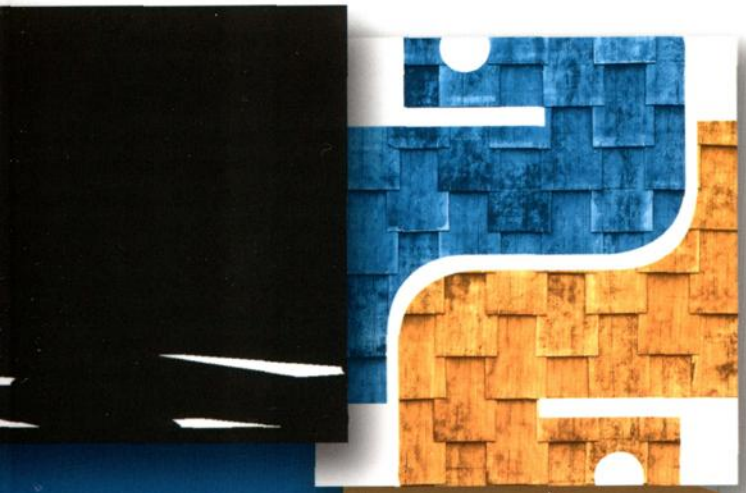
Владимир Дронов

bhv®

Django: практика создания Web-сайтов на Python



- ЯЗЫК Python И БИБЛИОТЕКА Django
- МОДЕЛИ, КОНТРОЛЛЕРЫ И ШАБЛОНЫ
- ФОРМЫ И ВЫГРУЗКА ФАЙЛОВ
- РАЗГРАНИЧЕНИЕ ДОСТУПА И КОММЕНТИРОВАНИЕ
- ФОРМАТИРОВАНИЕ BBCode
- ОТПРАВКА ЭЛЕКТРОННОЙ ПОЧТЫ
- ВСТРОЕННЫЙ АДМИНИСТРАТИВНЫЙ САЙТ
- ПОЛНОФУНКЦИОНАЛЬНЫЙ САЙТ НА Ajax

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+ Материалы на www.bhv.ru

Владимир Дронов

Django: практика создания Web-сайтов на Python

Санкт-Петербург
«БХВ-Петербург»
2016

УДК 004.738.5+004.438Python
ББК 32.973.26-018.1
Д75

Дронов В. А.

Д75 Django: практика создания Web-сайтов на Python. — СПб.: БХВ-Петербург, 2016. — 528 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-0421-8

Книга посвящена разработке Web-сайтов на популярном языке программирования Python с использованием библиотеки Django. Описывается создание моделей, контроллеров и шаблонов, применение форм для ввода данных и выгрузки на сайт файлов, реализация разграничения доступа, комментирование кода, работа со статичными страницами, применение сторонних библиотек для вывода миниатюр. Рассказывается о форматировании текста тегами BBCode, привязке к позициям тегов и выполнении поиска по тегам. Рассматриваются инструменты для генерирования каналов новостей RSS и Atom, рассылки электронной почты и настройка встроенного административного сайта Django под свои нужды. Детально описывается процесс разработки и публикации полнофункционального коммерческого Web-сайта, использующего, в том числе, технологию AJAX. Все исходные коды доступны для загрузки с сайта издательства.

Для широкого круга Web-программистов

УДК 004.738.5+004.438Python
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Капалыгина*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*

Подписано в печать 31.08.15.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 42,57.
Тираж 1000 экз. Заказ № 1043
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-5-9775-0421-8

Оглавление

Введение

Википедия — свободная интернет-энциклопедия — знает все. Давайте зайдем на нее и выполним поиск по слову `Python`. Что гласит статья, посвященная этому языку программирования?

Язык программирования Python

- Красивое лучше, чем уродливое.
- Явное лучше, чем неявное.
- Простое лучше, чем сложное.
- Читаемость имеет значение.
- Должен существовать один — и, желательно, только один — очевидный способ сделать это.

Здесь, конечно, приведены не все правила, определяющие так называемую *философию Python*, а лишь, на взгляд автора, самые значимые. Но ведь и в самом деле красивое всегда лучше уродливого, простое лучше сложного, желательно иметь только один способ сделать что-либо (чтобы лишний раз не ломать голову), а за хорошую читаемость программного кода, в особенности чужого, многие программисты, пожалуй, отдали бы все на свете.

И язык Python вполне соответствует этим постулатам. Простой, ясный с первого взгляда, непротиворечивый синтаксис, использование традиционной алгебраической записи и привычных по другим языкам конструкций, простые средства для работы со сложными типами данных, строгая модульность и компактность кода, богатые возможности объектно-ориентированного программирования — все это присутствует. А если добавить сюда еще и высокое (разумеется, для интерпретируемого языка) быстродействие, богатейшую стандартную библиотеку и наличие огромного числа сторонних библиотек, выполняющих самые разные задачи, становится совсем не удивительно, что Python в настоящее время столь популярен.

С помощью этого языка можно писать настольные приложения, в том числе имеющие графический интерфейс, системные утилиты, интернет-приложения — и командные сценарии для других программных пакетов. А еще на Python можно разрабатывать Web-серверные приложения — например, Web-сайты.

Библиотека Web-программирования Django

Специально для разработки сайтов и создана библиотека Django, ставшая сейчас, вероятно, самой популярной библиотекой такого рода из всех, написанных для использования с Python.

Вот лишь некоторые из ее возможностей:

- реализация архитектуры «модель-контроллер-шаблон»;
- реализация принципа DRY (Don't Repeat Yourself, не повторяйся), в результате чего однажды написанный код может быть использован где угодно;
- унифицированные средства для работы с базами данных любых поддерживаемых форматов: SQLite, MySQL, PostgreSQL, Oracle, Microsoft SQL Server, Firebird и др;
- мощный шаблонизатор, основанный на специальных тегах, с возможностью наследования шаблонов;
- богатые средства для работы с формами;
- инструменты для реализации разграничения доступа;
- встроенные средства для поддержки возможности комментирования, пагинации, генерирования каналов RSS и Atom, рассылки электронной почты и многие другие;
- встроенный административный сайт с возможностями настройки, который можно использовать для работы с данными;
- простая и ясная структура создаваемых сайтов: каждый раздел сайта представляет собой отдельное приложение, которое может быть отчуждено и использовано в другом сайте; все приложения объединяются в проект, собственно, и представляющий собой сайт.

Осталось добавить к этому хорошую поддержку со стороны сообщества разработчиков и наличие в Интернете большого количества сайтов, посвященных Django, на которых можно найти статьи по программированию, ответы на часто возникающие вопросы и готовые примеры кода. Так что даже неопытный Django-программист не останется один на один со своими проблемами.

Эта книга посвящена разработке Web-сайтов общего назначения на языке программирования Python с применением библиотеки Django.

Некоторые замечания от автора

Нужно сразу сказать, что книга не ставит своей целью полностью описать все возможности Python и, тем более, его гигантской стандартной библиотеки, равно как и Django. Для этого есть другие книги — огромные и очень дорогие тома, где рассказано и продемонстрировано на примерах кода абсолютно все.

Автор описывает здесь лишь основные инструменты Python и Django, без которых не обойтись при разработке обычного Web-сайта, содержащего стандартный набор

разделов: перечень товаров, разделенных на категории, гостевую книгу, новости, блог, главную страницу и набор совсем простых страниц с дополнительными сведениями (списком контактов, сведениями о разработчиках и др.).

Цель автора — не описать функциональность Python и Django полностью, а на конкретных примерах научить читателя создавать с их помощью реально работающие сайты.

В качестве практики мы создадим полнофункциональный сайт — интернет-представительство гипотетической фирмы «Веник-Торг», торгующей вениками, щетками и метлами. Этот сайт можно свободно использовать как основу для разработки других, более сложных и более, так сказать, приближенных к жизни решений.

> ***Материалы полнофункционального сайта***
>
> Электронный архив с материалами сайта фирмы «Веник-Торг» можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977504218.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение 2*).

Автор предполагает, что читатели этой книги знакомы с языком разметки Web-страниц HTML, технологией каскадных таблиц стилей CSS, языком программирования Web-сценариев JavaScript и библиотекой jQuery. Описания всех этих интернет-технологий в книге приводиться не будут.

Автор применял в работе следующие версии ПО:

- Microsoft Windows 7, русская 32-разрядная редакция со всеми установленными обновлениями;
- Notepad++ 6.5.3;
- Python 3.3.4;
- Django 1.6.2;
- setuptools 3.4.1;
- pytz 2014.2;
- Pillow 2.3.0;
- easy-thumbnails 2.0;
- django-taggit 0.12.0;
- django-precise-bbcode 0.4.2;
- MySQL 5.6.17.0;
- PostgreSQL 9.3.4;
- Psycopg 2.5.3.

В случае если какая-либо библиотека поставлялась в виде Windows-дистрибутива, выбиралась ее редакция, предназначенная для 32-разрядной редакции Python версий 3.3.*.

Типографские соглашения

В книге будут постоянно приводиться форматы написания различных конструкций, применяемых в языке Python. В них использованы особые типографские соглашения, которые мы сейчас изучим.

> ***Внимание!***
>
> Все эти типографские соглашения применяются автором только в форматах описания языковых конструкций Python. 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:

  ```
  def <имя функции>(<список параметров функции>):
  ```

  Здесь вместо подстроки *<имя функции>* должно быть подставлено реальное имя функции, а вместо подстроки *<список параметров функции>* — реальный список ее параметров.

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

  ```
  if <условие 1>:
    <блок 1>
  [elif <условие 2>:
    <блок 2>]
  ```

  Здесь фрагмент кода

  ```
  elif <условие 2>:
    <блок 2>
  ```

  может присутствовать, а может и отсутствовать.

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

  ```
  name = models.CharField(max_length = 50, unique = True, ↳
  verbose_name = "Название")
  ```

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

- Троеточием (. . .) помечены пропущенные по тем или иным причинам фрагменты кода.

  ```
  class Good(models.Model):
    name = models.CharField(max_length = 50, unique = True, ↳
    verbose_name = "Название")
    . . .
    in_stock = models.BooleanField(default = True, db_index = True, ↳
    verbose_name = "В наличии")
  ```

Здесь пропущен ряд выражений между объявлениями свойств `name` и `in_stock` класса модели `Good`.

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 изначального фрагмента, в его конец или в середину, между уже присутствующими в нем выражениями.

ЕЩЕ РАЗ ВНИМАНИЕ!

Все приведенные здесь типографские соглашения имеют смысл только в форматах описания конструкций языка Python. В коде примеров используется только знак ↪ и троеточие.

Благодарности

Автор приносит благодарности своим родителям, знакомым и коллегам по работе.

- Белову Алексею Васильевичу, начальнику отдела ОИТ Волжского гуманитарного института (г. Волжский Волгоградской обл.), где работает автор, — за понимание и поддержку.
- Всем работникам отдела ОИТ — за понимание и поддержку.
- Родителям — за терпение, понимание и поддержку.
- Архангельскому Дмитрию Борисовичу — за дружеское участие.
- Шапошникову Игорю Владимировичу — за побуждение начать писательскую деятельность.
- Рыбакову Евгению Евгеньевичу, заместителю главного редактора издательства «БХВ-Петербург», — за неоднократные побуждения к работе, без которых автор давно бы обленился.
- Издательству «БХВ-Петербург» — за издание моих книг.
- Разработчикам Python и Django и всех прочих использованных автором программных продуктов — за их труд.
- Всем своим читателям и почитателям — за прекрасные отзывы о моих книгах.
- Всем, кого я забыл здесь перечислить, — за все хорошее.

ЧАСТЬ I

Web-приложения. Язык Python. Библиотека Django

ГЛАВА 1

Введение в серверное Web-программирование

Не откладывая дела в долгий ящик, сразу же приступим к рассмотрению основных принципов серверного Web-программирования. Мы узнаем, что такое собственно серверные Web-приложения, как они работают, что есть базы данных, таблицы, поля, записи, индексы, модели, контроллеры и шаблоны. Без всего этого мы просто не поймем, о чем пойдет речь в следующих главах книги.

