание новой сущности (т. е. обрабатывает методы GET и POST);
- `RetrieveUpdateDestroyAPIView` — выполняет вывод сведений об отдельной сущности, правку и удаление сущностей (обрабатывает методы GET, PUT, PATCH и DELETE);
- `RetrieveUpdateAPIView` — выполняет вывод сведений об отдельной сущности и правку сущностей (обрабатывает методы GET, PUT и PATCH);
- `RetrieveDestroyAPIView` — выполняет вывод сведений об отдельной сущности и удаление сущностей (обрабатывает методы GET и DELETE).

Как минимум, в таких классах нужно задать набор записей, который будет обрабатываться, и сериализатор, что будет применяться для пересылки данных клиенту. Набор записей указывается в атрибуте `queryset`, а сериализатор — в атрибуте `serializer_class`.

Листинг 27.9 показывает новую реализацию контроллеров-классов `APIRubrics` и `APIRubricDetail` — основанную на классах `ListCreateAPIView` и `RetrieveUpdateDestroyAPIView`.

Листинг 27.9. Пример использования классов `ListCreateAPIView` и `RetrieveUpdateDestroyAPIView`

```
from rest_framework import generics

class APIRubrics(generics.ListCreateAPIView):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer
```

```
class APIRubricDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer
```

Если интерфейс веб-службы предусматривает выполнение только какого-либо одного действия (например, вывода списка рубрик), использование комбинированных классов будет избыточным. В таких случаях удобнее задействовать более простые классы, выполняющие только одно действие:

- `ListAPIView` — выполняет вывод перечня сущностей (т. е. обрабатывает метод GET);
- `RetrieveAPIView` — выполняет вывод сведений об отдельной сущности (обрабатывает метод GET);
- `CreateAPIView` — выполняет создание новой сущности (метод POST);
- `UpdateAPIView` — выполняет правку сущностей (методы PUT и PATCH);
- `DestroyAPIView` — выполняет удаление сущностей (метод DELETE).

Используются они точно так же, как и комбинированные контроллеры-классы. Например, код контроллера, выводящего список рубрик, может быть таким:

```
class APIRubricList(generics.ListAPIView):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer
```

27.5. Метаконтроллеры

Высокоуровневые комбинированные контроллеры-классы значительно упрощают реализацию элементарных действий. Но Django REST framework идет еще дальше, предлагая метаконтроллеры (в оригинальной документации — viewsets).

Метаконтроллер можно рассматривать как комбинированный контроллер-класс, реализующий всю необходимую функциональность по выводу списка сущностей, сведений об отдельной сущности, добавлению, правке и удалению сущностей. Как видим, он может заменить два обычных комбинированных контроллера-класса высокого уровня, наподобие показанных в листинге 27.9, и сам по себе представляет полноценный программный интерфейс. Помимо этого, метаконтроллер предоставляет средства для генерирования маршрутов, указывающих на отдельные функции этого интерфейса.

Метаконтроллер, связанный с моделью, создается как подкласс класса `ModelViewSet`, объявленного в модуле `rest_framework.viewsets`. В нем с применением атрибутов `queryset` и `serializer_class` указываются набор записей, с которым будет выполняться работа, и сериализатор, управляющий отправкой данных клиенту.

Код метаконтроллера `APIRubricViewSet`, работающего со списком рубрик, показан в листинге 27.10. Как видим, он очень прост и компактен.

Листинг 27.10. Пример метаконтроллера

```
from rest_framework.viewsets import ModelViewSet

class APIRubricViewSet(ModelViewSet):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer
```

Чтобы сгенерировать набор маршрутов, которые указывают на отдельные функции метаконтроллера, нужно выполнить три действия:

- получить объект *генератора* таких *маршрутов*, представляющий собой экземпляр класса `DefaultRouter` из модуля `rest_framework.routers`. Конструктор этого класса вызывается без параметров;
- зарегистрировать в генераторе маршрутов метаконтроллер, связав его с выбранным префиксом. Это выполняется вызовом метода `register()` класса `DefaultRouter` с передачей ему префикса и ссылки на класс метаконтроллера;
- добавить сгенерированные маршруты в список уровня приложения или проекта, воспользовавшись известной нам по *разд. 8.3* функцией `include()`. Сами маршруты можно извлечь из атрибута `urls` генератора маршрутов.

Вот пример генерирования набора маршрутов для метаконтроллера `APIRubricViewSet` из листинга 27.10:

```
from rest_framework.routers import DefaultRouter
from django.urls import path, include

router = DefaultRouter()
router.register('rubrics', APIRubricViewSet)

urlpatterns = [
    . . .
    path('api/', include(router.urls)),
    . . .
]
```

В результате в список будут добавлены два следующих маршрута:

- **api/rubrics/** — выполняет при запросе с применением HTTP-метода:
  - GET — извлечение списка рубрик;
  - POST — добавление новой рубрики;
- **api/rubrics/<*ключ*>** — выполняет при запросе с применением HTTP-метода:
  - GET — извлечение рубрики с указанным *ключом*;
  - PUT или PATCH — правку рубрики с указанным *ключом*;
  - DELETE — удаление рубрики с указанным *ключом*.

Помимо класса `ModelViewSet`, библиотека Django REST framework предлагает разработчикам класс `ReadOnlyModelViewSet`, объявленный в том же модуле `rest_framework.viewsets`. Он реализует функциональность только по выводу списка сущностей и отдельной сущности и подходит для случаев, когда необходимо дать сторонним приложениям возможность только получения данных с сайта, но не их добавления и правки. Пример метаконтроллера, обрабатывающего список рубрик и позволяющего только считывать данные, показан в листинге 27.11.

Листинг 27.11. Пример метаконтроллера, реализующего только считывание данных

```
from rest_framework.viewsets import ReadOnlyModelViewSet

class APIRubricViewSet(ReadOnlyModelViewSet):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer
```

27.6. Разграничение доступа

Django REST framework предлагает весьма развитые средства для разграничения доступа, использующие в работе подсистему разграничения доступа Django. Мы можем указывать, кому предоставляется доступ к тому или иному ресурсу: всем пользователям, включая гостей, только зарегистрированным пользователям, пользователям со статусом персонала или же пользователям, имеющим права на работу с какой-либо моделью.

27.6.1. Третий принцип REST: данные клиента хранятся на стороне клиента

Когда в *главах 1* и *2* мы разрабатывали наш тестовый сайт, то сразу заметили одну примечательную вещь. При выполнении входа на сайт мы вводили имя и пароль пользователя только однажды — в веб-форме на странице входа. Впоследствии, чтобы получить доступ к административному сайту Django, мы не заносили их повторно.

Дело в том, что в сайтах, построенных по традиционной архитектуре, данные клиента хранятся на стороне сервера. Как только клиент выполнил вход на сайт, на сервере, в составе данных сессии, сохраняется ключ записи, хранящей сведения об этом пользователе. Именно поэтому для посещения страниц, доступных только для зарегистрированных пользователей, клиенту не нужно вводить свои имя и пароль снова и снова.

Но в веб-службах REST данные клиента не сохраняются на стороне сервера — они должны храниться самим клиентом. Применительно к нашему сайту, сам клиент должен сохранить где-то имя и пароль, введенные пользователем, и потом отсылать их серверу в каждом запросе, чтобы сервер смог проверить, имеет ли этот пользователь права на доступ к данным.

Имя и пароль пользователя можно сохранить в локальном хранилище DOM, cookie или, в конце концов, в обычной переменной. Но как передать их серверу? В заголовке `Authorization`, значением которого должна быть строка вида "`Basic <имя и пароль>`", где *имя и пароль* представляются в формате `<имя>:<пароль>` и кодируются в кодировке base64. Для кодирования можно использовать метод `btoa()`, поддерживаемый окном веб-обозревателя.

Вот пример подготовки имени и пароля пользователя к отправке серверу и самой их отправки:

```
var username = 'editor';
var password = '1988win1993';
var credentials = window.btoa(username + ':' + password);
. . .
rubricListLoader.open('GET', domain + 'api/rubrics/', true);
rubricListLoader.setRequestHeader('Authorization',
                                  'Basic ' + credentials);
rubricListLoader.send();
```

Таким образом реализуется *основная аутентификация* (basic authentication), при которой в каждом клиентском запросе серверу пересылаются непосредственно имя и пароль пользователя.

27.6.2. Классы разграничения доступа

Если с клиентом придется повозиться, реализуя получение от пользователя его имени и пароля и их сохранение, то на стороне сервера нам понадобится добавить в контроллер всего одну строчку кода. А именно, указать для контроллера перечень классов, реализующих разграничение доступа.

Классы разграничения доступа объявлены в модуле `rest_framework.permissions`. Наиболее часто используемые из них приведены далее:

- `AllowAny` — разрешает доступ к данным всем — и зарегистрированным пользователям, и гостям. Этот класс используется по умолчанию;
- `IsAuthenticated` — разрешает доступ к данным только зарегистрированным пользователям;
- `IsAuthenticatedOrReadOnly` — предоставляет полный доступ к данным только зарегистрированным пользователям, гости получат доступ лишь на чтение;
- `IsAdminUser` — разрешает доступ к данным только зарегистрированным пользователям со статусом персонала;
- `DjangoModelPermissions` — разрешает доступ к данным только зарегистрированным пользователям, имеющим необходимые права на работу с этими данными (о правах пользователей рассказывалось в *главе 15*);
- `DjangoModelPermissionsOrAnonReadOnly` — предоставляет полный доступ к данным только зарегистрированным пользователям, имеющим необходимые права на работу с ними. Все прочие пользователи, включая гостей, получат доступ только на чтение.

Перечень классов разграничения доступа записывается в виде кортежа. Указать его можно:

- в контроллере-функции — с помощью декоратора `permission_classes(<перечень классов>)`, объявленного в модуле `rest_framework.decorators`:

```
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated

@api_view(['GET', 'POST'])
@permission_classes((IsAuthenticated,))
def api_rubrics(request):
    . . .
```

- в контроллере-классе — в атрибуте `permission_classes`:

```
from rest_framework.permissions import IsAuthenticated

class APIRubricViewSet(ModelViewSet):
    . . .
    permission_classes = (IsAuthenticated,)
```

Также имеется возможность задать перечень классов разграничения доступа, используемый по умолчанию всеми контроллерами, что присутствуют в проекте. Этот перечень указывается в настройках проекта (в модуле settings.py пакета конфигурации) следующим образом:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    )
}
```

Чтобы пользователь получил доступ к данным, он должен «пройти» через все классы разграничения доступа, указанные в перечне. Можно также сказать, что ограничения, накладываемые этими классами, объединяются по правилу логического И. Так, если указать классы `IsAdminUser` и `DjangoModelPermissions`, доступ к данным получат только пользователи со статусом персонала, имеющие необходимые права на работу с этими данными.

> ***На заметку***
>
> Библиотека Django REST framework поддерживает множество других программных инструментов: сериализаторы, не связанные с моделями, иные способы аутентификации (жетонную, при которой от клиента серверу пересылаются не имя и пароль пользователя, а однозначно идентифицирующий его электронный жетон, и сессионную, традиционную, при которой сведения о клиенте сохраняются на стороне сервера), дополнительные классы разграничения доступа и т. п. К сожалению, ограниченный объем книги не позволяет рассказать обо всем этом.

ГЛАВА 28

Средства диагностики и отладки

На протяжении целых 27 глав мы говорим о разработке сайтов с применением фреймворка Django. А ведь в процессе разработки необходимо выполнять отладку написанного кода с целью выяснить, работает ли он, и если не работает, то по какой причине. Для этого нам понадобятся специализированные инструменты.

Аналогичные инструменты пригодятся и для диагностики работы уже написанного и опубликованного в Интернете сайта. Нам придется время от времени проверять, не содержит ли код сайта не найденные в процессе разработки ошибки, не возникают при его функционировании нештатные ситуации.

Эта глава будет посвящена средствам диагностики и отладки, предоставляемым Django.

28.1. Средства диагностики

Средства диагностики применяются как в процессе отладки, для выявления ошибок, так и для слежения за работой уже опубликованного сайта.

28.1.1. Настройка средств диагностики

Настройки средств диагностики записываются в модуле settings.py пакета конфигурации. Все они указываются в параметре `LOGGING` в виде словаря, ключи элементов которого указывают названия различных параметров, а значения элементов задают значения этих параметров.

Вот параметры средств диагностики, доступные для указания:

- `version` — номер версии стандарта, в котором записываются настройки средств диагностики, в виде целого числа. На данный момент поддерживается только версия 1;
- `formatters` — перечень доступных для использования форматировщиков. *Форматировщик* определяет формат, в котором будет представлено каждое сообще-

ние об ошибке, нештатной ситуации, предупреждение или информационное сообщение, выводимое средствами диагностики;

- `filters` — перечень доступных для использования фильтров сообщений. *Фильтры* отбирают для вывода только те сообщения, что удовлетворяют определенным условиям, или же выводят сообщения, основываясь на выполнении или невыполнении каких-либо условий, не относящихся к самим сообщениям;
- `handlers` — перечень доступных обработчиков. *Обработчики* непосредственно выполняют вывод сообщений определенным способом (на консоль, в файл, по электронной почте и др.) в формате, заданном форматировщиком, возможно, с использованием фильтров;
- `loggers` — перечень доступных для использования регистраторов. *Регистратор* собирает все сообщения, отправленные какой-либо подсистемой Django или сразу несколькими подсистемами, и передает их указанным обработчикам для вывода;
- `disable_existing_loggers` — если `True`, средства диагностики, используемые по умолчанию, работать не будут, если `False` — будут. Значение по умолчанию — `True`.

28.1.2. Объект сообщения

Сообщение, формируемое и выводимое средствами диагностики Django, представляется в виде экземпляра класса `LogRecord` из модуля `logging`. Этот модуль и класс принадлежат не Django, а стандартной библиотеке языка Python.

Давайте рассмотрим атрибуты класса `LogRecord`, хранящие полезную для нас информацию о сообщении:

- `message` — текст сообщения в виде строки;
- `levelname` — обозначение уровня сообщения в виде строки: `"DEBUG"`, `"INFO"`, `"WARNING"`, `"ERROR"` или `"CRITICAL"`. Все эти уровни сообщений описаны в табл. 22.1;
- `levelno` — обозначение уровня сообщения в виде целого числа;
- `pathname` — полный путь выполняемого в данный момент файла в виде строки;
- `filename` — имя выполняемого в данный момент файла в виде строки;
- `module` — имя выполняемого в данный момент модуля, полученное из имени файла путем удаления у него расширения, в виде строки;
- `lineno` — порядковый номер выполняемой в данный момент строки программного кода в виде целого числа;
- `funcName` — имя выполняемой в данный момент функции в виде строки;
- `asctime` — дата и время создания сообщения в виде строки;
- `created` — дата и время создания сообщения в виде вещественного числа, представляющего собой количество секунд, что прошли с полуночи 1 января

1970 года. Для формирования этой величины применяется функция `time()` из модуля `time` Python;

- `msecs` — миллисекунды из времени создания сообщения в виде целого числа;
- `relativeCreated` — количество миллисекунд, прошедших между запуском регистратора и созданием текущего сообщения, в виде целого числа;
- `exc_info` — кортеж из трех значений: ссылки на класс исключения, самого объекта исключения и объекта, хранящего стек вызова. Для формирования этого кортежа применяется функция `exc_info()` из модуля `sys` Python;
- `stack_info` — объект, хранящий стек вызовов;
- `process` — идентификатор процесса в виде целого числа (если таковой удается определить);
- `processName` — имя процесса в виде строки (если таковое удается определить);
- `thread` — идентификатор потока в виде целого числа (если таковой удается определить);
- `threadName` — имя потока в виде строки (если таковое удается определить);
- `name` — имя регистратора, оставившего это сообщение, в виде строки.

28.1.3. Форматировщики

Форматировщик задает формат, в котором представляется сообщение, отправленное подсистемой диагностики.

Перечень форматировщиков в параметре `formatters` указывается в виде словаря. Ключами его элементов служат имена объявляемых форматировщиков, а значения элементов задают значения параметров соответствующего форматировщика.

Доступны следующие параметры форматировщиков:

- `format` — строка формата для формирования текста сообщения. Для вставки в текст значений атрибутов объекта сообщения (они были приведены в *разд. 28.1.2*) применяются языковые конструкции вида `%(`*<имя атрибута>*`)s`;
- `datefmt` — строка формата для формирования значения даты и времени. В ней должны использоваться специальные символы, поддерживаемые функцией `strftime()` из модуля `time`. Значение по умолчанию: `"%Y-%m-%d %H:%M:%S,uuu"`.

Пример объявления простого форматировщика с именем `simple`:

```
LOGGING = {
    . . .
    'formatters': {
        'simple': {
            'format': '[%(asctime)s] %(levelname)s: %(message)s',
            'datefmt': '%Y.%m.%d %H:%M:%S',        },
    },
    . . .
}
```

Он будет выводить сообщения в формате [*<дата и время>*] *<уровень>*: *<текст>*, а дата и время создания события будут иметь формат *<год>*.*<месяц>*.*<число>* *<часы>*:*<минуты>*:*<секунды>*.

28.1.4. Фильтры

Фильтр отбирает для вывода только те сообщения, что удовлетворяют определенным условиям, или же выводит сообщения только в том случае, если выполняется какое-либо условие, не связанное с самими сообщениями.

Перечень фильтров записывается в таком же формате, что и перечень форматировщиков (см. *разд. 28.1.3*). Для каждого объявленного фильтра мы можем задать обязательный параметр `()` (две круглые скобки), указывающий строку с именем класса фильтра. Если конструктор этого класса принимает какие-либо параметры, мы можем задать их там же — в настройках фильтра.

Классы фильтров, предлагаемых Django разработчикам, объявлены в модуле `django.utils.log`. Вот они:

- `RequireDebugTrue` — выводит сообщения только в том случае, если включен отладочный режим (параметру `DEBUG` настроек проекта присвоено значение `True`). Об отладочном и эксплуатационном режимах сайта рассказывалось в *разд. 3.3.1*;
- `RequireDebugFalse` — выводит сообщения только в том случае, если включен эксплуатационный режим (параметру `DEBUG` настроек проекта присвоено значение `False`).

  Пример использования этих классов фильтров:

```
LOGGING = {
    . . .
    'filters': {
        'require_debug_false': {
            '()': 'django.utils.log.RequireDebugFalse',
        },
        'require_debug_true': {
            '()': 'django.utils.log.RequireDebugTrue',
        },
    },
    . . .
}
```

- `CallbackFilter(callback=`*<функция>*`)` — отбирает для вывода только те сообщения, для которых указанная в параметре `callback` *функция* вернет `True`. *Функция* должна в качестве единственного параметра принимать сообщение, представленное экземпляром класса `LogRecord` (см. *разд. 28.1.2*).

  Пример объявления фильтра `info_filter`, отбирающего только сообщения уровня `INFO`:

```
def info_filter(message):
    return message.levelname == 'INFO'
. . .
LOGGING = {
    . . .
    'filters': {
        'info_filter': {
            '()': 'django.utils.log.CallbackFilter',
            'callback': info_filter,
        },
    },
    . . .
}
```

28.1.5. Обработчики

Обработчики, как мы уже знаем, непосредственно выполняют вывод сообщений на поддерживаемые ими устройства: на консоль, в файл или куда-либо еще.

Перечень обработчиков записывается в таком же формате, что и перечень форматировщиков (см. *разд. 28.1.3*). Для каждого из обработчиков мы можем задать такие параметры:

- `class` — строка с именем класса обработчика, который будет выполнять вывод сообщений. Поддерживаемые Django классы обработчиков мы рассмотрим позже;
- `level` — минимальный уровень сообщений в виде строкового обозначения. Обработчик станет выводить сообщения, чей уровень не меньше заданного, сообщения меньшего уровня выводиться не будут. Если параметр не указан, обработчик будет выводить сообщения всех уровней;
- `formatter` — форматировщик, что будет применяться для формирования сообщений. Если параметр не указан, сообщения будут иметь формат по умолчанию: *<текст сообщения>*;
- `filters` — список фильтров, через которые будут проходить выводимые обработчиком сообщения. Чтобы сообщение было выведено, оно должно пройти через все включенные в список фильтры. Если параметр не указан, фильтры использоваться не будут;
- если конструктор класса обработчика принимает какие-либо именованные параметры, мы также можем указать их здесь, в настройках обработчика.

Пример указания фильтра, выводящего сообщения уровня `ERROR` и выше на консоль с применением фильтра `require_debug_true` и форматировщика `simple`:

```
LOGGING = {
    . . .
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
```

```
            'level': 'ERROR',
            'formatter': 'simple',
            'filters': ['require_debug_true'],
        },
    },
    . . .
}
```

Теперь приведем наиболее часто используемые классы обработчиков, которые мы можем использовать.

- `logging.StreamHandler` — выводит сообщения в консоли;
- `logging.FileHandler(filename=<путь к файлу>[, mode='a'][, encoding=None][, delay=False])` — сохраняет сообщения в файле с заданным *путем*. Размер получающегося файла ничем не ограничен.

  Параметр `mode` задает режим открытия файла, если он не указан, файл открывается для добавления. Параметр `encoding` указывает кодировку файла, если он опущен, Python сам выберет кодировку.

  Если параметру `delay` дать значение `True`, файл будет открыт только в момент вывода самого первого сообщения. Если же присвоить ему значение `False`, файл будет открыт непосредственно при инициализации класса обработчика (поведение по умолчанию).

  Пример использования этого класса:

```
LOGGING = {
    . . .
    'handlers': {
        'file': {
            'class': 'logging.FileHandler',
            'level': 'INFO',
            'filename': 'd:/logs/django-site.log',
        },
    },
    . . .
}
```

- `logging.handlers.RotatingFileHandler` — то же самое, что `FileHandler`, но вместо одного большого файла создает набор файлов заданного размера. Как только размер очередного файла приближается к указанному пределу, создается новый файл. Формат вызова конструктора этого класса:

```
RotatingFileHandler(filename=<путь к файлу>[, maxBytes=0][,
                    backupCount=0][, mode='a'][, encoding=None][,
                    delay=False])
```

  Параметр `maxBytes` устанавливает размер файла, по превышении которого будет создан новый файл с сообщениями, в байтах. Если задать для него значение `0`, класс обработчика будет сохранять все сообщения в один файл неограниченного размера, т. е. вести себя, как класс `FileHandler`.

Параметр `backupCount` указывает количество ранее созданных файлов, которые будут сохраняться на диске. К расширениям этих файлов будут добавляться последовательно увеличивающиеся целые числа. Так, если сообщения записываются в файл django-site.log, предыдущие файлы получат имена django-site.log.1, django-site.log.2 и т. д. Если параметру `backupCount` присвоить значение `0`, все сообщения будут сохраняться в один файл неограниченного размера (при этом значение параметра `maxBytes` будет проигнорировано).

О назначении остальных параметров конструктора говорилось в описании класса `FileHandler`.

Пример:

```
LOGGING = {
    . . .
    'handlers': {
        'file': {
            'class': 'logging.handlers.RotatingFileHandler',
            'level': 'INFO',
            'filename': 'd:/logs/django-site.log',
            'maxBytes': 1048576,
            'backupCount': 10,
        },
    },
    . . .
}
```

- `logging.handlers.TimedRotatingFileHandler` — то же самое, что `RotatingFileHandler`, только начинает запись в новый файл не при превышении указанного размера файла, а при прошествии заданного временного промежутка. Формат вызова конструктора:

```
TimedRotatingFileHandler(filename=<путь к файлу>[, when='H'][,
                         interval=1][, utc=False][, atTime=None][,
                         backupCount=0][, encoding=None][,
                         delay=False])
```

Параметр `when` указывает разновидность промежутка времени, через который следует начинать запись сообщений в следующий файл. Доступны значения:

- `"S"` — секунды;
- `"M"` — минуты;
- `"H"` — часы;
- `"D"` — дни;
- `"W<номер дня недели>"` — каждый день недели с указанным *номером*. В качестве *номера дня недели* нужно указать целое число от `0` (понедельник) до `6` (воскресенье);
- `"midnight"` — каждый день в полночь.

Параметр `interval` задает количество промежутков времени заданной разновидности, по прошествии которых нужно начинать запись в новый файл.

Примеры:

```
# Создавать новый файл каждый день
'when': 'D',
# Создавать новый файл каждые шесть часов
'interval': 6,
# Создавать новый файл каждые десять дней
'when': 'D',
'interval': 10,
# Создавать новый файл каждую субботу
'when': 'W5',
```

К расширениям ранее созданных файлов с сообщениями будут добавляться строки формата *<год>-<месяц>-<число>*[_*<часы>-<минуты>-<секунды>*], причем вторая половина, с *часами*, *минутами* и *секундами*, может отсутствовать, если задан временной интервал, превышающий один день.

Если параметру `utc` присвоить значение `True`, будет использоваться всемирное координированное время (UTC). Присвоение значения `False` укажет Django использовать местное время.

Значение параметра `atTime` принимается во внимание только в том случае, если для параметра `when` указано значение "W*<номер дня недели>*" или "midnight". Значением параметра `atTime` должна быть отметка времени в виде объекта типа `time` из модуля `datetime`, которая укажет время, в которое следует начать запись в новый файл.

О назначении остальных параметров конструктора говорилось в описании классов `FileHandler` и `RotatingFileHandler`.

Пример:

```
LOGGING = {
    . . .
    'handlers': {
        'file': {
            'class': 'logging.handlers.TimedRotatingFileHandler',
            'level': 'INFO',
            'filename': 'd:/logs/django-site.log',
            'when': 'D',
            'interval': 10,
            'utc': True,
            'backupCount': 10,
        },
    },
    . . .
}
```

- `django.utils.log.AdminEmailHandler([include_html=False][,][email_backend=None])` — отправляет сообщения по электронной почте по адресам, приведенным в списке параметра `ADMINS` настроек проекта (см. *разд. 24.3.3*).

  Если параметру `include_html` присвоить значение `True`, в письмо будет включена в виде вложения веб-страница с полным текстом сообщения об ошибке. Значение `False` приводит к отправке обычного сообщения.

  Посредством параметра `email_backend` можно выбрать другой класс, реализующий отправку электронных писем. Список доступных классов такого назначения, равно как и формат значения параметра, приведены в *разд. 24.1*, в описании параметра `EMAIL_BACKEND` настроек проекта.

- `logging.handlers.SMTPHandler` — отправляет сообщения по электронной почте на произвольный адрес. Формат конструктора:

```
SMTPHandler(mailhost=<интернет-адрес SMTP-сервера>,
            fromaddr=<адрес отправителя>,
            toaddrs=<адреса получателей>, subject=<тема>[,
            credentials=None][, secure=None][, timeout=1.0])
```

  Интернет-адрес SMTP-сервера может быть задан в виде:

  - строки — если сервер работает через стандартный TCP-порт;
  - кортежа из собственно интернет-адреса и номера TCP-порта — если сервер работает через нестандартный порт.

  Адреса получателей указываются в виде списка. *Адрес отправителя* и *тему* отправляемого письма нужно задать в виде строк.

  Параметр `credentials` указывает кортеж из имени и пароля для подключения к SMTP-серверу. Если сервер не требует аутентификации, параметр нужно опустить.

  Доступные значения для параметра `secure`:

  - `None` — если протоколы SSL и TLS не используются (значение по умолчанию);
  - «пустой» кортеж — если используется протокол SSL или TLS;
  - кортеж из пути к файлу с закрытым ключом;
  - кортеж из пути к файлу с закрытым ключом и пути к файлу с сертификатом.

  Параметр `timeout` указывает промежуток времени, в течение которого класс-отправитель будет пытаться установить соединение с SMTP-сервером, в виде целого числа в секундах.

  Пример:

```
LOGGING = {
    . . .
    'handlers': {
        'file': {
            'class': 'logging.handlers.SMTPHandler',
```

```
            'mailhost': 'mail.supersite.ru',
            'fromaddr': 'site@supersite.ru',
            'toaddr': ['admin@supersite.ru',
                       'webmaster@othersite.ru'],
            'subject': 'Проблема с сайтом!',
            'credentials': ('site', 's1i2t3e4'),
        },
    },
    . . .
}
```

- `logging.NullHandler` — вообще не выводит сообщения. Применяется для подавления вывода сообщений определенного уровня.

> ***На заметку***
>
> Здесь были описаны не все классы обработчиков, поддерживаемые Django. Описание всех классов такого рода можно найти по интернет-адресу: **https://docs.python.org/3/library/logging.handlers.html**.

28.1.6. Регистраторы

Регистраторы занимаются сбором всех сообщений, отправляемых какой-либо подсистемой Django или целой группой подсистем.

Перечень регистраторов записывается в виде словаря. В качестве ключей его элементов указываются имена регистраторов, поддерживаемых фреймворком, а значениями элементов должны быть словари, задающие настройки этих регистраторов.

Django поддерживает следующие регистраторы:

- `django` — собирает сообщения от всех подсистем фреймворка;
- `django.request` — собирает сообщения от подсистемы обработки запросов и формирования ответов. Ответы с кодами статуса `5XX` создают сообщения с уровнем `ERROR`, сообщения с кодами `4XX` — сообщения уровня `WARNING`.

  Объект сообщения, в дополнение к приведенным в *разд. 28.1.2*, получит следующие атрибуты:

  - `status_code` — числовой код статуса ответа;
  - `request` — объект запроса;
- `django.server` — то же самое, что `django.request`, но работает только под отладочным веб-сервером Django;
- `django.template` — собирает сообщения об ошибках, присутствующих в коде шаблонов. Такие сообщения получают уровень `DEBUG`;
- `django.db.backends` — собирает сообщения обо всех операциях с базой данных сайта. Такие сообщения получают уровень `DEBUG`.

  Объект сообщения, в дополнение к приведенным в *разд. 28.1.2*, получит следующие атрибуты:

- `sql` — SQL-код команды, отправленной СУБД;
- `duration` — продолжительность выполнения этой команды;
- `params` — параметры, переданные вместе с этой командой;

- `django.db.backends.schema` — собирает сообщения обо всех операциях, производимых над базой данных в процессе выполнения миграций.

  Объект сообщения, в дополнение к приведенным в *разд. 28.1.2*, получит следующие атрибуты:

  - `sql` — SQL-код команды, отправленной СУБД;
  - `params` — параметры, переданные вместе с этой командой;

- `django.security.<класс исключения>` — собирает сообщения о возникновении исключений указанного *класса*. Поддерживаются только класс исключения `SuspiciousOperation` и все его подклассы (`DisallowedHost`, `DisallowedModelAdminLookup`, `DisallowedModelAdminToField`, `DisallowedRedirect`, `InvalidSessionKey`, `RequestDataTooBig`, `SuspiciousFileOperation`, `SuspiciousMultipartForm`, `SuspiciousSession` и `TooManyFieldsSent`);

- `django.security.csrf` — собирает сообщения о несовпадении электронных жетонов безопасности, указанных в веб-формах посредством тега `csrf_token`, с ожидаемыми подсистемой обеспечения безопасности фреймворка.

А вот параметры, которые мы можем указать для каждого такого регистратора:

- `handlers` — список обработчиков, которым регистратор будет пересылать собранные им сообщения для вывода;
- `propagate` — если `True`, регистратор будет передавать собранные сообщения более универсальным регистраторам (обычно это регистратор `django`). Если `False`, сообщения передаваться не будут. Значение по умолчанию — `False`;
- `level` — минимальный уровень сообщений в виде строкового обозначения. Регистратор станет собирать сообщения, чей уровень не меньше заданного, сообщения меньшего уровня будут отклоняться. Если параметр не указан, регистратор будет собирать сообщения всех уровней.

  У универсального регистратора, принимающего сообщения от регистраторов более специализированных, значение параметра `level`, судя по всему, во внимание не принимается. Универсальный регистратор будет собирать сообщения любого уровня, полученные от специализированных регистраторов, независимо от значения этого параметра;
- `filters` — список фильтров, через которые будут проходить собираемые регистратором сообщения. Чтобы сообщение было воспринято, оно должно пройти через все включенные в список фильтры. Если параметр не указан, фильтры использоваться не будут.

28.1.7. Пример настройки диагностических средств

В листинге 28.1 показан пример кода, задающего настройки диагностических средств, который можно использовать на практике. Не забываем, что настройки средств диагностики, равно как и любые прочие настройки проекта, записываются в модуле settings.py пакета конфигурации.

Листинг 28.1. Пример настройки диагностических средств Django

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'filters': {
        'require_debug_false': {
            '()': 'django.utils.log.RequireDebugFalse',
        },
        'require_debug_true': {
            '()': 'django.utils.log.RequireDebugTrue',
        },
    },
    'formatters': {
        'simple': {
            'format': '[%(asctime)s] %(levelname)s: %(message)s',
            'datefmt': '%Y.%m.%d %H:%M:%S',
        }
    },
    'handlers': {
        'console_dev': {
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
            'filters': ['require_debug_true'],
        },
        'console_prod': {
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
            'level': 'ERROR',
            'filters': ['require_debug_false'],
        },
        'file': {
            'class': 'logging.handlers.RotatingFileHandler',
            'filename': 'd:/django-site.log',
            'maxBytes': 1048576,
            'backupCount': 10,
            'formatter': 'simple',
        },
    },
```

```
    'loggers': {
        'django': {
            'handlers': ['console_dev', 'console_prod'],
        },
        'django.server': {
            'handlers': ['file'],
            'level': 'INFO',
            'propagate': True,
        },
    }
}
```

Сначала мы отключаем все средства диагностики, заданные по умолчанию, дав параметру `disable_existing_loggers` значение `True`. Фактически мы назначаем нашему проекту полностью свои собственные диагностические средства.

Затем объявляем два фильтра: `require_debug_false`, пропускающий сообщения только в эксплуатационном режиме, и `require_debug_true`, который будет пропускать сообщения только в режиме отладочном.

Форматировщик `simple` станет выводить сообщения в формате `[`*<дата и время создания>*`] `*<уровень>*`: `*<текст>*.

Обработчиков в нашей конфигурации целых три:

- `console_dev` — будет выводить на консоль сообщения любого уровня, прошедшие через фильтр `require_debug_true`, посредством форматировщика `simple`;
- `console_prod` — будет выводить на консоль сообщения уровня `ERROR`, прошедшие через фильтр `require_debug_false`, посредством форматировщика `simple`;
- `file` — будет сохранять в файл d:\django-site.log сообщения любого уровня, посредством форматировщика `simple`. При превышении файлом размера в 1 Мбайт (1 048 576 байт) будет создан новый файл. Всего будет одновременно храниться 10 таких файлов с сообщениями.