А начнем мы с того, что разберемся, в чем принципиальная разница между статичными Web-страницами и Web-приложениями, и рассмотрим их преимущества и недостатки.

Статичные Web-страницы и Web-приложения — две эпохи в развитии Интернета

В более чем тридцатилетней истории Интернета можно выделить две эпохи. Эпоха первая, давно прошедшая, представляла собой царство статичных Web-страниц — тех самых, что пишутся на языке *HTML* (HyperText Markup Language, язык гипертекстовой разметки) и хранятся в обычных текстовых файлах с расширениями htm или html. Эпоха вторая, которая продолжается и сейчас, ознаменована господством серверных Web-приложений, особых программ, которые получают данные из базы, обрабатывают их и генерируют на основе результатов обработки Web-страницы.

Но почему произошел такой резкий переход от статики к, можно сказать, динамике? Чем Web-дизайнеров не устраивали старые добрые Web-страницы?

Статичные Web-страницы

Написать статичную Web-страницу (для краткости их называют просто Web-страницами) — пара пустяков. Необходимый для их создания язык HTML сейчас знают даже школьники, равно как и язык *CSS* (Cascading Style Sheets, каскадные таблицы стилей), на котором в виде каскадных таблиц стилей и описывается их оформление.

WEB-СКРИПТЫ НА ЯЗЫКЕ JAVASCRIPT

Существуют также *Web-сценарии*, или *Web-скрипты*, с помощью которых программируется поведение страниц или отдельных их элементов в ответ на действия посетителя или каких-либо событий, происходящих в Web-обозревателе. Они пишутся на языке программирования JavaScript. Однако в этой книге мы их касаться не будем.

А для разработки Web-страниц подойдет любой текстовый редактор — например, Блокнот, поставляемый в составе Windows.

Давайте создадим Web-страницу с вот таким кодом:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Страница 1</title>
  </head>
  <body>
    <h1>Это страница 1</h1>
    <p>Щелкните гиперссылку внизу, чтобы перейти на страницу 2.</p>
    <p><a href="pages/2.html">Перейти</a>.</p>
  </body>
</html>
```

Эта страница очень проста — она включает лишь заголовок, два абзаца и гиперссылку для перехода на вторую страницу, которую мы скоро сделаем.

Сохраним нашу первую Web-страницу в какой-либо папке в кодировке UTF-8 под именем 1.html.

Теперь создадим еще одну страницу. Ее HTML-код будет таким:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Страница 2</title>
  </head>
  <body>
    <h1>Это страница 2</h1>
    <p>Щелкните гиперссылку внизу, чтобы вернуться на страницу 1.</p>
    <p><a href="../1.html">Перейти</a>.</p>
  </body>
</html>
```

Здесь все то же самое, за единственным отличием — гиперссылка ведет на первую страницу.

Создадим в папке, где хранится первая страница, вложенную папку pages и сохраним в этой папке вторую страницу также в кодировке UTF-8, дав ей имя 2.html.

Откроем страницу 1.html в Web-обозревателе. Выглядеть она будет так, как показано на рис. 1.1.

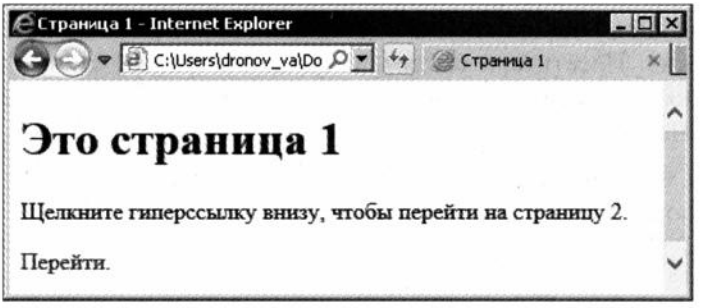


Рис. 1.1. Статичная Web-страница

Щелкнем на гиперссылке, чтобы перейти на вторую страницу, после чего вернемся на первую. Если мы не допустили в HTML-коде ошибок, все должно получиться.

Как видим, сайт на статичных Web-страницах делается очень просто — мы создаем страницы и раскладываем их по папкам. Структура папок отражает структуру самого сайта.

Обрабатываются статичные страницы также очень просто. Когда посетитель набирает в строке ввода адреса Web-обозревателя интернет-адрес нашего сайта и нажимает клавишу <Enter>, Web-обозреватель отправляет компьютеру, располагающемуся в Сети по этому адресу, особый запрос. Операционная система удаленного компьютера передает этот запрос программе *Web-сервера*. Та извлекает из запроса имя файла запрошенной Web-страницы, считывает этот файл и отправляет его Web-обозревателю. Последнему остается лишь получить файл, обработать его и вывести страницу на экран.

Ключевых преимуществ у статичных Web-страниц два. Во-первых, их очень просто создавать — для этого достаточно знать языки HTML и CSS, о которых говорилось ранее. Во-вторых, развернуть на компьютере статичный Web-сайт не составляет ни малейшего труда — нужно лишь установить на него и настроить программу Web-сервера, что можно сделать буквально за пять минут.

Сайты, основанные на статичных Web-страницах, активно создаются до сих пор. Это всевозможные домашние страницы, рекламные сайты и сайты-визитки.

Web-приложения

А теперь представим себе такую ситуацию. Мы создали основанный на статичных страницах корпоративный Web-сайт некоей фирмы, содержащий в своем составе каталог товаров. Какое-то время этот сайт работал и благополучно привлекал клиентов, пока начальство не поставило нам следующие задачи:

1. Реализовать в каталоге фильтрацию товаров по ключевым словам и сортировку их по наименованиям и цене.
2. Создать гостевую книгу.
3. Создать блог для сотрудников фирмы, где они смогли бы публиковать тематические статьи.

Конечно, соорудить некое подобие гостевой книги и блога на статичных Web-страницах можно. Посетители сайта и сотрудники фирмы станут присылать нам свои сообщения и статьи по электронной почте, а мы будем вставлять их в код Web-страниц. Неудобно, но вполне реализуемо.

Но как сделать фильтрацию и сортировку товаров в каталоге? Средствами HTML и CSS это реализовать всяко не получится.

Выход здесь один — создать некую программу, которая будет:

- работать совместно с Web-сервером;
- перехватывать запросы, получаемые Web-сервером, и активизироваться лишь при обращении к интернет-адресу списка товаров;
- извлекать из запросов введенные посетителем ключевые слова для поиска и критерий сортировки (такие данные обычно отправляют как часть интернет-адреса — методом GET);
- считывать из базы данных список товаров, сортировать его согласно заданным критериям и отфильтровывать лишь те товары, чьи наименования включают указанное ключевое слово;
- формировать на основе результирующего списка товаров обычную Web-страницу и передавать ее Web-серверу, который, в свою очередь, отправит ее посетителю.

В этом случае Web-страницы нашего сайта не будут храниться в файлах на жестких дисках компьютера, а станут генерироваться на «лету» особым приложением. Сразу видно, что такой подход решит огромное количество проблем, более того, мы, немного поразмыслив, без труда сможем создать на сайте, в том числе, и гостевую книгу с блогом, написав соответствующие программы.

Такие приложения, работающие совместно с Web-сервером и генерирующие страницы на основе данных, хранящихся в базе и введенных посетителем, носят название *Web-приложений*. Точнее, *серверных Web-приложений*, поскольку они функционируют на стороне Web-сервера.

> ***Клиентские Web-приложения***
>
> Существуют также *клиентские Web-приложения*, представляющие собой сложные Web-сценарии и обрабатывающие данные, как полученные от серверного Web-приложения, так и введенные посетителем, на стороне клиента — Web-обозревателя. Но, поскольку в этой книге не рассматривается клиентское Web-программирование, мы говорить о них не будем.

Как правило, сложные сайты включают в себя сразу несколько серверных приложений. Так, отдельное приложение формирует *главную Web-страницу*, которая выводится при обращении непосредственно по адресу сайта, отдельные же приложения генерируют список товаров, гостевую книгу, блог, фотогалерею и прочие разделы сайта. Обычные страницы, наподобие сведений о фирме и сайте или списка контактов, могут также генерироваться разными приложениями, равно как и создаваться одним «универсальным» приложением — это зависит от реализации.

Как мы уже знаем, Web-сайт, основанный на статичных страницах, представляет собой набор файлов и папок. Получив запрос, Web-сервер просто извлекает из него имя запрошенного файла, после чего считывает его с диска и пересылает Web-обозревателю. Например, получив запрос **http://www.somesite.ru/pages/3.html**, Web-сервер прочтет файл 3.html из папки pages, что находится в папке, где помещается сам сайт.

В Web-сайте, основанном на серверных приложениях, все несколько иначе. Каждое составляющее его приложение привязывается к определенному интернет-адресу, который можно рассматривать как «папку», не существующую в реальности. Web-сервер, получив запрос на обращение к такой «папке», запускает соответствующее ей серверное приложение. Скажем, если обратиться по интернет-адресу **http://www.somesite.ru/goods/**, будет запущено приложение, выводящее список товаров.

Самое интересное, что «структура» таких «папок» совершенно не обязана совпадать со структурой файлов и папок, составляющих сайт. С одной стороны, это может внести некоторую путаницу, но, с другой, позволяет реализовать весьма интересные сценарии работы...

Подробный рассказ об интернет-адресах и привязке их к приложениям еще впереди. А пока что давайте рассмотрим преимущества сайтов, основанных на серверных приложениях.

- Возможность хранить данные отдельно (как правило, в базе данных), или, говоря другими словами, отделить их от обработки и вывода. Это может быть полезно, если эти данные используются не только в сайте, но и где-либо еще, скажем, в бухгалтерском или складском приложении.
- Возможность реализовать обработку данных перед их выводом: фильтровать их, сортировать и объединять в группы. Мы можем даже считать количество позиций в списке, вычислять средние значения для параметров позиций и пр.
- Возможность получить данные от посетителя и сохранить их в базе.
- Повысить безопасность, заблокировав доступ к файлам сайта, не предназначенным для чужих глаз (той же базе данных).

Недостатков у подобных сайтов всего два, и оба не являются критичными:

- для создания серверных приложений необходимо дополнительно изучить язык программирования, на котором будут писаться эти приложения, и платформу их разработки. Что, впрочем, несложно;
- помимо собственно Web-сервера, нам понадобится также установить и настроить программное ядро соответствующей платформы. Сделать это также несложно — весь процесс установки и первоначальной настройки занимает несколько минут.

Платформ для разработки серверных приложений существует несколько. Прежде всего, это PHP (Pretty Home Page, симпатичная домашняя страница) — на данный момент имеющая наибольшую популярность. Приложения для этой платформы создаются на особом языке, который также называется PHP. А чтобы облегчить

труд Web-программиста, написано несколько библиотек, реализующих типовые задачи, которые в противном случае придется решать самому разработчику: Zend Framework, Yii и многие другие.

Мы же будем использовать платформу Django. Эта платформа служит для создания Web-приложений на языке Python (и сама, кстати, написана на этом языке).

Базы данных. Реляционные базы данных

Ранее мы уже говорили, что данные, с которыми работают серверные Web-приложения, хранятся в базах данных. Настала пора узнать, что это такое.

Что такое реляционная база данных?

Итак, *база данных* — это файл (или набор файлов), хранящий структурированную определенным образом информацию. Для обработки этой информации используются особые программы, называемые *системами управления базами данных*, или *СУБД*. Примеры СУБД: MySQL, PostgreSQL, Oracle, Microsoft SQL Server и Microsoft Access.

Базы данных делятся на несколько видов по способу структурирования содержащейся в них информации. Мы будем пользоваться *реляционными базами данных* — они служат для хранения информации, организованной в виде связанных друг с другом таблиц, и в настоящее время имеют наибольшее распространение.

Кстати, все упомянутые ранее СУБД обрабатывают реляционные базы данных.

Что хранит реляционная база данных?

Реляционная база хранит структуры, относящиеся к трем различным типам. Давайте их рассмотрим.

Таблицы, поля и записи

Таблица — это набор структурированных данных. Пример таблицы представлен на рис. 1.2.

Эта таблица содержит список товаров из пяти столбцов: категория товара, его наименование, описание, цена и признак того, имеется ли товар в наличии.

Каждая таблица, хранящаяся в базе данных, должна иметь уникальное в пределах этой базы имя. Это нужно для того, чтобы СУБД (да и мы сами) смогла найти эту таблицу.

Отдельная строка таблицы, содержащая реальные данные, называется *записью*. (Строка заголовка, выделенная на рис. 1.2 черным фоном, записью не является, т. к. содержит не реальные данные, описывающие какую-либо статью, а служебные сведения — заголовки столбцов.) На каждый товар, занесенный в таблицу, отводится одна запись.

category	name	description	price	in_stock
Веники	В-1	Классическая конструкция из экологически чистых материалов	100	*
Веники	В-2М	Усовершенствованная конструкция, выполненная с применением особо прочной синтетики	200	
Веники	В-2000	Современная конструкция, выполненная с применением нанотехнологий	2000	*
Метлы	М-1	Классическая конструкция из экологически чистых материалов	150	*
Метлы	М-2014О	Выпущена в честь Зимней олимпиады в Сочи 2014 года	15000	*

Рис. 1.2. Таблица — список товаров

Отдельная ячейка отдельной строки-записи называется *поле́м*. Можно сказать, что поле — это порция данных, составляющих запись. А сами данные, помещенные в поле, называются его *значением*.

Каждое поле обязано иметь уникальное в пределах таблицы имя. Имена полей, кстати, и приведены в строке заголовка таблицы на рис. 1.2.

Поле способно хранить данные какого-то одного типа: строки, числа, даты и т. п. *Тип* хранимых в поле данных задается при создании поля (и может быть потом изменен, если был задан ошибочно). СУБД не позволит записать, скажем, дату в поле, предназначенное для хранения строк. Типы данных, поддерживаемые большинством баз данных, приведены в табл. 1.1.

***Таблица 1.1.** Типы данных, поддерживаемые большинством форматов баз данных*

Название	Описание
Строковый	Текст фиксированной длины, содержащий любые символы: буквы, цифры, знаки препинания, пробелы и пр. Максимальная длина текста, хранимого в таком поле, задается при его создании
Целочисленный	Целые числа
С плавающей точкой	Дробные числа
Логический	Значения вида «истина» (true) или «ложь» (false)
Дата	Значения даты
Дата и время	Объединенное значение даты и времени
Memo	Текст произвольной длины, содержащий любые символы: буквы, цифры, знаки препинания, пробелы и пр. Длина хранимого в таком поле текста не ограничена (по крайней мере, очень велика)
Счетчик	Постепенно увеличивающиеся и уникальные в пределах таблицы целые числа. Поля такого типа используются для специальных целей, в частности, в качестве ключевого поля (см. далее)

Несколько разновидностей типов данных

На самом деле, существует несколько разновидностей целочисленного типа данных и типа с плавающей точкой, различающихся величинами чисел, которые могут быть записаны в поле данного типа. Мы поговорим о них потом.

Посмотрим еще раз на рис. 1.2. Представленная там таблица имеет пять полей. А какого они типа? Давайте подумаем.

- Поле категории (`category`) получит строковый тип, поскольку название категории представляет собой слово. (Для него еще нужно указать предельную длину, но это можно сделать и потом.)
- Поле наименования товара (`name`) — также строкового типа. (Ему тоже следует задать предельную длину.)
- Полю описания товара (`description`) мы дадим тип memo. Такого рода данные могут иметь достаточно большой размер.
- В случае поля цены (`price`) вариант один — число с плавающей точкой. (Впрочем, раз цены на наши товары указываются в целых рублях, то можно использовать и целочисленный тип.)
- В случае поля признака, указывающего на наличие товара (`in_stock`), у нас тоже нет особого выбора — логический тип подходит для этого наилучшим образом.

Большинство форматов баз данных позволяют задать для поля *правила* — характеристики, которым должны удовлетворять записываемые туда данные. Такими условиями могут быть:

- обязательное наличие в поле какого-либо значения (*обязательное поле*);
- значение, которое должно быть помещено в поле при создании новой записи (*значение поля по умолчанию*);
- дополнительные условия (например, диапазон значений, в которые должно укладываться число).

Осталось только сказать, что набор полей с их именами, типами данных и условиями называется *структурой* таблицы. Сами реальные данные — содержимое полей и записей — в структуру не входят.

Индексы и ключи

Предположим, что мы создали таблицу, изображенную на рис. 1.2, заполнили ее данными и теперь пишем приложение, которое выводит на экран список товаров. И нам требуется отсортировать товары по какому-либо полю — например, по цене. В этом случае СУБД будет вынуждена:

1. Прочитать из базы данных все записи и поместить их в оперативную память, создав особый список.
2. Создать в памяти еще один список, содержащий все значения поля, по которому выполняется сортировка.
3. Переупорядочить эти значения, чтобы они шли по нарастанию или убыванию, и создать на их основе еще один список — третий по счету.

4. Соответственно переупорядочить записи таблицы и поместить их в отдельный список, который станет уже четвертым.

Если записей в таблице мало, этот процесс не займет много ни времени, ни оперативной памяти. А если записей там уже пара сотен?

Чтобы ускорить обработку записей, мы можем указать СУБД создать в базе отдельный массив данных, включающий все значения определенного поля таблицы, которые уже упорядочены нужным нам образом, и ссылки на соответствующие им записи. Понятно, что сортировка в этом случае будет выполняться много быстрее — ведь эти значения уже отсортированы, и СУБД остается лишь:

1. Прочитать из базы данных содержимое этого массива.
2. Прочитать из базы данных все записи таблицы.
3. Соответственно переупорядочить записи.

В этом случае, помимо ускорения обработки, потребуется еще и заметно меньше оперативной памяти — ведь будут созданы три списка, а не четыре, как ранее.

Такой список значений называется *индексом*, а поле, значения которого хранятся в индексе, — *индексированным*.

Индексы поддерживаются абсолютно всеми форматами баз данных и используются очень часто. В самом деле, индекс занимает немного места на диске и в памяти, а ускоряет операцию сортировки очень заметно. Единственный недостаток: при добавлении, изменении или удалении любой записи СУБД будет вынуждена соответственно изменить индекс, что отнимает некоторое время. Поэтому не стоит без необходимости создавать слишком много индексов.

Кроме сортировки, индексы также могут помочь при выполнении фильтрации записей. СУБД считывает индекс в память, ищет в нем значения, удовлетворяющие заданному критерию, и извлекает нужные записи из таблицы. Просто и быстро!

Изначально, при открытии таблицы, СУБД не считывает ни один индекс — они задействуются только при сортировке и фильтрации. Но имеется возможность сделать один из индексов загружаемым при открытии таблицы — при этом таблица будет изначально отсортирована согласно этому индексу. Такой индекс называется *ключевым*, или *ключом*, а задействованное в нем поле — *ключевым*. Ключевой индекс может быть только один на всю таблицу.

Ключевое поле должно удовлетворять следующим условиям:

- оно должно содержать значение (т. е. быть обязательным полем);
- оно должно содержать уникальные в пределах таблицы значения (быть уникальным полем).

Обычно в качестве ключевого применяется поле типа счетчика (см. табл. 1.1). Такие поля подходят для этого наилучшим образом. (Хотя, конечно, никто не мешает нам применить для этого поле любого другого типа.)

Ключевые индексы используются для того, чтобы однозначно идентифицировать какую-либо запись для изменения хранящихся в ней значений и ее удаления. Они также применяются для установления межтабличных связей.

Связи

Кстати, поговорить о межтабличных связях сейчас самое время. И вот почему...

Давайте посмотрим на таблицу, что показана на рис. 1.2. В частности, на поле `category`, где хранится категория товара. Чем примечательны хранящиеся в нем значения? И чем неоптимален такой способ указания категории?

Тем, что в этом поле записывается само ее название. Во-первых, оно довольно длинное и, соответственно, занимает немало места в базе данных. (Да, сейчас у нас категории имеют короткие названия. Но это сейчас...) Во-вторых, при вводе мы можем по ошибке указать название категории неправильно и тем самым нарушить работу приложения. В-третьих, если решим изменить название какой-либо категории, нам придется перебрать все относящиеся к ней товары и внести нужные правки в соответствующие им записи, что отнимет много времени.

Простое и красивое решение предлагает нам сам формат реляционных баз, который представляет данные как набор связанных таблиц.

Создадим в базе еще одну таблицу — для хранения списка категорий (рис. 1.3). Она будет содержать следующие поля:

- `id` — уникальный идентификатор записи, тип — счетчик, ключевое;
- `name` — название категории, тип — строковый.

id	name
1	Веники
2	Метлы

Рис. 1.3. Таблица — список категорий товаров

Теперь удалим из таблицы-списка товаров поле `category` и создадим в ней поле с тем же именем, но целочисленного типа. В этом поле будет храниться значение поля `id` таблицы-списка категорий, соответствующее данному товару. То есть вместо того, чтобы хранить в списке товаров сами названия категорий, мы будем помещать туда лишь ссылки на них.

Преимуществ у такого подхода два. Во-первых, мы храним в таблице не длинную строку, а короткое целое число, за счет чего размер базы данных станет меньше, а процесс ее обработки — быстрее. Во-вторых, мы не будем иметь никаких проблем, если вдруг захотим поменять название категорий, — ведь для этого нам потребуется исправить всего одну запись в таблице — списке категорий.

Можем себя поздравить — мы только что создали первую в нашей практике *связь* между таблицами (рис. 7.4).

В нашем случае одна запись списка категорий связана с произвольным количеством записей списка товаров. (В реальности так и бывает — в одну категорию могут входить множество товаров.) Это связь *один-ко-многим*. При этом таблица-список категорий будет *первичной*, или *родительской*, поскольку она, можно сказать, под-

чиняет себе связанные записи. А таблица-список товаров, напротив, станет *вторичной* или *дочерней*. Обе таблицы при этом станут *связанными*.

Поле вторичной таблицы, содержащее ссылки на записи первичной таблицы (у нас это поле `category`), называется *внешним индексом*. «Внешний» — потому что это поле внешнее по отношению к первичной таблице, а «индекс» — потому что практически всегда на основе этого поля создается индекс.

category	name	description	price	in_stock
1	B-1	Классическая конструкция из экологически чистых материалов	100	*
1	B-2M	Усовершенствованная конструкция, выполненная с применением особо прочной синтетики	200	
1	B-2000	Современная конструкция, выполненная с применением нанотехнологий	2000	*
2	M-1	Классическая конструкция из экологически чистых материалов	150	*
2	M-2014O	Выпущена в честь Зимней олимпиады в Сочи 2014 года	15000	*

id	name
1	Веники
2	Метлы

Рис. 1.4. Таблица-список категорий, связанная с таблицей-списком товаров

Правила построения реляционных баз данных требуют выделения одинаковых значений в отдельные связанные таблицы. В таком случае нам самим будет легче как поддерживать базу впоследствии, так и писать приложения, которые станут с ней работать.

Но что случится, если мы попытаемся удалить запись первичной таблицы, на которую ссылаются записи таблицы вторичной? СУБД просто не позволит это сделать, выведя нам сообщение об ошибке нарушения ссылочной целостности. Так что разорвать связь между записями, случайно или преднамеренно, у нас не получится.

Еще мы можем создать между таблицами связь типа *один-к-одному*, когда на одну запись первичной таблицы может ссылаться только одна запись таблицы вторичной. Если же мы попытаемся привязать к записи первичной таблицы еще одну запись вторичной, то, опять же, получим сообщение об ошибке. Однако такие связи на практике применяются довольно редко.

На этом ударный курс теории баз данных можно считать законченным.

Основные принципы разработки серверных Web-приложений

На очереди — не менее ударный курс теории программирования серверных Web-приложений.

Каждое более или менее сложное приложение (это относится к программам любого типа, не только к серверным) состоит из нескольких программных модулей, выполняющих различные задачи. И это понятно — ведь написать большое приложение, состоящее из одного модуля, очень трудно, если вообще возможно.