Наконец, мы объявили два регистратора:

- `django` — универсальный регистратор, станет собирать сообщения из всех подсистем фреймворка и выводить их посредством обработчиков `console_dev` и `console_prod`;
- `django.server` — станет собирать сообщения уровней `INFO` и выше от подсистемы обработки запросов, когда запущен отладочный веб-сервер, и выводить их через обработчик `file`.

В результате, если включен отладочный режим, в консоли будут выводиться все сообщения, а при активном эксплуатационном режиме — только сообщения о критических ошибках. А сообщения от подсистемы обработки запросов в любом случае будут дополнительно записываться в файл.

28.2. Средства отладки

Django предлагает весьма удобные средства отладки, позволяющие выяснить, в каком месте исходного кода допущена ошибка. Еще мы можем отключить кэширование статических файлов, что может пригодиться при разработке таблиц стилей для сайта.

28.2.1. Веб-страница сообщения об ошибке

Если при исполнении программного кода произошла ошибка, неважно, синтаксическая, допущенная при программировании или возникшая при собственно выполнении, Django выводит стандартную веб-страницу с сообщением об этой ошибке. Верхнюю, наиболее полезную часть этой страницы можно увидеть на рис. 28.1.

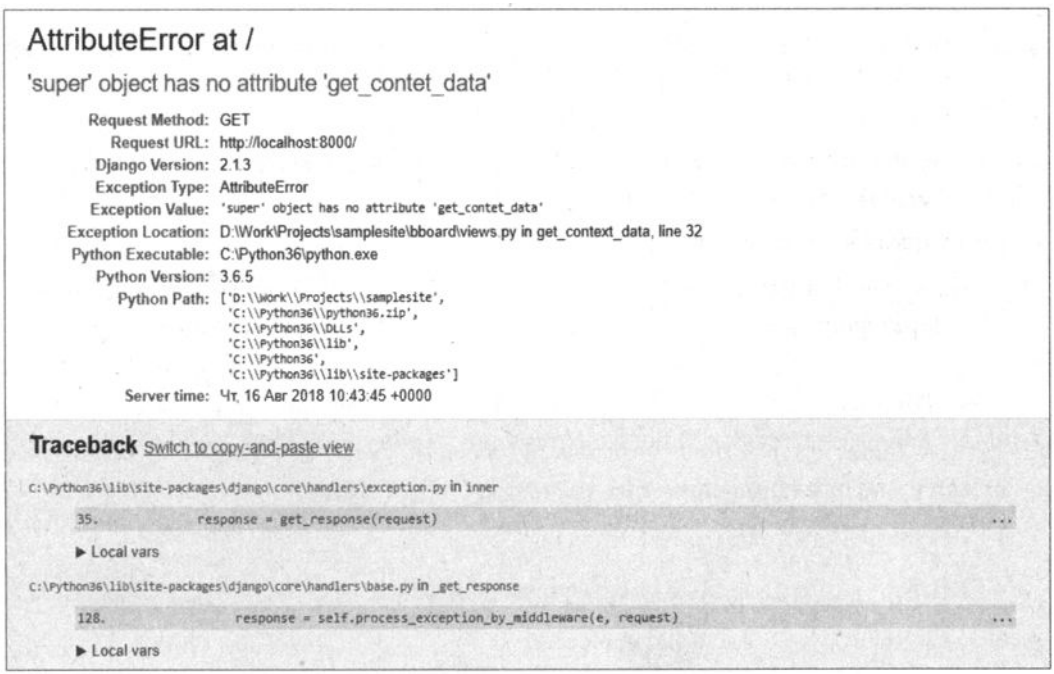


Рис. 28.1. Веб-страница с сообщением об ошибке

Содержимое этой страницы разделено на отдельные области, представляющие различную информацию:

- Общие сведения об ошибке — наиболее важная, выделенная на странице желтым фоном. В ней присутствуют следующие полезные для нас сведения:
  - имя класса исключения, возбужденного при возникновении ошибки, и запрошенный клиентом интернет-адрес;
  - текстовое описание ошибки;

- **Request Method** — HTTP-метод, посредством которого был выполнен запрос;
- **Request URL** — полный интернет-адрес, запрошенный клиентом, с указанием домена и номера порта;
- **Exception Type** — имя класса исключения;
- **Exception Value** — текстовое описание ошибки;
- **Exception Location** — полный путь к программному файлу, в коде которого была допущена ошибка, с указанием номера строки кода.

Также в этой области выводятся номера версий Django и Python, путь к файлу исполняющей среды Python, список путей, по которым исполняющая среда ищет библиотеки, и время обнаружения ошибки.

- **Traceback** — стек вызовов. Организован в виде набора разделов, в каждом из которых выводятся полный путь к программному файлу, строка исходного кода и список локальных переменных с их значениями.

  Каждая строка исходного кода в таком разделе представляет собой заголовок спойлера (раскрывающейся панели). Если щелкнуть на расположенном в его правой части многоточии, появится фрагмент исходного кода, в котором находится эта строка.

  Список локальной переменной также представляет собой спойлер с заголовком **Local vars**.

- **Request information** — сведения о полученном запросе. Здесь присутствуют следующие сведения:
  - **USER** — имя текущего пользователя или **AnonymousUser**, если это гость;
  - **GET** — GET-параметры и их значения;
  - **POST** — POST-параметры и их значения;
  - **FILES** — отправленные посетителем файлы;
  - **COOKIES** — cookie и их значения;
  - **META** — дополнительные сведения, отправленные в составе запроса, сведения об исполняющей среде Python и операционной системе;
  - **Settings** — настройки проекта.

Внимание!

Описанная здесь страница со сведениями об ошибке выводится только в том случае, если активен отладочный режим. Если сайт переведен в эксплуатационный режим, будет выведена обычная страница с сообщением об ошибке 503 (внутренняя ошибка сервера).

28.2.2. Отключение кэширования статических файлов

Очень часто бывает так, что в процессе разработки программного кода сайта также дорабатывается его представление, в частности, используемые им таблицы стилей. Но здесь нас подстерегает серьезная проблема: при отправке статических сайтов клиенту Django устанавливает для них очень большое время кэширования. И, если исправить таблицу стилей и перезагрузить страницу, веб-обозреватель использует кэшированную старую копию таблицы стилей, и мы не увидим на странице никаких изменений.

В качестве решения можно очищать кэш веб-обозревателя после каждого изменения таблиц стилей или вообще отключить кэширование. Но много удобнее указать Django, чтобы он запрещал клиенту кэшировать статические файлы.

Сделать это совершенно не сложно — достаточно выполнить всего три простых действия:

- добавить в модуль urls.py пакета конфигурации такой код (выделен полужирным шрифтом):

```
from django.contrib.staticfiles.views import serve
from django.views.decorators.cache import never_cache

urlpatterns = [
    . . .
]

if settings.DEBUG:
    urlpatterns.append(path('static/<path:path>', never_cache(serve)))
```

Этот код создает маршрут, связывающий шаблонный путь вида **static/*<путь к статическому файлу>*** с контроллером-функцией `serve()` из модуля `django.contrib.staticfiles.views`, который занимается обработкой запросов к статическим файлам. Обратим внимание, что для этого контроллера указан декоратор `never_cache()`, запрещающий кэширование на стороне клиента (за подробностями — к *разд. 25.2.4*). Созданный таким образом маршрут добавляется в список маршрутов уровня проекта, если сайт функционирует в отладочном режиме.

Обратим внимание, что шаблонный путь для этого маршрута записан с учетом того, что в качестве префикса, добавляемого к интернет-адресу статического файла, используется значение по умолчанию: `"/static/"` (этот префикс указывается в параметре `STATIC_URL` настроек проекта и описан в *разд. 11.6.1*). Если в настройках проекта задан другой префикс, шаблонный путь нужно исправить соответствующим образом.

И, наконец, отметим, что этот маршрут добавляется в список, только если сайт работает в отладочном режиме (параметру `DEBUG` настроек проекта присвоено значение `True`). В эксплуатационном режиме он просто не нужен, т. к. в нем обработкой статических файлов занимается сторонний веб-сервер;

- обязательно очистить кэш веб-обозревателя. Если этого не сделать, веб-обозреватель продолжит использовать копии статических файлов, сохраненные в своем кэше;
- запустить отладочный веб-сервер с ключом `--nostatic`, отключающим обработку статических файлов по умолчанию:

```
manage.py runserver --nostatic
```

Как только работа над таблицами стилей сайта будут закончена, рекомендуется вновь включить кэширование статических файлов. Для этого достаточно закомментировать строки, добавленные в модуль urls.py пакета конфигурации на первом шаге, и перезапустить отладочный веб-сервер уже без ключа `--nostatic`.

ГЛАВА 29

Публикация готового веб-сайта

Разработка веб-сайта — процесс долгий и по-своему увлекательный. Но рано или поздно он подходит к концу. Сайт написан, проверен, возможно, наполнен какими-либо рабочими данными — и теперь его предстоит опубликовать в Сети.

В процессе разработки сайта мы запускали его под управлением отладочного веб-сервера Django. Это замечательная программа, прекрасно подходящая для разработки под Django и, главное, всегда имеющаяся под рукой. Но применять его для публикации сайта в Интернете настоятельно не рекомендуется. Для этого применяется сторонний веб-сервер, а именно Apache с одним весьма специфическим дополнительным модулем.

29.1. Подготовка веб-сайта к публикации

Перед публикацией веб-сайта предварительно надо выполнить некоторые подготовительные работы: написать шаблоны страниц с сообщениями об ошибках, удалить ненужные данные и указать кое-какие специфические настройки, не рассмотренные нами ранее. Наконец, нужно соответствующим образом позаботиться о статических файлах. Этим-то мы сейчас и займемся.

29.1.1. Веб-страницы с сообщениями об ошибках и их шаблоны

Из *разд. 28.2.1* мы знаем, что, если сайт работает в отладочном режиме, при возникновении ошибки клиенту будет отправлена стандартная страница с детальным описанием возникшей проблемы. Эту страницу создает сам Django, основываясь на уже имеющемся в его составе шаблоне.

Но для сайта, работающего в эксплуатационном режиме, нам самим придется предусмотреть страницы с сообщениями об ошибках. Таких страниц четыре, и шаблонов нам придется создать также четыре штуки:

- 404.html — шаблон страницы с сообщением об ошибке с кодом статуса 404 (запрошенная страница отсутствует). Обычно такая страница содержит текст вида «Страница не найдена» и гиперссылку на главную страницу сайта.

  Служебный контроллер, выводящий эту страницу, создает в контексте шаблона две переменные:

  - `request_path` — путь, выделенный из интернет-адреса, который был получен в составе запроса;
  - `exception` — строка с текстом сообщения об отсутствии запрошенной страницы.

  Помимо этого, шаблон 404.html имеет доступ ко всем переменным, добавленным в контекст шаблона зарегистрированными обработчиками контекста (о них рассказывалось в *разд. 11.1*);

- 500.html — шаблон страницы с сообщением об ошибке 500 (внутренняя ошибка сервера). Обычно такая страница содержит текст «Внутренняя ошибка сервера» и предложение попытаться обновить страницу спустя некоторое время.

  Служебный контроллер, выводящий эту страницу, передает шаблонизатору пустой контекст шаблона без каких-либо переменных;

- 403.html — шаблон страницы с сообщением об ошибке с кодом статуса 403 (доступ к запрошенной странице запрещен. В частности, эта ошибка возникает при обращении гостя к странице, к которой имеют доступ только зарегистрированные пользователи). Обычно такая страница содержит текст вида «Страница недоступна», предложение выполнить процедуру входа на сайт и гиперссылки на страницу входа и главную страницу сайта.

  Служебный контроллер, выводящий эту страницу, создает в контексте шаблона переменную `exception`, в которой хранится строка с текстом сообщения о недоступности запрошенной страницы;

- 400.html — шаблон страницы с сообщением об ошибке 400 (клиентский запрос некорректно сформирован). Обычно такая страница содержит текст вида «Некорректный запрос».

  Служебный контроллер, выводящий эту страницу, передает шаблонизатору пустой контекст шаблона без каких-либо переменных.

Все эти шаблоны помещаются непосредственно в папку templates пакета приложения или в одну из папок, чей путь указан в параметре `DIRS` настроек текущего шаблонизатора (см. *разд. 11.1*).

> ***Внимание!***
>
> Стандартное приложение `django.contrib.admin` (административный веб-сайт) содержит в своем составе шаблоны *404.html* и *500.html*. Чтобы наш сайт использовал созданные нами шаблоны, а не принадлежащие этому приложению, мы можем прибегнуть к переопределению шаблонов (см. *разд. 18.4*).

Если какой-либо из упомянутых шаблонов отсутствует, Django отправит клиенту пустой ответ с кодом статуса, соответствующим возникшей ошибке. В результате веб-обозреватель выведет встроенную в него страницу с описанием ошибки.

29.1.2. Указание настроек эксплуатационного режима

Следующий шаг — указание настроек проекта, которые будут действовать в эксплуатационном режиме:

- DEBUG — этому параметру, указывающему режим работы сайта, нужно присвоить значение False, задающее эксплуатационный режим;
- ALLOWED_HOSTS — очень важный параметр, указывающий перечень хостов, с которых Django будет принимать отправленные клиентами данные. Если сайт получит данные с хоста, отсутствующего в этом перечне, он возбудит исключение SuspiciousOperation из модуля django.core.exceptions, что приведет к выдаче страницы с сообщением об ошибке 400 (некорректно сформированный запрос).

  Перечень должен быть представлен в виде списка, элементами которого должны быть строки, указывающие разрешенные хосты. Эти строки могут быть:

  - доменными именами;
  - IP-адресами в формате IPv4 или IPv6;
  - шаблонами доменных имен. В таких шаблонах можно применять специальный символ * (звездочка), который обозначает произвольное количество любых знаков.

  Пример:

```
ALLOWED_HOSTS = ['www.supersite.ru', 'blog.supersite.ru',
                                     '*.shop.supersite.ru']
```

  Здесь в список разрешенных занесены хосты **www.supersite.ru**, **blog.supersite.ru** и все хосты вида ***<произвольные символы>*.shop.supersite.ru** (**technics.shop.supersite.ru**, **furniture.shop.supersite.ru** и т. п.).

  Значения этого параметра по умолчанию:

  - в отладочном режиме — список ['localhost', '127.0.0.1', '[::1]'] (т. е. локальный хост, представленный доменным именем и IP-адресами стандартов IPv4 и IPv6);
  - в эксплуатационном режиме — «пустой» список. Поэтому перед запуском сайта в эксплуатацию этот параметр обязательно следует задать, иначе сайт работать не будет;
- DATABASES — необходимо указать параметры базы данных, которая будет использоваться сайтом в эксплуатационном режиме (подробнее об их указании рассказывалось в *разд. 3.3.2*). Поскольку сайт, как правило, публикуется на компьютере, отличном от того, на котором он разрабатывался, параметры базы данным (по крайней мере, параметры соединения с серверной СУБД) там, скорее всего, будут иными;
- STATIC_ROOT — возможно, понадобится изменить путь к папке, в которой хранятся статические файлы сайта (за подробностями — к *разд. 11.6*);

- MEDIA_ROOT — возможно, понадобится изменить путь к папке, в которой хранятся файлы, выгруженные на сайт посетителями (о работе с выгруженными файлами рассказывалось в *главе 19*);
- настройки подсистемы отправки электронных писем — следует изменить их на те, что будут использоваться сайтом, пребывающем в режиме эксплуатации (описание этих параметров см. в *разд. 24.1*);
- CACHES — следует указать параметры подсистемы кэширования уровня сервера (см. *главу 25*), которая будет использоваться при эксплуатации сайта;
- LOGGING — понадобится задать окончательные настройки для подсистемы диагностики (она была описана в *разд. 28.1*);
- ADMINS — здесь нужно задать перечень адресов электронной почты, принадлежащих администраторам;
- MANAGERS — и не забудем об адресах редакторов.

  Как указываются перечни электронных адресов администраторов и редакторов, было рассказано в *разд. 24.3.3*;
- SECRET_KEY — на всякий случай удостоверимся, что секретный ключ, задаваемый этим параметром, кроме нашего сайта, не применяется более нигде.

Также нам следует открыть модуль urls.py пакета конфигурации и удалить или закомментировать код, который создает маршруты, указывающие на контроллеры для обработки статических и выгруженных посетителями файлов (см. *разд. 19.1.2* и *28.2.2*). Впрочем, если при их создании мы следовали указаниям, записанным в соответствующих разделах книги, никаких правок в код нам вносить не придется.

29.1.3. Подготовка статических файлов

Как правило, разработчики сайтов располагают статические файлы в папках static пакетов приложений. Отладочный веб-сервер, встроенный в Django, «умеет» обрабатывать эти папки.

Но сторонний веб-сервер, тот же Apache, под которым будет работать опубликованный в Сети сайт, на такое не способен. Он требует, чтобы все статические файлы, что имеются в составе проекта, включая и входящие в состав стандартных приложений Django, находились в одной папке. Следовательно, нам нужно позаботиться об этом.

Сначала нужно создать папку, в которой будут храниться статические файлы. Путь к этой папке записывается в параметре STATIC_ROOT настроек проекта. Вот пример указания пути к папке static, находящейся непосредственно в папке проекта:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

Для сбора всех статических файлов в этой папке предназначена команда collectstatic утилиты manage.py:

```
manage.py collectstatic [--ignore|-i <шаблон>] [--clear|-c] [--link|-l]
[--noinput|--no-input] [--no-default-ignore] [--dry-run]
```

По умолчанию все статические файлы, найденные в папках static пакетов приложений, а также в папках, пути к которым указаны в параметре STATICFILES_DIRS, копируются в папку, чей путь задан в параметре STATIC_ROOT настроек проекта. Структура папок, в которые вложены эти файлы, при этом сохраняется.

Приложения просматриваются в том порядке, в котором они указаны в списке зарегистрированных в проекте. Если в разных папках static присутствуют файлы с одним и тем же именем, будет использован файл, найденный первым.

При последующих вызовах команды collectstatic в папку будут скопированы только новые и изменившиеся после предыдущего копирования файлы. Перед перезаписью имеющегося в папке назначения файла утилита выдаст предупреждение и предложит ввести слово «yes» для перезаписи или «no» для отказа от этого.

По умолчанию копируются все статические файлы, за исключением файлов с именами CVS, а также именами, которые совпадают с шаблонами .* и *~.

Поддерживаемые командные ключи:

- --ignore или -i — указывает *шаблон* для имен файлов, которые не должны копироваться:

  ```
  manage.py collectstatic --ignore *.tmp
  ```

  Можно указать произвольное количество таких *шаблонов* — каждый в своем ключе:

  ```
  manage.py collectstatic --ignore *.tmp --ignore *.bak
  ```

- --clear или -c — перед началом копирования очистить папку назначения;
- --link или -l — вместо копирования файла создать символическую ссылку на него;
- --noinput или --no-input — имеющийся в папке назначения файл будет перезаписан без выдачи предупреждения;
- --no-default-ignore — также копировать файлы с именами CVS, .* и *~;
- --dry-run — выводит на экран сведения о файлах, подлежащих копированию, но не копирует их.

> ***Внимание!***
>
> Статических файлов в папке назначения может оказаться довольно много, особенно если в проекте используются сложные приложения и библиотеки, наподобие административного веб-сайта Django и Django REST framework. Поэтому во многих случаях имеет смысл рассмотреть вариант с созданием символических ссылок на статические файлы вместо их копирования.

Также может оказаться полезной команда findstatic утилиты manage.py, которая ищет статические файлы с указанными именами:

```
manage.py findstatic <имя файла 1> <имя файла 2> . . . <имя файла n>
[--first]
```

На экран будут выведены полные пути всех найденных файлов. Если указать командный ключ --first, будет выведен только путь к первому обнаруженному файлу.

29.1.4. Удаление ненужных данных

Часть данных, образующихся в процессе работы типового Django-сайта, либо носят временный характер (устаревшие CAPTCHA, сессии и пр.), либо впоследствии безболезненно могут быть созданы повторно (например, миниатюры). Перед публикацией сайта такие данные лучше удалить для уменьшения его объема, особенно если сайт будет переноситься на целевой компьютер по сети.

Вот список команд утилиты manage.py, служащих для удаления ненужных и маловажных данных:

- `captcha_clean` — удаляет просроченные CAPTCHA из хранилища (подробности — в *разд. 17.4.4*);
- `thumbnail_cleanup` — удаляет файлы с миниатюрами: все или сгенерированные в течение указанного количества дней (см. *разд. 19.6.5*);
- `clearsessions` — удаляет устаревшие сессии (см. *разд. 22.2.3*).

29.1.5. Окончательная проверка веб-сайта

По окончании подготовительных работ неплохо было бы провести проверку, все ли мы сделали как надо. Провести ее нам поможет команда `check` утилиты manage.py:

```
manage.py check [<псевдоним приложения 1> <псевдоним приложения 2> . . .
<псевдоним приложения n>] [--tag|-t <группа проверок>] [--list-tags]
[--deploy] [--fail-level <уровень неполадки>]
```

По умолчанию выполняется проверка всех приложений, имеющихся в проекте. Но мы можем задать выполнение проверки только приложений с заданными *псевдонимами*, указав их через пробел. Пример:

```
manage.py check bboard testapp restapi
```

Поддерживаемые командные ключи:

- `--tag` или `-t` — указывает *группу проверок*, которые необходимо провести. Доступны следующие *группы*:
  - `admin` — все, связанное с административным веб-сайтом Django (редакторы, обычные и встроенные, действия и др.);
  - `caches` — настройки подсистемы кэширования;
  - `compatibility` — потенциальные проблемы при переходе на следующую версию Django;
  - `database` — настройки используемых баз данных;
  - `models` — объявления моделей, диспетчеров записей и наборов записей;
  - `security` — настройки, затрагивающие безопасность и защиту от сетевых атак;
  - `signals` — объявления сигналов и привязка к ним обработчиков;

- `staticfiles` — настройки подсистемы, обрабатывающей статические файлы;
- `templates` — настройки шаблонизаторов;
- `urls` — списки маршрутов.

Пример:

```
manage.py check --tag urls
```

Можно указать произвольное количество *групп проверок* — каждую в отдельном ключе:

```
manage.py check --tag database --tag staticfiles --tag urls
```

Если ключ не задан, выполняется проверка по всем группам, за исключением `database`;

- `--list-tags` — выводит список всех поддерживаемых групп проверки;
- `--deploy` — выполняет дополнительные проверки, актуальные только для сайтов, что предназначены для публикации;
- `--fail-level` — указывает *уровень* найденной *неполадки*, после которого проверка прекращается, и утилита завершает работу с выдачей соответствующего сообщения. Доступны уровни неполадок `DEBUG`, `INFO`, `WARNING`, `ERROR` и `CRITICAL`. Если не указан, проверка завершается по выявлении неполадки уровня `ERROR`.

29.2. Публикация веб-сайта с использованием веб-сервера Apache

Итак, последние манипуляции, призванные сделать сайт работоспособным, выполнены. Настал волнующий момент, когда наше веб-творение, ранее созерцаемое лишь нами, разработчиками, явит себя миру.

К сожалению, процесс публикации Django-сайта нельзя назвать простым и беспроблемным. Готового программного продукта для этой задачи не существует, и нам придется бóльшую часть работ выполнять вручную.

29.2.1. Подготовка платформы для публикации

Сначала нам нужно подготовить программную платформу для публикации нашего сайта, а именно веб-сервер и программный модуль, выступающий в качестве коннектора между веб-сервером и Django-сайтом.

И начнем мы с поиска подходящей редакции модуля-коннектора, носящего название mod_wsgi. Далее из описания станет понятно, почему дело обстоит именно так.

Дистрибутивные комплекты различных редакций модуля mod_wsgi находятся по интернет-адресу **https://www.lfd.uci.edu/~gohlke/pythonlibs/#mod_wsgi**. Имена файлов с этими комплектами включают в себя следующие фрагменты символов:

- ap<*две цифры*> — где *две цифры* обозначают версию Apache;
- vc<*одна или две цифры*> — *одна или две цифры* показывают внутреннюю версию среды разработки Microsoft Visual C++, в которой компилировался модуль-коннектор;
- cp<*две цифры*> — *две цифры* обозначают версию Python;
- win32 — если это 32-разрядная редакция модуля;
- win_amd64 — если это 64-разрядная редакция модуля.

Например, mod_wsgi-4.6.4+ap24vc10-cp34-cp34m-win32.whl — это редакция для 32-разрядного Python 3.4 и Apache 2.4, откомпилированного в Microsoft Visual C++ версии VC10.

Нам нужно выбрать ту редакцию mod_wsgi, которая соответствует версии и редакции установленного у нас Python. Так, если у нас 64-разрядный Python 3.6, мы выберем и загрузим файл mod_wsgi-4.6.4+ap24vc14-cp36-cp36m-win_amd64.whl.

С расширением whl сохраняются файлы формата WHL (от англ. wheel) — дистрибутивные пакеты дополнительных библиотек для Python. Чтобы выполнить установку библиотеки из такого файла, достаточно отдать команду формата:

```
pip install <имя файла с дистрибутивом в формате WHL>
```

Теперь можно загрузить и установить веб-сервер Apache. Найти его дистрибутивный комплект можно на сайте **http://www.apachelounge.com/** или **https://www.apachehaus.com/**. Только при выборе дистрибутива учтем две очень важные вещи:

1. Следует выбрать редакцию Apache, которая откомпилирована в той же версии Microsoft Visual C++, что и установленный нами ранее модуль-коннектор mod_wsgi. Так, если мы установили модуль mod_wsgi-4.6.4+ap24vc14-cp36-cp36m-win_amd64.whl, который, если судить по символам vc14 в имени его файла, откомпилирован в Microsoft Visual C++ версии VC14, нам следует выбрать дистрибутив Apache 2.4.x OpenSSL 1.0.2 VC14.
2. В случае установки 32-разрядной редакции Python нужно выбирать только 32-разрядную редакцию Apache, а в случае 64-разрядной редакции Python — только 64-разрядную редакцию Apache.

Дистрибутив Apache поставляется в виде обычного архива формата ZIP, содержимое которого следует распаковать в корневую папку диска. Как правило, веб-сервер начинает работать непосредственно после установки, без указания каких-либо настроек.

> ***На заметку***
>
> Полная документация по Apache находится на его домашнем сайте: **http://httpd.apache.org/**.

> ***Внимание!***
>
> Если мы установим не совпадающие друг с другом редакции Python, mod_wsgi и Apache, у нас, скорее всего, ничего не заработает.

Установив веб-сервер, вернемся к командной строке и отдадим в ней команду:

```
mod_wsgi-express module-config
```

Утилита mod_wsgi-express.exe устанавливается в составе модуля mod_wsgi, а команда `module-config` этой утилиты выведет на экран строки, которые нужно добавить в файл конфигурации Apache, чтобы установленный нами модуль заработал. Автор книги после отдачи команды получил следующие конфигурационные строки:

```
LoadFile "c:/python36/python36.dll"
LoadModule wsgi_module "c:/python36/lib/site-packages/mod_wsgi/server/↳
mod_wsgi.cp36-win_amd64.pyd"
WSGIPythonHome "c:/python36"
```

Первая строка предписывает сразу при запуске веб-сервера загрузить программное ядро исполняющей среды Python, которое необходимо для успешной работы mod_wsgi. Вторая строка выполняет загрузку самого этого модуля-коннектора. Третья строка указывает коннектору, где установлена исполняющая среда Python.

Все эти три строки следует добавить в файл конфигурации Apache, который носит имя httpd.conf и находится по пути *<путь, по которому установлен Apache>*\conf.

29.2.2. Конфигурирование веб-сайта

В файл конфигурации httpd.conf, что хранится по пути *<путь, по которому установлен Apache>*\conf, нам также нужно внести код, который задаст параметры самого публикуемого Django-сайта. Этот код не назовешь компактным, и он довольно сложен, так что давайте рассмотрим его по частям:

- ```
  Alias <префикс интернет-адреса статических файлов> ↳
  "<путь к папке статических файлов>"
  ```

  Директива `Alias` указывает Apache, что файлы, чьи интернет-адреса имеют заданный префикс, следует искать в папке с указанным путем. Следовательно, если мы укажем в этой директиве *префикс интернет-адреса статических файлов*, взятый из параметра `STATIC_URL` настроек проекта, и *путь к папке статических файлов*, Apache станет искать статические файлы в этой папке. То, что нам и нужно;

- ```
  <Directory "<путь к папке статических файлов>">
      Require all granted
  </Directory>
  ```

  Директива `Directory` устанавливает права Apache на содержимое папки с заданным путем. Здесь мы указываем, что веб-сервер может иметь доступ ко всем файлам, что хранятся в папке статических файлов. Если мы не сделаем этого, сервер не сможет загрузить ни один файл из этой папки;

- ```
  Alias <префикс интернет-адреса выгруженных файлов> ↳
  "<путь к папке выгруженных файлов>"
  <Directory "<путь к папке выгруженных файлов>">
      Require all granted
  </Directory>
  ```
  ```

Аналогичным образом конфигурируем обработку файлов, выгруженных посетителями.

- `WSGIScriptAlias <префикс для сайта> ↳`
  `"<путь к папке пакета конфигурации проекта>/wsgi.py"`

  Эта директива свяжет указанный *префикс* с самим Django-сайтом, представленным в виде модуля **wsgi.py**, который хранится в пакете конфигурации проекта. Например, если мы укажем префикс **/bboard**, сайт станет доступен по интернет-адресу **http://*<хост>*/bboard/**. Чтобы связать сайт с «корнем», следует задать префикс / (слеш);

- `WSGIPythonPath "<путь к папке проекта>"`

  Это очень важная директива, и она задаст *путь к папке проекта*, чтобы исполняющая среда Python «знала», где находятся все пакеты и модули, составляющие сайт;

- ```
  <Directory "<путь к папке пакета конфигурации проекта>">
      <Files wsgi.py>
          Require all granted
      </Files>
  </Directory>
  ```

  А эта директива задает права на доступ к модулю **wsgi.py**. Она также очень важна — если ее не указать, Apache не сможет запустить этот файл;

- `WSGIPassAuthorization On`

  Если в состав сайта входит веб-служба, обрабатывающая AJAX-запросы и реализующая аутентификацию, эта директива обязательно должна присутствовать в конфигурационном коде. Она указывает модулю-коннектору пропускать любые заголовки, содержащие имя и пароль пользователя, а не удалять их, как это он делает по умолчанию.

Готовый код, конфигурирующий Django-сайт, представлен в листинге 29.1. Его можно использовать как шаблон для написания своей конфигурации.

Листинг 29.1. Пример конфигурации Django-сайта

```
Alias /static/ "c:/sites/samplesite/static/"
Alias /media/ "c:/sites/samplesite/media/"

<Directory "c:/sites/samplesite/static/">
    Require all granted
</Directory>

<Directory "c:/sites/samplesite/media/">
    Require all granted
</Directory>

WSGIScriptAlias / "c:/sites/samplesite/samplesite/wsgi.py"
WSGIPythonPath "c:/sites/samplesite"
```
```

```
<Directory "c:/sites/samplesite/samplesite">
    <Files wsgi.py>
        Require all granted
    </Files>
</Directory>

WSGIPassAuthorization On
```

29.2.3. Особенности публикации веб-сайта, работающего по протоколу HTTPS

Любой Django-сайт может быть опубликован для работы по защищенному протоколу HTTPS. Причем никаких правок в конфигурацию, записываемую в файле httpd.conf, для этого вносить не нужно.

Однако сами разработчики фреймворка настоятельно рекомендуют указать в модуле settings.py пакета конфигурации некоторые настройки, напрямую затрагивающие безопасность сайта и защиту от сетевых атак. Давайте их рассмотрим:

- SECURE_SSL_REDIRECT — если True, сайт станет предписывать веб-обозревателям при попытке доступа к нему по протоколу HTTP выполнять перенаправление по тому же интернет-адресу, но с использованием протокола HTTPS. Значение по умолчанию — False.

  Если сайт должен работать исключительно по протоколу HTTPS, необходимо установить этот параметр в True;

- SECURE_HSTS_SECONDS — указывает время в секундах, на которое сайт запретит веб-обозревателю доступ к себе через незащищенный протокол HTTP. Если задать 0, веб-обозреватель сможет пользоваться для работы с сайтом протоколом HTTP беспрепятственно. Значение по умолчанию: 0.

  Этот параметр нужно указывать, если сайт должен работать исключительно по протоколу HTTPS, чтобы усилить защиту от сетевых атак. Сначала в целях проверки работоспособности имеет смысл задать относительно небольшое значение, например, 3600 (1 час), а потом, удостоверившись, что сайт полностью работоспособен, увеличить его до 31536000 (1 года);

- SECURE_CONTENT_TYPE_NOSNIFF — если True, сайт будет подавлять попытки некоторых веб-обозревателей угадать тип загруженного файла по его содержимому, игнорируя тип, указанный в заголовке Content-Type полученного ответа. Значение по умолчанию — False.

  Указание значения True позволяет предотвратить некоторые типы сетевых атак, связанных с загрузкой клиентом небезопасных файлов (например, веб-страниц с вредоносными веб-сценариями), замаскированных под безопасные (например, изображения или архивы);

- SECURE_BROWSER_XSS_FILTER — если True, сайт будет указывать веб-обозревателям блокировать любой веб-сценарий, получающий в составе загруженных с сайта данных другие веб-сценарии. Значение по умолчанию — False;

- `CSRF_COOKIE_SECURE` — если `True`, электронные жетоны, используемые в веб-формах для идентификации получаемых данных, будут пересылаться посредством подписанных cookie. Значение по умолчанию — `False`.

  Этим параметрам также нужно дать значение `True`, чтобы обезопасить сайт и его посетителей от сетевых атак;

- `SESSION_COOKIE_SECURE` — этот параметр уже рассматривался в *разд. 22.2.1*. Ему нужно дать значение `True`, чтобы cookie сессий загружались только по протоколу HTTPS;

- `X_FRAME_OPTIONS` — указывает, разрешает ли сайт веб-обозревателям открывать какое-либо содержимое во фреймах. Доступны два строковых значения:

  - `"SAMEORIGIN"` — во фреймах могут быть открыты только страницы текущего сайта (значение по умолчанию);
  - `"DENY"` — полный запрет на открытие во фреймах чего-либо.

  Если страницы сайта не предусматривают легальную возможность загрузки какого бы то ни было содержимого во фреймах, этому параметру нужно дать значение `"DENY"`.