Все модули, из которых состоит серверное приложение, можно разделить на четыре разновидности: модели, контроллеры, шаблоны и служебные. Рассмотрим их поочередно.

Модели

Модель — это программный модуль, входящий в состав приложения, который служит своего рода посредником между остальными его модулями и базой данных. Или, говоря другими словами, модель — суть представление базы данных, ее таблиц, полей, индексов и связей в терминологии языка программирования, на котором пишется данное приложение.

Модель выполняет следующие задачи:

- описывает таблицы базы данных и их структуру в терминологии используемого языка программирования. Благодаря этому мы можем получать данные из базы, не прибегая к сторонним средствам;
- представляет считанные из базы данные в терминологии используемого языка программирования. Так что мы, считав с помощью модели какую-либо запись таблицы, сможем обработать ее средствами выбранного нами языка, опять же, не привлекая сторонние инструменты;
- реализует механизм выборки данных, их фильтрации и сортировки;
- реализует механизм добавления в таблицы новых записей, а также правки и удаления существующих;
- следит за корректностью данных, позволяя обрабатывать возникающие ошибки средствами выбранного языка программирования;
- возможно, расширяет набор средств, предоставляемых принятым форматом баз данных, добавляя к нему дополнительные инструменты, которые созданы разработчиком приложения или сторонними программистами.

Если уж совсем коротко, то модель — наш пропуск в базу данных.

Приложение может включать в свой состав произвольное количество моделей. Обычно каждая модель соответствует определенной таблице в базе данных.

Отметим сразу, что модели в приложении всегда играют подчиненную роль. Они вызываются другими модулями, относящимися к другой разновидности — кон-

троллерам, когда им требуется обратиться к базе данных, и вызываются явно, указанием в программном коде особой команды.

Контроллеры

Контроллер — это модуль приложения, выполняющий непосредственно обработку данных. Это главная действующая часть приложения, его сердце.

Обязанности у контроллера следующие:

- выборка данных из базы посредством явно вызываемых моделей;
- обработка полученных данных: фильтрация и сортировка;
- получение данных, отправленных пользователем;
- занесение полученных от пользователя данных в базу, опять же, посредством явно вызванных моделей, или иная их обработка;
- запуск формирования на основе результата обработки данных Web-страницы, которую увидит посетитель сайта.

Приложение может содержать произвольное количество контроллеров. Каждый контроллер соответствует определенному действию, выполняемому приложением, — так, выборка списка товаров выполняется одним контроллером, а добавление нового товара в список — другим.

Каждый контроллер ставится в соответствие определенному интернет-адресу приложения. Например, контроллер, выводящий список товаров, ставится в соответствие интернет-адресу **/goods/**, а контроллер, который добавляет товар в список, — интернет-адресу **/goods/add/**. Отслеживанием запрошенных посетителем интернет-адресов занимается особый служебный модуль, входящий в программное ядро приложения, он же выполняет запуск нужного контроллера при обращении к «его» адресу.

Контроллеры в процессе работы загружают и запускают остальные модули: модели и шаблоны. Так что их вполне можно назвать хозяевами положения... или приложения...

Шаблоны

Шаблон — это модуль приложения, единственное назначение которого — принять подготовленные контроллером данные и сформировать на их основе результирующую Web-страницу. Если совсем коротко, то шаблоны занимаются выводом данных.

Если модели и контроллеры представляют собой программы, написанные на языке программирования, то шаблон — это фактически обычная Web-страница, созданная на языке HTML. За единственным исключением — в ее код вставлены особые теги (*теги шаблона*), которые указывают, какие данные и в каком формате следует сюда поместить.

Приложение может включать в свой состав произвольное количество шаблонов. Правило здесь очень простое — каждой Web-странице, формируемой приложением, соответствует свой шаблон.

Как и модели, шаблоны явно вызываются контроллерами, когда им потребуется вывести обработанные данные.

Служебные модули

Что касается служебных модулей, входящих в состав приложения, то они включают в себя:

- модуль, отслеживающий запрошенные посетителем интернет-адреса и запускающий соответствующие им контроллеры (*диспетчер*);
- модуль, обрабатывающий теги шаблонов, т. е. подставляющий на их место отформатированные указанным образом данные (*шаблонизатор*);
- модуль настроек приложения;
- разнообразные модули, выполняющие типовые задачи Web-программирования: разграничение доступа, кэширование и пр.

Служебные модули составляют так называемое *программное ядро* приложения. Оно обеспечивает само функционирование приложения, не зависит от структуры создаваемого сайта и выполняемых им задач, вследствие чего может быть использовано в самых разных сайтах, обычно пишется один раз и модифицируется крайне редко, как правило, лишь с целью существенно расширить его функциональность.

Служебные модули могут как вызываться явно, когда в них возникнет нужда, так и постоянно функционировать «за кулисами». Одни служебные модули задействуются в любом случае (например, диспетчер), а другие могут включаться и отключаться в настройках приложения в зависимости от того, наличествует ли в них необходимость или нет.

Осталось лишь сказать, что принцип построения приложения, или, говоря другими словами, его *архитектура*, когда функциональность разделяется между моделями, контроллерами и шаблонами, носит название *модель-контроллер-шаблон*. (В зарубежной литературе применяется термин *model-view-controller*, *MVC*.)

Что дальше?

В этой главе мы выяснили, что такое Web-приложения и какие преимущества они несут по сравнению со статичными Web-страницами, что такое база данных и что она хранит. Еще мы узнали об архитектуре построения приложений модель-контроллер-шаблон и познакомились с моделями, контроллерами и шаблонами.

Следующая глава будет целиком посвящена языку программирования Python, на котором мы будем писать наши Web-приложения. Язык этот очень прост, в чем мы скоро и убедимся.

ГЛАВА 2

Язык программирования Python

В предыдущей главе мы рассмотрели основные принципы серверного Web-программирования и теорию баз данных. В этой главе мы приблизимся к практике, приступив к изучению языка программирования Python.

Python — высокоуровневый, объектно-ориентированный, тьюринг-полный язык программирования, одинаково хорошо подходящий для разработки как простейших командных скриптов, так и сложных настольных и Web-приложений. В комплекте с ним поставляется богатейшая стандартная библиотека, включающая мощные средства для обработки текстов, поддержки шифрования, работы с файлами, реализации обмена данных через Интернет и многое другое.

Но нас пока что интересуют базовые возможности Python, его синтаксис, поддерживаемые им типы данных, управляющие структуры и инструменты для работы с классами и объектами. Ими-то мы здесь и займемся.

Интерактивный интерпретатор Python

Разговор об этом языке будет сопровождаться большим количеством примеров. Чтобы проверить их в действии, мы можем использовать *интерактивный интерпретатор* Python, входящий в комплект его поставки.

После установки Python на компьютер в системном меню **Пуск** появится группа с именем вида **Python *<номер версии без последней цифры>***. (Например, у автора, установившего версию 3.3.4, эта папка носит имя **Python 3.3**.) В ней имеется ярлык **IDLE (Python GUI)**, который и запускает интерактивный интерпретатор.

Окно этой программы показано на рис. 2.1. Оно содержит большую область редактирования, куда вводится Python-код, и вследствие этого напоминает окно текстового редактора.

Введем в это окно следующее выражение:

```
2 + 3
```

и нажмем клавишу <Enter>. Тем самым мы дадим Python указание вывести на экран результат сложения чисел 2 и 3. Результат этот, равный 5, появится в следующей строке.

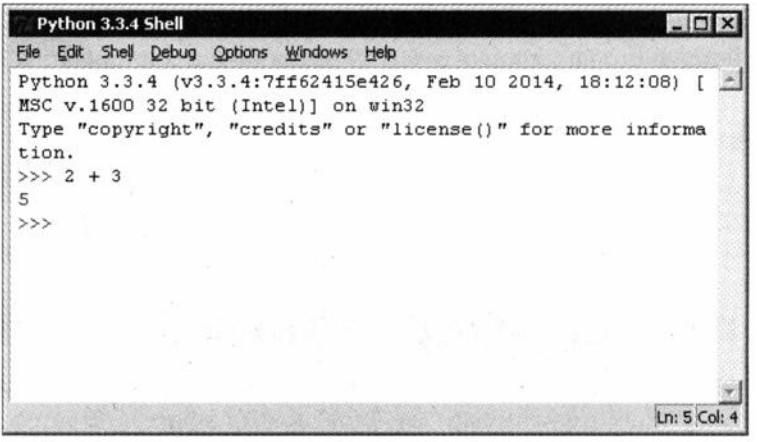


Рис. 2.1. Окно интерактивного интерпретатора Python

Что ж, по крайней мере, складывать числа этот язык умеет. Посмотрим, на что он еще способен...

Основные понятия Python

Начнем мы с самых простых понятий Python-программирования. Это выражение, оператор, функция и переменная.

Выражения

Ранее мы назвали команду `2 + 3`, введенную в окне интерактивного интерпретатора, *выражением*. И тем самым озвучили первый термин из всех, что нам предстоит запомнить.

Выражение в терминологии программирования — это команда, выполняющая законченное действие. Таким действием может быть вычисление некоего значения (как в нашем случае), создание какой-либо структуры данных, команда, управляющая выполнением программного кода, вызов функции или метода (о них мы поговорим потом) или что-то иное.

Любое выражение в Python должно завершаться символами возврата каретки и перевода строки, которые вставляются в программный код нажатием клавиши <Enter>.

Давайте рассмотрим примеры еще нескольких выражений.

- `3 * 4 + 8`

  Умножаем 3 на 4, прибавляем к получившейся сумме 8 и получаем 20. Операция умножения выполняется перед операцией сложения, т. к. она имеет больший приоритет.

- `5 / 6`

  Делим 5 на 6 и получаем длинный результат — 0.8333333333333334.

- `(3 + 7) * (6 + 4)`

  Складываем 3 и 7, складываем 6 и 4 и перемножаем получившиеся суммы. На выходе вполне ожидаемо окажется 100. Отметим, что операции сложения будут выполнены перед операциями умножения, т. к. они заключены в скобки.

- `round(1.8)`

  Округляем 1,8, что на выходе даст 2.

Операторы. Порядок выполнения и приоритет операторов

В самом первом нашем выражении мы использовали символ `+`, чтобы указать Python выполнить операцию сложения двух чисел, находящихся левее и правее этого символа. В данном случае `+` — это *оператор*, команда, выполняющая элементарное действие над одним или двумя значениями-*операндами* и выдающая, или, как говорят программисты, возвращающая, результат.

Также мы познакомились с операторами умножения `*` и деления `/`. А не знакомый нам пока оператор вычитания обозначается дефисом (`-`).

Как видим, операторы арифметических действий (*арифметические операторы*) обозначаются так же, как в обычных математических формулах. Это сделано для простоты.

Операторы в Python имеют *приоритет*, задающий порядок их выполнения. Так, операторы умножения и деления имеют более высокий приоритет, нежели операторы сложения и вычитания, поэтому выполняются в первую очередь (что мы и наблюдали ранее).

Операторы с одинаковым приоритетом выполняются в порядке слева направо.

Для изменения порядка выполнения операторов применяются круглые скобки — оператор, который требуется выполнить в первую очередь, помещается в них вместе с операндами:

```
(3 + 7) * (6 + 4)
```

Это выражение, как мы помним, даст в качестве результата 20.

Но если мы уберем скобки:

```
3 + 7 * 6 + 4
```

то Python сначала умножит 7 на 6, после чего прибавит к полученному произведению 3 и 4. На выходе мы получим число 49 — совсем другой результат!

Функции

Помимо операторов, в выражениях активно используются *функции*, которые выполняют уже более сложные действия над значениями, чем простое сложение или умножение. Одна из них нам уже знакома — это функция `round`, выполняющая округление числа с плавающей точкой: `round(1.8)`.

Обратим внимание на две особенности функций Python.

- Во-первых, значения, которые функция будет обрабатывать, или ее *параметры*, записываются за ее именем и заключаются в круглые скобки. Если таких параметров несколько, они разделяются запятыми.

  ```
  somefunction(1, 2, 345)
  ```

  Здесь мы передали гипотетической функции `somefunction` сразу три параметра.

  В некоторых функциях параметры имеют имена (*именованные параметры*). Чтобы передать таким функциям данные, используется вот такая запись:

  ```
  otherfunction(arg1 = 10, arg2 = 20)
  ```

  Здесь мы передали функции `otherfunction` два параметра с именами `arg1` и `arg2`.

- Во-вторых, если операторы всегда возвращают один результат, то функции могут вернуть сразу несколько. Во многих случаях это может оказаться полезным.

> ***Создание собственных функций***
>
> Функция `round` встроена в сам язык Python. Однако программист может создавать свои функции — как это делается, будет рассмотрено позже.

Переменные

Рассмотренные нами выражения вычисляли нужное нам значение сразу же, за один проход. Однако так получается далеко не всегда. Во многих случаях, чтобы вычислить какое-либо значение, приходится задействовать несколько идущих подряд выражений. Или, как вариант, некое значение, вычисленное в одном выражении, может не выводиться на экран, а использоваться в другом выражении для вычисления другого значения.

И возникает потребность как-то сохранить это значение для будущих вычислений.

Для такого случая предназначены *переменные*, которые можно рассматривать как ячейки памяти, имеющие уникальные имена. В переменную можно записать любое значение, а впоследствии извлечь его и использовать в дальнейших вычислениях.

Переменная может хранить только одно значение. Если поместить в переменную другое значение, предыдущее будет утеряно.

Давайте создадим, или, как говорят профессиональные программисты, *объявим*, нашу первую переменную:

```
a = 10
```

Здесь мы поместили в переменную с именем `a` число 10.

Знак равенства =, стоящий между именем переменной и заносимым в нее значением, — это *оператор присваивания*. Он присваивает переменной, чье имя стоит слева, значение, стоящее справа.

Отметим, что при объявлении переменной никакого результата на экран не выводится, поскольку оператор присваивания не возвращает результата.

Теперь мы можем использовать только что объявленную переменную:

```
a + 5
```

и получим 15.

> ***Объявление переменной***
>
> Перед тем как использовать переменную, ее следует объявить. Если мы попытаемся обратиться к еще не объявленной переменной, то получим сообщение об ошибке.

Мы можем присвоить переменной результат выполнения выражения:

```
b = 2 + 3 / 4
```

или результат, возвращенный функцией:

```
c = somefunction(1, 2, 345)
```

Мы также можем объявить в одном выражении сразу несколько переменных:

```
a, b, c = 10, 2 + 3 / 4, somefunction(1, 2, 345)
```

Как видим, имена объявляемых переменных и присваиваемые им значения разделяются запятыми. Первая переменная получит первое значение, вторая переменная — второе значение и т. д.

Python также предоставляет нам возможность удалить не нужную более переменную. Это делает оператор `del`. Имена удаляемых переменных перечисляются за ним и разделяются запятыми.

```
del a, b
```

Здесь мы удаляем переменные `a` и `b`.

В программе может быть объявлено сколько угодно переменных. Следует помнить лишь две вещи: во-первых, имена переменных должны быть уникальными, а во-вторых, переменные расходуют оперативную память, поэтому лишних переменных объявлять не следует.

Типы данных и операции с ними

Следующий наш урок посвящен стандартным типам данных, поддерживаемым языком Python, и операциям, которые мы можем выполнять над ними. Таких типов четыре, причем один из них имеет три разновидности. Мы также рассмотрим особое значение `None`, которое может нам пригодиться в некоторых случаях.

Числа

Разумеется, любой язык программирования обязан поддерживать числа, или *числовой тип данных*. Python — не исключение.

Числа могут быть как целыми, так и с плавающей точкой. В последнем случае в качестве десятичного разделителя используется точка. Примеры задания чисел: `12`, `27`, `15000`, `2.3`, `3.3333333334`.

Если мы хотим записать целочисленное значение как число с плавающей точкой, то должны поставить в его конце десятичный разделитель (точку) и 0: `2.0`, `65.0`, `-3.0`. Если мы этого не сделаем, Python посчитает его целым числом.

Для записи чисел с плавающей точкой можно использовать экспоненциальную нотацию вида *<мантисса>*e*<порядок>*. Например: `1e3` (1000), `3e-4` (0,0003), `4.5e6` (4500000).

Для выполнения арифметических действий над числами мы можем пользоваться операторами, перечисленными в табл. 2.1. Многие из них нам уже знакомы.

***Таблица 2.1.** Арифметические операторы*

Оператор	Описание	Пример	Результат
`x + y`	Сложение	`10 + 5`	15
`x - y`	Вычитание	`3 - 8`	-5
`x * y`	Умножение	`2 * 4`	8
`x / y`	Обычное деление	`5 / 2`	2,5
`x // y`	Деление нацело	`5 // 2`	2
`x % y`	Остаток от деления	`5 % 2`	1
`-x`	Смена знака	`a = 3` `-a`	-3
`x ** y`	Возведение в степень	`5 ** 2`	25

Наконец, Python предоставляет пару полезных функций для обработки чисел. Они представлены в табл. 2.2.

***Таблица 2.2.** Функции для обработки чисел*

Функция	Описание	Пример	Результат
`abs(x)`	Получение абсолютного значения числа	`abs(-5)`	5
`round(x[, n])`	Округление числа. Количество знаков задается вторым параметром; если он не указан, число округляется до целого	`round(3.5)` `round(3.337, 2)`	4 3,34

Строки

В списке самых популярных типов данных строки оспаривают первое место с числами. Неудивительно, что Python предоставляет мощные средства для обработки *данных строкового типа*.

Запись строк

Строки заключаются либо в одинарные, либо в двойные кавычки.

```
'Python'
"Django 1.6.2"
```

Если строка заключена в одинарные кавычки, в ней не допускаются символы одинарных кавычек. А если строка заключена в двойные кавычки, в ней не допускаются символы двойных кавычек. Например, такие строки:

```
'Python '3.3.4''
"Django "1.6.2""
```

приведут к возникновению ошибки.

Если же нам все же потребуется вставить такой символ в строку, мы предварим его обратным слешем \:

```
'Python \'3.3.4\''
"Django \"1.6.2\""
```

Комбинация символов \n помещает в строковое значение символы возврата каретки и перевода строки:

```
'Python\nDjango'
```

В этом случае слово Django будет выведено в следующей строке.

Имеется также возможность задать строковое значение, которое представляет собой целый фрагмент текста, разбитый на строки. Такое значение предваряется комбинацией символов ''' или """, записывается, начиная со следующей строки, и завершается также комбинацией символов ''' или """, которая должна быть записана в следующей строке:

```
'''
Web-программирование:
  Python
  Django
'''
```

Это значение будет выведено на экран следующим образом:

```
Web-программирование:
  Python
  Django
```

Если записать подряд несколько строковых значений:

```
"Python" " 3.3.4"
```

они будут слиты в одну строку:

```
Python 3.3.4
```

Отметим, что таким образом можно объединить лишь сами строки, но никак не строковые значения, хранящиеся в переменных. Так, если мы запишем:

```
s1 = "Python"
s2 = " 3.3.4"
s1 s2
```

то получим сообщение об ошибке.

Кстати, процесс объединения нескольких строк в одну в программировании называется *конкатенацией*.

Обработка строк

Для работы со строками Python предоставляет довольно богатые стандартные средства (и еще более богатые инструменты, включенные в состав стандартной библиотеки).

Прежде всего, это оператор конкатенации +. Да, он записывается так же, как оператор арифметического сложения! И работает как с самими строками, так и со строковыми значениями, хранящимися в переменных:

```
"Python" + " 3.3.4"
s1 = "Python"
s2 = " 3.3.4"
s1 + s2
```

Строки можно умножать на целые числа. В этом случае результат будет представлять собой строку, повторенную соответствующее число раз:

```
s1 * 2
```

Результатом станет строка `PythonPython`.

Мы можем получить любой символ строки, просто указав его номер (*индекс*) в квадратных скобках после имени строковой переменной. Нужно только иметь в виду, что символы в строке нумеруются, начиная с нуля:

```
s1[0]
```

Это выражение выведет первый символ строки `s1` — букву `P`.

```
s1[2] + s1[5]
```

А это выражение выведет строку `tn`, составленную из третьего и шестого символов строки `s1`.

Если указать отрицательный индекс, символы будут отсчитываться не с начала строки, как обычно, а с ее конца. Запомним, что в этом случае символы начинают нумероваться с –1:

```
s1[-2]
```

Выполнив это выражение, мы получим второй с конца символ строки `s1` — букву `o`.

Мы можем выделить из строки фрагмент (*подстроку*), указав индексы ее начального и конечного символа в формате *<строка>*`[`*<начало>*`:`*<конец>*`]`. Здесь *начало* — это индекс начального символа выделяемой подстроки, а *конец* — индекс символа, ко-

торый следует за ее конечным символом. Другими словами, начальный символ включается в выделяемую подстроку, а конечный — нет.

Если не указан индекс начального символа, в подстроку включаются все символы с начала строки. Если же не указать конечный символ, в подстроку будут включены все оставшиеся символы строки:

```
s1[1:4]
```

Результатом будет подстрока, включающая второй, третий и четвертый символы строки `s1`, — `yth`.

```
s1[3:]
```

Здесь мы получим подстроку со всеми символами строки `s1`, начиная с четвертого, — `hon`.

```
s1[:3]
```

А здесь — подстроку со всеми символами строки `s1`, заканчивая третьим, — `Pyt`. (Не забываем, что конечный указанный символ не включается в подстроку.)

Теперь нужно уяснить один важный момент. Строки в языке Python неизменяемы. Это значит, что у нас не получится, скажем, заменить один из символов строки с помощью выражения вида:

```
s1[0] = "S"
```

Попытавшись его выполнить, мы получим ошибку.

Для работы со строками нам могут пригодиться четыре функции, представленные в табл. 2.3.

Таблица 2.3. Функции для обработки строк

Функция	Описание	Пример	Результат
`len(s)`	Получение длины строки	`len("Python")`	6
`int(s)`	Преобразование числа, представленного в виде строки, в целое число	`int("2")` `int("2.33")`	2 2
`float(s)`	Преобразование числа, представленного в виде строки, в число с плавающей точкой	`float("2")` `float("2.33")`	2,0 2,33
`str(x)`	Преобразование числа в строку	`str(2)` `str(2.33)`	"2" "2,33"

Списки

При разработке Web-сайтов с применением Python и Django мы часто будем манипулировать разнообразными списками. В самом деле, перечни категорий товаров, наборы записей в гостевой книге и статей в блоге — все это списки.

Python предлагает развитые инструменты для обработки *данных списочного типа*, или просто *списков*. Такие списки могут включать произвольное количество позиций (*элементов*) любого типа. (Хотя обычно список включает однотипные элементы.)

Обычные списки

Рассказ о списках мы начнем с *обычных списков* — они применяются в Python-программах чаще всего.

Создать список очень просто — достаточно заключить все входящие в него элементы в квадратные скобки, разделив их запятыми.

- `ln = [2, 54, 7.5, 890]`

  Мы только что создали список, включающий четыре числа.

- `ls = ["D", "j", "a", "n", "g", "o"]`

  А этот список включает все символы названия библиотеки, знакомство с которой мы начнем в следующей главе.

- `lc = ["c"]`

  Мы создали список из одного элемента.

- `le = []`

  А это пустой список, не содержащий ни одного элемента.

- `lq = [ln, ls, lc, le]`

  И, наконец, список, чьи элементы представляют собой созданные нами ранее списки, которые в данном случае будут называться *вложенными*.

Мы можем получить любой элемент списка тем же образом, что и любой символ строки:

```
ln[1]
```

Здесь мы получим число 54 — второй элемент списка `ln`.

Тем же способом мы можем получить список, включающий указанные нами элементы другого списка:

```
ls[1:3]
```

На выходе будет список ["j", "a"], включающий второй и третий элементы списка `ls`.

Списки Python, в отличие от строк, являются изменяемыми. Так что мы можем с легкостью изменить значение любого из его элементов:

```
ln[2] = 8.11
```

Меняем значение третьего элемента списка `ln`.