ЧАСТЬ IV

Практическое занятие: разработка веб-сайта

ГЛАВА 30

Дизайн. Вспомогательные веб-страницы

На протяжении трех частей книги мы изучали теорию, разбавляя ее небольшими практическими упражнениями. Четвертая же часть представляет собой полностью практическое упражнение — разработку полнофункционального и, в принципе, готового к публикации веб-сайта электронной доски объявлений.

30.1. План веб-сайта

Наша электронная доска объявлений позволит зарегистрированным пользователям публиковать объявления о продаже чего-либо. Объявления будут разноситься по рубрикам, причем структура рубрик будет иметь два уровня иерархии: на первом уровне расположатся рубрики общего плана («недвижимость», «транспорт» и пр.), а на втором — более конкретные («жилье», «гаражи», «дачи», «легковой», «грузовой», «специальный»).

Для вывода списка объявлений мы применим пагинацию, т. к. объявлений может оказаться очень много, и страница, содержащая все объявления, окажется слишком большой. Также мы предусмотрим возможность поиска объявлений по введенному посетителем слову.

Под любым объявлением (на странице сведений об объявлении) может быть оставлено произвольное количество комментариев. Оставлять комментарии будет позволено любому пользователю, в том числе и гостю.

В составе объявления пользователь может поместить основную графическую иллюстрацию, которая будет выводиться и в списке объявлений, и в составе сведений об объявлении, а также произвольное количество дополнительных иллюстраций, которые можно будет увидеть лишь на странице сведений об объявлении. И основная, и дополнительные иллюстрации не являются обязательными к размещению.

Процедура регистрации нового пользователя на сайте будет разбита на два этапа. На первом этапе посетитель вводит свои данные на странице регистрации, после чего на указанный им адрес электронной почты приходит письмо с гиперссылкой,

ведущей на страницу активации. На втором этапе посетитель переходит по гиперссылке, полученной в письме, и попадает на страницу активации, уведомляющей его, что теперь он является зарегистрированным пользователем сайта.

Сайт доски объявлений включит в себя следующие страницы:

- главная — показывающая десять последних опубликованных объявлений без разбиения их на рубрики;
- страница списка объявлений — показывающая (с использованием пагинации) объявления, относящиеся к определенной рубрике. Также она будет содержать форму для поиска объявления по введенному слову;
- страница сведений о выбранном объявлении — помимо сведений об объявлении, выведет все оставленные для него комментарии и форму для добавления нового комментария;
- страницы регистрации и активации нового пользователя;
- страницы входа и выхода;
- страница профиля зарегистрированного пользователя — выведет список объявлений, оставленных текущим пользователем;
- страницы добавления, правки, удаления объявлений;
- страницы изменения пароля, правки и удаления пользовательского профиля;
- страницы сведений о сайте, о правах его разработчика, пользовательского соглашения и пр.

Таков, в самых общих чертах, план разрабатываемого нами сайта. Всевозможные мелочи мы уточним по ходу дела.

30.2. Подготовка проекта и приложения *main*

Сейчас мы создадим проект нашего сайта, незатейливо назвав его `bboard`, и приложение `main`, которое реализует всю функциональность сайта, за исключением веб-службы (последняя будет удостоена особой части — в *главе 35* мы создадим под нее отдельное приложение `api`).

30.2.1. Создание и настройка проекта

Подготовим папку, в которой мы создадим проект, запустим командную строку, перейдем в эту папку и отдадим команду на создание проекта, который мы назовем `bboard`;

```
django-admin startproject bboard
```

Когда проект будет создан, откроем модуль настроек проекта settings.py из пакета конфигурации и внесем в него следующие правки:

- изменим имя файла, в котором будет храниться база данных сайта, — на bboard.data;

- изменим код языка для вывода системных сообщений и страниц административного сайта — на "ru-ru".

Исправленные фрагменты кода модуля settings.py должны выглядеть так:

```
. . .
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'bboard.data'),
    }
}
. . .
LANGUAGE_CODE = 'ru-ru'
. . .
```

Пока этого достаточно. Остальные необходимые правки в настройки проекта мы внесем потом, по ходу работы.

30.2.2. Создание и настройка приложения *main*

Теперь можно создать приложение `main`, которое реализует всю функциональность сайта, кроме веб-службы. Перейдем в папку проекта и отдадим команду:

```
manage.py startapp main
```

В пакете приложения найдем модуль настроек приложения app.py и откроем его. Добавим в объявление конфигурационного класса `MainConfig` атрибут `verbose_name` с названием приложения:

```
class MainConfig(AppConfig):
    name = 'main'
    verbose_name = 'Доска объявлений'
```

Вернемся к модулю настроек проекта settings.py и добавим только что созданное приложение в список зарегистрированных в проекте:

```
INSTALLED_APPS = [
    . . .
    'main.apps.MainConfig',
]
```

30.3. Базовый шаблон

Самое время заняться базовым шаблоном, который будет лежать в основе всех страниц нашего сайта. Создадим в пакете приложения `main` папку templates, а в ней — папку layout. Именно там мы и сохраним наш шаблон.

Для оформления страниц мы используем популярный CSS-фреймворк Bootstrap 4. Он позволит нам быстро создать разметку и указать представление для различных элементов страниц без необходимости записывать для этого бесконечные CSS-

стили. Использовать фреймворк Bootstrap 4 в Django нам позволит дополнительная библиотека django-bootstrap4, так что установим ее (если ранее не сделали этого), отдав в командной строке команду:

```
pip install django-bootstrap4
```

Добавим в список зарегистрированных в проекте приложение `bootstrap4` — программное ядро библиотеки django-bootstrap4:

```
INSTALLED_APPS = [
    . . .
    'bootstrap4',
]
```

Код базового шаблона показан в листинге 30.1. Сохраним его в файле **templates\layout\basic.html**.

Листинг 30.1. Код базового шаблона templates\layout\basic.html

```
{% load bootstrap4 %}
{% load static %}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8">
        <meta name="viewport"
        content="width=device-width, initial-scale=1, shrink-to-fit=no">
        <title>{% block title %}Главная{% endblock %} - Доска
        объявлений</title>
        {% bootstrap_css %}
        <link rel="stylesheet" type="text/css"
        href="{% static 'main/style.css' %}">
        {% bootstrap_javascript jquery='slim' %}
    </head>
    <body class="container-fluid">
        <header class="mb-4">
            <h1 class="display-1 text-center">Объявления</h1>
        </header>
        <div class="row">
            <ul class="col nav justify-content-end border">
                <li class="nav-item"><a class="nav-link"
                href="#">Регистрация</a></li>
                <li class="nav-item dropdown">
                    <a class="nav-link dropdown-toggle"
                    data-toggle="dropdown" href="#"
                    role="button" aria-haspopup="true"
                    aria-expanded="false">Профиль</a>
                    <div class="dropdown-menu">
```

```
                        <a class="dropdown-item" href="#">Мои
                        объявления</a>
                        <a class="dropdown-item" href="#">Изменить личные
                        данные</a>
                        <a class="dropdown-item" href="#">Изменить
                        пароль</a>
                        <div class="dropdown-divider"></div>
                        <a class="dropdown-item" href="#">Выйти</a>
                        <div class="dropdown-divider"></div>
                        <a class="dropdown-item" href="#">Удалить</a>
                    </div>
                </li>
                <li class="nav-item"><a class="nav-link"
                href="#">Вход</a></li>
            </ul>
        </div>
        <div class="row">
            <nav class="col-md-auto nav flex-column border">
                <a class="nav-link root"
                href="{% url 'main:index' %}">Главная</a>
                <span class="nav-link root font-weight-bold">
                Недвижимость</span>
                <a class="nav-link" href="#">Жилье</a>
                <a class="nav-link" href="#">Склады</a>
                <a class="nav-link" href="#">Гаражи</a>
                <span class="nav-link root font-weight-bold" href="#">
                Транспорт</span>
                <a class="nav-link" href="#">Легковой</a>
                <a class="nav-link" href="#">Грузовой</a>
            </nav>
            <section class="col border py-2">
                {% bootstrap_messages %}
                {% block content %}
                {% endblock %}
            </section>
        </div>
        <footer class="mt-3">
            <p class="text-right font-italic">&copy; читатели,
            2018 г.</p>
        </footer>
    </body>
</html>
```

Код базового шаблона очень велик и сложен. Помимо этого, он включает большое количество тегов и стилевых классов, формирующих разметку и оформление в стиле Bootstrap. Давайте рассмотрим наиболее значимые фрагменты этого кода и выясним, что они делают:

- ```
  <meta name="viewport"
  content="width=device-width, initial-scale=1, shrink-to-fit=no">
  ```

  Этот метатег, помещенный нами в секцию заголовка страницы (в парный тег `<head>`), необходим для того, чтобы Bootstrap правильно обработал страницу;

- ```
  <title>{% block title %}Главная{% endblock %} -
  Доска объявлений</title>
  ```

  В теге `<title>` мы создаем первый блок, назвав его `title`. С его помощью мы выведем название у каждой страницы сайта;

- ```
  {% bootstrap_css %}
  ```

  Здесь мы привязываем к странице таблицы стилей Bootstrap;

- ```
  <link rel="stylesheet" type="text/css"
  href="{% static 'main/style.css' %}">
  ```

  Также не забываем привязать таблицу стилей static\main\style.css, которую создадим чуть позже. В ней мы запишем некоторые специфические для нашего сайта стили;

- ```
  {% bootstrap_javascript jquery='slim' %}
  ```

  Привязываем файлы веб-сценариев с программным кодом Bootstrap. Также выполним привязку сокращенной редакции библиотеки jQuery, без которой не заработает созданное нами раскрывающееся меню (AJAX и анимацию мы использовать не собираемся, так что полная редакция jQuery нам не нужна);

- ```
  <body class="container-fluid">
      . . .
  </body>
  ```

  К телу страницы (тегу `<body>`) мы привязываем стилевой класс `container-fluid`. Это необходимо делать со всяким тегом, чье содержимое будет верстаться с помощью Bootstrap;

- ```
  <header class="mb-4">
      <h1 class="display-1 text-center">Объявления</h1>
  </header>
  ```

  Стилевой класс `mb-4`, привязанный к элементу страницы, установит у него достаточно большой внешний отступ снизу. А стилевые классы `display-1` и `text-center`, привязанные к заголовку, предпишут веб-обозревателю вывести текст увеличенным шрифтом и выровнять его посередине;

- ```
  <div class="row">
      <ul class="col . . .">
          . . .
      </ul>
  </div>
  ```

  Этот код реализует одну из наиболее впечатляющих возможностей, предлагаемых Bootstrap, — табличную верстку, выполняемую без участия таблиц. Стиле-

вой класс row, привязанный к элементу страницы, вынуждает его вести себя как строка таблицы, а стилевой класс col — как ячейка в этой строке.

В нашем случае мы создаем строку (сформированную блоком — тегом) с единственной ячейкой — маркированным списком. Мы сделали так для того, чтобы убрать у создаваемого элемента страницы просветы слева и справа, которые выглядят очень некрасиво;

```
<ul class=". . . nav justify-content-end border">
    <li class="nav-item"><a class="nav-link"
    href="#">Регистрация</a></li>
    . . .
</ul>
```

Маркированный список, о котором шла речь ранее, мы используем для создания горизонтальной полосы навигации. Для этого мы привяжем к нему стилевой класс nav. Стилевой класс justify-content-end укажет выровнять пункты полосы навигации по правому краю, а стилевой класс border создаст рамку вокруг нее.

Пункты полосы навигации создаются так же, как и пункты списков, — тегами <li> — с привязанным стилевым классом nav-item. Внутри этих тегов должны находиться гиперссылки с привязанными стилевыми классами nav-link;

```
<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" data-toggle="dropdown"
    href="#" role="button" aria-haspopup="true"
    aria-expanded="false">Профиль</a>
    <div class="dropdown-menu">
        <a class="dropdown-item" href="#">Мои объявления</a>
        . . .
        <div class="dropdown-divider"></div>
        <a class="dropdown-item" href="#">Удалить</a>
    </div>
</li>
```

А этот весьма объемистый код создает в полосе навигации пункт с раскрывающимся меню. Для этого внутрь тега <li> помещается, помимо гиперссылки, еще и блок, который создаст само меню. Обратим внимание на стилевые классы, привязываемые к различным элементам, и дополнительные атрибуты, которые обязательно должны присутствовать в тегах.

Пункты раскрывающегося меню формируются с помощью обычных гиперссылок, к которым необходимо привязать стилевой класс dropdown-item. А «пустой» блок со стилевым классом dropdown-divider создаст разделитель между пунктами;

```
<div class="row">
    <nav class="col-md-auto . . .">
        . . .
    </nav>
```

```
        <section class="col . . .">
            . . .
        </section>
    </div>
```

Здесь мы снова применили табличную верстку, но на этот раз из двух «ячеек»: семантической панели навигации (тега `<nav>`) и семантической секции страницы (тега `<section>`).

Знакомый нам стилевой класс `col` при привязке к элементам-«ячейкам» создаст «ячейки» одинаковой ширины. Если нужно сделать так, чтобы ширина какой-либо ячейки соответствовала ширине ее содержимого, нужно привязать к создающему эту ячейку элементу страницы стилевой класс `col-md-auto`. Мы привязали его к панели навигации;

```
<nav class=". . . nav flex-column border">
    <a class="nav-link root" href="{% url 'main:index' %}">Главная</a>
    <span class="nav-link root font-weight-bold">Недвижимость</span>
    <a class="nav-link" href="#">Жилье</a>
    . . .
</nav>
```

Еще мы привязали к семантической панели навигации стилевые классы `nav` и `flex-column`, чтобы превратить ее в вертикальную панель навигации в стиле Bootstrap, а также создающий рамку стилевой класс `border`.

Разные пункты панели навигации мы формируем тремя разными способами:

- пункты, ведущие на служебные страницы, — гиперссылками с двумя привязанными стилевыми классами: `nav-link`, который нам уже знаком, и `root`, который мы запишем в таблице стилей static\main\style.css (он задаст увеличенный размер шрифта);
- пункты, обозначающие рубрики верхнего уровня (надрубрики), — тегами `<span>` со стилевыми классами `nav-link`, `root` и `font-weight-bold`. Последний стилевой класс задаст полужирное начертание шрифта — так мы визуально выделим пункты этого типа. Обратим внимание, что такие пункты не являются гиперссылками;
- пункты, ведущие на рубрики нижнего уровня (подрубрики), — гиперссылками со стилевым классом `nav-link`;

```
<section class=". . . border py-2">
    {% bootstrap_messages %}
    {% block content %}
    {% endblock %}
</section>
```

К семантической секции мы также привязали стилевые классы `border` и `py-2`. Первый нам уже знаком, а второй установит для тега небольшие внутренние отступы сверху и внизу, чтобы содержимое тега не примыкало к рамке вплотную.

В семантическую секцию мы поместили код, выводящий всплывающие сообщения, и блок `content`, в котором будет выводиться основное содержимое страниц;

- ```
  <footer class="mt-3">
      © читатели, 2018 г.
  </footer>
  ```

  «Поддон» сайта мы создаем специализированным тегом `<footer>`. Стилевой класс `mt-3` укажет для него средних размеров внешний отступ сверху, чтобы отделить его от вышерасположенных элементов. Чтобы выровнять текст абзаца по правому краю и вывести его курсивом, мы привязали к тегу `<p>` стилевые классы `text-right` и `font-italic`.

Основная особенность Bootstrap — большое количество стилей буквально на все случаи жизни. Пользуясь ими, мы можем выполнять типовые задачи по верстке веб-страниц исключительно средствами этого CSS-фреймворка, не написав ни единого стиля.

Однако сейчас нам все-таки придется написать пару стилей. Создадим в папке пакета приложения папку static, в ней — папку main. В последнюю запишем файл style.css с кодом из листинга 30.2.

Листинг 30.2. Таблица стилей static\main\style.css

```
header h1 {
    background: url("bg.jpg") left / auto 100% no-repeat, ↳
    url("bg.jpg") right / auto 100% no-repeat;
}
.root {
    font-size: larger;
}
```

Что делает второй стиль (со стилевым классом `root` в качестве селектора), мы уже знаем — увеличивает размер шрифта. А первый стиль создаст у заголовка сайта фон в виде двух изображений доски объявлений, выведенных слева и справа.

Найдем в Интернете подходящее изображение, которое можно использовать в качестве фона, и также сохраним его в папке static\main пакета приложения. Дадим файлу с изображением имя bg.jpg.

30.4. Главная веб-страница

Закончив с базовым шаблоном, можно приступать к написанию контроллера и шаблона главной страницы. Сделаем их как можно проще — только чтобы удостовериться, работает ли наш сайт.

Контроллер мы реализуем в виде функции — так проще. Дадим ему имя `index()`. Код контроллера можно увидеть в листинге 30.3.
```

Листинг 30.3. Код контроллера-функции index()

```
from django.shortcuts import render

def index(request):
    return render(request, 'main/index.html')
```

В папке templates пакета приложения создадим папку main, в которой будем сохранять шаблоны страниц. Первым из них станет шаблон приложения index.html, код которого показан в листинге 30.4.

Листинг 30.4. Код шаблона templates\main\index.html

```
{% extends "layout/basic.html" %}

{% block content %}
<h2>Последние 10 объявлений</h2>
{% endblock %}
```

Теперь нужно написать маршруты. Заодно мы сделаем так, чтобы статические файлы сайта не кэшировались веб-обозревателем — ведь работа над таблицей стилей еще не закончена.

Начнем со списка маршрутов уровня проекта. Откроем модуль urls.py пакета конфигурации и исправим его код так, чтобы он выглядел, как показано в листинге 30.5.

Листинг 30.5. Код модуля urls.py пакета конфигурации

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.contrib.staticfiles.views import serve
from django.views.decorators.cache import never_cache

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('main.urls', namespace='')),
]

if settings.DEBUG:
    urlpatterns.append(path('static/<path:path>', never_cache(serve)))
```

Здесь нам все знакомо и никаких пояснений не требует. Отметим только, что мы установили приложение `main` в качестве корневого.

Приступим к созданию списка маршрутов уровня приложения. Создадим в пакете приложения модуль urls.py и запишем в него код из листинга 30.6.

Листинг 30.6. Код модуля urls.py пакета приложения

```
from django.urls import path

from .views import index

app_name = 'main'
urlpatterns = [
    path('', index, name='index'),
]
```

Теперь можно проверить написанное нами в работе. Сохраним все вновь созданные и исправленные файлы и запустим отладочный веб-сервер Django с отключенной обработкой статических файлов:

```
manage.py runserver --nostatic
```

Запустим веб-обозреватель, выполним переход по интернет-адресу **http://localhost:8000/** и попадем на главную страницу нашего сайта (рис. 30.1).

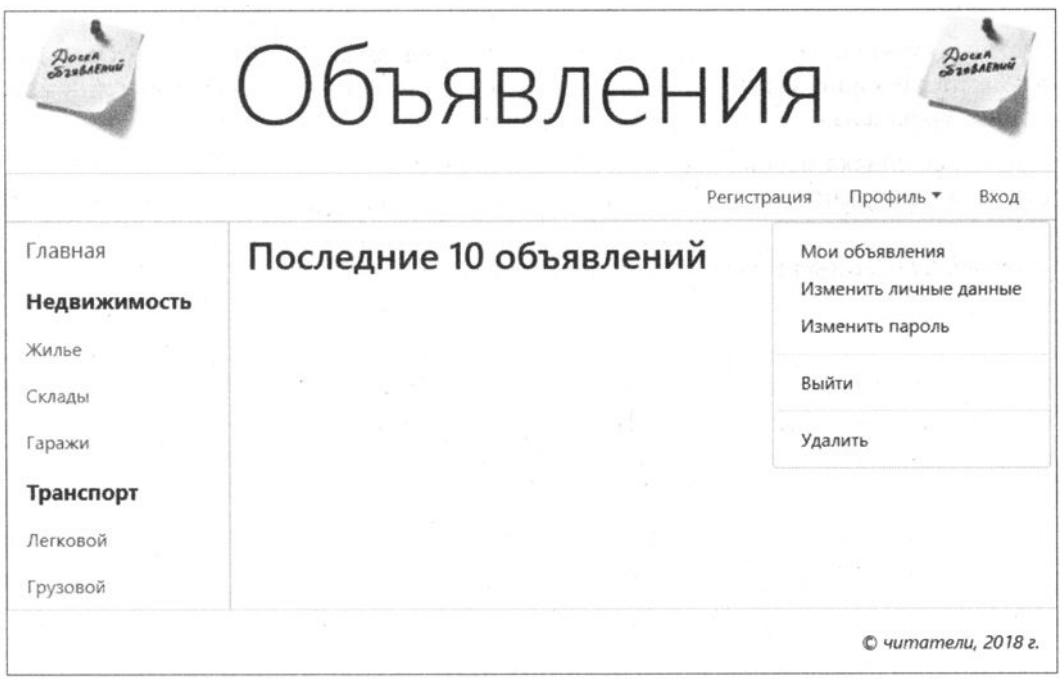


Рис. 30.1. Главная веб-страница сайта доски объявлений

Здесь мы видим заголовок сайта, украшенный изображениями стилизованной доски объявлений, горизонтальную полосу навигации с тремя пунктами, причем третий имеет раскрывающееся меню (на рис. 30.1 оно как раз открыто), вертикальную панель навигации и «поддон». Отметим, что мы создали это минимумом CSS-кода, в основном, пользуясь средствами Bootstrap.

30.5. Вспомогательные веб-страницы

Сразу же создадим вспомогательные веб-страницы — по крайней мере, страницу со сведениями о сайте и правах его разработчиков. Сделать их можно двумя способами.

Первый способ, самый очевидный, заключается в том, что для каждой страницы пишется отдельный контроллер и отдельный маршрут. Этот способ весьма трудоемок, поскольку придется писать несколько контроллеров с практически одинаковым кодом, и подходит лишь для тех случаев, когда страницы должны выводить какие-либо данные, извлекаемые из базы или формируемые программно.

Второй способ — реализация вывода всех страниц с применением одного контроллера и, соответственно, одного маршрута. Какой-либо идентификатор страницы, предназначенной к выводу на экран, передается контроллеру с URL-параметром. Трудоемкость работы в таком случае существенно снижается, поскольку нужно написать всего один контроллер.

Давайте реализуем вывод вспомогательных страниц вторым способом. В качестве идентификатора использует имя формирующего ее шаблона без пути и без расширения — так проще.

В список маршрутов уровня приложения, что хранится в модуле urls.py пакета приложения, добавим такой код (выделен полужирным шрифтом):

```
from .views import other_page
. . .
urlpatterns = [
    path('<str:page>/', other_page, name='other'),
    path('', index, name='index'),
]
```

Имя шаблона выводимой страницы мы передаем через URL-параметр `page`. А контроллер, выводящий вспомогательные страницы, назовем `other_page` и реализуем в виде функции. Вообще, контроллеры-функции — идеальный инструмент для написания чего-либо нестандартного, специфического.

В модуль views.py пакета приложения, где хранится код контроллеров, добавим код контроллера-функции `other_page()`, представленный в листинге 30.7.

Листинг 30.7. Код контроллера-функции other-page()

```
from django.http import HttpResponse, Http404
from django.template import TemplateDoesNotExist
from django.template.loader import get_template

def other_page(request, page):
    try:
        template = get_template('main/' + page + '.html')
```

```
    except TemplateDoesNotExist:
        raise Http404
    return HttpResponse(template.render(request=request))
```

Здесь мы получаем имя выводимой страницы из параметра `page`, добавляем к нему путь и расширение, получив тем самым полный путь к нужному шаблону, и пытаемся загрузить его вызовом функции `get_template()`. Если загрузка выполнилась успешно, мы формируем на основе этого шаблона страницу.

Если же шаблон загрузить не удалось, функция `get_template()` возбудит исключение `TemplateDoesNotExist`. Мы перехватываем это исключение и возбуждаем другое исключение — `Http404`, которое приведет к отправке страницы с сообщением об ошибке 404 (запрошенная страница не существует).

Для начала давайте сделаем страницу со сведениями о самом сайте и правах его разработчиков и дадим ей имя `about`.

Создадим шаблон templates\main\about.html с кодом из листинга 30.8.

Листинг 30.8. Код шаблона templates\main\about.html

```
{% extends "layout/basic.html" %}

{% block title %}О сайте{% endblock %}

{% block content %}
<h2>О сайте</h2>
<p>Сайт для публикации объявлений о продаже, разбитых на рубрики.</p>
<p>Все права принадлежат читателям книги &quot;Django 2.1&quot;.</p>
{% endblock %}
```

После чего откроем базовый шаблон templates\layout\basic.html и добавим в панель навигации такой код (выделен полужирным шрифтом):

```
<nav class="col-md-auto nav flex-column border">
    . . .
    <a class="nav-link root" href="{% url 'main:other' page='about' %}">
    О сайте</a>
</nav>
```

Сохраним все исправленные и вновь созданные файлы, обновим открытую в веб-обозревателе главную страницу и перейдем по гиперссылке **О сайте**. Если мы не допустили ошибок, то увидим только что созданную страницу со сведениями о сайте.

ГЛАВА 31

Работа с пользователями и разграничение доступа

Первое, что следует создать непосредственно после изготовления «костяка» сайта — базового шаблона и главной страницы, — инструменты для работы с пользователями и разграничения доступа. Сейчас мы этим и займемся.

Мы создадим страницы для входа и выхода, регистрации, активации, страницы пользовательского профиля, смены данных о пользователе, смены его пароля и удаления профиля. Работы предстоит довольно много, так что не будем терять времени.

31.1. Модель пользователя

Стандартная модель пользователя `User`, предлагаемая стандартным же приложением `django.contrib.auth`, нам не подходит, поскольку нам нужно хранить дополнительные данные о пользователе. Поэтому мы создадим свою собственную модель, сделав ее производной от стандартной абстрактной модели `AbstractUser`, объявленной в модуле `django.contrib.auth`.

Наша модель будет носить название `AdvUser`. Список полей, которые мы объявим в ней, приведен в табл. 31.1.

***Таблица 31.1.** Структура модели `AdvUser`*

Имя	Тип	Дополнительные параметры	Описание
`is_activated`	`BooleanField`	Значение по умолчанию — `True`, индексированное	Признак, прошел ли пользователь процедуру активации
`send_messages`	`BooleanField`	Значение по умолчанию — `True`	Признак, желает ли пользователь получать уведомления о новых комментариях

Код, объявляющий эту модель, показан в листинге 31.1. Его мы запишем в модуль models.py пакета приложения.

Листинг 31.1. Код модели AdvUser

```
from django.db import models
from django.contrib.auth.models import AbstractUser

class AdvUser(AbstractUser):
    is_activated = models.BooleanField(default=True, db_index=True,
                                       verbose_name='Прошел активацию?')
    send_messages = models.BooleanField(default=True,
                    verbose_name='Слать оповещения о новых комментариях?')

    class Meta(AbstractUser.Meta):
        pass
```

Сразу же укажем ее как модель пользователя, используемую подсистемой разграничения доступа Django. Для этого откроем модуль settings.py пакета конфигурации и добавим в него строку:

```
AUTH_USER_MODEL = 'main.AdvUser'
```

Остановим отладочный веб-сервер Django и отдадим в командной строке команду сначала на создание миграций:

```
manage.py makemigrations
```

а потом — на их выполнение:

```
manage.py migrate
```

Как только миграции будут выполнены, создадим суперпользователя, отдав команду:

```
manage.py createsuperuser
```

Введем выбранные нами имя, адрес электронной почты и пароль создаваемого пользователя.

Напоследок откроем модуль admin.py пакета приложения, в который заносится код, объявляющий классы-редакторы и регистрирующий модели в административном сайте. Зарегистрируем нашу модель пользователя, добавив в этот модуль код:

```
from .models import AdvUser

admin.site.register(AdvUser)
```

Запустим отладочный веб-сервер, опять же, отключив обработку статических файлов. Откроем административный веб-сайт, набрав интернет-адрес **http://localhost:8000/admin/**, и попытаемся выполнить вход от имени только что созданного суперпользователя. У нас все должно получиться.

31.2. Основные веб-страницы: входа, профиля и выхода

Подготовив нашу собственную модель пользователя, приступим в разработке страниц входа, пользовательского профиля и выхода — основных страниц, связанных с разграничением доступа.

31.2.1. Веб-страница входа

Для реализации входа мы используем контроллер-класс `LoginView`. Мы создадим на его основе подкласс, в котором запишем все необходимые для работы контроллера параметры. Ради простоты давайте, по возможности, следовать установленным Django соглашениям.

Код контроллера-класса, выполняющего вход и носящего имя `BBLoginView`, показан в листинге 31.2. Добавим его в модуль views.py пакета приложения.

Листинг 31.2. Код контроллера-класса BBLoginView

```
from django.contrib.auth.views import LoginView

class BBLoginView(LoginView):
    template_name = 'main/login.html'
```

Как видим, код очень прост. Единственное, что он делает, — задает путь к файлу шаблона, занеся его в атрибут `template_name`. Остальные параметры мы не указываем, т. к. собираемся использовать значения, заданные для них по умолчанию, и, таким образом, следовать соглашениям (вообще, на взгляд автора, это лучшая политика в Django-программировании, да и в целом в разработке сайтов).

Отметим один примечательный момент. Шаблон страницы входа login.html мы поместили в папку templates\main — туда же, где и находятся все остальные, созданные к настоящему моменту, шаблоны. Поскольку наш сайт не отличается значительной сложностью и содержит относительно немного страниц, давайте хранить их в одной папке — так нам будет проще в сопровождении.

Запишем новый маршрут, указывающий на контроллер `BBLoginView`, в списке уровня приложения. Откроем модуль urls.py пакета приложения и добавим в него код:

```
from .views import BBLoginView
. . .
urlpatterns = [
    path('accounts/login/', BBLoginView.as_view(), name='login'),
    . . .
]
```

Здесь мы связываем с нашим новым контроллером маршрут **accounts/login/**. По умолчанию именно по этому маршруту Django выполняет перенаправление при

попытке гостя получить доступ к закрытой от него странице. То есть мы и в этом случае соблюдаем соглашения фреймворка.

На заметку

Мы могли бы записать маршрут и таким образом:

```
from django.contrib.auth.views import LoginView
. . .
urlpatterns = [
    path('accounts/login/',
         LoginView.as_view(template_name='main/login.html'),
         name='login'),
    . . .
]
```

то есть, указав в маршруте непосредственно класс `LoginView` и записав все необходимые параметры контроллера в вызове его метода `as_view()`. Но в таком случае код контроллеров окажется записан в двух модулях: *views.py* и *urls.py*, и в дальнейшем, при сопровождении сайта, в поисках нужного фрагмента кода нам придется просматривать оба этих модуля, что неудобно.

Поэтому давайте держать код контроллеров в модуле *views.py*, а код списка маршрутов — в модуле *urls.py*. Так нам будет проще.

Напишем шаблон страницы входа templates\main\login.html. Этот код можно увидеть в листинге 31.3.

Листинг 31.3. Код шаблона templates\main\login.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %}Вход{% endblock %}

{% block content %}
<h2>Вход</h2>
{% if user.is_authenticated %}
<p>Вы уже выполнили вход.</p>
{% else %}
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    <input type="hidden" name="next" value="{{ next }}">
    {% buttons submit='Войти' %}{% endbuttons %}
</form>
{% endif %}
{% endblock %}
```

Мы использовали для вывода формы инструменты, предлагаемые библиотекой django-bootstrap4. Поскольку поля ввода, в которые будут заноситься имя и пароль

пользователя, невелики, мы вывели форму в «горизонтальной» разметке, когда надпись и относящийся к ней элемент управления располагаются по горизонтали.

Напоследок внесем весьма значительные правки в базовый шаблон templates\layout\basic.html. Найдем в нем фрагмент кода, создающего пункт **Вход** и пункт с раскрывающимся меню **Профиль** горизонтальной панели навигации, и исправим его следующим образом (добавленный и исправленный код выделен полужирным шрифтом):

```
<ul class="col nav justify-content-end border">
    <li class="nav-item"><a .. . .>Регистрация</a></li>
    {% if user.is_authenticated %}
    <li class="nav-item dropdown">
        . . .
    </li>
    {% else %}
    <li class="nav-item"><a . . .>Вход</a></li>
    {% endif %}
</ul>
```

В результате пункт **Вход** будет выводиться только гостям, а пункт **Профиль** — только пользователям, выполнившим вход.

И запишем в тег <a>, создающий гиперссылку **Вход**, интернет-адрес, который ведет на страницу входа:

```
<a . . . href="{% url 'main:login' %}">Вход</a>
```

Сохраним все новые и исправленные файлы, обновим открытую в веб-обозревателе главную страницу и щелкнем на гиперссылке **Вход**. Если мы все сделали без ошибок, то сразу же попадем на страницу входа.

Но пока не будем выполнять вход, иначе возникнет ошибка. Давайте сначала сделаем страницы профиля и выхода.

31.2.2. Веб-страница пользовательского профиля

Контроллер, который выведет страницу пользовательского профиля, мы реализуем в виде функции и назовем `profile()`. Его код чрезвычайно прост — см. листинг 31.4.

Листинг 31.4. Код контроллера-функции `profile()`

```
from django.contrib.auth.decorators import login_required

@login_required
def profile(request):
    return render(request, 'main/profile.html')
```

На странице пользовательского профиля у нас будет выводиться список объявлений, оставленных текущим пользователем. Но объявлений у нас пока что нет

(более того, сама функциональность по их написанию еще даже не создавалась), так что пока выводить на этой странице совершенно нечего.

Поскольку страница пользовательского профиля должна быть доступна только зарегистрированным пользователям, успешно выполнившим вход на сайт, мы пометили контроллер-функцию `profile()` декоратором `login_required()`.

Добавим в список маршрутов уровня приложения (модуль urls.py пакета приложения) маршрут, который укажет на контроллер `profile()`:

```
from .views import profile
. . .
urlpatterns = [
    path('accounts/profile/', profile, name='profile'),
    . . .
]
```

Мы указали в этом маршруте шаблонный путь **accounts/profile/** — по этому пути Django по умолчанию выполняет перенаправление после успешного входа. Тем самым мы продолжаем следовать соглашениям, принятым во фреймворке.

Код шаблона templates\main\profile.html, формирующего страницу профиля, можно увидеть в листинге 31.5.

Листинг 31.5. Код шаблона templates\main\profile.html

```
{% extends "layout/basic.html" %}

{% block title %}Профиль пользователя{% endblock %}

{% block content %}
<h2>Профиль пользователя {{ user.username }}</h2>
{% if user.first_name and user.last_name %}
<p>Здравствуйте, {{ user.first_name }} {{ user.last_name }}!</p>
{% else %}
<p>Здравствуйте!</p>
{% endif %}
<h3>Ваши объявления</h3>
{% endblock %}
```

Если пользователь при регистрации написал свои имя и фамилию, на странице будет выведено персонализированное приветствие. Если же пользователь не ввел эти данные, будет выведено приветствие простое.

В шаблоне templates/layout/basic.html найдем тег `<a>`, выводящий гиперссылку **Мои объявления**, и вставим в него интернет-адрес страницы профиля:

```
<a . . . href="{% url 'main:profile' %}">Мои объявления</a>
```

31.2.3. Веб-страница выхода

Контроллер выхода мы реализуем в виде класса `BBLogoutView`, производного от класса `LogoutView`. Код этого класса показан в листинге 31.6.

Листинг 31.6. Код контроллера-класса `BBLogoutView`

```
from django.contrib.auth.views import LogoutView
from django.contrib.auth.mixins import LoginRequiredMixin

class BBLogoutView(LoginRequiredMixin, LogoutView):
    template_name = 'main/logout.html'
```

Нам нужно сделать так, чтобы страница выхода была доступна только зарегистрированным пользователям, выполнившим вход. Для этого мы добавили в число суперклассов контроллера-класса `BBLogoutView` примесь `LoginRequiredMixin`.

В списке маршрутов уровня приложения (он записывается в модуле urls.py пакета приложения) запишем маршрут, ведущий на этот контроллер:

```
from .views import BBLogoutView
. . .
urlpatterns = [
    path('accounts/logout/', BBLogoutView.as_view(), name='logout'),
    . . .
]
```

Напишем шаблон templates\main\logout.html, очень простой код которого показан в листинге 31.7.

Листинг 31.7. Код шаблона templates\main\logout.html

```
{% extends "layout/basic.html" %}

{% block title %}Выход{% endblock %}

{% block content %}
<h2>Выход</h2>
<p>Вы успешно вышли с сайта.</p>
{% endblock %}
```

В шаблоне templates/layout/basic.html найдем тег `<a>`, выводящий гиперссылку **Выход**, и поместим в него интернет-адрес страницы выхода:

```
<a . . . href="{% url 'main:logout' %}">Выйти</a>
```

Сохраним все файлы, подождем, пока отладочный веб-сервер не перезапустится, и обновим страницу входа, открытую в веб-обозревателе. Занесем в форму имя

и пароль созданного ранее суперпользователя и выполним вход. Посмотрим на страницу профиля и выполним выход.

31.3. Веб-страницы правки личных данных пользователя

Следующие на очереди — страницы для правки личных данных пользователя: основных (имени, фамилии, адреса электронной почты) и пароля.

31.3.1. Веб-страница правки основных сведений

Эта страница предоставит пользователю возможность исправить его имя (логин), адрес электронной почты, реальные имя, фамилию и признак, хочет ли он получать по электронной почте оповещения о появлении новых комментариев к опубликованным им объявлениям. При этом мы сделаем адрес электронной почты обязательным к заполнению.

Первое, что нам нужно сделать, — объявить форму, связанную с моделью `AdvUser`. В эту форму, которую мы назовем `ChangeUserInfoForm`, пользователь будет заносить новые личные данные. Код формы можно увидеть в листинге 31.8. Его мы запишем во вновь созданный модуль forms.py пакета приложения.

Листинг 31.8. Код формы `ChangeUserInfoForm`

```
from django import forms

from .models import AdvUser

class ChangeUserInfoForm(forms.ModelForm):
    email = forms.EmailField(required=True,
                             label='Адрес электронной почты')

    class Meta:
        model = AdvUser
        fields = ('username', 'email', 'first_name', 'last_name',
                  'send_messages')
```

Мы комбинируем быстрое и полное объявления формы. Так как мы хотим, чтобы пользователь обязательно вносил значение в поле `email` модели `AdvUser`, то выполним полное объявление поля `email` формы. А, поскольку параметры остальных полей формы: `username`, `first_name`, `last_name` и `send_messages` — у нас не меняются, в их отношении мы применим быстрое объявление.

Теперь что касается контроллера. Он должен выполнять правку записи модели, так что мы можем написать его на базе высокоуровневого класса `UpdateView`. Готовый код контроллера-класса `ChangeUserInfoView` показан в листинге 31.9.

Листинг 31.9. Код контроллера-класса `ChangeUserInfoView`

```
from django.views.generic.edit import UpdateView
from django.contrib.messages.views import SuccessMessageMixin
from django.urls import reverse_lazy
from django.shortcuts import get_object_or_404

from .models import AdvUser
from .forms import ChangeUserInfoForm

class ChangeUserInfoView(SuccessMessageMixin, LoginRequiredMixin,
                                               UpdateView):
    model = AdvUser
    template_name = 'main/change_user_info.html'
    form_class = ChangeUserInfoForm
    success_url = reverse_lazy('main:profile')
    success_message = 'Личные данные пользователя изменены'

    def dispatch(self, request, *args, **kwargs):
        self.user_id = request.user.pk
        return super().dispatch(request, *args, **kwargs)

    def get_object(self, queryset=None):
        if not queryset:
            queryset = self.get_queryset()
        return get_object_or_404(queryset, pk=self.user_id)
```

В процессе работы этот контроллер должен извлечь из модели `AdvUser` запись, представляющую текущего пользователя, для чего ему нужно предварительно получить ключ текущего пользователя. А получить его можно из объекта текущего пользователя, хранящегося в атрибуте `user` объекта запроса.

Вероятно, наилучшее место для получения ключа текущего пользователя — метод `dispatch()`, наследуемый всеми контроллерами-классами от их общего суперкласса `View`. Этот метод выполняется в самом начале исполнения контроллера-класса и получает объект запроса в качестве одного из параметров. В переопределенном методе `dispatch()` мы извлечем ключ пользователя и сохраним его в атрибуте `user_id`.

Извлечение исправляемой записи выполняется в методе `get_object()`, которую контроллер-класс унаследовал от примеси `SingleObjectMixin`. В переопределенном методе мы сначала учитываем тот момент, что набор записей, из которого следует извлечь искомую запись, может быть передан методу с параметром `queryset`, а может быть и не передан — в этом случае набор записей следует получить вызовом метода `get_queryset()`. После чего непосредственно ищем запись, представляющую текущего пользователя.

В качестве одного из суперклассов этого контроллера-класса мы указали примесь LoginRequiredMixin, запрещающую доступ к контроллеру гостям, и примесь SuccessMessageMixin, которая применяется для вывода всплывающих сообщений об успешном выполнении операции. Зря мы, что ли, вставили в шаблон templates\layout\basic.html код, выводящий всплывающие сообщения...

Откроем модуль urls.py пакета приложения и вставим код, создающий соответствующий маршрут (добавленный код выделен полужирным шрифтом):

```
from .views import ChangeUserInfoView
. . .
urlpatterns = [
    . . .
    path('accounts/profile/change/', ChangeUserInfoView.as_view(),
                                     name='profile_change'),
    path('accounts/profile/', profile, name='profile'),
    . . .
]
```

Код шаблона templates\main\change_user_info.html, создающего страницу, показан в листинге 31.10. Ничего особо сложного в нем нет.

Листинг 31.10. Код шаблона templates\main\change_user_info.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %}Правка личных данных{% endblock %}

{% block content %}
<h2>Правка личных данных пользователя {{ user.username }}</h2>
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% buttons submit='Сохранить' %}{% endbuttons %}
</form>
{% endblock %}
```

В шаблоне templates\layout\basic.html отыщем тег <a>, выводящий гиперссылку **Изменить личные данные**, и вставим в него интернет-адрес, ведущий на страницу правки основных данных:

```
<a . . . href="{% url 'main:profile_change' %}">Изменить личные
данные</a>
```

Сохраним файлы, перейдем на страницу пользовательского профиля и проверим только что созданную страницу правки личных данных в работе.

31.3.2. Веб-страница правки пароля

Здесь все совсем просто. Контроллер мы делаем на основе контроллера-класса `PasswordChangeView`, который реализует смену пароля. Код нашего контроллера-класса `BBPasswordChangeView`, очень простой и уже знакомый нам по предыдущим контроллерам, показан в листинге 31.11.

Листинг 31.11. Код контроллера-класса `BBPasswordChangeView`

```
from django.contrib.auth.views import PasswordChangeView
. . .
class BBPasswordChangeView(SuccessMessageMixin, LoginRequiredMixin,
                                                PasswordChangeView):
    template_name = 'main/password_change.html'
    success_url = reverse_lazy('main:profile')
    success_message = 'Пароль пользователя изменен'
```

После успешной смены пароля мы выполняем перенаправление на страницу профиля пользователя с выводом соответствующего всплывающего сообщения. Так нам не придется готовить отдельную страницу с сообщением об успешной смене пароля.

И добавим в список маршрутов уровня приложения (модуль urls.py пакета приложения) маршрут, который укажет на новый контроллер:

```
from .views import BBPasswordChangeView
. . .
urlpatterns = [
    . . .
    path('accounts/password/change/', BBPasswordChangeView.as_view(),
                                      name='password_change'),
    . . .
]
```

Листинг 31.12 показывает код шаблона страницы для смены пароля templates\main\password_change.html.

Листинг 31.12. Код шаблона `templates\main\password_change.html`

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %}Смена пароля{% endblock %}

{% block content %}
<h2>Смена пароля пользователя {{ user.username }}</h2>
```

```
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% buttons submit='Сменить пароль' %}{% endbuttons %}
</form>
{% endblock %}
```

Осталось только в коде шаблона templates\layout\basic.html найти код, создающий гиперссылку **Изменить пароль**, и поместить в нее правильный интернет-адрес:

```
<a . . . href="{% url 'main:password_change' %}">Изменить пароль</a>
```

Вот теперь можно сохранить все файлы и проверить, работает ли страница замены пароля.

31.4. Веб-страницы регистрации и активации пользователей

С этого момента начинается настоящая работа, и весьма сложная. Нам нужно написать страницу, на которой посетитель, желающий превратиться в зарегистрированного пользователя, сможет оставить свои личные данные, и страницу для активации нового пользователя. А еще нам придется решать вопросы с занесением новых пользователей в список, формированием интернет-адресов, ведущих на страницу активации, и отправкой электронных писем.

31.4.1. Веб-страницы регистрации нового пользователя

Для выполнения регистрации нам нужно написать форму для ввода сведений о новом пользователе, контроллеры и шаблоны для страниц непосредственно регистрации и подтверждения об успешной регистрации.

Для отправки письма о необходимости активации мы объявим свой сигнал. Сразу же условимся, что называться он будет `user_registrated` и получит в качестве единственного параметра `instance` объект вновь созданного пользователя.

Сигнал будет объявлен в модуле models.py пакета приложения. Этот модуль выполняется непосредственно при инициализации приложения и, таким образом, является идеальным местом для записи кода, объявляющего сигналы.

31.4.1.1. Форма для занесения сведений о новом пользователе

Начнем мы с класса формы, который назовем `RegisterUserForm`. Его код, показанный в листинге 31.13, весьма велик, поскольку нам придется выполнять проверку на корректность занесенного пароля. Мы запишем его в модуль forms.py пакета приложения.

Листинг 31.13. Код формы `RegisterUserForm`

```
from django.contrib.auth import password_validation
from django.core.exceptions import ValidationError

from .models import user_registrated

class RegisterUserForm(forms.ModelForm):
    email = forms.EmailField(required=True,
                             label='Адрес электронной почты')
    password1 = forms.CharField(label='Пароль',
      widget=forms.PasswordInput,
      help_text=password_validation.password_validators_help_text_html())
    password2 = forms.CharField(label='Пароль (повторно)',
      widget=forms.PasswordInput,
      help_text='Введите тот же самый пароль еще раз для проверки')

    def clean_password1(self):
        password1 = self.cleaned_data['password1']
        if password1:
            password_validation.validate_password(password1)
        return password1

    def clean(self):
        super().clean()
        password1 = self.cleaned_data['password1']
        password2 = self.cleaned_data['password2']
        if password1 and password2 and password1 != password2:
            errors = {'password2': ValidationError(
              'Введенные пароли не совпадают', code='password_mismatch')}
            raise ValidationError(errors)

    def save(self, commit=True):
        user = super().save(commit=False)
        user.set_password(self.cleaned_data['password1'])
        user.is_active = False
        user.is_activated = False
        if commit:
            user.save()
        user_registrated.send(RegisterUserForm, instance=user)
        return user

    class Meta:
        model = AdvUser
        fields = ('username', 'email', 'password1', 'password2',
                  'first_name', 'last_name', 'send_messages')
```

Здесь мы также комбинируем быстрое и полное объявление полей. Полное объявление мы используем для создания полей электронной почты (поскольку хотим сделать его обязательным для заполнения) и обоих полей для занесения пароля. Согласно общепринятой практике, мы отведем для занесения пароля два поля, в которые нужно ввести один и тот же пароль.

В качестве дополнительного поясняющего текста у первого поля пароля мы используем объединенный текст с требованиями к вводимому паролю, предоставленный всеми доступными в системе валидаторами, — там новый пользователь сразу поймет, какие требования предъявляются к паролю.

В методе `clean_password1()` мы выполняем валидацию пароля, введенного в первое поле, с применением доступных в системе валидаторов пароля. Проверять таким же образом пароль, введенный во второе поле, нет нужды — если пароль из первого поля некорректен, не имеет значения, является ли корректным пароль из второго поля.

В переопределенном методе `clean()` мы проверяем, совпадают ли оба введенных пароля. Отметим, что эта проверка будет проведена после проверки на корректность пароля из первого поля.

При сохранении нового пользователя нам нужно занести значения `False` в поля `is_active` (признак, является ли пользователь активным) и `is_activated` (признак, выполнил ли пользователь процедуру активации), тем самым сообщая фреймворку, что этот пользователь еще не может выполнять вход на сайт. Также нам нужно записать в модель закодированный пароль и отправить сигнал `user_registrated`, чтобы отослать пользователю письмо с требованием активации. Все это мы выполняем в переопределенном методе `save()`.

31.4.1.2. Средства для регистрации пользователя

Теперь можно приступить к написанию контроллеров и шаблонов для страниц, выполняющих регистрацию пользователя. Таковых будет два: один зарегистрирует пользователя и инициирует отправку письма с сообщением об активации, а другой выведет сообщение о том, что регистрация завершена и письмо отослано.

Контроллер-класс, регистрирующий пользователя, мы назовем `RegisterUserView` и сделаем производным от контроллера-класса `CreateView`. Код нашего контроллера показан в листинге 31.14. Не забываем, что код всех контроллеров записывается в модуле views.py пакета приложения.

Листинг 31.14. Код контроллера-класса `RegisterUserView`

```
from .forms import RegisterUserForm

class RegisterUserView(CreateView):
    model = AdvUser
    template_name = 'main/register_user.html'
    form_class = RegisterUserForm
    success_url = reverse_lazy('main:register_done')
```

Контроллер, который выведет сообщение об успешной регистрации, будет называться `RegisterDoneView` и, в силу его исключительной простоты, станет производным от класса `TemplateView`. Его код можно увидеть в листинге 31.15.

Листинг 31.15. Код контроллера-класса `RegisterDoneView`

```
from django.views.generic.base import TemplateView

class RegisterDoneView(TemplateView):
    template_name = 'main/register_done.html'
```

Откроем модуль urls.py пакета приложения, где записывается список маршрутов уровня приложения, и вставим в этот список два маршрута, ведущих на только что написанные нами контроллеры:

```
from .views import RegisterUserView, RegisterDoneView
. . .
urlpatterns = [
    path('accounts/register/done/', RegisterDoneView.as_view(),
                                    name='register_done'),
    path('accounts/register/', RegisterUserView.as_view(),
                                    name='register'),
    path('accounts/login/', BBLoginView.as_view(), name='login'),
    . . .
]
```

Код шаблонов templates\main\register_user.html и templates\main\register_done.html, которые формируют страницы регистрации и сообщения о ее успешном прохождении, представлен в листингах 31.16 и 31.17 соответственно.

Листинг 31.16. Код шаблона templates\main\register_user.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %}Регистрация{% endblock %}

{% block content %}
<h2>Регистрация нового пользователя</h2>
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% buttons submit='Зарегистрироваться' %}{% endbuttons %}
</form>
{% endblock %}
```

Листинг 31.17. Код шаблона templates\main\register_done.html

```
{% extends "layout/basic.html" %}

{% block title %}Регистрация завершена{% endblock %}

{% block content %}
<h2>Регистрация</h2>
<p>Регистрация пользователя завершена.</p>
<p>На адрес электронной почты, указанный пользователем, выслано письмо
для активации.</p>
{% endblock %}
```

В шаблоне templates\layout\basic.html найдем фрагмент, создающий гиперссылку **Регистрация**, и вставим в нее интернет-адрес, ведущий на страницу регистрации:

```
<a . . . href="{% url 'main:register' %}">Регистрация</a>
```

31.4.1.3. Средства для отправки писем с требованиями активации

Чтобы все написанное нами заработало, нам еще нужно написать код, который выполнит отправку электронных писем с оповещениями о необходимости выполнить активацию.

Условимся, что непосредственную рассылку электронных писем будет выполнять функция `send_activation_notification()`, которую мы объявим чуть позже, во вновь созданном модуле utilities.py. Эта функция еще пригодится нам, когда мы будем писать редактор для модели `AdvUser`.

Откроем модуль models.py пакета приложения и запишем в него код, который объявит сигнал `user_registrated` и привяжет к нему обработчик:

```
from django.dispatch import Signal

from .utilities import send_activation_notification

user_registrated = Signal(providing_args=['instance'])

def user_registrated_dispatcher(sender, **kwargs):
    send_activation_notification(kwargs['instance'])

user_registrated.connect(user_registrated_dispatcher)
```

Создадим в пакете приложения модуль utilities.py. Занесем в него код из листинга 31.18.

Листинг 31.18. Код, реализующий отправку писем с оповещениями об активации

```
from django.template.loader import render_to_string
from django.core.signing import Signer
```

```
from bboard.settings import ALLOWED_HOSTS

signer = Signer()

def send_activation_notification(user):
    if ALLOWED_HOSTS:
        host = 'http://' + ALLOWED_HOSTS[0]
    else:
        host = 'http://localhost:8000'
    context = {'user': user, 'host': host,
               'sign': signer.sign(user.username)}
    subject = render_to_string('email/activation_letter_subject.txt',
                               context)
    body_text = render_to_string('email/activation_letter_body.txt',
                                 context)
    user.email_user(subject, body_text)
```

Чтобы сформировать интернет-адрес, ведущий на страницу подтверждения активации, нам понадобится, во-первых, домен, на котором находится наш сайт, а во-вторых, некоторое значение, уникально идентифицирующее только что зарегистрированного пользователя и при этом устойчивое к попыткам его подделать.

Домен мы можем извлечь из списка разрешенных доменов, который записан в параметре `ALLOWED_HOSTS` настроек проекта. В нашем случае мы используем самый первый домен, что присутствует в списке. Если же список доменов пуст, мы задействуем интернет-адрес, используемый отладочным веб-сервером Django.

В качестве уникального и стойкого к подделке идентификатора пользователя мы применяем его имя, защищенное цифровой подписью. Создание цифровой подписи мы выполняем посредством класса `Signer`.

Текст темы и тела письма мы формируем с применением шаблонов templates\email\activation_letter_subject.txt и templates\email\activation_letter_body.txt соответственно. Код этих шаблонов показан в листингах 31.19 и 31.20.

Листинг 31.19. Код шаблона templates\email\activation_letter_subject.txt

```
Активация пользователя {{ user.username }}
```

Листинг 31.20. Код шаблона templates\email\activation_letter_body.txt

```
Уважаемый пользователь {{ user.username }}!

Вы зарегистрировались на сайте "Доска объявлений".
Вам необходимо выполнить активацию, чтобы подтвердить свою личность.
Для этого пройдите, пожалуйста, по ссылке
```

```
{{ host }}{% url 'main:register_activate' sign=sign %}

До свидания!

С уважением, администрация сайта "Доска объявлений".
```

31.4.2. Веб-страницы активации пользователя

Чтобы реализовать активацию нового пользователя, мы напишем один контроллер и целых три шаблона. Они создадут страницы с сообщением об успешной активации, о том, что активация была выполнена ранее, и о том, что цифровая подпись у идентификатора пользователя, полученного в составе интернет-адреса, скомпрометирована.

Контроллер мы реализуем в виде функции `user_activate()`. Ее код показан в листинге 31.21.

Листинг 31.21. Код контроллера-функции `user_activate()`

```
from django.core.signing import BadSignature

from .utilities import signer

def user_activate(request, sign):
    try:
        username = signer.unsign(sign)
    except BadSignature:
        return render(request, 'main/bad_signature.html')
    user = get_object_or_404(AdvUser, username=username)
    if user.is_activated:
        template = 'main/user_is_activated.html'
    else:
        template = 'main/activation_done.html'
        user.is_active = True
        user.is_activated = True
        user.save()
    return render(request, template)
```

Подписанный идентификатор пользователя, который приходит нам в составе интернет-адреса, мы получаем с параметром `sign`. Далее мы извлекаем из него имя пользователя, ищем пользователя с этим именем, делаем его активным, присвоив значения `True` полям `is_active` и `is_activated` модели, и выводим страницу с сообщением об успешной активации. Если цифровая подпись оказалась скомпрометированной, мы выводим страницу с сообщением о неуспехе активации, а если пользователь был активирован ранее (поле `is_activated` уже хранит значение `True`) — страницу с сообщением, что активация уже произошла.

Для обработки подписанного значения мы используем экземпляр класса `Signer`, созданный в модуле utilities.py и хранящийся в переменной `signer`. Так мы сэкономим оперативную память.

В списке маршрутов уровня приложения (модуль urls.py пакета приложения) запишем маршрут, ведущий на этот контроллер (добавленный код выделен полужирным шрифтом):

```
from .views import user_activate
. . .
urlpatterns = [
    path('accounts/register/activate/<str:sign>/', user_activate,
         name='register_activate'),
    path('accounts/register/done/', RegisterDoneView.as_view(),
         name='register_done'),
    . . .
]
```

Код шаблонов templates\main\activation_done.html, templates\main\bad_signature.html и templates\main\user_is_activated.html страниц с сообщениями, соответственно, об успешной, неуспешной и выполненной ранее активации, показан в листингах 31.22—31.24.

Листинг 31.22. Код шаблона templates\main\activation_done.html

```
{% extends "layout/basic.html" %}

{% block title %}Активация выполнена{% endblock %}

{% block content %}
<h2>Активация</h2>
<p>Пользователь с таким именем успешно активирован.</p>
<p><a href="{% url 'main:login' %}">Войти на сайт</a></p>
{% endblock %}
```

Листинг 31.23. Код шаблона templates\main\bad_signature.html

```
{% extends "layout/basic.html" %}

{% block title %}Ошибка при активации{% endblock %}

{% block content %}
<h2>Активация</h2>
<p>Активация пользователя с таким именем прошла неудачно.</p>
<p><a href="{% url 'main:register' %}">Зарегистрироваться
повторно</a></p>
{% endblock %}
```

Листинг 31.24. Код шаблона templates\main\user_is_activated.html

```
{% extends "layout/basic.html" %}

{% block title %}Пользователь уже активирован{% endblock %}

{% block content %}
<h2>Активация</h2>
<p>Пользователь с таким именем был активирован ранее.</p>
<p><a href="{% url 'main:login' %}">Войти на сайт</a></p>
{% endblock %}
```

Теперь можно все проверить в действии. Только предварительно запишем в модуле settings.py пакета конфигурации параметры используемого нами SMTP-сервера — иначе Django не сможет отправить ни одного электронного письма.

Сохраним файлы с исходным кодом, перейдем на страницу регистрации, введем сведения о новом пользователе и подождем, пока не придет письмо с требованием активации. Перейдем по находящемуся в этом письме интернет-адресу, удостоверимся, что активация прошла успешно, и попытаемся выполнить вход от имени только что созданного пользователя.

Добавим таким же образом еще двух или трех пользователей — они пригодятся нам для реализации удаления пользователей.

31.5. Веб-страница удаления пользователя

Сайты, на которых может зарегистрироваться в качестве пользователя кто угодно, должны предоставлять возможность удаления зарегистрированного пользователя. Давайте и мы сделаем такую штуку, благо уже предусмотрели для нее гиперссылку.

Контроллер-класс `DeleteUserView`, который и выполнит удаление текущего пользователя, мы создадим как производный от класса `DeleteView`. Его код можно увидеть в листинге 31.25.

Листинг 31.25. Код контроллера-класса `DeleteUserView`

```
from django.views.generic.edit import DeleteView
from django.contrib.auth import logout
from django.contrib import messages

class DeleteUserView(LoginRequiredMixin, DeleteView):
    model = AdvUser
    template_name = 'main/delete_user.html'
    success_url = reverse_lazy('main:index')
```

```
    def dispatch(self, request, *args, **kwargs):
        self.user_id = request.user.pk
        return super().dispatch(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        logout(request)
        messages.add_message(request, messages.SUCCESS,
                                    'Пользователь удален')
        return super().post(request, *args, **kwargs)

    def get_object(self, queryset=None):
        if not queryset:
            queryset = self.get_queryset()
        return get_object_or_404(queryset, pk=self.user_id)
```

Здесь мы использовали те же программные приемы, что и в контроллере `ChangeUserInfoView` (см. листинг 31.9). В переопределенном методе `dispatch()` мы сохранили ключ текущего пользователя, а в переопределенном методе `get_object()` отыскали по этому ключу пользователя, подлежащего удалению.

Нужно отметить два момента. Во-первых, перед удалением текущего пользователя необходимо выполнить выход, что мы и сделали в переопределенном методе `post()`. Во-вторых, всплывающее сообщение, созданное примесью `SuccessMessageMixin` перед выходом, после выполнения выхода пропадет, поэтому нам придется создать всплывающее сообщение об успешном удалении пользователя самостоятельно — в том же методе `post()`.

В списке маршрутов уровня приложения (он, как мы помним, хранится в модуле urls.py пакета приложения) запишем следующий код (он выделен полужирным шрифтом), который добавит новый маршрут:

```
from .views import DeleteUserView
. . .
urlpatterns = [
    . . .
    path('accounts/login/', BBLoginView.as_view(), name='login'),
    path('accounts/profile/delete/', DeleteUserView.as_view(),
                                    name='profile_delete'),
    path('accounts/profile/change/', ChangeUserInfoView.as_view(),
                                    name='profile_change'),
    . . .
]
```

Напишем шаблон templates\main\delete_user.html страницы для удаления пользователя. Его код приведен в листинге 31.26.

Листинг 31.26. Код шаблона templates\main\delete_user.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %}Удаление пользователя{% endblock %}

{% block content %}
<h2>Удаление пользователя {{ object.username }}</h2>
<form method="post">
    {% csrf_token %}
    {% buttons submit='Удалить' %}{% endbuttons %}
</form>
{% endblock %}
```

Нам останется только записать в шаблоне templates\layout\basic.html интернет-адрес, ведущий на страницу удаления пользователя:

```
<a . . . href="{% url 'main:profile_delete' %}">Удалить</a>
```

Для проверки попробуем войти на сайт от имени одного из ранее созданных пользователей (только не суперпользователя — иначе придется создавать его заново) и выполнить его удаление.

31.6. Инструменты для администрирования пользователей

Напоследок подготовим инструменты для администрирования пользователей, а именно редактор, посредством которого администрация сайта будет работать с зарегистрированными пользователями. В редактор мы добавим возможность фильтрации пользователей по именам, адресам электронной почты, настоящим именам и фамилиям. Также мы реализуем вывод пользователей, уже выполнивших активацию, не выполнивших ее в течение 3 дней и недели. А еще мы добавим в редактор действие, выполняющее отправку выбранным в списке пользователям электронных писем с уведомлениями о необходимости выполнить активацию.

Полный код класса редактора `AdvUserAdmin`, вспомогательного класса и функции показан в листинге 31.27. Этот код следует занести в модуль admin.py пакета приложения, заменив им имеющийся там код.

Листинг 31.27. Код редактора `AdvUserAdmin`

```
from django.contrib import admin
import datetime

from .models import AdvUser
from .utilities import send_activation_notification
```

```
def send_activation_notifications(modeladmin, request, queryset):
    for rec in queryset:
        if not rec.is_activated:
            send_activation_notification(rec)
    modeladmin.message_user(request, 'Письма с оповещениями отправлены')
send_activation_notifications.short_description = 'Отправка писем с ' + \
'оповещениями об активации'

class NonactivatedFilter(admin.SimpleListFilter):
    title = 'Прошли активацию?'
    parameter_name = 'actstate'

    def lookups(self, request, model_admin):
        return (
                    ('activated', 'Прошли'),
                    ('threedays', 'Не прошли более 3 дней'),
                    ('week', 'Не прошли более недели'),
                )

    def queryset(self, request, queryset):
        val = self.value()
        if val == 'activated':
            return queryset.filter(is_active=True, is_activated=True)
        elif val == 'threedays':
            d = datetime.date.today() - datetime.timedelta(days=3)
            return queryset.filter(is_active=False, is_activated=False,
                                   date_joined__date__lt=d)
        elif val == 'week':
            d = datetime.date.today() - datetime.timedelta(weeks=1)
            return queryset.filter(is_active=False, is_activated=False,
                                   date_joined__date__lt=d)

class AdvUserAdmin(admin.ModelAdmin):
    list_display = ('__str__', 'is_activated', 'date_joined')
    search_fields = ('username', 'email', 'first_name', 'last_name')
    list_filter = (NonactivatedFilter,)
    fields = (('username', 'email'), ('first_name', 'last_name'),
              ('send_messages', 'is_active', 'is_activated'),
              ('is_staff', 'is_superuser'),
              'groups', 'user_permissions',
              ('last_login', 'date_joined'))
    readonly_fields = ('last_login', 'date_joined')
    actions = (send_activation_notifications,)

admin.site.register(AdvUser, AdvUserAdmin)
```

Здесь в списке записей мы указываем выводить строковое представление записи (имя пользователя — как реализовано в модели `AbstractUser`, от которой наследует наша модель), поле признака, выполнил ли пользователь активацию, дату и время его регистрации. Также мы разрешаем выполнять фильтрацию по полям имени, адреса электронной почты, настоящих имени и фамилии.

Для выполнения фильтрации пользователей, выполнивших активацию, не выполнивших ее в течение 3 дней и недели, мы используем класс `NonactivatedFilter`. Обратим внимание на код, непосредственно фильтрующий пользователей по значению даты их регистрации.

Мы явно указываем список полей, которые должны выводиться в формах для правки пользователей, чтобы выстроить их в удобном для работы порядке. Поля даты регистрации пользователя и последнего его входа на сайт мы делаем доступными только для чтения.

Наконец, мы регистрируем действие, которое разошлет пользователям письма с предписаниями выполнить активацию. Это действие реализовано функцией `send_activation_notifications()`. В ней мы перебираем всех выбранных пользователей и для каждого, кто не выполнил активацию, вызываем функцию `send_activation_notification()`, объявленную нами ранее в модуле utilities.py и непосредственно выполняющую отправку писем.

На этом все. Сохраним весь исправленный код и проверим написанный нами редактор в действии.

В качестве домашнего задания реализуйте на сайте сброс пароля и авторизацию через социальную сеть «ВКонтакте». Все необходимые для этого сведения были даны в предыдущих главах этой книги, так что вы, уважаемые читатели, справитесь с этим без особого труда.

ГЛАВА 32

Рубрики

Закончив с инструментами для разграничения доступа, приступим к наделению нашего сайта рубриками.

Как условились в *главе 30*, для удобства мы реализуем двухуровневую структуру рубрик: более общие рубрики верхнего уровня и вложенные в них рубрики нижнего уровня. Чтобы не путаться, назовем их, соответственно, надрубриками и подрубриками.

В этой главе мы напишем базовую модель, в которой будут храниться и надрубрики, и подрубрики, две производные от нее прокси-модели: для надрубрик и подрубрик, а также редакторы для административного сайта. И не забудем написать код, который будет выводить список рубрик в вертикальной панели навигации на каждой странице сайта.

32.1. Модели рубрик

Итак, нам понадобится написать три модели: базовую и две производные. Причем последние мы реализуем в виде прокси-моделей.

32.1.1. Базовая модель рубрик

Базовой модели, в которой будут храниться и надрубрики, и подрубрики, мы дадим имя `Rubric`. Ее структура приведена в табл. 32.1.

Таблица 32.1. *Структура модели* `Rubric`

Имя	Тип	Дополнительные параметры	Описание
`name`	`CharField`	Длина — 20 символов, индексированное	Название
`order`	`IntegerField`	Значение по умолчанию — 0, индексированное	Порядок
`super_rubric`	`ForeignKey`	Необязательное, запрет каскадного удаления	Надрубрика

Назначение первого поля понятно без дополнительных пояснений. Второе поле будет хранить целое число, обозначающее порядок следования рубрик друг за другом: при выводе рубрики сначала будут сортироваться по возрастанию значения порядка, а уже потом — по их названиям. А вот с третьим полем все сложнее...

Третье поле с именем `super_rubric` будет хранить надрубрику, к которой относится текущая подрубрика. Оно будет иметь следующие важные особенности:

- это поле будет заполняться только в том случае, если запись хранит подрубрику. Если запись хранит надрубрику, поле заполнять не нужно (собственно, отсутствие значения в этом поле является признаком надрубрики — ведь надрубрика в принципе не может ссылаться на надрубрику). Следовательно, мы сделаем это поле необязательным к заполнению;
- нужно обязательно запретить каскадное удаление записей, чтобы пользователь по ошибке не удалил надрубрику вместе со всеми подрубриками;
- связь, создаваемая этим полем, должна устанавливаться с моделью надрубрик, которую мы объявим чуть позже. Условимся назвать эту модель `SuperRubric`.

Код класса модели `Rubric` показан в листинге 32.1. Не забываем, что код всех моделей заносится в модуль models.py пакета приложения.

Листинг 32.1. Код модели `Rubric`