Также мы можем удалить любой элемент списка, воспользовавшись уже знакомым нам оператором `del`:

```
del ls[-1]
```

Удаляем последний элемент списка `ls`.

Мы можем складывать списки, пользуясь оператором +:

```
ls + [" ", "1", ".", "6", ".", "2"]
```

в результате чего получится список, содержащий все элементы из списков-операндов.

А уже знакомая нам по строкам функция `len` позволит узнать длину списка — количество его элементов.

Кортежи

Список с несколько неуклюжим названием *кортеж* (tuple) отличается от обычного списка тем, что является неизменяемым. Создав его, мы не сможем в будущем не изменить значения его элементов, ни удалить их.

Кортежи создаются так же, как обычные списки, за исключением того, что вместо квадратных скобок здесь ставятся круглые:

```
tn = (2, 54, 7.5, 890)
```

Создаем кортеж из четырех элементов.

Впрочем, скобки можно и не ставить:

```
tn = 2, 54, 7.5, 890
```

Кортеж из одного элемента создается следующим образом:

```
ts = (3,)
```

То есть после значения единственного его элемента ставится запятая — так мы сообщим Python, что хотим создать именно кортеж.

А пустой, не содержащий элементов кортеж создается с помощью пустых скобок:

```
te = ()
```

Правда, не совсем понятно, зачем он нужен...

Важным достоинством кортежа является то, что он может быть присвоен сразу нескольким переменным, которые получат значения, хранящиеся в его элементах.

```
n1, n2, n3, n4 = tn
```

В этом случае переменная `n1` получит значение первого элемента кортежа `tn`, переменная `n2` — значение его второго элемента и т. д.

Операция присваивания кортежа нескольким переменным называется *распаковкой*.

В разделе, посвященном переменным, мы говорили, что можем в одном выражении объявить сразу несколько переменных, записав через запятую имена переменных и присваиваемые им значения. Фактически при этом мы создавали кортеж, который сразу же распаковывали.

Словари

Словарь — это особый список, элементы которого имеют заданные нами индексы. В качестве их индексов могут выступать числа, строки и кортежи, включающие

числа и строки; однако на практике для этого используются строки. Кстати, такие заданные программистом индексы называются *ключами* (не путать с ключевыми индексами, описанными в *главе 1*).

Словарь создается почти так же, как обычный список, за двумя исключениями. Во-первых, список элементов берется не в квадратные, а в фигурные скобки. Во-вторых, сами элементы записываются в формате *<индекс>:<значение>*:

```
d1 = {"name":"Python", "version":"3.3.4", "category":1}
d2 = {"name":"Django", "version":"1.6.2", "category":2}
```

Мы создали два словаря, каждый из которых имеет по три элемента со строковыми ключами `"name"`, `"version"` и `"category"`.

Пустой словарь создается указанием «пустых» фигурных скобок:

```
de = {}
```

Мы можем получить доступ к любому элементу словаря по его ключу:

```
v = d2["version"]
```

и изменить его значение:

```
d2["version"] = "1.6.3"
```

что говорит о том, что словари, как и обычные списки, являются изменяемыми.

Обычно словари применяются для хранения данных, относящихся к какой-то одной сущности. В нашем примере мы так и поступили, описав в каждом словаре сведения об используемых нами платформах разработки Web-приложений.

Присваивание списков. Ссылки

А теперь рассмотрим один крайне важный вопрос. Он касается присваивания списков.

Давайте создадим вот такой простой список:

```
l1 = [1, 2, 3]
```

и присвоим его другой переменной:

```
l2 = l1
```

Теперь сменим значение первого элемента списка на другое:

```
l1[0] = 6
```

И проверим, что хранится во втором списке, набрав в интерактивном интерпретаторе имя хранящей его переменной `l2`.

Что же мы увидим? А то, что значение первого элемента второго списка также изменилось на 6! Выходит, что в переменных `l1` и `l2` хранится один и тот же список!

Так и есть! Как только мы создаем список и присваиваем его переменной, последняя получает в качестве значения не сам этот список, а *ссылку* на него, которую можно считать указателем на фрагмент оперативной памяти, где он содержится.

А когда мы присваиваем значение переменной, где хранится список, другой переменной, туда копируется эта ссылка. В результате обе переменные фактически хранят один и тот же список.

Кстати, то же самое касается чисел, строк и логических величин (о них — чуть позже). При их присваивании другой переменной в нее копируется не сама величина, а лишь ссылка на нее.

В языке Python все типы данных, даже самые простые, наподобие чисел и строк, являются ссылочными.

Логические величины

Логическая величина — это признак вида «да-нет», «включено-выключено» или «истина-ложь». Обычно *данные логического типа* используются в особых выражениях, управляющих порядком выполнения выражений (мы рассмотрим их позже).

Запись логических величин

Логическая величина записывается с помощью *ключевых слов* (особых слов языка Python, применяемых для специальных целей) `True` и `False`. Первое слово означает «истина», второе — «ложь».

```
l1 = True
l2 = False
```

Но, как правило, логические величины являются результатом вычисления особых выражений, которые называются *логическими*. В них применяются операторы двух типов, которые мы сейчас рассмотрим.

Операторы сравнения

Операторы сравнения, как следует из их названия, сравнивают между собой два операнда. Каждый тип данных поддерживает свой набор операторов сравнения.

Операторы сравнения чисел перечислены в табл. 2.4.

Таблица 2.4. *Операторы сравнения чисел*

Оператор	Описание	Пример	Результат
`x == y`	Равно	`2 == 2`	True
		`3.5 == 6.8`	False
`x != y`	Не равно	`2 != 2`	False
		`3.5 != 6.8`	True
`x < y`	Меньше	`1 < 2`	True
		`34 < 27`	False
`x > y`	Больше	`1 > 2`	False
		`34 > 27`	True

Таблица 2.4 (окончание)

Оператор	Описание	Пример	Результат
x <= y	Меньше или равно	1 <= 2	True
		34 <= 27	False
		155 <= 155	True
x >= y	Больше или равно	1 >= 2	False
		34 >= 27	True
		155 >= 155	True

Вот примеры:

```
n1 = 30
n2 = 40
l1 = n1 + 2 > n2
l2 = n1 != n2
```

После выполнения этого кода в переменной `l1` окажется значение `False`, а в переменной `l2` — значение `True`.

Отметим, что операторы сравнения имеют меньший приоритет, чем арифметические, поэтому выполняются после них.

Для сравнения строк служат операторы == и !=.

```
s1 = "Python"
s2 = "Django"
l = s1 != s2
```

После выполнения этого кода в переменной `l` окажется значение `True`.

Списки поддерживают два оператора сравнения, указанные в табл. 2.5.

Таблица 2.5. Операторы сравнения списков

Оператор	Описание	Пример	Результат
x in l	Входит ли значение x в состав списка l	2 in [2, 3, 4]	True
		1 in [2, 3, 4]	False
x not in l	Отсутствует ли значение x в списке l	2 not in [2, 3, 4]	False
		1 not in [2, 3, 4]	True

Помимо этого, списки поддерживают все операторы сравнения, перечисленные в табл. 2.4. В этом случае Python выполняет сравнение элементов списков последовательно. Если списки имеют разное количество элементов, то сравнению подвергаются лишь те элементы большего списка, что имеются в меньшем списке; остальные элементы большего списка в расчет не принимаются:

```
d1 = [2, 3, 4]
d2 = [1, 2]
l1 = d1 == d2
```

В переменной `l1` окажется `False`, т. к. оба элемента списка `d1` не равны первым двум элементам списка `d2`.

```
l2 = d1 < d2
```

А переменной `l2` будет присвоено значение `True`, т. к. оба элемента списка `d1` меньше первых двух элементов списка `d2`.

Кстати, если условное выражение возвращает `True`, программисты говорят, что оно выполняется, в противном случае — не выполняется.

Логические операторы

Часто бывает необходимо задействовать в логическом выражении не одно сравнение, а сразу несколько, причем по определенным правилам. Скажем, может возникнуть необходимость указать, что два условия должны выполниться обязательно или, наоборот, должно выполниться лишь одно из заданных двух условий.

Для таких случаев Python предусматривает так называемые *логические операторы*. Всего их три, и увидеть их можно в табл. 2.6.

***Таблица 2.6.** Логические операторы*

Оператор	Описание	Пример	Результат
`l1 and l2`	Возвращает `True`, если значения `l1` и `l2` равны `True`, и `False` в противном случае	`2 < 3 and 4 >= 1`	True
		`2 < 3 and 4 <= 1`	False
`l1 or l2`	Возвращает `True`, если значение `l1` или `l2` равно `True`, и `False` в противном случае	`2 > 3 or 4 >= 1`	True
		`2 > 3 or 4 <= 1`	False
`not l`	Возвращает `True`, если значение `l` равно `False`, и наоборот	`not 4 >= 1`	False
		`not 4 <= 1`	True

Вот примеры:

```
n1 = 30
n2 = 40
l1 = n1 + 2 > 10 and n2 / 2 <= 80
```

Здесь переменная `l1` получит значение `True`.

```
l2 = n1 - n2 > 0 or not n1 / n2 != 1
```

А здесь переменная `l2` получит значение `False`.

Значение *None*

Еще Python поддерживает особое значение `None`. Оно обозначает отсутствие данных любого типа.

```
n = None
```

Мы будем пользоваться этим значением нечасто и лишь в специальных случаях.

Преобразования типов

Мы почти закончили со стандартными типами данных. Осталось лишь рассмотреть один важный вопрос.

Если мы сложим два целых числа:

```
2 + 2
```

то получим целое число. А если сложить целое число и число с плавающей точкой? А если нам захочется сложить строку и число? Что произойдет в этом случае?

В этом случае Python выполнит *преобразование типов данных*. Оно подчиняется законам, которые мы сейчас рассмотрим.

При выполнении арифметических операций с целыми числами и числами с плавающей точкой все целые числа преобразуются в числа с плавающей точкой:

```
1 + 3.5 + 2.5
```

Получится число с плавающей точкой `7.0`.

В условных выражениях (о них мы скоро поговорим) выполняется преобразование чисел, строк и списков в логические величины. В значение `False` преобразуются:

- математический ноль;
- пустая строка;
- пустой список любой разновидности;
- значение `None`.

Все прочие значения преобразуются в `True`:

```
s = "Django"
if 2 < 3 and s:
  n = 1
else:
  n = 2
```

В переменной `n` окажется значение 1.

> ***Преобразование логических величин***
>
> Преобразование логических величин выполняется только в условных выражениях.

Во всех остальных случаях преобразование типов не выполняется. Так что мы не сможем просто так, например, объединить строку и число — нам придется явно преобразовать последнее в строковый тип:

```
"Python " + str(3)
```

Управление выполнением кода. Управляющие выражения

Очень часто в программировании бывает необходимо выполнить или не выполнить какие-либо выражения при выполнении или невыполнении определенного условия или выполнить некий код несколько раз подряд. Для таких случаев Python предусматривает несколько разновидностей *управляющих выражений*, о которых сейчас и пойдет речь.

Блоки

Но сначала нужно сказать пару слов о *блоках*. Такое название носит фрагмент кода, входящий в состав управляющих выражений (а также объявлений функций и классов, но об этом потом).

Выражения, входящие в блок, выделяются отступом слева. Этот отступ создается при помощи пробелов и может иметь любую длину — важно лишь помнить, что все выражения блока должны иметь одинаковый отступ. Однако на практике длина отступа чаще всего равна двум или четырем пробелам:

```
s1 = "Python"
s2 = "Django"
  n1 = 30
  n2 = 40
  l1 = n1 + 2 > 10 and n2 / 2 <= 80
l = s1 != s2
```

Здесь мы создали блок из трех выражений, выделив их отступом.

Условные выражения

Условные выражения позволяют выполнить или не выполнить блок при выполнении или невыполнении какого-либо условия. Это условие задается в виде логической величины.