```
class Rubric(models.Model):
    name = models.CharField(max_length=20, db_index=True, unique=True,
                            verbose_name='Название')
    order = models.SmallIntegerField(default=0, db_index=True,
                                     verbose_name='Порядок')
    super_rubric = models.ForeignKey('SuperRubric',
                   on_delete=models.PROTECT, null=True, blank=True,
                   verbose_name='Надрубрика')
```

Обратим внимание, что мы не задаем никаких параметров самой модели — поскольку пользователи не будут работать с этой моделью непосредственно, здесь это совершенно излишне.

32.1.2. Модель надрубрик

Для работы с надрубриками мы объявим модель `SuperRubric` — прокси-модель, производную от `Rubric` (как мы помним, прокси-модель позволяет менять лишь функциональность модели, но не набор объявленных в ней полей, однако нам и надо изменить лишь функциональность модели). Мы сделаем так, чтобы она обрабатывала только надрубрики.

Чтобы изменить состав обрабатываемых моделью записей, нужно задать для нее свой диспетчер записей, который и укажет необходимые условия фильтрации. Следовательно, нам нужно еще и написать класс диспетчера записей.

Код обоих классов: и модели SuperRubric, и диспетчера записей SuperRubricManager — представлен в листинге 32.2.

Листинг 32.2. Код модели SuperRubric и диспетчера записей SuperRubricManager

```
class SuperRubricManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(super_rubric__isnull=True)

class SuperRubric(Rubric):
    objects = SuperRubricManager()

    def __str__(self):
        return self.name

    class Meta:
        proxy = True
        ordering = ('order', 'name')
        verbose_name = 'Надрубрика'
        verbose_name_plural = 'Надрубрики'
```

Условия фильтрации записей мы указываем в переопределенном методе get_queryset() класса диспетчера записей SuperRubricManager. Мы выбираем только записи с пустым полем super_rubric, т. е. надрубрики.

В самом классе модели SuperRubric мы задаем диспетчер записей SuperRubricManager в качестве основного. И не забываем объявить метод __str__(), который станет генерировать строковое представление надрубрики — ее название.

Как и условились ранее, мы указываем для модели порядок сортировки записей сначала по возрастанию значения порядка, а потом — по названию.

32.1.3. Модель подрубрик

Модель подрубрик SubRubric мы создадим тем же образом, что и модель надрубрик. Только теперь диспетчер записей, который мы объявим для нее, будет возвращать лишь подрубрики.

Код классов модели SubRubric и диспетчера записей SubRubricManager можно увидеть в листинге 32.3.

Листинг 32.3. Код модели SubRubric и диспетчера записей SubRubricManager

```
class SubRubricManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(super_rubric__isnull=False)

class SubRubric(Rubric):
    objects = SubRubricManager()
```

```
    def __str__(self):
        return '%s - %s' % (self.super_rubric.name, self.name)

    class Meta:
        proxy = True
        ordering = ('super_rubric__order', 'super_rubric__name', 'order',
                    'name')
        verbose_name = 'Подрубрика'
        verbose_name_plural = 'Подрубрики'
```

Диспетчер записей `SubRubricManager` будет отбирать для нас лишь записи с непустым полем `super_rubric` (т. е. подрубрики). Строковое представление, которое станет создавать для нас модель, будет выполнено в формате *<название надрубрики> - <название подрубрики>*. А сортировку записей мы укажем по порядку надрубрики, названию надрубрики, порядку подрубрики и названию подрубрики.

Объявив все необходимые классы, мы можем, предварительно остановив отладочный веб-сервер, произвести создание и выполнение миграций:

```
manage.py makemigrations
manage.py migrate
```

32.2. Инструменты для администрирования рубрик

Поскольку вся работа над надрубриками и подрубриками будет проводиться средствами административного сайта, нам понадобится написать для обеих наших моделей редакторы и кое-какие вспомогательные классы.

Для надрубрик мы создадим встроенный редактор, чтобы пользователь, добавив новую надрубрику, смог сразу же заполнить ее подрубриками. Из формы для ввода и правки надрубрик мы исключим поле надрубрики (`super_rubric`), поскольку оно там совершенно не нужно и, более того, собьет пользователя с толку.

Листинг 32.4 показывает код классов редактора `SuperRubricAdmin` и встроенного редактора `SubRubricInline`. Не забываем, что код редакторов, обычных и встроенных, равно как и код, регистрирующий модели и редакторы в подсистеме административного сайта, должен записываться в модуль admin.py пакета приложения.

Листинг 32.4. Код редактора `SuperRubricAdmin` и встроенного редактора `SubRubricInline`

```
from .models import SuperRubric, SubRubric

class SubRubricInline(admin.TabularInline):
    model = SubRubric
```

```
class SuperRubricAdmin(admin.ModelAdmin):
    exclude = ('super_rubric',)
    inlines = (SubRubricInline,)

admin.site.register(SuperRubric, SuperRubricAdmin)
```

Для работы с подрубриками нам понадобится сделать поле надрубрики (super_rubric) обязательным для заполнения. Для этого мы объявим форму SubRubricForm, записав ее код, показанный в листинге 32.5, в модуле forms.py пакета приложения.

Листинг 32.5. Код формы SubRubricForm

```
from .models import SuperRubric, SubRubric

class SubRubricForm(forms.ModelForm):
    super_rubric = forms.ModelChoiceField(
                   queryset=SuperRubric.objects.all(), empty_label=None,
                   label='Надрубрика', required=True)

    class Meta:
        model = SubRubric
        fields = '__all__'
```

Мы убрали у раскрывающегося списка, с помощью которого пользователь будет выбирать подрубрику, «пустой» пункт, присвоив параметру empty_label конструктора класса поля ModelChoiceField значение None. Так мы дополнительно дадим понять, что в это поле обязательно должно быть занесено значение.

Теперь мы можем написать код, объявляющий класс редактора SubRubricAdmin. Этот код можно увидеть в листинге 32.6.

Листинг 32.6. Код редактора SubRubricAdmin

```
from .forms import SubRubricForm

class SubRubricAdmin(admin.ModelAdmin):
    form = SubRubricForm

admin.site.register(SubRubric, SubRubricAdmin)
```

Сохраним весь исправленный код, запустим отладочный веб-сервер (не забыв при этом отключить обработку статических файлов), войдем на административный сайт и добавим несколько надрубрик и подрубрик.

32.3. Вывод списка рубрик в панели навигации

Напоследок займемся вертикальной панелью навигации на страницах нашего сайта. Нам нужно сделать так, чтобы в ней выводились пункты, представляющие все созданные к данному моменту рубрики.

Первое, что нам необходимо сделать, — поместить в состав контекста каждого шаблона переменную, в которой хранится список подрубрик (именно на его основе мы будем формировать пункты панели навигации). Можно создавать такую переменную в каждом контроллере, но это очень трудоемко. Поэтому мы объявим и зарегистрируем в проекте обработчик контекста, в котором и будет формироваться список подрубрик.

Условимся, что список подрубрик будет помещаться в переменную `rubrics` контекста шаблона.

Создадим в пакете приложения модуль middlewares.py и запишем в него код обработчика контекста `bboard_context_processor()`, показанный в листинге 31.7.

Листинг 31.7. Код обработчика контекста `bboard_context_processor()`

```
from .models import SubRubric

def bboard_context_processor(request):
    context = {}
    context['rubrics'] = SubRubric.objects.all()
    return context
```

Откроем модуль settings.py пакета конфигурации и зарегистрируем только что написанный обработчик контекста. Для этого добавим имя этого обработчика в список `context_processors` из параметра `OPTIONS` (добавленный код выделен полужирным шрифтом):

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            'context_processors': [
                . . .
                'main.middlewares.bboard_context_processor',
            ],
        },
    },
]
```

Теперь выполним еще пару подготовительных действий. Во-первых, в модуле views.py пакета приложения объявим контроллер by_rubric():

```
def by_rubric(request, pk):
    pass
```

Во-вторых, добавим в список маршрутов уровня приложения (он хранится в модуле urls.py пакета приложения) маршрут, ведущий на этот контроллер:

```
from .views import by_rubric
. . .
urlpatterns = [
    . . .
    path('<int:pk>/', by_rubric, name='by_rubric'),
    path('<str:page>/', other_page, name='other'),
    . . .
]
```

Объявление этого маршрута мы поместим перед объявлением маршрута, ведущего на контроллер other_page(), что выводит на экран вспомогательные страницы. Если же мы поместим его после упомянутого ранее маршрута, при просмотре списка маршрутов Django примет присутствующий в интернет-адресе ключ рубрики за имя шаблона страницы и запустит контроллер other_page(), что приведет к ошибке 404.

Зачем мы объявляли эти контроллер и маршрут? Чтобы прямо сейчас сформировать в панели навигации гиперссылки с правильными интернет-адресами. В *главе 33* мы заменим контроллер-«заглушку» другим, выполняющим полезную работу — вывод объявлений из выбранной посетителем рубрики.

Откроем шаблон layout\basic.html (давайте ради краткости не указывать папку templates в путях к шаблонам — мы уже давно знаем, что шаблоны хранятся в папке templates) и исправим код панели навигации следующим образом (исправленный код выделен полужирным шрифтом):

```
<a class="nav-link root" href="{% url 'main:index' %}">Главная</a>
{% for rubric in rubrics %}
{% ifchanged rubric.super_rubric.pk %}
<span class="nav-link root font-weight-bold">
{{ rubric.super_rubric.name }}</span>
{% endifchanged %}
<a class="nav-link" href="{% url 'main:by_rubric' pk=rubric.pk %}">
{{ rubric.name }}</a>
{% endfor %}
<a class="nav-link root" href="{% url 'main:other' page='about' %}">О
сайте</a>
```

Здесь все просто. Мы перебираем список подрубрик, хранящийся в переменной rubrics контекста шаблона (эту переменную создал наш обработчик контекста bboard_context_processor()), и для каждой подрубрики выводим:

- если ключ связанной надрубрики изменился (т. е., если начали выводиться подрубрики из другой надрубрики) — пункт с именем надрубрики;
- пункт-гиперссылку с именем подрубрики.

Сохраним код, на всякий случай перезапустим отладочный веб-сервер, обновим открытую в веб-обозревателе страницу и полюбуемся на список рубрик. Только не будем щелкать на них — ни к чему хорошему это пока не приведет.

ГЛАВА 33

Объявления

Теперь можно приступать к работе над объявлениями. Мы создадим модель самих объявлений, модель дополнительных иллюстраций к ним, страницу для просмотра списка объявлений, относящихся к выбранной рубрике, с поддержкой пагинации и поиска, страницу сведений о выбранном объявлении, страницы для добавления, правки и удаления объявлений. И не забудем сделать так, чтобы на странице профиля выводился список объявлений, оставленных текущим пользователем.

33.1. Подготовка к обработке выгруженных файлов

В составе объявлений у нас будет присутствовать графическое изображение с основной иллюстрацией к продаваемому товару. Помимо этого, пользователь может создать для объявления произвольное количество дополнительных иллюстраций, также представляющих собой изображения.

Чтобы Django смог обработать выгруженные посетителями файлы, нам необходимо установить три дополнительных библиотеки: Pillow (обеспечивает поддержку графики), Easy Thumbnails (создает миниатюры) и django-cleanup (удаляет выгруженные файлы после удаления хранящих их записей моделей). Установим их, отдав команды:

```
pip install pillow
pip install easy-thumbnails
pip install django-cleanup
```

Добавим программные ядра двух последних библиотек: приложения `easy_thumbnails` и `django_cleanup` — в список зарегистрированных в проекте. Для этого откроем модель settings.py пакета конфигурации и добавим в список параметра `INSTALLED_APPS` псевдонимы этих приложений:

```
INSTALLED_APPS = [
    . . .
```

```
    'django_cleanup',
    'easy_thumbnails',
]
```

Для хранения самих выгруженных файлов мы отведем папку media, которую создадим в папке проекта. Для хранения миниатюр мы создадим в ней папку thumbnails.

В модуле settings.py укажем путь к папке media и префикс для интернет-адресов выгруженных файлов:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

И сразу же добавим туда настройки приложения easy_thumbnails:

```
THUMBNAIL_ALIASES = {
    '': {
        'default': {
            'size': (96, 96),
            'crop': 'scale',
        },
    },
}
THUMBNAIL_BASEDIR = 'thumbnails'
```

Мы создали для миниатюр один-единственный пресет, используемый по умолчанию для всех случаев. Он укажет создавать миниатюры простым масштабированием до размеров 96×96 пикселов. Также мы задали имя вложенной папки, в которой будут храниться миниатюры, — thumbnails.

И наконец откроем модель urls.py пакета конфигурации, где хранится список маршрутов уровня проекта, и добавим в него код, создающий маршрут для обработки выгруженных файлов (выделен полужирным шрифтом):

```
from django.conf.urls.static import static

urlpatterns = [
    . . .
]

if settings.DEBUG:
    urlpatterns.append(path('static/<path:path>',
                            never_cache(serve)))
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

33.2. Модели объявлений и дополнительных иллюстраций

Теперь можно приступать к работе над обеими моделями: объявлений и дополнительных иллюстраций.

33.2.1. Модель самих объявлений

Модель, хранящая объявления, будет называться `Bb`. Ее структура приведена в табл. 33.1.

***Таблица 33.1.** Структура модели `Bb`*

Имя	Тип	Дополнительные параметры	Описание
`rubric`	`ForeignKey`	Запрет каскадного удаления	Подрубрика
`title`	`CharField`	Длина — 40 символов	Название товара
`content`	`TextField`		Описание товара
`price`	`IntegerField`	Значение по умолчанию — 0	Цена товара
`contacts`	`TextField`		Контакты
`image`	`ImageField`	Необязательное	Основная иллюстрация к объявлению
`author`	`ForeignKey`		Пользователь, оставивший объявление
`is_active`	`BooleanField`	Значение по умолчанию — `True`, индексированное	Признак, показывать ли объявление в списке
`created_at`	`DateTimeField`	Подставляется текущее значение даты и времени, индексированное	Дата и время публикации объявления

В поле `rubric`, устанавливающем связь с моделью подрубрик `SubRubric`, мы указали запрет каскадного удаления, чтобы предотвратить случайное удаление подрубрики вместе со всеми оставленными в ней объявлениями.

Графические файлы, сохраняемые в поле `image` модели, будут иметь в качестве имен текущие отметки времени. Так мы приведем имена к единому типу и заодно устраним ситуацию, когда выгруженный файл имеет настолько длинное имя, что оно не помещается в отведенное для хранения файла поле.

При разработке модели объявления нам нужно учесть еще один момент. Чуть позже мы напишем модель дополнительных иллюстраций, которую свяжем с моделью объявлений связью «один-со-многими». Если при объявлении этой связи мы разрешим каскадное удаление, то при удалении объявления будут удалены все относящиеся к ней дополнительные иллюстрации. Но это удаление выполнит не Django, а СУБД, отчего приложение `django_cleanup` не получит сигнала об удалении записей и не сможет в ответ удалить хранящиеся в них графические файлы. В результате эти файлы останутся на диске бесполезным мусором.

Мы обязательно решим эту проблему, но для этого нам понадобится условиться об имени модели дополнительных иллюстраций. Пусть эта модель называется `AdditionalImage`.

В *главе 31* мы создали в пакете приложения модуль utilities.py, в который записали объявление функции, выполняющей отправку писем. Этот модуль — отличное

место для сохранения кода, не относящегося напрямую ни к моделям, ни к редакторам, ни к контроллерам. Давайте поместим в него объявление функции `get_timestamp_path()`, которая будет генерировать имена сохраняемых в модели выгруженных файлов (листинг 31.1).

Листинг 33.1. Код функции `get_timestamp_path()`

```
from datetime import datetime
from os.path import splitext

def get_timestamp_path(instance, filename):
    return '%s%s' % (datetime.now().timestamp(), splitext(filename)[1])
```

Теперь мы можем написать код, объявляющий сам класс модели `Bb`. Готовый код показан в листинге 33.2.

Листинг 33.2. Код модели `Bb`

```
from .utilities import get_timestamp_path

class Bb(models.Model):
    rubric = models.ForeignKey(SubRubric, on_delete=models.PROTECT,
                                          verbose_name='Рубрика')
    title = models.CharField(max_length=40, verbose_name='Товар')
    content = models.TextField(verbose_name='Описание')
    price = models.FloatField(default=0, verbose_name='Цена')
    contacts = models.TextField(verbose_name='Контакты')
    image = models.ImageField(blank=True, upload_to=get_timestamp_path,
                              verbose_name='Изображение')
    author = models.ForeignKey(AdvUser, on_delete=models.CASCADE,
                               verbose_name='Автор объявления')
    is_active = models.BooleanField(default=True, db_index=True,
                                    verbose_name='Выводить в списке?')
    created_at = models.DateTimeField(auto_now_add=True, db_index=True,
                                      verbose_name='Опубликовано')

    def delete(self, *args, **kwargs):
        for ai in self.additionalimage_set.all():
            ai.delete()
        super().delete(*args, **kwargs)

    class Meta:
        verbose_name_plural = 'Объявления'
        verbose_name = 'Объявление'
        ordering = ['-created_at']
```

Единственное, на что стоит обратить внимание, — переопределенный метод `delete()`. В нем, перед тем как удалить текущую запись, мы перебираем и вызовом метода `delete()` удаляем все дополнительные иллюстрации, связанные с этой записью. При вызове метода `delete()` возникает сигнал `post_delete`, обрабатываемый приложением `django_cleanup`, которое в ответ удалит все файлы, хранящиеся в только что удаленной записи.

33.2.2. Модель дополнительных иллюстраций

Модель дополнительных иллюстраций мы назовем, как условились в *разд. 33.2.1*, `AdditionalImage`. Она будет существенно проще предыдущей — достаточно посмотреть на табл. 33.2, где представлена ее структура.

Таблица 33.2. *Структура модели* `AdditionalImage`

Имя	Тип	Описание
bb	ForeignKey	Объявление, к которому относится иллюстрация
image	ImageField	Собственно иллюстрация

Графические файлы, сохраняемые в поле `image` этой модели, также получат в качестве имен текущие отметки времени. Для формирования имен файлов мы применим функцию `get_timestamp_path()` из модуля utilities.py.

Готовый код модели `AdditionalImage` можно увидеть в листинге 33.3.

Листинг 33.3. Код модели `AdditionalImage`

```
class AdditionalImage(models.Model):
    bb = models.ForeignKey(Bb, on_delete=models.CASCADE,
                           verbose_name='Объявление')
    image = models.ImageField(upload_to=get_timestamp_path,
                              verbose_name='Изображение')

    class Meta:
        verbose_name_plural = 'Дополнительные иллюстрации'
        verbose_name = 'Дополнительная иллюстрация'
```

Сохраним код моделей, произведем создание и выполнение миграций. И сделаем еще одно очень важное дело.

33.2.3. Реализация удаления объявлений в модели пользователя

В *разд. 33.2.1* мы сделали так, чтобы при удалении объявления явно удалялись все связанные с ним дополнительные иллюстрации. Это нужно для того, чтобы приложение `django_cleanup` удалило все файлы, содержащие удаленные иллюстрации.

Сейчас нам нужно предусмотреть явное удаление объявлений, оставленных пользователем, при удалении этого пользователя. Для этого мы добавим в код модели `AdvUser` следующий фрагмент (выделен полужирным шрифтом):

```
class AdvUser(AbstractUser):
    . . .
    def delete(self, *args, **kwargs):
        for bb in self.bb_set.all():
            bb.delete()
        super().delete(*args, **kwargs)
    . . .
```

33.3. Инструменты для администрирования объявлений

Для пользователей со статусом персонала, которые будут работать с объявлениями посредством административного сайта, мы создадим инструменты для администрирования объявлений. Мы объявим редактор `BbAdmin`, обеспечивающий работу с самими объявлениями, и встроенный редактор `AdditionalImageInline`, посредством которого пользователь сможет работать с дополнительными иллюстрациями непосредственно на страницах добавления и правки объявлений.

Листинг 33.4. Код редактора `BbAdmin` и встроенного редактора `AdditionalImageInline`

```
from .models import Bb, AdditionalImage

class AdditionalImageInline(admin.TabularInline):
    model = AdditionalImage

class BbAdmin(admin.ModelAdmin):
    list_display = ('rubric', 'title', 'content', 'author', 'created_at')
    fields = (('rubric', 'author'), 'title', 'content', 'price',
              'contacts', 'image', 'is_active')
    inlines = (AdditionalImageInline,)

admin.site.register(Bb, BbAdmin)
```

На страницах добавления и правки объявлений мы выведем раскрывающиеся списки подкатегории и пользователя в одну строку — ради компактности.

Сохраним код, запустим отладочный веб-сервер, войдем на административный сайт и добавим несколько объявлений, обязательно с дополнительными иллюстрациями. Попробуем исправить одно объявление и удалить другое, проверив, действительно ли при этом будут удалены все файлы, хранящиеся в самом объявлении, и связанные с ним иллюстрации.

33.4. Вывод объявлений

Занеся в базу данных сайта несколько объявлений (хотя бы три), мы можем приступать к реализации вывода объявлений. Мы создадим две страницы:

- страницу списка объявлений, относящихся к выбранной посетителем рубрике, с поддержкой пагинации и поиска объявлений по введенному слову;
- страницу сведений о выбранном объявлении, на которой будут выводиться также и дополнительные иллюстрации.

Кроме того, нам нужно реализовать вывод на главной странице десяти наиболее «свежих» объявлений.

33.4.1. Вывод списка объявлений

Вывод списка объявлений — достаточно сложная задача. Нам понадобится форма, в которую посетитель будет заносить искомое слово (разумеется, это должна быть обычная форма, не привязанная к модели), контроллер и шаблон. А еще нам придется решить весьма серьезную проблему корректного возврата, о которой мы поговорим чуть позже.

33.4.1.1. Форма поиска и контроллер списка объявлений

Прежде чем писать код формы, давайте условимся, что искомое слово, введенное посетителем, мы будем пересылать контроллеру методом GET в GET-параметре с именем `keyword`. Следовательно, поле в форме для ввода искомого слова мы назовем точно так же.

Код формы поиска `SearchForm` очень прост — убедимся в этом сами, взглянув на листинг 33.5.

Листинг 33.5. Код формы `SearchForm`

```
class SearchForm(forms.Form):
    keyword = forms.CharField(required=False, max_length=20, label='')
```

Поскольку посетитель может ввести в поле `keyword` искомое слово, а может и не ввести (чтобы отменить выполненный ранее поиск и вновь вывести все объявления из списка), мы объявили это поле необязательным к заполнению. А еще мы убрали у этого поля надпись, присвоив параметру `label` пустую строку — все равно такого рода поля выводятся без надписей.

В *главе 32*, чтобы проверить написанный тогда код, мы создали ничего не делающий контроллер-функцию `by_rubric()`. Поскольку наш контроллер будет относительно нетривиальным, а нетривиальные вещи проще реализовывать в виде контроллеров-функций, давайте оставим его в таком виде. Только перепишем код контроллера, чтобы он выглядел, как представлено в листинге 33.6.

Листинг 33.6. Код контроллера-функции by_rubric()

```
from django.core.paginator import Paginator
from django.db.models import Q

from .models import SubRubric, Bb
from .forms import SearchForm

def by_rubric(request, pk):
    rubric = get_object_or_404(SubRubric, pk=pk)
    bbs = Bb.objects.filter(is_active=True, rubric=pk)
    if 'keyword' in request.GET:
        keyword = request.GET['keyword']
        q = Q(title__icontains=keyword) | Q(content__icontains=keyword)
        bbs = bbs.filter(q)
    else:
        keyword = ''
    form = SearchForm(initial={'keyword': keyword})
    paginator = Paginator(bbs, 2)
    if 'page' in request.GET:
        page_num = request.GET['page']
    else:
        page_num = 1
    page = paginator.get_page(page_num)
    context = {'rubric': rubric, 'page': page, 'bbs': page.object_list,
               'form': form}
    return render(request, 'main/by_rubric.html', context)
```

Здесь мы извлекаем выбранную посетителем рубрику — нам понадобится вывести на странице ее название. Затем извлекаем объявления, относящиеся к этой рубрике и помеченные для вывода (те, у которых поле `is_active` хранит значение `True`). После чего выполняем фильтрацию уже отобранных объявлений по введенному посетителем искомому слову. Это слово, как мы условились ранее, передается через GET-параметр `keyword`.

Вот фрагмент кода, «отвечающий» за фильтрацию объявлений по введенному посетителем слову:

```
if 'keyword' in request.GET:
    keyword = request.GET['keyword']
    q = Q(title__icontains=keyword) | Q(content__icontains=keyword)
    bbs = bbs.filter(q)
else:
    keyword = ''
form = SearchForm(initial={'keyword': keyword})
```

Ради простоты мы получаем искомое слово непосредственно из GET-параметра `keyword`. Затем формируем на основе полученного слова условие фильтрации, применив объект `Q`, и выполняем фильтрацию объявлений.

Следующее, что нам нужно сделать, — создать экземпляр формы `SearchForm`, чтобы вывести ее на экран. Конструктору ее класса в параметре `initial` мы передаем полученное из GET-параметра `keyword` искомое слово — так что это слово будет присутствовать в выведенной на экран форме.

Не забываем создать пагинатор, указав количество записей в одной его части равным `2`. Это позволит нам проверить, работает ли пагинация, имея в базе данных всего три-четыре объявления. Наконец, выводим страницу со списком объявлений, применив шаблон main\by_rubric.html. С написанием которого нам придется подождать...

33.4.1.2. Реализация корректного возврата

Предположим, что мы написали весь код, что будет выводить на экран и списки объявлений, разбитые на рубрики, и сведения о выбранном объявлении. И вот, посетитель заходит на наш сайт, выбирает какую-либо рубрику, пролистывает несколько частей, сформированных пагинатором, находит нужное ему объявление и щелкает на гиперссылке, чтобы просмотреть это объявление полностью. Открывается страница со сведениями об объявлении, посетитель смотрит их, после чего щелкает на гиперссылке для возврата на список объявлений... И попадает на самую первую часть этого списка.

То же самое произойдет, если посетитель выполнит поиск, а уже потом отправится смотреть сведения о каком-либо объявлении. Когда он щелкнет на гиперссылке возврата, то вернется в изначальный список объявлений, в котором не был выполнен поиск.

Как избежать этой проблемы, в общем, понятно. Номер выводимой части и искомое слово у нас передаются посредством GET-параметров с именами `page` и `keyword` соответственно. Тогда, чтобы вернуться на нужную часть списка уже отфильтрованных по заданному слову объявлений, нам нужно передать эти параметры странице сведений об объявлении.

Конечно, готовый набор GET-параметров можно получить из элемента с ключом `QUERY_STRING` словаря, что хранится в атрибуте `META` объекта запроса. Но в нашем случае нежелательно передавать параметр `page`, если его значение равно 1, и параметр `keyword` с «пустой» строкой в качестве значения, — это их значения по умолчанию.

Также можно формировать набор GET-параметров в контроллере. Но, по принятым в Django соглашениям, весь код, «ответственный» за формирование страниц, следует помещать в шаблон, посредник или — наш случай! — обработчик контекста.

Откроем модуль middlewares.py пакета приложения и найдем в нем код обработчика контекста `bboard_context_processor()`, написанный нами в *главе 32*. Вставим в него фрагмент, который создаст в контексте шаблона две переменные:

- `keyword` — с полностью сформированным GET-параметром `keyword`, который понадобится нам для генерирования интернет-адресов в гиперссылках навигатора;

- `all` — с полностью сформированными GET-параметрами `keyword` и `page`, которые мы добавим к интернет-адресам гиперссылок, указывающих на страницы сведений об объявлениях.

Добавленный нами код, как обычно, выделен полужирным шрифтом и не использует никаких особо сложных приемов программирования:

```
def bboard_context_processor(request):
    context = {}
    context['rubrics'] = SubRubric.objects.all()
    context['keyword'] = ''
    context['all'] = ''
    if 'keyword' in request.GET:
        keyword = request.GET['keyword']
        if keyword:
            context['keyword'] = '?keyword=' + keyword
            context['all'] = context['keyword']
    if 'page' in request.GET:
        page = request.GET['page']
        if page != '1':
            if context['all']:
                context['all'] += '&page=' + page
            else:
                context['all'] = '?page=' + page
    return context
```

33.4.1.3. Шаблон страницы списка объявлений

Теперь мы можем приступать к написанию кода шаблона main\by_rubric.html, который сформирует страницу списка объявлений. Код этого шаблона можно увидеть в листинге 33.7.

Листинг 33.7. Код шаблона main\by_rubric.html

```
{% extends "layout/basic.html" %}

{% load thumbnail %}
{% load static %}
{% load bootstrap4 %}

{% block title %}{{ rubric }}{% endblock %}

{% block searchform %}
{% endblock %}

{% block content %}
<h2 class="mb-2">{{ rubric }}</h2>
```

```
<div class="container-fluid mb-2">
    <div class="row">
        <div class="col"> </div>
        <form class="col-md-auto form-inline">
            {% bootstrap_form form show_label=False %}
            {% bootstrap_button content='Искать' button_type='submit' %}
        </form>
    </div>
</div>
{% if bbs %}
<ul class="list-unstyled">
    {% for bb in bbs %}
    <li class="media my-5 p-3 border">
        {% url 'main:detail' rubric_pk=rubric.pk pk=bb.pk as url %}
        <a href="{{ url }}{{ all }}">
        {% if bb.image %}
        <img class="mr-3" src="{% thumbnail bb.image 'default' %}">
        {% else %}
        <img class="mr-3" src="{% static 'main/empty.jpg' %}">
        {% endif %}
        </a>
        <div class="media-body">
          <h3><a href="{{ url }}{{ all }}">
          {{ bb.title }}</a></h3>
          <div>{{ bb.content }}</div>
          <p class="text-right font-weight-bold">{{ bb.price }} руб.</p>
          <p class="text-right font-italic">{{ bb.created_at }}</p>
        </div>
    </li>
    {% endfor %}
</ul>
{% bootstrap_pagination page url=keyword %}
{% endif %}
{% endblock %}
```

Здесь есть несколько примечательных моментов, на которые нужно обратить внимание.

Чтобы вывести форму поиска, прижав ее к правой части страницы, мы используем конструкцию следующего вида:

```
<div class="container-fluid mb-2">
    <div class="row">
        <div class="col"> </div>
        <form class="col-md-auto . . .">
            . . .
        </form>
    </div>
</div>
```

Знакомый нам по *главе 30* стилевой класс `container-fluid` заставляет элемент, к которому он привязан, вести себя как обычная таблица HTML (стилевой класс `mb-3` устанавливает средней величины внешний отступ снизу, отделяющий форму от собственно списка объявлений). Элемент с привязанным стилевым классом `row`, вложенный в этот элемент, ведет себя как строка таблицы. Вложенный в элемент-«строку» элемент со стилевым классом `col` ведет себя как ячейка таблицы, растягивающаяся на всю доступную ширину, а элемент со стилевым классом `col-md-auto` — как ячейка с шириной, равной ширине ее содержимого. Последним как раз и является форма поиска, которая в результате окажется у правого края форма.

Посмотрим на код самой формы:

```
<form class=". . . form-inline">
    {% bootstrap_form form show_label=False %}
    {% bootstrap_button content='Искать' button_type='submit' %}
</form>
```

Стилевой класс `form-inline` укажет веб-обозревателю вывести все элементы управления формы в одну строку. В самой форме мы не помещаем электронный жетон защиты (который генерируется тегом шаблонизатора `csrf_token`), поскольку он там совершенно не нужен, а для вывода кнопки применяем тег `bootstrap_button` (тег `buttons`, что мы использовали ранее, добавляет лишний блок, который в этом случае также не нужен).

Для вывода очередной части списка объявлений мы применяем особые перечни Bootstrap. Взглянем на код, который их формирует:

```
<ul class="list-unstyled">
    <li class="media my-5 p-3 border">
        <img class="mr-3" . . .>
        <div class="media-body">
            . . .
        </div>
    </li>
</ul>
```

Сам перечень создается маркированным списком HTML (тегом `<ul>`) со стилевым классом `list-unstyled`. Отдельная позиция перечня, как логично предположить, формируется пунктом списка (тегом `<li>`), к которому должен быть привязан стилевой класс `media` (стилевой класс `my-5` задает большие внешние отступы сверху и снизу, стилевой класс `p-3` — внутренние отступы среднего размера со всех сторон, а стилевой класс `border` — рамку вокруг элемента). В пункте помещается графическое изображение (тег `<img>`) со стилевым классом `mr-3` и любое количество других элементов.

Гиперссылки, указывающие на страницу сведений об объявлении, мы создадим на базе основной иллюстрации и названия товара. Чтобы не генерировать интернет-адрес для этих гиперссылок дважды, мы сохраним его в переменной `url`:

```
{% url 'main:detail' rubric_pk=rubric.pk pk=bb.pk as url %}
```

Основная иллюстрация к объявлению у нас является необязательной к указанию. Поэтому мы обязаны предусмотреть случай, когда пользователь, оставляя объявление, не станет указывать в нем основную иллюстрацию. Для этого мы написали такой код:

```
<a href="{{ url }}{{ all }}">
{% if bb.image %}
<img class="mr-3" src="{% thumbnail bb.image 'default' %}">
{% else %}
<img class="mr-3" src="{% static 'main/empty.jpg' %}">
{% endif %}
</a>
```

Если основная иллюстрация в объявлении указана, будет выведена ее миниатюра. Если же автору объявления нечем его иллюстрировать, будет выведено изображение из статического файла main\empty.jpg.

Теперь посмотрим на самую первую строку приведенного ранее фрагмента. Там, в создающем гиперссылку теге `<a>`, мы сформировали интернет-адрес, объединив содержимое только что созданной в шаблоне переменной `url` и переменной `all` контекста шаблона, в которой хранятся полностью сформированные GET-параметры `keyword` и `page`. Так мы получим для гиперссылки полный интернет-адрес, включающий искомое слово и номер части.

Далее нет ничего интересного, за исключением стилевого класса `font-italic`, который задает для элемента курсивное начертание шрифта.

Напоследок посмотрим на тег шаблонизатора, создающий пагинатор:

```
{% bootstrap_pagination page url=keyword %}
```

В качестве базового интернет-адреса мы указываем GET-параметр `keyword`, хранящий искомое слово. В результате при переходе на другую часть пагинатора контроллер получит это слово и сможет выполнить фильтрацию объявлений.

Найдем в Интернете какое-либо подходящее графическое изображение и сохраним его под именем empty.jpg в папке static\main папки проекта (если это изображение хранится в формате, отличном от JPEG, необходимо соответственно изменить расширение имени файла в коде шаблона).

33.4.2. Вывод сведений о выбранном объявлении

Сразу же создадим контроллер, маршрут и шаблон для страницы со сведениями о выбранном посетителем объявлении.

И начнем мы с маршрута. Поместим его непосредственно перед маршрутом, ведущим на контроллер `by_rubric()`, который выводит список объявлений:

```
from .views import detail
. . .
urlpatterns = [
    . . .
```

```
    path('<int:rubric_pk>/<int:pk>/', detail, name='detail'),
    path('<int:pk>/', by_rubric, name='by_rubric'),
    . . .
]
```

Здесь интернет-адреса показывают своего рода иерархию рубрик и отдельных объявлений:

- страницы со списками объявлений, принадлежащих определенной *рубрике*, имеют интернет-адреса формата */<рубрика>/*;
- страницы с отдельными объявлениями, принадлежащими определенной рубрике, имеют интернет-адреса формата */<рубрика>/<объявление>/*.

Контроллер `detail()` лучше реализовать в виде функции, поскольку в *главе 34* нам придется подготавливать в нем еще и список комментариев, что выходит за рамки типовой задачи по выводу отдельной записи. Код контроллера-функции показан в листинге 33.8.

Листинг 33.8. Код контроллера-функции `detail()`

```
def detail(request, rubric_pk, pk):
    bb = get_object_or_404(Bb, pk=pk)
    ais = bb.additionalimage_set.all()
    context = {'bb': bb, 'ais': ais}
    return render(request, 'main/detail.html', context)
```

Помимо самого объявления, которое мы помещаем в переменную контекста шаблона с именем `bb`, мы также подготавливаем список дополнительных иллюстраций к объявлению, записав его в переменную `ais`.

Код шаблона main\detail.html, выводящего страницу сведений об объявлении, можно увидеть в листинге 33.9.

Листинг 33.9. Код шаблона main\detail.html

```
{% extends "layout/basic.html" %}

{% block title %}{{ bb.title }} - {{ bb.rubric.name }}{% endblock %}

{% block content %}
<div class="container-fluid mt-3">
    <div class="row">
        {% if bb.image %}
        <div class="col-md-auto"><img src="{{ bb.image.url }}"
        class="main-image"></div>
        {% endif %}
        <div class="col">
            <h2>{{ bb.title }}</h2>
            <p>{{ bb.content }}</p>
```

```
            <p class="font-weight-bold">{{ bb.price }} руб.</p>
            <p>{{ bb.contacts }}</p>
            <p class="text-right font-italic">Объявление добавлено
            {{ bb.created_at }}</p>
        </div>
    </div>
</div>
{% if ais %}
<div class="d-flex justify-content-between flex-wrap mt-5">
    {% for ai in ais %}
    <div>
        <img class="additional-image" src="{{ ai.image.url }}">
    </div>
    {% endfor %}
</div>
{% endif %}
<p><a href="{% url 'main:by_rubric' pk=bb.rubric.pk %}{{ all }}">
Назад</a></p>
{% endblock %}
```

Код, выводящий основные сведения об объявлении (название, описание и цену товара, контакты, дату и время добавления объявления, основную иллюстрацию, если таковая указана), располагается между вот этими тегами:

```
<div class="container-fluid mt-3">
    . . .
</div>
```

Аналогичный код мы рассматривали уже дважды (причем второй раз — непосредственно в текущей главе), так что он уже должен быть нам знаком.

А вот код, что выводит дополнительные иллюстрации, заслуживает более пристального рассмотрения. Вот он:

```
<div class="d-flex justify-content-between flex-wrap mt-5">
    {% for ai in ais %}
    <div>
        <img class="additional-image" src="{{ ai.image.url }}">
    </div>
    {% endfor %}
</div>
```

К блоку, в котором будут выводиться дополнительные иллюстрации, привязаны следующие стилевые классы:

- `d-flex` — устанавливает для элемента так называемую *гибкую* разметку, при которой дочерние элементы выстраиваются внутри родителя по горизонтали;
- `justify-content-between` — указывает, что дочерние элементы — собственно дополнительные иллюстрации — должны располагаться внутри родителя на равномерном расстоянии друг от друга;

- `flex-wrap` — если дочерним элементам не хватит места, чтобы выстроиться по горизонтали, не помещающиеся элементы будут перенесены на следующую строку;
- `mt-5` — большой внешний отступ сверху, чтобы отделить дополнительные иллюстрации от основной информации.

Как видим, CSS-фреймворк Bootstrap включает в себя стилевые классы буквально почти на все случаи жизни. Но, увы, не на все... С его помощью мы не сможем установить нужные нам размеры основной и дополнительных иллюстраций. Нам придется открыть таблицу стилей main\style.css и добавить в нее фрагмент кода:

```
img.main-image {
    width: 300px;
}
img.additional-image {
    width: 180px;
}
```

Так мы установим ширину основной иллюстрации в 300 пикселов, а дополнительных иллюстраций — в 180 пикселов. Этого должно хватить.

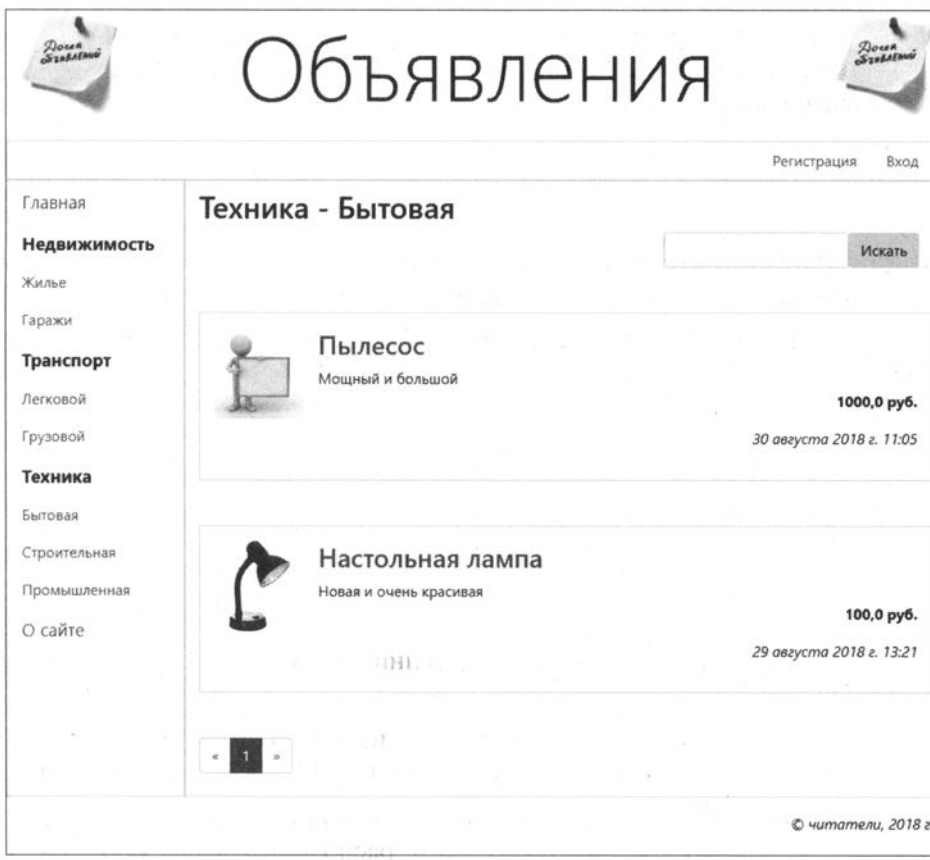


Рис. 33.1. Веб-страница списка объявлений

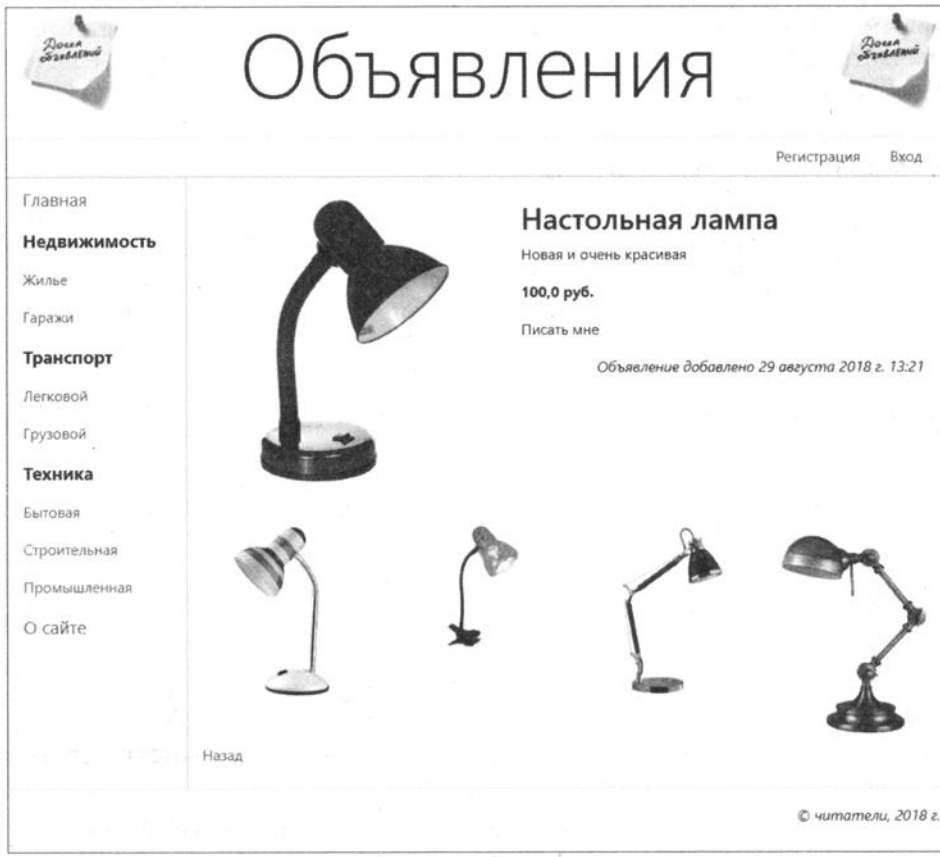


Рис. 33.2. Веб-страница сведений об объявлении

Сохраним весь код, перезапустим отладочный веб-сервер и зайдем на страницу со списком объявлений, относящихся к какой-либо из категорий. Если мы все сделали правильно, страница будет выглядеть так, как показано на рис. 33.1. После чего попробуем перейти на страницу со сведениями об объявлении (рис. 33.2).

33.4.3. Вывод последних 10 объявлений на главной веб-странице

Вывод списка объявлений и сведений о выбранном объявлении мы сделали. Так что реализовать вывод 10 последних объявлений на главной странице для нас труда не составит.

Сначала найдем код контроллера-функции `index()`, который выводит главную страницу, и добавим в него фрагмент, выбирающий из базы последние 10 объявлений:

```
def index(request):
    bbs = Bb.objects.filter(is_active=True)[:10]
    context = {'bbs': bbs}
    return render(request, 'main/index.html', context)
```

И допишем в шаблоне main\index.html код, который, собственно, и выведет эти объявления. Сделайте это самостоятельно. За основу можете взять аналогичный код из шаблона main\by_rubric.html, показанный в листинге 33.7.

33.5. Работа с объявлениями

Последнее, что нам нужно сделать, — подготовить инструменты, посредством которых зарегистрированные пользователи будут просматривать перечень оставленных ими объявлений, добавлять, править и удалять объявления.

33.5.1. Вывод объявлений, оставленных текущим пользователем

Вывести объявления, написанные текущим пользователем, совсем не сложно. Давайте посмотрим на код, который мы добавим в контроллер-функцию `profile()`:

```
@login_required
def profile(request):
    bbs = Bb.objects.filter(author=request.user.pk)
    context = {'bbs': bbs}
    return render(request, 'main/profile.html', context)
```

Мы выполняем здесь фильтрацию объявлений по значению поля `author` (ключ автора объявления, которым является зарегистрированный пользователь), сравнивая это значение с ключом текущего пользователя.

Для вывода списка объявлений мы вставим в шаблон main\profile.html, формирующий страницу пользовательского профиля, код, который мы практически без изменений можем взять из шаблона main\by_rubric.html (см. листинг 33.7).

Если пользователь зайдет на страницу своего профиля и щелкнет на объявлении, он попадет на общедоступную страницу сведений об объявлении, написанную нами в *разд. 33.4.2*. Возможно, это окажется не очень удобно, так что давайте создадим другую, административную страницу сведений об объявлении, предназначенную лишь для зарегистрированных пользователей.

Сначала объявим маршрут, который укажет на эту страницу, поместив его непосредственно перед маршрутом, что ведет на страницу профиля:

```
from .views import profile_bb_detail
. . .
urlpatterns = [
    . . .
    path('accounts/profile/<int:pk>/', profile_bb_detail,
                                       name='profile_bb_detail'),
    path('accounts/profile/', profile, name='profile'),
    . . .
]
```

Контроллер-функцию `profile_bb_detail()`, который выведет страницу сведений об объявлении, напишем самостоятельно, взяв за основу контроллер `detail()`

(см. листинг 33.8). Не забудем сделать его доступным только для зарегистрированных пользователей, выполнивших вход. Также самостоятельно напишем шаблон, который формирует эту страницу и за основу которого можно взять шаблон main\detail.html (см. листинг 33.9).

33.5.2. Добавление, правка и удаление объявлений

Чтобы реализовать добавление нового объявления, нам сначала нужно объявить форму для ввода самого объявления и набор форм, в которые будут заноситься дополнительные иллюстрации. Форма будет связана с моделью объявления `Bb`, а набор форм — с моделью дополнительной иллюстрации `AdditionalImage`.

Код, объявляющий форму `BbForm` и набор форм `AIFormSet`, представлен в листинге 33.10.

Листинг 33.10. Код формы `BbForm` и набора форм `AIFormSet`

```
from django.forms import inlineformset_factory
from .models import Bb, AdditionalImage

class BbForm(forms.ModelForm):
    class Meta:
        model = Bb
        fields = '__all__'
        widgets = {'author': forms.HiddenInput}

AIFormSet = inlineformset_factory(Bb, AdditionalImage, fields='_all_')
```

В форме мы будем выводить все поля модели `Bb`. Для поля автора объявления `author` мы зададим в качестве элемента управления `HiddenInput`, т. е. скрытое поле — все равно значение туда будет заноситься автоматически.

Контроллер, добавляющий объявление, мы реализуем в виде функции (в виде класса его реализовать будет сложнее) и назовем `profile_bb_add()`. Его код приведен в листинге 33.11.

Листинг 33.11. Код контроллера-функции `profile_bb_add()`

```
from django.shortcuts import redirect

from .forms import BbForm, AIFormSet

@login_required
def profile_bb_add(request):
    if request.method == 'POST':
        form = BbForm(request.POST, request.FILES)
        if form.is_valid():
            bb = form.save()
```

```
            formset = AIFormSet(request.POST, request.FILES, instance=bb)
            if formset.is_valid():
                formset.save()
                messages.add_message(request, messages.SUCCESS,
                                     'Объявление добавлено')
                return redirect('main:profile')
    else:
        form = BbForm(initial={'author': request.user.pk})
        formset = AIFormSet()
    context = {'form': form, 'formset': formset}
    return render(request, 'main/profile_bb_add.html', context)
```

Здесь нужно отметить три важных момента. Во-первых, при создании формы перед выводом страницы сохранения мы заносим в поле `author` формы ключ текущего пользователя. Так мы свяжем заносимое объявление с этим пользователем.

Во-вторых, во время сохранения введенного объявления, при создании объектов формы и набора форм, мы должны передать конструкторам их классов вторым позиционным параметром словарь со всеми полученными файлами (он хранится в атрибуте `FILES` объекта запроса). Если мы не сделаем этого, отправленные пользователем иллюстрации окажутся потерянными.

В-третьих, при сохранении мы сначала выполняем валидацию и сохранение формы самого объявления. Метод `save()` в качестве результата возвращает сохраненную запись, и эту запись мы должны передать через параметр `instance` конструктору класса набора форм. Это нужно для того, чтобы все дополнительные иллюстрации после сохранения оказались связанными с объявлением.

Напишем маршрут, который укажет на страницу добавления, поместив его перед маршрутом, указывающим на страницу профиля:

```
from .views import profile_bb_add
. . .
urlpatterns = [
    . . .
    path('accounts/profile/add/', profile_bb_add, name='profile_bb_add'),
    path('accounts/profile/<int:pk>/', profile_bb_detail,
                                       name='profile_bb_detail'),
    . . .
]
```

Теперь займемся шаблоном main\profile_bb_add.html, который создаст страницу добавления объявления. Его код можно увидеть в листинге 33.12.

Листинг 33.12. Код шаблона main\profile_bb_add.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}
```

```
{% block title %}Добавление объявления -
Профиль пользователя{% endblock %}

{% block content %}
<h2>Добавление объявления</h2>
<form method="post" enctype="multipart/form-data">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% bootstrap_formset formset layout='horizontal' %}
    {% buttons submit='Добавить' %}{% endbuttons %}
</form>
{% endblock %}
```

Обязательно укажем у формы метод кодирования данных `multipart/form-data`. Если мы не сделаем этого, занесенные в форму файлы не будут отправлены. А набор форм выведем с помощью тега шаблонизатора `bootstrap_formset`.

Наконец, в шаблон страницы профиля main\profile.html добавим гиперссылку, указывающую на страницу добавления объявления:

```
<p><a href="{% url 'main:profile_bb_add' %}">Добавить объявление</a></p>
```

После чего можно сохранить код и попробовать функциональность по добавлению новых объявлений в деле.

Контроллеры `profile_bb_change()` и `profile_bb_delete()`, которые, соответственно, правят и удаляют объявление, написать не очень сложно. Давайте посмотрим на объявляющий их код, представленный в листинге 33.13.

Листинг 33.13. Код контроллеров-функций `profile_bb_change()` и `profile_bb_delete()`

```
@login_required
def profile_bb_change(request, pk):
    bb = get_object_or_404(Bb, pk=pk)
    if request.method == 'POST':
        form = BbForm(request.POST, request.FILES, instance=bb)
        if form.is_valid():
            bb = form.save()
            formset = AIFormSet(request.POST, request.FILES, instance=bb)
            if formset.is_valid():
                formset.save()
                messages.add_message(request, messages.SUCCESS,
                                     'Объявление исправлено')
                return redirect('main:profile')
    else:
        form = BbForm(instance=bb)
        formset = AIFormSet(instance=bb)
```

```
    context = {'form': form, 'formset': formset}
    return render(request, 'main/profile_bb_change.html', context)

@login_required
def profile_bb_delete(request, pk):
    bb = get_object_or_404(Bb, pk=pk)
    if request.method == 'POST':
        bb.delete()
        messages.add_message(request, messages.SUCCESS,
                             'Объявление удалено')
        return redirect('main:profile')
    else:
        context = {'bb': bb}
        return render(request, 'main/profile_bb_delete.html', context)
```

Для простоты на странице удаления объявления мы не будем выводить дополнительные иллюстрации — они там не особо нужны.

Объявим необходимые маршруты:

```
from .views import profile_bb_change, profile_bb_delete
. . .
urlpatterns = [
    . . .
    path('accounts/profile/change/<int:pk>/', profile_bb_change,
                                              name='profile_bb_change'),
    path('accounts/profile/delete/<int:pk>/', profile_bb_delete,
                                              name='profile_bb_delete'),
    path('accounts/profile/add/', profile_bb_add, name='profile_bb_add'),
    . . .
]
```

Шаблоны main\profile_bb_change.html и main\profile_bb_delete.html вы можете написать самостоятельно, использовав в качестве основы ранее написанные шаблоны.

В шаблон main\profile.html нужно добавить код, создающий гиперссылки для правки и удаления каждого из занесенных пользователем в базу объявлений. Этот код может выглядеть следующим образом:

```
<div class="media-body">
    <p>Рубрика: {{ bb.rubric }}</p>
    . . .
    <p class="text-right mt-2">
        <a href="{% url 'main:profile_bb_change' pk=bb.pk %}">
        Исправить</a>
        <a href="{% url 'main:profile_bb_delete' pk=bb.pk %}">Удалить</a>
    </p>
</div>
```

Напоследок следует проверить в действии реализованную функциональность по правке и удалению объявлений.

ГЛАВА 34

Комментарии

К каждому из опубликованных на нашем сайте объявлений посетители могут оставить комментарии. Сейчас самое время заняться этой функцией.

34.1. Подготовка к выводу CAPTCHA

Мы сделаем так, чтобы зарегистрированные пользователи могли оставлять комментарии беспрепятственно, а гости должны были бы дополнительно занести в форму CAPTCHA. Поэтому нам предварительно нужно установить библиотеку Django Simple Captcha:

```
pip install django-simple-captcha
```

Добавим в список зарегистрированных в проекте приложение `captcha` — программное ядро этой библиотеки:

```
INSTALLED_APPS = [
    . . .
    'captcha',
]
```

И объявим в списке маршрутов уровня проекта (модуль urls.py пакета конфигурации) маршрут, указывающий на это приложение:

```
urlpatterns = [
    . . .
    path('captcha/', include('captcha.urls')),
    path('', include('main.urls', namespace='')),
]
```

Миграции пока выполнять не будем — все равно нам еще нужно объявить модель для хранения комментариев.

34.2. Модель комментария

Модель, хранящая комментарии, мы назовем `Comment`. Ее структура приведена в табл. 34.1.

***Таблица 34.1.** Структура модели `Comment`*

Имя	Тип	Дополнительные параметры	Описание
bb	ForeignKey	Каскадное удаление разрешено	Объявление, к которому оставлен комментарий
author	CharField	Длина — 30 символов	Имя автора
content	TextField		Содержание
is_active	BooleanField	Значение по умолчанию — True, индексированное	Признак, показывать ли комментарий в списке
created_at	DateTimeField	Подставляется текущее значение даты и времени, индексированное	Дата и время публикации комментария

Код, объявляющий модель `Comment`, приведен в листинге 34.1.

Листинг 34.1. Код модели `Comment`

```
class Comment(models.Model):
    bb = models.ForeignKey(Bb, on_delete=models.CASCADE,
                           verbose_name='Объявление')
    author = models.CharField(max_length=30, verbose_name='Автор')
    content = models.TextField(verbose_name='Содержание')
    is_active = models.BooleanField(default=True, db_index=True,
                                    verbose_name='Выводить на экран?')
    created_at = models.DateTimeField(auto_now_add=True, db_index=True,
                                      verbose_name='Опубликован')

    class Meta:
        verbose_name_plural = 'Комментарии'
        verbose_name = 'Комментарий'
        ordering = ['created_at']
```

Для комментариев мы указываем сортировку по увеличению даты и времени их добавления. В результате более старые комментарии будут располагаться в начале списка, а более новые — в его конце.

Создадим миграции, после чего выполним их (при этом также будут выполнены миграции из библиотеки Django Simple Captcha).

34.3. Вывод и добавление комментариев

Выводить сам список комментариев и форму для добавления комментария мы будем на общедоступной странице сведений об объявлении. Впоследствии мы сделаем так, чтобы комментарии выводились и на административной странице того же рода.

Сначала мы объявим две формы, в которые будут заносить комментарии, соответственно, зарегистрированные пользователи, выполнившие вход, и гости. Код форм `UserCommentForm` и `GuestCommentForm` приведен в листинге 34.2.

Листинг 34.2. Код форм `UserCommentForm` и `GuestCommentForm`

```
from captcha.fields import CaptchaField

from .models import Comment

class UserCommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        exclude = ('is_active',)
        widgets = {'bb': forms.HiddenInput}

class GuestCommentForm(forms.ModelForm):
    captcha = CaptchaField(label='Введите текст с картинки',
              error_messages={'invalid': 'Неправильный текст'})

    class Meta:
        model = Comment
        exclude = ('is_active',)
        widgets = {'bb': forms.HiddenInput}
```

Понятно, что эти формы связаны с моделью `Comment`. Поле `is_active` (признак, будет ли комментарий выводиться на странице) мы уберем из форм, поскольку оно требуется лишь администрации сайта. Для поля `bb`, хранящего ключ объявления, с которым связан комментарий, мы указали в качестве элемента управления скрытое поле — так оно, с одной стороны, все же будет присутствовать в форме, а с другой, не появится на экране.

В форме `GuestCommentForm` дополнительно присутствует поле `captcha`. Оно позволит нам хотя бы в какой-то степени обезопасить наш сайт от атаки служб рассылки спама. В форме `UserCommentForm` это поле отсутствует, поскольку зарегистрированным пользователям мы доверяем больше, чем гостям.

Теперь нам необходимо существенно обновить код контроллера-функции `detail()`, написанного в *главе 33*. В этом контроллере мы реализуем и вывод уже добавленных к объявлению комментариев, и добавление нового комментария. Код обновленного контроллера `detail()` показан в листинге 34.3.

Листинг 34.3. Код обновленного контроллера-функции detail()

```
from .models import Comment
from .forms import UserCommentForm, GuestCommentForm

def detail(request, rubric_pk, pk):
    bb = Bb.objects.get(pk=pk)
    ais = bb.additionalimage_set.all()
    comments = Comment.objects.filter(bb=pk, is_active=True)
    initial = {'bb': bb.pk}
    if request.user.is_authenticated:
        initial['author'] = request.user.username
        form_class = UserCommentForm
    else:
        form_class = GuestCommentForm
    form = form_class(initial=initial)
    if request.method == 'POST':
        c_form = form_class(request.POST)
        if c_form.is_valid():
            c_form.save()
            messages.add_message(request, messages.SUCCESS,
                                 'Комментарий добавлен')
        else:
            form = c_form
            messages.add_message(request, messages.WARNING,
                                 'Комментарий не добавлен')
    context = {'bb': bb, 'ais': ais, 'comments': comments, 'form': form}
    return render(request, 'main/detail.html', context)
```

Сразу после того, как мы подготовим объявление, которое следует вывести на странице, список дополнительных иллюстраций к ней и список оставленных под ним комментариев (отметим, что мы включаем в этот список только активные комментарии — те, чье поле `is_active` хранит значение `True`), перед нами возникнет задача создать форму для добавления комментария. При этом, если текущий пользователь выполнил вход на сайт, нам нужно занести его имя в поле `author` формы комментария, чтобы пользователю было удобнее. Далее, в любом случае нам нужно занести в поле `bb` формы ключ выводящегося в настоящий момент объявления. Наконец, если текущий пользователь выполнил вход на сайт, форма должна создаваться на основе класса `UserCommentForm`, а если не выполнил — на основе класса `GuestCommentUser`. Все эти действия выполняет фрагмент кода:

```
initial = {'bb': bb.pk}
if request.user.is_authenticated:
    initial['author'] = request.user.username
    form_class = UserCommentForm
else:
    form_class = GuestCommentForm
form = form_class(initial=initial)
```

Объект формы мы сохранили в переменной с именем `form`. Форма из этой переменной впоследствии будет выведена на странице сведений об объявлении. Запомним это.

Далее, если полученный запрос был отправлен HTTP-методом POST, т. е. посетитель ввел комментарий и отправил его на сохранение, мы создаем еще один объект формы, передав конструктору полученные данные. Второй объект формы сохраняется в переменной `c_form`, что нам также следует запомнить. После этого мы выполняем валидацию второй формы.

Если валидация прошла успешно, мы сохраняем введенный комментарий и выводим соответствующее случаю всплывающее сообщение. Когда страница будет выведена, она будет содержать новый комментарий и пустую форму ввода комментария из переменной `form`.

Если же валидация прошла неуспешно, мы переносим форму из переменной `c_form` в переменную `form`. Эта форма, хранящая некорректные данные и сообщения об ошибках ввода, впоследствии будет выведена на экран, так что посетитель сразу сможет увидеть, что он ввел не так. И, разумеется, мы выводим всплывающее сообщение о неуспешном добавлении комментария.

В шаблоне main\detail.html отыщем тег шаблонизатора `block content . . . endblock` и вставим непосредственно перед закрывающим тегом следующий код, который и выведет комментарии:

```
<h4 class="mt-5">Новый комментарий</h4>
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% buttons submit='Добавить' %}{% endbuttons %}
</form>
{% if comments %}
<div class="mt-5">
    {% for comment in comments %}
    <div class="my-2 p-2 border">
        <h5>{{ comment.author }}</h5>
        <p>{{ comment.content }}</p>
        <p class="text-right font-italic">{{ comment.created_at }}</p>
    </div>
    {% endfor %}
</div>
```

Не знакомый нам пока еще стилевой класс `my-2` задает небольшие внешние отступы сверху и снизу, а стилевой класс `p-2` — небольшие внутренние отступы со всех сторон. Мы используем их, чтобы создать просветы между отдельными комментариями.

Закончив программирование, попробуем открыть страницу со сведениями о каком-либо объявлении и добавить один или два комментария. После этого выполним вход на сайт и снова попытаемся добавить комментарий. Отметим, что во втором случае форма для добавления комментария не включает поле ввода CAPTCHA.

Аналогичным образом добавим список комментариев на административную страницу сведений об объявлении. Сделайте это самостоятельно. Форму для ввода комментария и соответствующую функциональность в контроллере можно не делать — вряд ли автор объявления будет комментировать его сам...

34.4. Отправка уведомлений о появлении новых комментариев

Для отправки уведомлений о появлении новых комментариев мы станем обрабатывать сигнал `post_save`, возникающий после сохранения записи в модели. Причем обрабатывать мы будем только сигналы, возникающие при сохранении записи модели `Comment`, т. е. комментариев.

Откроем модуль utilities.py пакета приложения, которому мы назначили быть хранилищем вспомогательного кода, и добавим в него объявление функции `send_new_comment_notification()`, которая и выполнит отправку уведомления. Ее код похож на код аналогичной функции `send_activation_notification()` (см. листинг 31.18) и приведен в листинге 34.4.

Листинг 34.4. Код функции `send_new_comment_notification()`

```
def send_new_comment_notification(comment):
    if ALLOWED_HOSTS:
        host = 'http://' + ALLOWED_HOSTS[0]
    else:
        host = 'http://localhost:8000'
    author = comment.bb.author
    context = {'author': author, 'host': host, 'comment': comment}
    subject = render_to_string('email/new_comment_letter_subject.txt',
                               context)
    body_text = render_to_string('email/new_comment_letter_body.txt',
                                 context)
    author.email_user(subject, body_text)
```

Написание шаблонов email\new_comment_letter_subject.txt и email\new_comment_letter_body.txt, которые создадут тему и тело электронного письма, пусть будет вашим, уважаемые читатели, домашним заданием. Их можно написать на основе аналогичных шаблонов email\activation_letter_subject.txt и email\activation_letter_body.txt (см. листинги 31.19 и 31.20). В письме нужно указать интернет-адрес административной страницы сведений об объявлении, к которому был оставлен новый комментарий.

Осталось лишь привязать к сигналу `post_save` обработчик, который будет вызывать функцию `send_new_comment_notification()` после добавления комментария. Код, выполняющий привязку, мы поместим в модуль models.py пакета приложения. Вот этот код:

```
from django.db.models.signals import post_save

def post_save_dispatcher(sender, **kwargs):
    author = kwargs['instance'].bb.author
    if kwargs['created'] and author.send_messages:
        send_new_comment_notification(kwargs['instance'])

post_save.connect(post_save_dispatcher, sender=Comment)
```

Перед тем как отправлять оповещение, следует проверить, не запретил ли пользователь их отправку, т. е. не хранит ли поле `send_messages` модели пользователя `AdvUser` значение `False`.

Попробуем еще раз добавить комментарий к какому-либо объявлению и удостоверимся, что уведомление о новом комментарии было отправлено.

Осталось только сделать инструменты для администрирования комментариев. Пусть это также станет домашним заданием для читателей этой книги.

ГЛАВА 35

Веб-служба REST

А в этой, заключительной, главе книги мы напишем веб-службу, работающую по принципам REST. Она будет выдавать список из 10 последних объявлений, сведения о выбранном объявлении, список комментариев, оставленных для выбранного объявления, а также даст посетителям возможность добавлять новые комментарии. Чтобы защититься от спамеров, мы разрешим оставлять комментарии только зарегистрированным на сайте пользователям.

35.1. Веб-служба

Заниматься программированием веб-службы мы будем с применением библиотеки Django REST framework.

35.1.1. Подготовка к разработке веб-службы

Сначала нам необходимо установить библиотеки Django REST framework и django-cors-headers, для чего мы отдадим команды:

```
pip install djangorestframework
pip install django-cors-headers
```

Сразу же создадим новое приложение api, в котором и реализуем функциональность веб-службы:

```
manage.py startapp api
```

Добавим приложения `rest_framework` и `corsheaders` — программные ядра только что установленных библиотек, — а также только что созданное приложение `api` в список зарегистрированных в проекте:

```
INSTALLED_APPS = [
    . . .
    'rest_framework',
    'corsheaders',
    'api.apps.ApiConfig',
]
```

Добавим в список зарегистрированных в проекте необходимый для работы посредник:

```
MIDDLEWARE = [
    . . .
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    . . .
]
```

Не забудем указать там же, в модуле settings.py пакета конфигурации, настройки, разрешающие доступ к веб-службе с любого домена:

```
CORS_ORIGIN_ALLOW_ALL = True
CORS_URLS_REGEX = r'^/api/.*$'
```

35.1.2. Список объявлений

Создадим в пакете приложения `api` модуль serializers.py. В нем мы запишем код нашего первого сериализатора, который будет формировать список рубрик и который мы назовем `BbSerializer` (см. листинг 35.1).

Листинг 35.1. Код сериализатора `BbSerializer`

```
from rest_framework import serializers

from main.models import Bb

class BbSerializer(serializers.ModelSerializer):
    class Meta:
        model = Bb
        fields = ('id', 'title', 'content', 'price', 'created_at')
```

В составе сведений о каждом объявлении мы ради компактности будем отправлять лишь ключ, название, описание, цену товара и дату создания объявления. Интернет-адрес основной иллюстрации и контакты мы отправим в составе сведений о выбранном объявлении.

Контроллер, который будет выдавать список объявлений, мы реализуем в виде функции и назовем `bbs()`. Его код показан в листинге 35.2.

Листинг 35.2. Код контроллера-функции `bbs()`

```
from rest_framework.response import Response
from rest_framework.decorators import api_view

from main.models import Bb
from .serializers import BbSerializer
```

```
@api_view(['GET'])
def bbs(request):
    if request.method == 'GET':
        bbs = Bb.objects.filter(is_active=True)[:10]
        serializer = BbSerializer(bbs, many=True)
        return Response(serializer.data)
```

Откроем список маршрутов уровня проекта (модуль urls.py пакета конфигурации) и добавим маршрут, указывающий на приложение `api`:

```
urlpatterns = [
    . . .
    path('api/', include('api.urls')),
    path('', include('main.urls', namespace='')),
]
```

В пакете приложения `api` создадим модель urls.py, в который запишем список маршрутов уровня этого приложения. Код этого модуля представлен в листинге 35.3.