Формат логического выражения таков:

```
if <условие 1>:
  <блок 1>
[elif <условие 2>:
  <блок 2>]
. . .
[else:
  <блок else>]
```

Сначала проверяется значение *условие 1*, расположенное после ключевого слова `if` (в секции `if`). Если оно возвращает `True` (выполняется), выполняется *блок 1*, после чего выполнение условного выражения прекращается, и Python начинает обработку кода, следующего за ним.

Если же значение *условие 1* равно `False` (не выполняется), проверяется *условие 2*, стоящее после ключевого слова `elif` (в секции `elif`). В случае равенства его `True` обрабатывается *блок 2*, и выполнение условного выражения прекращается. Отметим, что секция `elif` может отсутствовать.

Если же *условие 2* тоже оказалось равным `False`, обрабатываются остальные секции `elif`, если они есть. Происходит это так же, как было описано ранее.

Может случиться так, что ни одно условие из присутствующих в секциях `if` и `elif` не будет выполнено. Тогда выполняется *блок else*, присутствующий в секции `else`, если, конечно, он есть. После этого обработка условного выражения заканчивается.

Лучше один раз увидеть, чем семь раз услышать. Поэтому рассмотрим несколько примеров условных выражений, чтобы понять их работу.

- Пример 1:

```
if s == "Python":
  c = 1
elif s == "Django":
  c = 2
elif s == "Apache":
  c = 3
else:
  c = 0
```

  Это выражение будет присваивать переменной `c` числовое значение в зависимости от значения переменной `s`. Так, если `s` содержит строку `"Python"`, `c` получит значение 1, а если `"Django"` — то значение 2. Если же `s` содержит значение, не совпадающее ни с одним из сравниваемых, `c` присваивается 0.

- Пример 2:

```
if s == "Python":
  c = 1
else:
  c = 0
```

  Здесь мы присваиваем переменной `c` значение 1, только если `s` содержит строку `"Python"`, во всех остальных случаях `c` получит значение 0.

- Пример 3:

```
if s == "Python":
  c = 1
```

  Самый «куцый» вариант условного выражения, содержащего лишь секцию `if`. Здесь, если `s` содержит строку `"Python"`, `c` присваивается 1, в остальных случаях значение этой переменной не изменяется.

Циклы

Циклы позволяют выполнить блок кода несколько раз — либо пока не перестанет выполняться определенное условие, либо пока в списке не кончатся элементы.

Цикл с условием

Цикл с условием будет раз за разом выполнять блок, пока заданное нами условие возвращает `True`. Как только оно вернет `False`, начинает выполняться код, следующий за циклом.

Формат цикла с условием очень прост:

```
while <условие>:
  <блок>
```

Поскольку здесь применяется ключевое слово `while`, циклы такого рода получили название циклов `while`. А выполняемый блок часто называют *телом цикла*.

```
l = []
i = 2
while i <= 1024:
  l = l + [i]
  i = i * 2
```

Этот цикл создает список, чьими элементами станут степени числа 2: [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024].

Цикл по списку

Цикл по списку выполняет блок столько раз, сколько элементов включает в свой состав указанный в нем список. Формат его записи таков:

```
for <переменная элемента списка> in <список>:
  <блок>
```

В *переменную элемента списка* каждый раз будет помещаться значение очередного элемента *списка*. Эта переменная будет доступна только внутри тела цикла.

Цикл по списку имеет неофициальное название цикла `for-in`, т. к. он создается с применением ключевых слов `for` и `in`.

```
ls = [2, 8, 32, 128]
ld = []
for el in ls:
  ld = ld + [2 ** el]
```

Этот цикл создает список `ld`, элементы которого будут представлять собой число 2, возведенное в степени, чьи значения будут извлекаться из списка `ls`: [4, 256, 4294967296, 340282366920938463463374607431768211456].

Дополнительные возможности циклов

Иногда бывает необходимо прервать исполнение цикла досрочно. Для этого применяется *оператор прерывания цикла* `break`. Встретив его, Python тотчас перестает выполнять цикл и начинает обрабатывать код, который следует за ним.

```
ls = [2, 8, 32, 128]
ld = []
```

```
for el in ls:
  if el > 100:
    break
  ld = ld + [2 ** el]
```

В этом случае список `ld` будет содержать только элементы 4, 256 и 4294967296.

Как видим, оператор `break` не принимает операндов и сам по себе представляет собой полноценное выражение.

Похожий на него *оператор перезапуска цикла* `continue` указывает Python пропустить все выражения тела цикла, что следуют за ним, и начать следующий проход цикла.

```
ls = [2, 8, 32, 128]
ld = []
for el in ls:
  if el < 100:
    continue
  ld = ld + [2 ** el]
```

А в этом случае список `ld` будет содержать лишь одно значение — 340282366920938463463374607431768211456.

Функции

Собственно, с функциями мы уже знакомы. Мы вкратце рассмотрели их, когда разбирали основные принципы Python-программирования. Тогда мы использовали функции, встроенные в сам этот язык, и обмолвились, что программист сам может объявить свои собственные функции. Настала пора узнать, как это делается.

Объявление функции

Для описания функции применяется ключевое слово `def`. А формат самого объявления такой:

```
def <имя функции>(<список параметров функции>):
  <блок — тело функции>
```

Имя функции должно быть уникальным. Параметры в списке разделяются запятыми, если же функция не принимает параметров, после ее имени ставятся пустые круглые скобки.

Чтобы вернуть из функции вычисленный в ней результат, используется *оператор возврата значения* `return`. Возвращаемое в качестве результата значение ставится после этого оператора:

```
def pov2(n):
  return 2 ** n
```

Мы только что объявили нашу первую функцию. Она принимает один параметр, возводит число 2 в степень, равную значению этого параметра, и возвращает полученный результат.

Вызовем ее:

```
a = pov2(3)
```

и получим в переменной a значение 8.

> ***Объявление функции***
>
> Функция должна быть объявлена перед первым ее вызовом. Если мы попытаемся обратиться к еще не объявленной функции, получим сообщение об ошибке.

```
def somefunc(n1, n2):
  return n1 * 2, n2 / 2
```

А эта функция принимает сразу два параметра и возвращает в качестве результата кортеж из двух значений. Мы можем впоследствии распаковать его, присвоив двум переменным:

```
a, b = somefunc(4, 9)
```

Локальные переменные

Как и в обычном программном коде, мы можем объявлять в теле функции переменные. Как и в обычном программном коде, эти переменные могут содержать значения любого типа, и число их не ограничено.

Однако переменные, объявленные в теле функции (их, кстати, называют *локальными*), имеют некоторые отличия:

- локальные переменные существуют и доступны только внутри тела функции. Программный код, находящийся вне функции, получить к ним доступа не может;
- мы можем в теле функции получить доступ к переменным, объявленным «снаружи» (*глобальным переменным*), но только в том случае, если ранее не объявили локальную переменную с таким же именем, как у глобальной;
- если мы объявим локальную переменную, чье имя совпадает с именем глобальной переменной, то локальная переменная подменит, или *скроет*, собой глобальную.

```
CONSTANT = 2
def somefunc(n1, n2):
  a = n1 * CONSTANT
  b = n2 / CONSTANT
  return a, b
```

Здесь мы объявили глобальную переменную CONSTANT и функцию somefunc с двумя локальными переменными — a и b. Отметим, что в теле функции мы также можем получить доступ к глобальной переменной...

```
def otherfunc(n1, n2):
  CONSTANT = 3
  a = n1 * CONSTANT
```

```
  b = n2 / CONSTANT
  return a, b
```

...и даже записать в нее новое значение.

Значения параметров по умолчанию. Именованные параметры

Мы можем указать для какого-либо параметра функции значение по умолчанию. Это значение параметр получит, если он не был указан в вызове данной функции.

Значение параметра по умолчанию указывается после его имени через знак =.

```
def somefunc(n1, n2 = 3):
  return n1 * 2, n2 / 2
```

Как видим, здесь применяется синтаксис, аналогичный объявлению переменной.

Теперь, если мы вызовем нашу функцию, не указав значение второго параметра:

```
somefunc(4)
```

он получит заданное нами значение по умолчанию.

> ***Размещение параметров, имеющих значения по умолчанию***
>
> Параметры, имеющие значения по умолчанию, должны в объявлении функции располагаться после параметров, таких значений не имеющих. Нарушение этого правила приведет к возникновению ошибки.
>
> Например, такое объявление функции:
>
> ```
> def somefunc(n2 = 3, n1):
>   return n1 * 2, n2 / 2
> ```
>
> будет ошибочным.

Параметры, для которых указаны значения по умолчанию, автоматически станут *именованными*. И мы сможем обращаться к ним по именам.

```
def otherfunc (n, arg1 = 2, arg2 = 89, arg3 = "!!!"):
  . . .
otherfunc(6, arg2 = 487)
```

Именованные параметры — исключительно удобная вещь, особенно в функциях с большим числом параметров. Нам больше не нужно запоминать порядок следования параметров — достаточно запомнить их имена и значения по умолчанию.

Функции с произвольным количеством параметров. Необязательные параметры

И, наконец, Python позволяет нам объявлять функции, принимающие произвольное количество параметров, — как обычных, так и именованных.

Указать, что функция может принимать произвольное число обычных, неименованных параметров, можно, задав в ее объявлении единственный параметр вида

*<имя>:

```
def somefunc(*args):
  . . .
```

В теле функции будет создана локальная переменная с именем, которые мы указали. Значением этой переменной станет кортеж, включающий значения всех параметров, переданных данной функции:

```
def somefunc(*args):
  a = args[0]
  b = args[1]
  . . .
```

Теперь при вызове данной функции мы можем передать ей сколько угодно параметров:

```
somefunc(2, 7, 90, 45.7, "Python")
```

Мы можем объявить функцию, которая будет принимать некоторое количество обязательных параметров и произвольное количество *необязательных*. Например:

```
def somefunc(n1, n2, *args):
  . . .
```

Эта функция примет два обязательных параметра: первый и второй по счету, остальные параметры, которых может быть сколько угодно, станут необязательными.

Если нам нужно объявить функцию, принимающую произвольное количество необязательных именованных параметров, мы укажем в ее объявлении параметр вида `**`*<имя>*. Тогда в ее теле будет создана локальная переменная с указанным нами именем, значением которой станет словарь. Ключами элементов этого словаря окажутся имена параметров, переданных при вызове функции, а самими элементами — их значения:

```
def somefunc(**kwargs):
  keys = kwargs.keys()
  a = kwargs[keys[0]]
  b = kwargs[keys[1]]
  . . .
```

В теле этой функции мы получаем список всех ключей полученного словаря и, пользуясь им, извлекаем из словаря все элементы. (О методе `keys` класса `dict` и самом этом классе мы поговорим потом.)

- `somefunc(arg1 = 2, arg2 = 7, arg3 = "Python")`

  Вызываем только что объявленную функцию, передав ей три именованных параметра.

- `def otherfunc(n1, n2, **kwargs):`

  ```
    . . .
  ```

  Эта функция примет два обязательных неименованных параметра и произвольное количество именованных.

```
otherfunc(2.6, "3.3.4", arg1 = 79, arg2 = "Python")
```

Вызываем ее.

- `def otherfunc(n, *args, **kwargs):`

  . . .

  А эта функция примет один обязательный именованный параметр, произвольное количество необязательных неименованных и произвольное же количество необязательных именованных.

> ***Расположение обозначений *<имя> и **<имя>***
>
> Обозначения **<имя>* и ***<имя>*, указывающие списки необязательных параметров, должны стоять в объявлении функции в самом конце списка параметров, причем обозначение ***<имя>* должно находиться после **<имя>*. Нарушение этого правила приведет к возникновению ошибки.

Как видим, Python позволяет нам писать очень сложные функции. Если мы решим заняться созданием дополнительных библиотек, нам это должно пригодиться.

Классы и объекты

Обычно программы хранят данные, относящиеся к какой-либо одной сущности, в разных переменных. Например, мы будем хранить в разных переменных наименование товара, его категорию, описание, цену, признак того, есть ли товар в наличии, и всевозможные дополнительные данные (имя файла с изображением товара, процент скидки на него и пр.).