Листинг 35.3. Код модуля urls.py пакета приложения `api`

```
from django.urls import path

from .views import bbs

urlpatterns = [
    path('bbs/', bbs),
]
```

Пока что он содержит лишь маршрут, указывающий на только что написанный нами контроллер `bbs()`.

Сохраним код, запустим отладочный веб-сервер и попробуем получить список объявлений, перейдя по интернет-адресу **http://localhost:8000/api/bbs/**. Если мы все сделали правильно, то увидим веб-представление, показывающее последние 10 объявлений, что были оставлены посетителями нашего сайта.

35.1.3. Сведения о выбранном объявлении

В составе сведений о выбранном объявлении, помимо ключа записи, названия, описания, цены товара и даты создания объявления, мы должны выдать контакты и интернет-адрес основной иллюстрации. Учтем это при написании сериализатора `BbDetailSerializer` (см. листинг 35.4).

Листинг 35.4. Код сериализатора `BbDetailSerializer`

```
class BbDetailSerializer(serializers.ModelSerializer):
    class Meta:
```

```
        model = Bb
        fields = ('id', 'title', 'content', 'price', 'created_at',
                  'contacts', 'image')
```

Код сериализатора мы занесем в модуль serializers.py пакета приложения, который создали специально для этой цели.

Контроллер, который будет выдавать клиентам сведения о выбранном объявлении, мы реализуем в виде класса, производного от класса `RetrieveAPIView`, который выполнит за нас почти всю работу. Весьма компактный код нашего контроллера-класса `BbDetailView` приведен в листинге 35.5.

Листинг 35.5. Код контроллера-класса `BbDetailView`

```
from rest_framework.generics import RetrieveAPIView

from .serializers import BbDetailSerializer

class BbDetailView(RetrieveAPIView):
    queryset = Bb.objects.filter(is_active=True)
    serializer_class = BbDetailSerializer
```

Добавим в список маршрутов приложения маршрут, который укажет на наш новый контроллер:

```
from .views import BbDetailView

urlpatterns = [
    path('bbs/<int:pk>/', BbDetailView.as_view()),
    path('bbs/', bbs),
]
```

Сохраним код и попытаемся получить сведения об объявлении с ключом 1, для чего выполним переход по интернет-адресу **http://localhost:8000/api/bbs/1/**. Далее запросим сведения о паре других объявлений.

35.1.4. Вывод и добавление комментариев

Теперь займемся функциональностью для извлечения списка комментариев, оставленных под выбранным объявлением, и добавления новых комментариев.

Код сериализатора `CommentSerializer`, который будет отправлять список комментариев и добавлять новый комментарий, можно увидеть в листинге 35.6.

Листинг 35.6. Код сериализатора `CommentSerializer`

```
from main.models import Comment

class CommentSerializer(serializers.ModelSerializer):
```

```
    class Meta:
        model = Comment
        fields = ('bb', 'author', 'content', 'created_at')
```

Этот сериализатор отправит клиенту ключ объявления, с которым связан комментарий, имя автора, содержимое и дату создания комментария. Как видим, отправляются только те данные, которые действительно необходимы клиенту, чтобы вывести комментарий на экран. Исключение составляет лишь ключ объявления — он нужен, чтобы успешно добавить новый комментарий.

Вообще, сведения, которые клиенту не нужны, лучше ему не отправлять — так мы уменьшим объем пересылаемых по сети данных и избежим утечки сведений, которые могут оказаться конфиденциальными.

Код контроллера-функции `comments()`, выдающего список комментариев и добавляющий новый комментарий, показан в листинге 35.7.

Листинг 35.7. Код контроллера-функции `comments()`

```
from rest_framework.decorators import permission_classes
from rest_framework.status import HTTP_201_CREATED, HTTP_400_BAD_REQUEST
from rest_framework.permissions import IsAuthenticatedOrReadOnly

from main.models import Comment
from .serializers import CommentSerializer

@api_view(['GET', 'POST'])
@permission_classes((IsAuthenticatedOrReadOnly,))
def comments(request, pk):
    if request.method == 'POST':
        serializer = CommentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=HTTP_201_CREATED)
        else:
            return Response(serializer.errors,
                            status=HTTP_400_BAD_REQUEST)
    else:
        comments = Comment.objects.filter(is_active=True, bb=pk)
        serializer = CommentSerializer(comments, many=True)
        return Response(serializer.data)
```

Мы условились, что разрешим добавлять комментарии только зарегистрированным пользователям, а просматривать их, напротив, позволим всем. Поэтому мы пометили контроллер декоратором `permission_classes()`, в котором указали класс разграничения доступа `IsAuthenticatedOrReadOnly`. В остальном ничего особо сложного и нового в коде этого контроллера для нас нет.

Маршрут, который укажет на новый контроллер и который мы поместим в список уровня приложения, будет выглядеть так:

```
from .views import comments

urlpatterns = [
    path('bbs/<int:pk>/comments/', comments),
    . . .
]
```

Проверим, удастся ли нам получить список комментариев, оставленных под объявлением с ключом 1. Какой интернет-адрес при этом нужно набирать в веб-обозревателе, сообразим самостоятельно.

35.2. Тестовый клиентский веб-сайт

Для всесторонней проверки нашей веб-службы мы создадим простой тестовый веб-сайт, применив популярный клиентский веб-фреймворк Angular.

Angular — веб-фреймворк, предназначенный для написания клиентской части веб-сайтов. Он реализует вывод данных с применением шаблонов, предоставляет удобные средства для создания интерактивных эффектов и обеспечивает взаимодействие с серверной частью сайта, в качестве которой обычно применяется веб-служба REST.

Программный код Angular-сайтов пишется на языке TypeScript. Он является дальнейшим развитием JavaScript, вобравшего из других, более развитых объектно-ориентированных языков такие понятия, как классы, типизация, модули, импорт и экспорт. Файлы с программным кодом, написанным на TypeScript, имеют расширение ts. По ходу дела мы познакомимся с этим любопытным языком.

Написанный на TypeScript программный код не распознается веб-обозревателем и поэтому предварительно компилируется в обычный JavaScript с помощью входящего в состав Angular-проекта компилятора.

> ***На заметку***
>
> Полное описание фреймворка Angular можно отыскать на его домашнем сайте **https://angular.io/**. Существует домашний сайт и у языка TypeScript — он расположен по интернет-адресу **https://www.typescriptlang.org/**.

Веб-сайт, написанный на Angular, в терминологии этого фреймворка носит название *приложения*. Да, здесь приложением называется весь сайт, а не часть его функциональности, как в Django.

35.2.1. Подготовка к разработке тестового веб-сайта

Первое, что нам необходимо сделать, — установить исполняющую среду Node.js. Ее дистрибутивные комплекты можно отыскать по интернет-адресу **https://nodejs.org/en/download/current/**.

После этого необходимо установить утилиту командной строки ng, с помощью которой выполняется создание Angular-проектов, программных модулей различных типов, запуск отладочного веб-сервера Angular и ряд других служебных задач. Установить ng можно, отдав в командной строке команду:

```
npm install -g @angular/cli
```

Проект сайта, написанного на Angular, создается аналогично проекту Django-сайта. Пользуясь командной строкой, необходимо перейти в папку, в которой будет располагаться папка проекта, и отдать соответствующую команду, записав в ней имя создаваемого проекта. В результате будет создана папка проекта с набором файлов и папок, хранящих программный код и вспомогательные данные.

Создадим проект нашего тестового сайта, который назовем `bbclient`:

```
ng new bbclient
```

В папке bbclient, которая будет создана после этого, находятся с десяток файлов, хранящих всевозможные настройки и служебные данные, и три папки. Нам нужна папка src, в которой помещается исходный код сайта (папка e2e хранит тестовые модули, а папка node_modules — дополнительные библиотеки). Отыщем в папке src файл polyfills.ts и откроем его в текстовом редакторе. Найдем вот такой фрагмент:

```
// import 'core-js/es6/symbol';
// import 'core-js/es6/object';
// import 'core-js/es6/function';
// import 'core-js/es6/parse-int';
// import 'core-js/es6/parse-float';
// import 'core-js/es6/number';
// import 'core-js/es6/math';
// import 'core-js/es6/string';
// import 'core-js/es6/date';
// import 'core-js/es6/array';
// import 'core-js/es6/regexp';
// import 'core-js/es6/map';
// import 'core-js/es6/weak-map';
// import 'core-js/es6/set';
```

Эти выражения, изначально закомментированные, выполняют импорт модулей с библиотеками совместимости. Их нужно раскомментировать, иначе Angular-сайт не будет работать в Microsoft Internet Explorer и некоторых других старых веб-обозревателях:

```
import 'core-js/es6/symbol';
import 'core-js/es6/object';
. . .
```

Проверим также, раскомментированы ли вот эти выражения:

```
import 'core-js/es6/reflect';
import 'core-js/es7/reflect';
```

И не забудем сохранить исправленный модуль.

Более в папке src нет ничего интересного для нас. Перейдем во вложенную в нее папку app, в которой хранятся все модули TypeScript, содержащие программный код сайта. Изначально это компонент приложения и метамодуль приложения, составляющие ядро проекта.

Но одного компонента нам мало. Нам понадобятся еще три программных модуля, и мы создадим их прямо сейчас:

- компонент списка объявлений `BbListComponent`:

```
ng generate component bb-list
```

- компонент сведений о выбранном объявлении `BbDetailComponent` (он же выведет список комментариев, оставленных под этим объявлением, и форму для добавления нового комментария):

```
ng generate component bb-detail
```

- службу `BbService`, которая будет взаимодействовать с серверной веб-службой, написанной нами в *разд. 35.1*:

```
ng generate service bb
```

О компонентах, службах и метамодулях мы поговорим по ходу разработки тестового сайта.

35.2.2. Метамодули. Метамодуль приложения *AppModule*. Маршрутизация в Angular

Непосредственно в папке src\app папки нашего Angular-проекта `bbclient` находится модуль app.module.ts. В нем хранится код метамодуля приложения `AppModule`.

Метамодуль (в оригинальной документации — ngModule) — сущность, объединяющая в себе сущности более мелкого порядка: компоненты и службы Angular. Метамодуль выполняет инициализацию объявленных в нем компонентов и служб, после чего они могут быть использованы другими компонентами и службами. Также метамодуль может быть использован для создания списка маршрутов, указания параметров проекта и прочих служебных целей.

Сам фреймворк Angular представляет собой набор метамодулей, в которых объявлены всевозможные службы, директивы, фильтры и прочие сущности, которые могут понадобиться программисту.

Как и все прочие сущности (компоненты и службы), метамодуль в Angular реализуется в виде класса.

В Angular-проекте может присутствовать произвольное количество метамодулей. Один из них, запускаемый при открытии сайта и инициализирующий компонент приложения, называется *метамодулем приложения*. Класс метамодуля приложения всегда носит имя `AppModule`.

Итак, откроем модуль src\app\app.module.ts и исправим код метамодуля приложения AppModule таким образом, чтобы он выглядел, как показано в листинге 35.8.

Листинг 35.8. Код метамодуля приложения AppModule

```
import { BrowserModule } from '@angular/platform-browser';
import { LOCALE_ID, NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HttpClientModule } from '@angular/common/http';
import { registerLocaleData } from '@angular/common';
import localeRu from '@angular/common/locales/ru';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { BbListComponent } from './bb-list/bb-list.component';
import { BbDetailComponent } from './bb-detail/bb-detail.component';
import { BbService } from './bb.service';

registerLocaleData(localeRu, 'ru');

const appRoutes: Routes = [
    {path: ':pk', component: BbDetailComponent},
    {path: '', component: BbListComponent}
];

@NgModule({
    declarations: [
        AppComponent,
        BbListComponent,
        BbDetailComponent
    ],
    imports: [
        RouterModule.forRoot(appRoutes),
        BrowserModule,
        HttpClientModule,
        FormsModule
    ],
    providers: [
        {provide: LOCALE_ID, useValue: 'ru'},
        BbService
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Здесь очень много не знакомых JavaScript-программистам языковых конструкций, которые заслуживают обстоятельного рассказа.

Язык программирования TypeScript, как и Python, является модульным. Следовательно, чтобы использовать какую-либо сущность, объявленную в другом модуле, нужно предварительно импортировать ее. Более того, в отличие от Python, в TypeScript необходимо явно пометить сущность, которая впоследствии будет импортироваться другими модулями, как подлежащую экспорту.

Если в модуле подлежащими экспорту объявлены несколько сущностей, при импорте из такого модуля нужно явно указать импортируемую сущность или сущности. Вот пара примеров:

```
import { BrowserModule } from '@angular/platform-browser';
```

Здесь мы импортируем из модуля `@angular/platform-browser` класс метамодуля `BrowserModule`, в котором реализовано программное ядро Angular.

```
import { LOCALE_ID, NgModule } from '@angular/core';
```

А здесь импортируем из модуля `@angular/core` объект провайдера значения `LOCALE_ID` и декоратор `NgModule`, о которых мы поговорим позже.

В модуле можно объявить и всего одно значение, подлежащее экспорту. В таком случае в процессе импорта для него можно указать имя, под которым оно будет использоваться в импортирующем модуле. Пример:

```
import localeRu from '@angular/common/locales/ru';
```

Здесь мы импортируем единственное подлежащее экспорту значение из модуля `@angular/common/locales/ru` и даем ему имя `localeRu`. Забегая вперед скажем, что это значение хранит настройки локали русского языка.

Подобных выражений импорта в начале кода метамодуля довольно много. Они выполняют импорт сущностей, как входящих в состав Angular, так и созданных нами. Давайте оставим их в покое и рассмотрим выражение, находящееся непосредственно после них:

```
registerLocaleData(localeRu, 'ru');
```

Оно загружает импортированные ранее настройки русской локали в память. Несколько позже мы активизируем русскую локаль, и значения цен и даты у нас будут отображаться согласно заданным в ней правилам.

А вот еще одна интересная строка:

```
const appRoutes: Routes = . . .;
```

Язык TypeScript приносит в мир клиентского веб-программирования два важных нововведения. Во-первых, переменную, чье значение не изменяется после первого присваивания, можно пометить как *константу*, записав в ее объявлении ключевое слово `const`. Попытка присвоить такой константе другое значение вызовет ошибку компиляции.

Во-вторых, для переменной можно указать тип хранящегося в ней значения, записав его в объявлении переменной, после ее имени через двоеточие (*типизация*). Если для переменной указан тип, ей можно будет присвоить значения только ука-

занного типа, а попытка присваивания значения другого типа, опять же, вызовет ошибку компиляции.

В нашем случае переменной `appRoutes` можно однократно присвоить значение только типа `Routes`. Он обозначает массив экземпляров объекта `Object`, содержащих, по крайней мере, обязательное свойство `path`.

И константы, и типизация применяются для устранения ошибок, связанных с присваиванием переменным не тех значений.

Давайте же посмотрим, что мы присваиваем переменной `appRoutes`:

```
const appRoutes: Routes = [
    {path: ':pk', component: BbDetailComponent},
    {path: '', component: BbListComponent}
];
```

Это список маршрутов для встроенного в Angular маршрутизатора. «Корень» сайта мы связываем с компонентом `BbListComponent`, выводящим список объявлений, а пути вида */<**ключ объявления**>/* — с компонентом `BbDetailComponent`, который выведет сведения об объявлении с заданным *ключом*.

Далее следует такая запись:

```
@NgModule( . . . )
export class AppModule { }
```

Метамодуль, как мы уже знаем, представляет собой обычный класс TypeScript. Этот класс обязательно должен быть помечен декоратором `@NgModule` — в противном случае Angular не сможет его обработать. Также класс метамодуля с помощью ключевого слова `export` необходимо пометить как подлежащий экспорту.

Сам класс метамодуля практически всегда пуст, т. е. не содержит объявлений свойств и методов. Все необходимые параметры метамодуля записываются в вызове декоратора `@NgModule`. Давайте посмотрим на них:

```
@NgModule({
    declarations: [
        AppComponent,
        BbListComponent,
        BbDetailComponent
    ],
    imports: [
        RouterModule.forRoot(appRoutes),
        BrowserModule,
        HttpClientModule,
        FormsModule
    ],
    providers: [
        {provide: LOCALE_ID, useValue: 'ru'},
        BbService
    ],
    bootstrap: [AppComponent]
})
```

Единственным параметром декоратору передается экземпляр объекта `Object`, который и укажет сведения о метамодуле в своих свойствах. В приведенном ранее примере мы видим четыре таких свойства (порядок их записи был изменен):

- `declarations` — массив компонентов, зарегистрированных в метамодуле. При загрузке и активизации метамодуля все эти компоненты будут инициализированы и подготовлены к использованию. Если компонент не зарегистрирован ни в одном метамодуле, он не будет инициализирован и, соответственно, не может быть использован.

  Любопытно, что компоненты заносятся в этот массив утилитой ng при выполнении команды создания компонента. Так что нам самим не придется этого делать.

  В нашем метамодуле зарегистрированы компоненты `AppComponent` (компонент приложения, находящийся на самом верхнем уровне и представляющий сам Angular-сайт), `BbListComponent` и `BbDetailComponent`;

- `bootstrap` — компонент, который должен быть выведен на экран сразу после загрузки сайта. Практически всегда это компонент приложения;

- `providers` — массив служб, зарегистрированных в метамодуле. Здесь действует то же правило, что и в случае компонентов: службы, указанные в массиве, будут инициализированы и подготовлены к использованию, а если служба не зарегистрирована, работать она не будет.

  У нас зарегистрированы служба `BbService`, чье будущее предназначение — «общаться» с написанной нами ранее серверной веб-службой, и встроенная в Angular служба `LOCALE_ID`, которая сделает доступным для всех подсистем фреймворка настройки русской локали. Это те самые настройки, что мы загрузили ранее вызовом функции `registerLocaleData()` (вообще-то, `LOCALE_ID` — не совсем служба, но это уже тонкости).

- `imports` — массив метамодулей, подлежащих метаимпорту. В процессе *метаимпорта* Angular делает все сущности, зарегистрированные в метаимпортируемом метамодуле, доступными в текущем метамодуле.

  Мы выполняем метаимпорт следующих метамодулей:

  - `BrowserModule` — программное ядро Angular, реализующее все основные программные механизмы фреймворка;
  - `HttpClientModule` — подсистема взаимодействия с серверными веб-службами;
  - `FormsModule` — подсистема для работы с веб-формами;
  - программно сгенерированный метамодуль, возвращенный статическим методом `forRoot()` класса `RouterModule` и содержащий полностью готовый к работе маршрутизатор.

35.2.3. Компоненты. Компонент приложения *AppComponent*. Стартовая веб-страница

Интерфейс веб-сайта, разработанного с применением фреймворка Angular, строится из компонентов. *Компонент* — это отдельная часть интерфейса сайта, более или менее независимая от других таких же частей. Каждый компонент содержит в себе шаблон, задающий его внешний вид, таблицы стилей, указывающие его представление, и программный код, который определит его поведение. Программный код компонента реализуется в виде класса.

В виде компонентов обычно оформляются отдельные элементы страницы: перечень каких-либо позиций, набор сведений о выбранной позиции, веб-форма для добавления новой позиции, набор блоков, задающих разметку, и пр. Одни компоненты могут вкладываться в другие — так, компонент перечня может включать в себя набор компонентов, создающих отдельные позиции, и компонент формы поиска. Таким образом формируются целые иерархии компонентов, многократно вложенных друг в друга.

На самой вершине иерархии располагается *компонент приложения*, создающий весь интерфейс сайта. Все остальные компоненты вкладываются в него, непосредственно или опосредованно, путем вложения во вложенные компоненты. Класс компонента приложения всегда имеет имя `AppComponent`.

Внешний вид (содержание) компонента определяется его шаблоном. Как и шаблоны Django, шаблоны Angular-компонентов пишутся на языке HTML с вкраплениями специальных тегов, директив и фильтров.

Но как компонент вообще выводится на экран? С помощью особого парного тега, указанного в параметрах компонента, — *тега компонента*. Angular, встретив в HTML-коде такой тег, находит связанный с этим тегом компонент, создает объект его класса и выводит на экран в том месте, где присутствует этот тег. Единственное условие: чтобы Angular смог найти компонент по его тегу, этот компонент должен быть зарегистрирован в метамодуле.

Код класса компонента приложения хранится в файле src\app\app.component.ts. Откроем его и исправим, как показано в листинге 35.9.

Листинг 35.9. Код компонента приложения `AppComponent`

```
import { Component } from '@angular/core';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
}
```

Класс компонента обязательно должен быть помечен декоратором `@Component`, иначе Angular не сможет обработать его в качестве компонента. Также он должен быть помечен как подлежащий экспорту.

Сам класс компонента у нас «пуст», т. к. компонент приложения у нас всего лишь выводит заголовок сайта и помечает место, куда маршрутизатор будет выводить компоненты при навигации по сайту. Никаких данных, вычисляемых в процессе функционирования сайта или загружаемых с сервера, он не выводит и никакой интерактивности не предполагает.

Давайте обратим внимание на декоратор `@Component`, в котором записаны важные параметры компонента:

- `selector` — тег компонента;
- `templateUrl` — путь к файлу с шаблоном, определяющим внешний вид компонента;
- `styleUrls` — массив путей к файлам таблиц стилей, определяющих представление компонента. Изначально там присутствует одна таблица стилей — «пустая», т. е. не содержащая никакого кода.

Файл шаблона компонента `AppComponent` хранится в файле src\app\app.component.html. Откроем его и переделаем согласно листингу 35.10.

Листинг 35.10. Код шаблона компонента `AppComponent`

```
<header>
    <h1>Доска объявлений</h1>
</header>
<router-outlet></router-outlet>
```

Здесь мы просто создаем заголовок сайта и указываем место, в которое маршрутизатор фреймворка будет выводить компоненты при навигации. Это место носит название *выпуска* (outlet) и помечается специальным парным тегом `<router-outlet>`.

А теперь найдем файл src\index.html и откроем его. В этом файле хранится HTML-код *стартовой веб-страницы*, которая открывается в веб-обозревателе при переходе на Angular-сайт и в которой выводятся все компоненты, имеющиеся в составе сайта. Найдем там секцию тела (тег `<body>`):

```
<body>
  <app-root></app-root>
</body>
```

Тег `<app-root>` — это тег компонента приложения `AppComponent`. Так что, как видим, компонент приложения будет выведен на этой странице сразу же после ее открытия.

35.2.4. Службы. Служба *BbService*. Внедрение зависимостей

Прежде чем приступить к работе над остальными двумя компонентами, давайте займемся службой `BbService`.

Служба в терминологии Angular — сущность, напрямую не связанная с выводом, вводом данных и реализацией интерактивности, а ответственная за бизнес-логику. Служба реализуется в виде класса.

Обычно службы осуществляют взаимодействие с серверными веб-службами: получают от них данные, предназначенные для вывода, и отправляют им данные, введенные посетителем. Так, служба `BbService`, которую мы сейчас напишем, будет заниматься взаимодействием с серверной веб-службой, что мы разработали в *разд. 35.1*.

Код службы `BbService` хранится в файле **src\app\bb.service.ts**. Откроем его и перепишем содержимое, руководствуясь листингом 35.11.

Листинг 35.11. Код службы `BbService`

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable, of } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable({
    providedIn: 'root'
})
export class BbService {
    private urlPrefix: String = 'http://localhost:8000';

    constructor(private http: HttpClient) { }

    getBbs(): Observable<Object[]> {
        return this.http.get<Object[]>(`${this.urlPrefix}/api/bbs/`);
    }

    getBb(pk: Number): Observable<Object> {
        return this.http.get<Object>(`${this.urlPrefix}/api/bbs/${pk}/`);
    }

    getComments(pk: Number): Observable<Object[]> {
        return this.http.get<Object[]>(
               `${this.urlPrefix}/api/bbs/${pk}/comments/`);
    }
```

```
    handleError() {
        return (error: any): Observable<Object> => {
            window.alert(error.message);
            return of(null);
        }
    }

    addComment(bb, author, password, content): Observable<Object> {
        const comment = {'bb': bb, 'author': author, 'content': content};
        const httpOptions = {
            headers: new HttpHeaders(
                { 'Content-Type': 'application/json',
                'Authorization': 'Basic ' + window.btoa(author + ':' +
                password) }
            )
        };
        return this.http.post<Object>(
             `${this.urlPrefix}/api/bbs/${bb}/comments/`,
             comment, httpOptions).pipe(catchError(this.handleError()));
    }
}
```

Здесь применяются несколько приемов программирования, характерных как для языка TypeScript, так и для фреймворка Angular.

```
@Injectable({
    providedIn: 'root'
})
export class BbService {
    . . .
}
```

Прежде всего, класс службы должен быть помечен декоратором `@Injectable`. Только после этого фреймворк сможет обработать его в качестве службы.

```
export class BbService {
    private urlPrefix: String = 'http://localhost:8000';
    . . .
}
```

В классе службы мы объявляем частное свойство `urlPrefix`, в котором сохраняем домен серверной веб-службы (частные свойства, равно как и частные методы, в отличие от публичных, доступны только внутри класса).

```
export class BbService {
    . . .
    constructor(private http: HttpClient) { }
    . . .
}
```

Взаимодействие с сервером осуществляет встроенная во фреймворк служба `HttpClient`. По идее, нам нужно объявить в классе свойство для хранения объекта этой службы, в конструкторе класса создать сам объект и присвоить свойству, а в деструкторе уничтожить этот объект. Однако мы можем поступить так, как показано в приведенном фрагменте кода, — объявить свойство в составе списка параметров конструктора и указать ему в качестве типа класс службы. Angular сам создаст объект службы и присвоит этому свойству. Такая операция называется *внедрением зависимостей*.

Внедрение зависимостей позволяет уменьшить объем программного кода и устранить ряд ошибок, связанных с обращением к еще не созданному объекту. Внедрение зависимостей работает для всех служб, чьи классы были помечены декоратором `@Injectable` и зарегистрированы в метамодуле.

```
export class BbService {
    . . .
    getBbs(): Observable<Object[]> {
        return this.http.get<Object[]>(`${this.urlPrefix}/api/bbs/`);
    }
    . . .
}
```

Объявляем в классе службы метод `getBbs()`, который будет загружать и возвращать список объявлений.

При взаимодействии с серверными веб-службами приходится решать досадную проблему, связанную с тем, что невозможно предугадать момент времени, когда от сервера будет получен ответ. В Angular эта проблема решается очень изящным способом. Ответ сервера представляется в виде сущности, которую можно назвать *предположением*, представляет собой объект класса `Observable` и хранит значение, что будет получено в будущем. Чтобы получить это значение, достаточно подписаться на хранящее его предположение.

Обратим внимание, что мы указали тип возвращаемого методом `getBbs()` значения, записав его после списка параметров, после двоеточия:

```
getBbs(): Observable<Object[]>
```

то есть предположение, хранящее массив экземпляров объекта `Object`.

В теле метода `getBbs()` мы вызываем метод `get()` службы `HttpClient`, который получает данные от веб-службы, чей интернет-адрес передан ему в параметре. У метода `get()` мы указали тип получаемого значения `Object[]` — массив экземпляров объекта `Object`:

```
return this.this.http.get<Object[]>( . . . );
```

Теперь метод будет «знать», в какой тип следует преобразовать принятый от сервера массив объявлений, закодированный в формат JSON.

Методу `get()` мы в качестве параметра передаем странно выглядящую строку `` `${this.urlPrefix}/api/bbs/` ``. Это так называемая *шаблонная строка* TypeScript,

могущая содержать специальные символы вида ${*выражение*}, вместо которых в строку подставляется значение *выражения*. В нашем случае вместо специального символа ${this.urlPrefix} будет подставлено значение свойства urlPrefix. Шаблонные строки заключаются в обратные апострофы.

```
getBb(pk: Number): Observable<Object> {
    return this.http.get<Object>(`${this.urlPrefix}/api/bbs/${pk}/`);
}
```

В коде метода getBb(), загружающего с сервера сведения об объявлении с указанным ключом, тоже есть на что посмотреть. Во-первых, мы указали тип принимаемого методом параметра: getBb(pk: Number). Во-вторых, в качестве возвращаемого результата задали предположение, хранящее единичный экземпляр объекта Object (поскольку метод возвращает один объект, а не их массив):

```
getBb( . . . ): Observable<Object>
```

В-третьих, тот же единичный экземпляр объекта Object мы задали и в качестве типа получаемого значения, и у метода get():

```
this.http.get<Object>( . . . )
```

Наиболее сложен метод addComment(), добавляющий новый комментарий:

```
addComment(bb, author, password, content): Observable<Object> {
    const comment = {'bb': bb, 'author': author, 'content': content};
    const httpOptions = {
        headers: new HttpHeaders(
            { 'Content-Type': 'application/json',
            'Authorization': 'Basic ' + window.btoa(author + ':' +
            password) }
        )
    };
    return this.http.post<Object>(
           `${this.urlPrefix}/api/bbs/${bb}/comments/`,
           comment, httpOptions).pipe(catchError(this.handleError()));
}
```

Мы заносим в переменную comment экземпляр объекта Object, хранящий все необходимые для добавления комментария данные: ключ объявления, под которым оставляется комментарий, имя автора и содержание. Все эти сведения мы получаем с параметрами метода addComment(). Далее в переменную httpOptions заносим объект класса HttpHeaders, который задаст заголовки для отправляемого запроса: тип данных JSON, имя и пароль зарегистрированного пользователя. После чего вызовом метода post() службы HttpClient отправляем запрос.

Если при получении данных от веб-службы вероятность возникновения ошибки невелика, то при отправке — уже весьма значительна. Посетитель может ввести неправильные имя и (или) пароль — и мы уже имеем нештатную ситуацию. Поэтому здесь нужно предусмотреть обработку ошибок, и мы это сделали.

У предположения, возвращаемого методом `post()`, мы вызвали метод `pipe()`, в котором указали функции, выполняющие дополнительную обработку этого предположения. Точнее, одну функцию — `catchError()`, которая укажет функцию, вызываемую при возникновении ошибки. Для формирования этой функции в классе нашей службы мы объявили метод `handleError()`:

```
handleError() {
    return (error: any): Observable<Object> => {
        window.alert(error.message);
        return of(null);
    }
}
```

Он возвращает функцию, которая в качестве результата принимает объект ошибки любого типа (он обозначается ключевым словом `any`), выводит на экран окно-сообщение с текстом ошибки и возвращает предупреждение, хранящее «пустую» ссылку (`null`).

Для объявления функции, возвращаемой этим методом, мы использовали новый синтаксис создания функций, предлагаемый TypeScript. Сначала в круглых скобках записываются параметры функции, потом ставится тип возвращаемого ее результата, далее — последовательность символов `=>`, своего рода стрелка, и, наконец, в фигурных скобках записывается тело функции. Функции, объявленные подобным образом, носят название *функций-стрелок*.

35.2.5. Компонент списка объявлений *BbListComponent*. Директивы. Фильтры. Связывание данных

Вероятно, служба `BbService` — самый сложный модуль нашего Angular-сайта в плане реализации. С оставшимися двумя компонентами будет проще.

По умолчанию файлы с кодом компонентов, создаваемых уже после создания проекта, помещаются в отдельные папки. Откроем модуль **src\app\bb-list\bb-list.component.ts**, в котором хранится код компонента `BbListComponent`, выводящего список объявлений, и исправим согласно листингу 35.12.

Листинг 35.12. Код компонента `BbListComponent`

```
import { Component, OnInit } from '@angular/core';

import { BbService } from '../bb.service';

@Component({
    selector: 'app-bb-list',
    templateUrl: './bb-list.component.html',
    styleUrls: ['./bb-list.component.css']
})
```

```
export class BbListComponent implements OnInit {
    private bbs: Object[];

    constructor(private bbservice: BbService) { }

    ngOnInit() {
        this.bbservice.getBbs().subscribe(
            (bbs: Object[]) => {this.bbs = bbs;}
        );
    }
}
```

Многое здесь нам уже знакомо, но кое на что следует обратить внимание. Уже само объявление класса компонента содержит непонятный фрагмент:

```
export class BbListComponent implements OnInit {
    . . .
}
```

Это значит, что класс `BbListComponent` реализует интерфейс `OnInit`. *Интерфейсом* в TypeScript называется своего рода заготовка для объявления класса — набор свойств и методов, которые должны присутствовать в классе, реализующем этот интерфейс. Так, интерфейс `OnInit` содержит метод `ngOnInit()`, который обязательно должен быть объявлен в классе `BbListComponent`.

У интерфейсов два назначения. Первое: устранить ошибки, причиной которых может быть отсутствие в классе нужного свойства или метода. Второе: разграничить классы по реализуемым им интерфейсам, что может пригодиться, например, если необходимо выполнить над объектами класса, реализующего определенный интерфейс, какие-либо действия. Так, у объектов всех классов, реализующих интерфейс `OnInit`, при инициализации компонентов будет вызван метод `ngOnInit()`.

```
export class BbListComponent implements OnInit {
    private bbs: Object[];

    constructor(private bbservice: BbService) { }
    . . .
}
```

Объявляем частное свойство `bbs`, в котором будем хранить полученный от веб-службы массив объявлений, и свойство для объекта написанной нами ранее службы `BbService`. Да-да, созданные нами службы также обрабатываются подсистемой внедрения зависимостей Angular!

Поскольку у объектов всех классов, реализующих интерфейс `OnInit`, при инициализации вызывается метод `ngOnInit()`, в коде этого метода нашего класса можно реализовать загрузку списка объявлений. Так мы и сделали:

```
ngOnInit() {
    this.bbservice.getBbs().subscribe(
        (bbs: Object[]) => {this.bbs = bbs;}
    );
}
```

Мы вызываем метод `getBbs()` службы `BbService`. У возвращенного им результата — предположения — вызываем метод `subscribe()`, выполняющий подписку на хранимое этим предположением значение. В качестве параметра метод принимает функцию, которая получит и обработает готовое значение, — в нашем случае занесет загруженный с сервера список объявлений в свойство `bbs`.

Теперь откроем файл **src\app\bb-list\bb-list.component.html**, где хранится шаблон компонента. Исправим код шаблона так, чтобы он выглядел, как представлено в листинге 35.13.

Листинг 35.13. Код шаблона компонента `BbListComponent`

```
<h2>Последние 10 объявлений</h2>
<div *ngFor="let bb of bbs">
    <h3><a [routerLink]="[bb.id]">{{bb.title}}</a></h3>
    <p>{{bb.content}}</p>
    <p class="price">Цена: {{bb.price|currency:'RUR'}}</p>
    <p class="date">Объявление оставлено
    {{bb.created_at|date:'medium'}}</p>
</div>
```

Директива `*ngFor` по назначению аналогична тегу шаблонизатора `for . . . endfor`. Она перебирает указанный в ней массив, помещает очередной элемент в заданную переменную и выводит на экран очередную копию тега, в котором находится. В нашем случае:

```
<div *ngFor="let bb of bbs">
    . . .
</div>
```

директива `*ngFor` будет перебирать список объявлений из свойства `bbs`, помещать очередное объявление в переменную `bb` и для каждого объявления, что присутствует в списке, выводить на экран копию тега `<div>`, в котором находится.

Директива `{{`*`<переменная или свойство>`*`}}` выводит на экран значение указанной *переменной или свойства*. Так, директива `{{bb.title}}` выводит на экран название товара из очередного объявления, помещенного в переменную `bb` директивой `*ngFor`.

Любопытная деталь: если значение какой-либо *переменной или свойства*, указанного в директиве `{{`*`<переменная или свойство>`*`}}`, изменится, значение будет выведено повторно. Говорят, что такая директива реализует *связывание данных* между указанной переменной (свойством) и местом в HTML-коде шаблона, в котором она

присутствует. Поскольку отслеживается изменение значения только в переменной (свойстве), такое связывание данных является *односторонним*.

В директиве `{{bb.price|currency:'RUR'}}` используется фильтр `currency`, форматирующий выводимое значение цены товара как денежную сумму. Значение `"RUR"` этого фильтра указывает вывести в качестве обозначения денежной единицы строку `"р."`.

Фильтр `date` со значением `"medium"` форматирует выводимое значение даты и времени создания объявления как дату и время в компактном формате.

Теперь посмотрим на директиву `routerLink`, присутствующую в теге `<a>`, который создает гиперссылку на страницу сведений об объявлении:

```
<a [routerLink]="[bb.id]">{{bb.title}}</a>
```

Она формирует интернет-адрес из элементов массива, что передан ей в качестве значения. Поскольку в нашем случае передаваемый директиве массив содержит всего один элемент — ключ объявления, формируемый интернет-адрес будет иметь вид */<ключ объявления>/*. Именно в таком формате мы записали шаблонный путь в маршруте, указывающем на компонент сведений об объявлении `BbDetailComponent` (см. листинг 35.8).

Осталось оформить наш компонент. Откроем файл src\app\bb-list\bb-list.component.css, где хранится его таблица стилей, и запишем в нее код из листинга 35.14.

Листинг 35.14. Код таблицы стилей компонента `BbListComponent`

```
div {
    margin: 10px 0px;
    padding: 0px 10px;
    border: grey thin solid;
}
div p.price {
    font-size: larger;
    font-weight: bold;
    text-align: right;
}
div p.date {
    font-style: italic;
    text-align: right;
}
```

35.2.6. Компонент сведений об объявлении *BbDetailComponent*. Двустороннее связывание данных

И последний программный модуль, который нам нужно написать, — компонент `BbDetailComponent`, который выведет на экран сведения о выбранном посетителем объявлении.

Модуль с объявлением класса этого компонента хранится в файле src\app\bb-detail\bb-detail.component.ts. Исправим код класса, сообразуясь с листингом 35.15.

Листинг 35.15. Код компонента BbDetailComponent

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

import { BbService } from '../bb.service';

@Component({
    selector: 'app-bb-detail',
    templateUrl: './bb-detail.component.html',
    styleUrls: ['./bb-detail.component.css']
})
export class BbDetailComponent implements OnInit {
    private bb: any;
    private comments: Object[];
    private author: String = '';
    private password: String = '';
    private content: String = '';

    constructor(private bbservice: BbService,
                private ar: ActivatedRoute) { }

    getComments() {
        this.bbservice.getComments(this.bb.id).subscribe(
            (comments: Object[]) => {this.comments = comments;}
        );
    }

    submitComment() {
        this.bbservice.addComment(this.bb.id, this.author, this.password,
                                  this.content)
        .subscribe((comment: Object) => {
            if (comment) {
                this.content = '';
                this.getComments();
            }
        });
    }

    ngOnInit() {
        const pk = this.ar.snapshot.params.pk;
        this.bbservice.getBb(pk).subscribe((bb: Object) => {
            this.bb = bb;
            this.getComments();
        });
    }
}
```

В классе компонента объявлено целых пять свойств:

```
export class BbDetailComponent implements OnInit {
    private bb: any;
    private comments: Object[];
    private author: String = '';
    private password: String = '';
    private content: String = '';
    . . .
}
```

В свойстве `bb` будет храниться выводимое объявление, в свойстве `comments` — массив оставленных под ним комментариев, а в свойствах `author`, `password` и `content` — соответственно, автор, пароль и содержание нового комментария. Ключ объявления для отправки в составе данных нового комментария мы можем взять из свойства `id` объекта объявления, хранящегося в свойстве `bb`.

```
constructor(private bbservice: BbService, private ar: ActivatedRoute) { }

ngOnInit() {
    const pk = this.ar.snapshot.params.pk;
    this.bbservice.getBb(pk).subscribe((bb: Object) => {
        this.bb = bb;
        this.getComments();
    });
}
```

Ключ объявления, сведения о котором требуется вывести, передается в составе интернет-адреса. Чтобы извлечь его оттуда, мы воспользуемся встроенной во фреймворк службой `ActivatedRoute`, создав ее объект посредством внедрения зависимостей. Для получения ключа мы обратимся к свойству `snapshot` этой службы, хранящему объект со сведениями о маршруте, далее — к свойству `params` этого объекта, где хранится объект со значениями URL-параметров, и, наконец, к свойству `pk`, в котором и находится ключ объявления, извлеченный из интернет-адреса.

Как только сведения об объявлении будут загружены, мы вызываем метод `getComments()` компонента, выполняющий загрузку списка комментариев.

```
submitComment() {
    this.bbservice.addComment(this.bb.id, this.author, this.password,
                              this.content)
    .subscribe((comment: Object) => {
        if (comment) {
            this.content = '';
            this.getComments();
        }
    });
}
```

Метод `submitComment()` выполняется, когда посетитель, введя новый комментарий, нажмет в форме кнопку отправки данных. Этот метод вызывает метод `addComment()` службы `BbService` и получает от него результат. Если результатом окажется объект (т. е. добавление нового комментария увенчалось успехом), он производит два действия. Во-первых, очищает свойство `content`, где хранится содержание комментария, что послужит сигналом того, что новый комментарий сохранен, и подготовит почву для занесения еще одного комментария. Во-вторых, выполняет повторную загрузку списка комментариев.

Метод `getComments()` совершенно тривиален, и вы, уважаемые читатели, сами разберетесь, как он работает.

Код шаблона этого компонента хранится в файле src\app\bb-detail\bb-detail.component.html. Исправим его, как показано в листинге 35.16.

Листинг 35.16. Код шаблона компонента `BbDetailComponent`

```
<ng-container *ngIf="bb">
    <div class="image"><img src="{{bb.image}}"></div>
    <div class="others">
        <h2>{{bb.title}}</h2>
        <p>{{bb.content}}</p>
        <p class="price">Цена: {{bb.price|currency:'RUR'}}</p>
        <p>{{bb.contacts}}</p>
        <p class="date">Объявление оставлено
        {{bb.created_at|date:'medium'}}</p>
    </div>
</ng-container>
<h3>Новый комментарий</h3>
<form *ngIf="bb" (ngSubmit)="submitComment()">
    <p>Имя: <input [(ngModel)]="author" name="author" required></p>
    <p>Пароль: <input type="password" [(ngModel)]="password"
    name="password" required></p>
    <p>Содержание:<br><textarea [(ngModel)]="content" name="content"
    required></textarea></p>
    <p><input type="submit" value="Добавить"></p>
</form>
<div class="comment" *ngFor="let comment of comments">
    <h4>{{comment.author}}</h4>
    <p>{{comment.content}}</p>
    <p class="date">Комментарий оставлен
    {{comment.created_at|date:'medium'}}</p>
</div>
```

Уже в самом начале кода шаблона имеется нечто новенькое для нас:

```
<ng-container *ngIf="bb">
    . . .
</ng-container>
```

Специальный парный тег `<ng-container>` обозначает псевдоконтейнер, который не только не выводится на странице, но и никак не проявляет себя на ней. Он применяется для объединения других элементов страницы с минимумом побочных эффектов (которые вполне могут проявиться, если использовать для этого обычные элементы страницы, например, блоки).

Директива `*ngIf` эквивалентна тегу шаблонизатора `if . . . endif` и выводит тег, в котором присутствует, только при выполнении указанного в ней условия. В нашем случае эта директива выводит содержимое псевдоконтейнера только в том случае, если в свойстве `bb` хранится объект, т. е. выводимое в компоненте объявление было успешно загружено с сервера.

Директива `ngSubmit`, присутствующая в теге `<form>`:

```
<form *ngIf="bb" (ngSubmit)="submitComment()">
    . . .
</form>
```

привязывает к событию `submit` формы обработчик, указанный в качестве параметра, в нашем случае — метод `submitComment()` компонента, который мы уже рассмотрели.

В каждом теге, создающем элемент управления формы, имеется директива `ngModel`:

```
<input [(ngModel)]="author" name="author" required>
```

Она связывает элемент управления, в котором присутствует, со свойством компонента. Если изменится значение, хранящееся в свойстве, оно будет перенесено в элемент управления, а если посетитель изменит значение, выведенное в элементе управления, то изменившееся значение тут же будет перенесено в свойство. Как видим, реализуемое директивой `ngModel` связывание данных является *двусторонним*.

Осталось написать таблицу стилей для компонента `BbDetailComponent`. Она хранится в файле **src\app\bb-detail\bb-detail.component.css** и изначально «пуста». Сделайте это самостоятельно.

Как упоминалось ранее, Angular-проект включает в свой состав отладочный веб-сервер. Запустить его можно, перейдя в папку проекта и отдав в командной строке команду:

```
ng serve
```

Поскольку тестовый Angular-сайт в процессе работы «общается» с веб-службой, написанной на Django и Django REST framework, также необходимо запустить отладочный веб-сервер Django. Запустим его.

Отладочный веб-сервер Angular работает через TCP-порт 4200. Следовательно, для перехода на наш сайт нужно набрать интернет-адрес **http://localhost:4200/**.

Проверим наш небольшой тестовый сайтик в действии. После чего завершим работу обоих отладочных веб-серверов, для чего достаточно переключиться в окно командной строки, в котором запущен каждый из серверов, и нажать комбинацию клавиш <Ctrl>+<Break>.

Заключение

Вот и закончена книга, посвященная великолепному веб-фреймворку Django и разработке веб-сайтов с его помощью. Мы изучили практически все, что нужно для создания веб-проектов общего назначения, и даже сделали в качестве практического занятия полнофункциональный сайт электронной доски объявлений. И теперь можем с уверенностью и гордостью именовать себя настоящими программистами!

Автор описал в книге все основные возможности Django, без которых не обойтись. Однако невозможно объять необъятное, и кое-что все-таки осталось за кадром. В частности, не были описаны:

- подсистема GeoDjango, предназначенная для разработки геоинформационных систем;
- подсистема для разработки средствами фреймворка обычных, статических веб-сайтов;
- средства для создания карты сайта;
- средства для формирования лент новостей в формате RSS и Atom;
- инструменты для экспорта данных в форматах CSV и Adobe PDF;
- средства для написания своих миграций;
- множество полезных дополнительных библиотек;
- всевозможные вспомогательные инструменты;
- разработка дополнительных библиотек для Django.

Однако обо всем этом рассказано в официальной документации, представленной на домашнем сайте фреймворка. И любой желающий ознакомиться с этими инструментами всегда может обратиться к ней. Что касается дополнительных библиотек, то их в огромном количестве можно найти в PyPI — стандартном репозитории Python.

О да, на изучение всех возможностей Django стоит потратить время. Фреймворк Django, как и язык Python, на котором он написан, с честью выдержал проверку временем и занял свое место под солнцем. Он имеет огромную установочную базу — написанных с его применением сайтов, — и внушительную армию поклонников. К которым, автор смеет надеяться, присоединитесь и вы, уважаемые читатели.

И — автор полностью уверен в этом — Django еще долгое время будет применяться в веб-строительстве, и если и уйдет когда-нибудь, так сказать, в отставку, то лишь после появления достойного конкурента. Коих пока не предвидится...

Так что за будущее Django переживать не стоит — наш любимый фреймворк будет применяться еще очень и очень долго. И изучать его имеет смысл, как читая эту книгу, так и обращаясь к тематическим интернет-ресурсам. В табл. 3.1 приведен список таких ресурсов.

Таблица 3.1. *Интернет-ресурсы, посвященные Django*

Интернет-адрес	Описание
https://www.djangoproject.com/	Официальный сайт фреймворка Django. Дистрибутивы, документация, поддержка
https://www.djbook.ru/	Русскоязычная документация по Django
https://vk.com/django_framework	Группа «ВКонтакте», посвященная Django
https://www.python.org/	Официальный сайт языка Python. Дистрибутивы, документация, поддержка
https://pypi.org/	Официальный репозиторий Python, содержащий огромное количество дополнительных библиотек
https://pythonworld.ru/	Русскоязычный сайт для Python-программистов
https://vk.com/python_community	Группа «ВКонтакте», посвященная Python
https://vk.com/we_use_django	Группа «ВКонтакте», посвященная Python и Django

Интернет-адреса официальных сайтов использованных в книге сторонних библиотек приведены в тексте книги, в разделах, посвященных этим библиотекам.

Исходные коды разработанного в *части IV* книги веб-сайта электронной доски объявлений находятся в сопровождающем книгу электронном архиве, который можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977540582.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение*).

На этом всё. Автор книги теперь уже окончательно прощается и желает вам успешного программирования.

Владимир Дронов

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив к книге выложен на FTP-сервер издательства «БХВ-Петербург» по интернет-адресу: **ftp://ftp.bhv.ru/ 9785977540582.zip**. Ссылка на него доступна и со страницы книги на сайте **www.bhv.ru**.

Содержимое архива описано в табл. П.1.

***Таблица П.1.** Содержимое электронного архива*

Каталог, файл	**Описание**
bbclient	Папка с исходным кодом тестового веб-сайта, используемого для отладки веб-службы и написанного с применением веб-фреймворка Angular
bboard	Папка с исходным кодом веб-сайта электронной доски объявлений, разрабатываемого на протяжении *части IV* книги на Python и Django
readme.txt	Файл с описанием архива и инструкциями по развертыванию обоих веб-сайтов

Предметный указатель

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

К

М

Н

О

П

Р

С

Т

У

Ф

Ш

www.bhv.ru

Дронов В.

Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS

Отдел оптовых поставок:
E-mail: opt@bhv.spb.su

Быстрая разработка динамических сайтов — с Laravel это легко!

- Миграции
- Команды Blade
- Валидация данных
- Выгрузка файлов
- Разграничение доступа
- Использование CAPTCHA
- Поддержка BBCode
- Публикация сайта

Книга посвящена быстрой разработке профессиональных динамических Web-сайтов с применением популярнейшего в настоящее время PHP-фреймворка Laravel. Описаны технологии создания клиентской части сайта HTML5, CSS 3 и JavaScript, а для серверной части сайта — язык PHP и сервер данных MySQL. Рассказано о применении миграций Laravel для создания в базе данных таблиц, полей, индексов и связей, о написании моделей, маршрутов, контроллеров и шаблонов. Описаны средства Laravel для ввода и правки данных, встроенные во фреймворк средства валидации с применением запросов форм и инструменты для выгрузки файлов на сайт. Рассказано о подсистеме разграничения доступа Laravel и ее настройке под конкретные нужды, а также об использовании CAPTCHA. Даны практические примеры разработки дизайна страниц, интерактивных элементов — спойлера, лайтбокса и блокнота, создания универсального файлового хранилища, основанного на технологии AJAX, и реализации поддержки тегов BBCode для форматирования текста. Рассмотрен процесс разработки полнофункционального сайта и его публикации в Интернете. Все исходные коды доступны для загрузки с сайта издательства.

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор почти тридцати популярных компьютерных книг, в том числе «PHP, MySQL и Dreamweaver. Разработка интерактивных Web-сайтов», «JavaScript и AJAX в Web-дизайне», «Windows 8: разработка Metro-приложений для мобильных устройств», «Django: практика создания Web-сайтов на Python», «PHP, MySQL, HTML5 и CSS 3. Разработка современных динамических Web-сайтов» и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

www.bhv.ru

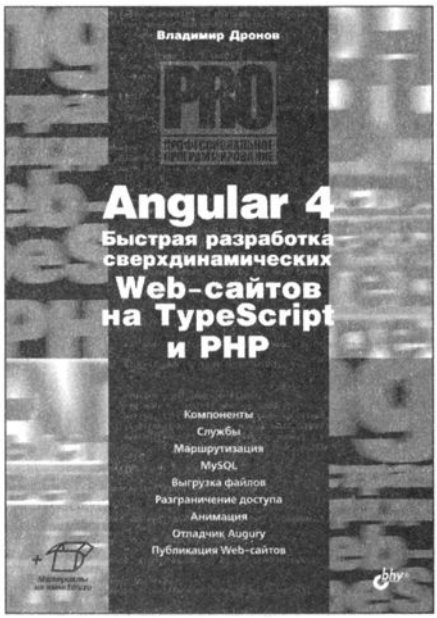
Дронов В.

Angular 4. Быстрая разработка сверхдинамических Web-сайтов на TypeScript и PHP

Отдел оптовых поставок

E-mail: opt@bhv.spb.su

Angular 4 — быстро и сверхдинамично!



- Компоненты
- Службы
- Маршрутизация
- MySQL
- Выгрузка файлов
- Разграничение доступа
- Анимация
- Отладчик Augury
- Публикация Web-сайтов

Книга посвящена быстрой разработке сверхдинамических одностраничных Web-сайтов на основе популярного фреймворка Angular 4 и языка программирования TypeScript. Дан вводный курс TypeScript, описаны типизация, классы и интерфейсы, модификаторы доступа, динамические свойства и разбиение программного кода на модули. Рассказано о создании интерфейса сайта посредством компонентов, реализации его бизнес-логики с помощью служб, структурировании программного кода сайта с применением метамодулей. Рассмотрены средства маршрутизации и навигация по сайту. Описано взаимодействие с серверной частью сайта, выгрузка файлов, программирование на языке PHP с применением баз данных MySQL. Рассказано о программировании инструментов разграничения доступа, средствах анимации, написании сложных таблиц стилей на языке LESS, тестировании сайтов с применением отладчика Augury и публикации готовых сайтов. Рассмотрен процесс создания полнофункционального сайта.

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор почти тридцати популярных компьютерных книг, в том числе «Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS», «Django: практика создания Web-сайтов на Python», «Python 3 и PyQt 5. Разработка приложений», «PHP, MySQL, HTML5 и CSS 3. Разработка современных динамических Web-сайтов», «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера. 4-е изд.», и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru». Его видеокурсы можно найти на сайте **http://osec.pro/**.

www.bhv.ru

Прохоренок Н., Дронов В.

HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера, 5-е изд.

Отдел оптовых поставок

E-mail: opt@bhv.spb.su

Объединение технологий — путь к вершинам мастерства

- HTML 5
- CSS 3
- PHP 7.2
- Web-сервер Apache
- phpMyAdmin
- AJAX
- Примеры и советы из практики

Прочитав книгу, вы научитесь не только основам технологий, но и самому главному — объединению этих технологий для создания единого целого — Web-сайта. Сотни примеров позволят наглядно увидеть весь процесс создания интерактивного сайта. Вы будете работать с базами данных, обрабатывать данные формы, отправлять письма с сайта, загружать файлы на сервер с помощью формы, сможете создать Личный кабинет для пользователей, гостевую книгу, форум и многое другое.

В 5-м издании содержится описание возможностей, предлагаемых PHP 7.2, новых инструментов JavaScript (включая рисование на холсте, средства геолокации и локальное хранилище данных) и всех нововведений, появившихся в актуальных на данный момент версиях HTML, CSS, Apache, MySQL и технологии AJAX.

Читатели о предыдущем издании:

- Превосходная книга. Главное ее достоинство в том, что описывается создание конкретного сайта, а не просто изложение PHP, JavaScript и т. д.
- Книга действительно очень хороша, написана толково и доступно, хорошо продумана структура, которая реально позволяет новичку в деле создания сайтов разобраться практически во всех аспектах этого процесса.
- Книга отличная, много конкретных и нужных примеров.

Прохоренок Николай Анатольевич, профессиональный программист, имеющий большой практический опыт создания и продвижения динамических сайтов с использованием HTML, JavaScript, PHP, Perl и MySQL. Автор книг «Python 3 и PyQt 5. Разработка приложений», «Python 3. Самое необходимое», «Основы Java», «OpenCV и Java. Обработка изображений и компьютерное зрение» и др.

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 20 популярных компьютерных книг, в том числе «Python 3 и PyQt 5. Разработка приложений», «Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS», «Angular 4. Быстрая разработка сверхдинамических Web-сайтов на TypeScript и PHP» и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

www.bhv.ru

Прохоренок Н., Дронов В.

Python 3 и PyQt 5. Разработка приложений, 2-е изд.

Отдел оптовых поставок

E-mail: opt@bhv.spb.su

Быстрое создание приложений с графическим интерфейсом

- Описание языка Python
- Объектно-ориентированное программирование
- Работа с файлами и каталогами
- Создание оконных приложений
- Работа с базами данных
- Мультимедиа
- Печать и экспорт в формат PDF
- Взаимодействие с Windows
- Сохранение настроек приложений
- Работающий пример: приложение «Судоку»

Если вы хотите научиться программировать на языке Python 3 и создавать приложения с графическим интерфейсом, эта книга для вас. В первой части книги описан базовый синтаксис языка Python 3: типы данных, операторы, условия, циклы, регулярные выражения, функции, инструменты объектно-ориентированного программирования, часто используемые модули стандартной библиотеки. Вторая часть книги посвящена библиотеке PyQt 5, позволяющей создавать приложения с графическим интерфейсом на языке Python 3. Рассмотрены средства для обработки сигналов, управления свойствами окна, разработки многопоточных приложений, описаны основные компоненты (кнопки, текстовые поля, списки, таблицы, меню, панели инструментов и др.), варианты их размещения внутри окна, инструменты для работы с базами данных, мультимедиа, вывода документов на печать и экспорта их в формате Adobe PDF, взаимодействия с Windows и сохранения настроек приложений.

Книга содержит большое количество практических примеров, помогающих начать программировать на языке Python самостоятельно. А в конце книги описывается процесс разработки приложения, предназначенного для создания и решения головоломок судоку. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

Прохоренок Николай Анатольевич, профессиональный программист, имеющий большой практический опыт создания и продвижения динамических сайтов с использованием HTML, JavaScript, PHP, Perl и MySQL. Автор книг «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Разработка Web-сайтов с помощью Perl и MySQL», «Python. Самое необходимое», «Python 3 и PyQt. Разработка приложений» и др.

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 20 популярных компьютерных книг, в том числе «Django: практика создания Web-сайтов на Python», «Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS», «Angular 4. Быстрая разработка сверхдинамических Web-сайтов на TypeScript и PHP» и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

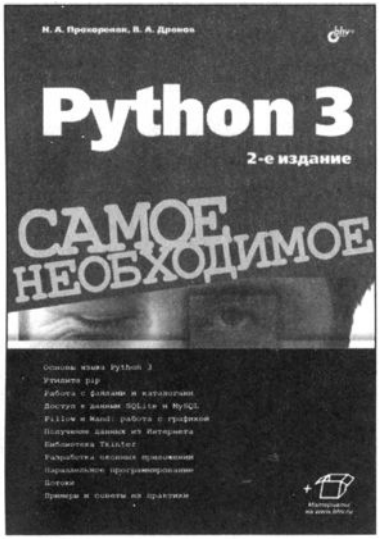
www.bhv.ru

Прохоренок Н., Дронов В.

Python 3. Самое необходимое, 2-е изд.

Отдел оптовых поставок:
e-mail: opt@bhv.spb.su

Быстро и легко осваиваем Python — самый стильный язык программирования



- Основы языка Python 3
- Утилита pip
- Работа с файлами и каталогами
- Доступ к данным SQLite и MySQL
- Pillow и Wand: работа с графикой
- Получение данных из Интернета
- Библиотека Tkinter
- Разработка оконных приложений
- Параллельное программирование
- Потоки
- Примеры и советы из практики

В книге описан базовый синтаксис языка Python 3: типы данных, операторы, условия, циклы, регулярные выражения, встроенные функции, объектно-ориентированнное программирование, обработка исключений, часто используемые модули стандартной библиотеки и установка дополнительных модулей с помощью утилиты pip. Даны основы SQLite, описан интерфейс доступа к базам данных SQLite и MySQL, рассказано об использовании ODBC для доступа к данным. Рассмотрена работа с изображениями с помощью библиотек Pillow и Wand, получение данных из Интернета, разработка оконных приложений с помощью библиотеки Tkinter, параллельное программирование и работа с архивными файлами различных форматов. Книга содержит более двухсот практических примеров, помогающих начать программировать на языке Python самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

Прохоренок Николай Анатольевич, профессиональный программист, имеющий большой практический опыт создания и продвижения динамических сайтов с использованием HTML, JavaScript, PHP, Perl и MySQL. Автор книг «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3 и PyQt 5. Разработка приложений», «Основы Java», «OpenCV и Java. Обработка изображений и компьютерное зрение» и др.

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 20 популярных компьютерных книг, в том числе «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3 и PyQt 5. Разработка приложений», «Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS», «Angular 4. Быстрая разработка сверхдинамических Web-сайтов на TypeScript и PHP» и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

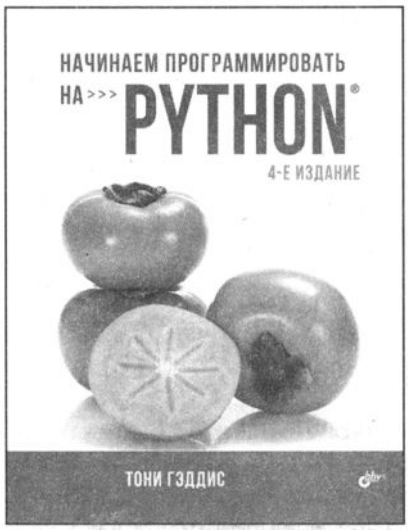
www.bhv.ru

Гэддис Т.

Начинаем программировать на Python, 4-е изд.

Отдел оптовых поставок

E-mail: opt@bhv.spb.su



- Краткое введение в компьютеры и программирование
- Ввод, обработка и вывод данных
- Управляющие структуры и булева логика
- Структуры с повторением и функции
- Файлы и исключения
- Списки и кортежи
- Строковые данные, словари и множества
- Классы и объектно-ориентированное программирование
- Наследование и рекурсия
- Функциональное программирование

В книге изложены принципы программирования, с помощью которых вы приобретете навыки алгоритмического решения задач на языке Python, даже если у вас нет опыта программирования. Для облегчения понимания сути алгоритмов широко использованы блок-схемы, псевдокод и другие инструменты. Приведено большое количество сжатых и практичных примеров программ. В каждой главе предложены тематические задачи с пошаговым анализом их решения.

Отличительной особенностью издания является его ясное, дружественное и легкое для понимания изложение материала.

Книга идеально подходит для вводного курса по программированию и разработке программного обеспечения на языке Python.

Тони Гэддис — ведущий автор всемирно известной серии книг «Начинаем программировать...» (Starting Out With) с двадцатилетним опытом преподавания курсов информатики в колледже округа Хейвуд, шт. Северная Каролина, удостоен звания «Преподаватель года», лауреат премии «Педагогическое мастерство».

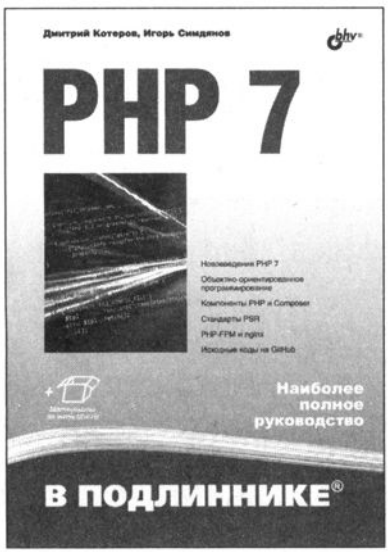
www.bhv.ru

Котеров Д., Симдянов И.

PHP 7

Отдел оптовых поставок:
e-mail: opt@bhv.spb.su

Современный подход к разработке Web-приложений



- Нововведения PHP 7
- Объектно-ориентированное программирование
- Компоненты PHP и Composer
- Стандарты PSR
- PHP-FPM и nginx
- Исходные коды на GitHub

Новую версию PHP 7 сообщество разработчиков ожидало более 10 лет. Предыдущее издание книги вышло более 8 лет назад. За это время язык и среда разработки изменились кардинально. PHP обогатился трейтами, пространством имен, анонимными функциями, замыканиями, элементами строгой типизации, генераторами, встроенным Web-сервером и многими другими возможностями. Версия PHP 7 дополняет язык новыми операторами, переработанным механизмом обработки ошибок, анонимными классами, расширенной поддержкой генераторов, кодировки UTF-8 и множеством более мелких изменений. Все возможности языка детально освещаются в книге.

За прошедшее время изменился и подход в Web-разработке. Революция, совершенная системой контроля версий Git и бесплатными Git-хостингами вроде GitHub, привела к совершенно новой системе распространения программных библиотек и их разработки. Современное Web-приложение собирается из независимых компонентов, управление которыми осуществляется менеджером Composer. Совместимость компонентов из разных фреймворков обеспечивают стандарты PSR, а рабочую среду — виртуальные машины. На страницах книги детально освещаются инструменты и приемы работы современного PHP-сообщества.

Котеров Дмитрий Владимирович, системный программист, Web-разработчик с пятнадцатилетним стажем работы в области Web-программирования, администрирования ОС Linux и Windows, ведущий разработчик хорошо известных в русскоязычном Интернете проектов Денвер (denwer.ru) и Orphus (orphus.ru). В повседневной практике использует языки PHP, JavaScript, Perl, C++, Java. Автор бестселлеров «Самоучитель PHP 4», «PHP 5» и более 50 статей, касающихся Web-программирования.

Симдянов Игорь Вячеславович, Web-разработчик, специалист с пятнадцатилетним стажем работы в области Web-программирования. Ведущий разработчик компании GoPromo, занимающейся разработкой сайтов газеты «Известия», телеканала LifeNews, радиостанции «Русская служба новостей». В повседневной практике использует языки PHP, Ruby, JavaScript, SQL. Автор двух десятков книг по Web-разработке.

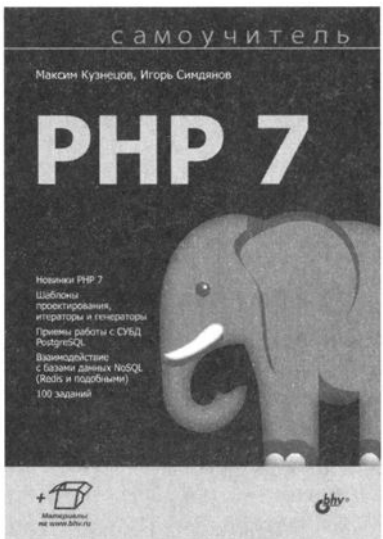
www.bhv.ru

Кузнецов М., Симдянов И.

Самоучитель PHP 7

Отдел оптовых поставок:
e-mail: opt@bhv.spb.su

Современный PHP — от простого к сложному



- Новинки PHP 7
- Шаблоны проектирования, итераторы и генераторы
- Приемы работы с СУБД PostgreSQL
- Взаимодействие с базами данных NoSQL (Redis и подобными)
- 100 заданий

Книга опытных разработчиков описывает последнюю, седьмую версию популярного языка Web-программирования PHP. Рассматриваются не только все нововведения языка, но и изменения в разработке современных Web-сайтов. Объектно-ориентированный подход, необязательный в PHP еще 10 лет назад, стал основной методологией. На смену традиционным базам данных MySQL и memcached приходят объектно-ориентированная СУБД PostgreSQL и базы данных NoSQL (Redis и подобные). Библиотеки в PHP теперь распространяются через Интернет при помощи менеджера пакетов Composer. Возможности языка PHP и сопутствующих технологий настолько возросли, что описать их в рамках одной книги становится затруднительно. По этой причине авторы ставили перед собой двойную цель: во-первых, систематически изложить язык PHP настолько полно, насколько это возможно, а во-вторых, снабдить каждую из глав заданиями, выполняя которые можно закрепить материал и познакомиться с неохваченными разделами языка и инструментами современного Web-разработчика. Книга будет интересна не только читателям, впервые знакомящимся с языком, но и профессионалам, заинтересованным в освоении современного PHP.

Кузнецов Максим Валерьевич, дважды лауреат стипендии Президента РФ, лауреат премии UNESCO, лауреат диплома I степени МГУ им. М. В. Ломоносова. Автор двух десятков книг по Web-разработке и более 50 научных работ.

Симдянов Игорь Вячеславович, ведущий разработчик группы компаний Rambler&Co с 15-летним стажем разработки Web-проектов (Известия, Life.ru, Rambler.ru). Автор двух десятков книг по Web-разработке.

Django 2.1. Практика создания веб-сайтов на Python

Python и Django — веб-разработка на высоком уровне

Книга посвящена разработке веб-сайтов на языке Python с использованием веб-фреймворка Django 2.1. Описаны основные функциональные возможности, необходимые для программирования сайтов общего назначения: модели, контроллеры, шаблоны, средства обработки пользовательского ввода, выгрузка файлов, разграничение доступа и др.

Рассказано о вспомогательных инструментах: посредниках, сигналах, средствах отправки электронной почты, подсистеме кэширования и пр. Описаны дополнительные библиотеки, производящие форматирование текста посредством BBCode, обработку CAPTCHA, вывод графических миниатюр, аутентификацию через социальные сети, интеграцию с Bootstrap. Рассмотрено программирование веб-служб REST, использование и настройка административного веб-сайта Django, описана публикация готового сайта.

Дан подробный практический пример разработки полнофункционального веб-сайта — электронной доски объявлений, в состав которого входит веб-служба.

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 30 популярных компьютерных книг, в том числе «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений», «Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS», «Angular 4. Быстрая разработка сверхдинамических Web-сайтов на TypeScript и PHP» и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

Исходные коды можно скачать по ссылке ftp://ftp.bhv.ru/9785977540582.zip, а также со страницы книги на сайте www.bhv.ru.

ISBN 978-5-9775-4058-2

9 785977 540582

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®