Соответственно, для манипулирования данными, которые относятся к одной сущности, в программах применяются и разные функции. Так, у нас, скорее всего, будет функция, рассчитывающая новую цену товара с учетом скидки, функция, выдающая сокращенное описание товара, и др.

Вследствие этого возникает желание свести и данные, и обрабатывающие их функции в некую единую сущность. Но как это сделать? Конечно, данные мы можем свести воедино, оформив их в виде словаря (см. ранее), но как добавить в него функции?

Решение есть. Мы можем использовать объекты.

Основные понятия и приемы работы

Объект — это сложная сущность, хранящая в себе как данные, так и программный код, который манипулирует ими. Данные представляют собой значения, хранящиеся в *свойствах* — своего рода переменных, являющихся «собственностью» объекта. А обрабатывающий эти значения программный код организован в виде *методов*, которые можно считать принадлежащими объекту функциями.

Как обычные значения принадлежат к определенному типу, так и объект должен принадлежать определенному классу. *Класс* описывает тип объекта — набор его свойств и методов.

Например, в составе стандартной библиотеки Python поставляется класс `Fraction`. Он позволяет хранить десятичные дроби и производить над ними всевозможные действия: сокращение, сложение, вычитание и пр.

```
from fractions import Fraction
```

Здесь мы импортируем класс `Fraction` из модуля `fractions`. (О модулях Python и импорте из них типов мы поговорим в конце этой главы.)

Чтобы создать объект какого-либо класса, мы запишем имя этого класса, поставим после него круглые скобки, в которых поместим параметры создаваемого объекта. Получится нечто похожее на вызов функции. Результат этой функции — готовый объект — мы присвоим переменной или используем в дальнейших вычислениях.

Параметрами создаваемого объекта класса `Fraction` будут числитель и знаменатель дроби:

```
f = Fraction(8, 6)
```

В переменной `f` окажется объект класса `Fraction` — дробь 8/6.

Для доступа к свойствам и методам объекта мы запишем имя переменной, где он хранится, поставим точку и укажем имя нужного свойства или метода.

Например, класс `Fraction` поддерживает свойства `numerator` и `denominator`, хранящие, соответственно, числитель и знаменатель дроби. Чтобы получить их значения, мы используем такие выражения:

```
a = f.numerator
b = f.denominator
```

В переменной `a` окажется значение числителя дроби `f`, а в переменной `b` — значение ее знаменателя. Они будут равны 4 и 3 соответственно — как видим, сам класс `Fraction` уже сократил нашу дробь.

Еще данный класс поддерживает метод `limit_denominator`. Он принимает в качестве единственного параметра знаменатель и возвращает новый объект того же класса, хранящий ближайшую к изначальной дробь с указанным знаменателем:

```
f2 = f.limit_denominator(2)
```

В переменной `f2` окажется объект класса `Fraction` — дробь 3/2.

Свойства `numerator` и `denominator` хранят данные объекта, а метод `limit_denominator` ими манипулирует. Вполне ожидаемо, что они вызываются у объектов. И поэтому носят название *свойств* и *методов объекта*.

Но существует и другая разновидность методов, которые вызываются не у объекта, а непосредственно у класса, — *методы класса*. (Свойства класса Python не поддерживает.) Обычно они создают другие объекты этого класса на основе переданных им параметров.

Класс `Fraction` поддерживает метод класса `from_float`. Он принимает в качестве единственного параметра число с плавающей точкой и возвращает новый объект данного класса, хранящий созданную на основе переданного числа дробь:

```
f3 = Fraction.from_float(3.5)
```

В переменной `f3` окажется объект — дробь 7/2.

Осталось лишь сказать, что объекты — это ссылочные типы, т. е. при присваивании объекта другой переменной в последнюю помещается ссылка на него:

```
f4 = f3
```

В результате переменные `f3` и `f4` будут указывать на один и тот же объект.

Объявление классов

Как и в случае функций, разработчик может объявлять свои собственные классы. Для этого используется следующий формат:

```
class <имя класса>:
  <блок объявлений свойств и методов>
```

Имя класса должно быть уникальным. Свойства объявляются так же, как и переменные, а методы — так же, как функции.

> ***Объявление класса***
>
> Класс должен быть объявлен перед созданием первого его объекта. Если мы попытаемся создать объект еще не объявленного класса, получим сообщение об ошибке.

Отметим, что каждый метод должен принимать, по меньшей мере, один параметр, который в списке должен стоять первым. Обычно он имеет имя `self`. Этот параметр получит в качестве значения ссылку на сам этот объект — мы используем его, чтобы получить доступ к свойствам и методам данного объекта. Значение в данный параметр подставляется самим Python при вызове метода, и нам самим этого делать не нужно.

```
class Square:
  width = 0
  height = 0
  def area(self):
    return self.width * self.height
```

Здесь мы объявили класс `Square`, хранящий сведения о прямоугольнике. Он поддерживает свойства `width` и `height`, где хранятся ширина и длина прямоугольника, и метод `area`, возвращающий его площадь.

```
sq = Square()
sq.width = 100
sq.height = 40
ar = sq.area()
```

Создаем объект только что объявленного класса, задаем значения размеров прямоугольника и получаем его площадь.

Объявить метод класса можно, предварив объявление самого метода декоратором `@staticmethod`. (*Декораторами* в Python называются команды, выполняющие особые действия над функциями или методами.) Этот декоратор должен быть указан в отдельной строке, находящейся перед строкой с объявлением метода. А сам

метод не должен принимать в качестве параметра ссылку на сам объект (поскольку этого объекта нет):

```
class Square:
  . . .
  @staticmethod
  def get_area(width, height):
    return width * height
```

Мы объявили в классе `Square` метод класса, вычисляющий площадь прямоугольника на основе переданных ему значений ширины и высоты.

Ранее, экспериментируя с классом `Fraction`, мы передавали параметры создаваемых дробей прямо при создании объектов. С нашим же классом в его изначальном виде такой номер не пройдет.

Чтобы получить возможность указывать значения свойств объекта прямо при его создании, мы должны объявить в классе особый метод — *конструктор*. Он должен иметь имя `__init__`:

```
class Square:
  . . .
  def __init__(self, width, height):
    self.width = width
    self.height = height
```

Наш конструктор принимает три параметра: обязательный для всех методов первый параметр, который получит в качестве значения ссылку на сам этот объект, ширину и высоту прямоугольника.

```
sq = Square(100, 40)
ar = sq.area()
```

Так гораздо удобнее — не нужно писать код для занесения значений в свойства.

Наследование классов

Одной из замечательных особенностей классов является *наследование*. Мы можем создать класс на основе другого класса, и второй класс получит в «наследство» все свойства и методы первого.

Класс, созданный на основе другого класса, получит название *потомка*, или *дочернего*. Соответственно, класс, на основе которого создается другой класс, называется *родителем*, или *родительским*.

Для создания класса-потомка применяется такой формат:

```
class <имя класса>(<имя класса-родителя>):
  <блок объявлений свойств и методов>
```

Если мы объявим в классе-потомке свойства и методы, чьи имена совпадают с именами таковых у класса-родителя, то свойства и методы потомка перекроют своих родительских «тезок». И если со свойствами такие штуки проделывать не рекомен-

дуется, то в случае методов эту особенность можно использовать для расширения унаследованной потомком функциональности.

Мы можем объявить в потомке метод с тем же именем, что у аналогичного метода родителя, но выполняющий дополнительные действия и, возможно, принимающий дополнительные параметры. В нужном месте этого метода мы вызовем одноименный метод родителя, использовав синтаксис *<имя класса-родителя>.<имя метода>(<список параметров>)*. Это избавит нас от необходимости копировать код «старого» метода в «новый»:

```
class Cube(Square):
  z = 0
  def __init__(self, width, height, z):
    Square.__init__(self, width, height)
    self.z = z
  def volume(self):
    return self.area(self) * self.z
```

Здесь мы объявили класс `Cube`, хранящий сведения о кубе, как потомок объявленного ранее класса `Square`. Мы добавили в него свойство `z`, хранящее высоту куба, и изменили конструктор, добавив в него выражение для сохранения переданного в качестве параметра значения высоты. Обратим внимание, как мы вызвали конструктор класса-родителя, который выполнит за нас всю работу по сохранению значений ширины и высоты. А еще мы добавили метод `volume`, который вернет значение объема куба.

```
c = Cube(100, 40, 5)
a = c.area()
v = c.volume()
```

В переменной `a` окажется число 4000, а в переменной `v` — число 20000.

Может случиться так, что вызываемый метод объявлен не непосредственно в родителе, а унаследован им от одного из его родителей, и мы не знаем, от кого именно. Тогда для вызова такого метода применяется синтаксис вида:

```
super(<имя текущего класса>, self).<имя метода>([<параметры>])
```

Отметим, что в *параметрах* в этом случае не указывается `self`.

Например, для нашего класса `Cube` вызов конструктора родителя будет выглядеть так:

```
super(Cube, self).__init__(width, height)
```

Осталось лишь сказать, что класс может иметь несколько родителей (так называемое *множественное наследование*). В этом случае их имена перечисляются через запятую:

```
class Child(Parent1, Parent2, Parent3):
  . . .
```

Стандартные типы Python как объекты

На закуску — самое интересное. Оказывается, стандартные типы данных Python: числа, строки, списки и логические величины — суть объекты соответствующих им классов!

Так, все целые числа являются объектами класса `int`, а все числа с плавающей точкой — объектами класса `float`. Класс `float` поддерживает полезный метод `is_integer`, который возвращает `True`, если данное число фактически является целым, и `False` в противном случае; параметров он не принимает.

```
n = 4.5
n.is_integer()
```

Эти два выражения дадут в качестве результата `False`, т. к. число 4,5 не целое.

```
2.0.is_integer()
```

А здесь мы вызвали данный метод не у переменной, а у самого числа. И вполне ожидаемо получили `True`, т. к. значение `2.0` хоть и обозначено как число с плавающей точкой, но на самом деле является целым.

Строки являются объектами класса `str`. Этот класс поддерживает огромное количество методов. Так, не принимающий параметров метод `capitalize` возвращает копию текущей строки с прописной первой буквой и строчными остальными.

Обычные списки являются объектами класса `list`. Он поддерживает, в частности, не принимающий параметров и не возвращающий результатов метод `clear`, который удаляет все элементы списка, аналогичный метод `sort`, сортирующий элементы в списке, и метод `count`, принимающий в качестве единственного параметра какое-либо значение и возвращающий количество элементов списка, чье значение равно указанному нами:

```
l = [1, 2, 3]
n = 4
if l.count(n) == 0:
  l = l + [n]
```

Кортежи являются объектами класса `tuple`, а словари — объектами класса `dict`. Последний класс поддерживает методы `keys` и `values` — они не принимают параметров и возвращают список всех ключей и значений словаря соответственно.

Логические величины являются объектами класса `bool`.

Обработка ошибок. Исключения

Как мы ни стараемся избежать возникновения в программах ошибок, они все равно будут возникать. Да, многие ошибки — например, неверно набранное ключевое слово, можно выявить сразу же. Однако другие, такие как деление на ноль, могут возникнуть в любой момент, и предотвратить их получается далеко не всегда.

Но ведь мы используем Python, один из популярнейших языков программирования! А он не стал бы таким популярным, не будь в нем мощных средств обработки ошибок.

Начать следует с того, что в случае возникновения ошибки Python создает особый объект, называемый *исключением*. По классу этого объекта мы можем определить, что случилось, и принять соответствующие меры.

Но как это сделать? С помощью особого выражения, называемого *блоком обработки исключений*. Формат его записи таков:

```
try:
  <блок try>
except <класс исключения 1>:
  <блок except 1>
[except <класс исключения 2>:
  <блок except 2>]
. . .
[else:
  <блок else>]
```

Поскольку для создания такого выражения применяются ключевые слова `try` и `except`, его еще называют блоком `try-except`.

Код, в котором может возникнуть ошибка, оформляется в виде *блока* `try`. Код, который должен сработать в случае возникновения исключения определенного класса, помещается в *блоке* `except`, где после ключевого слова `except` указывается имя нужного *класса исключения*. Таких блоков может быть сколько угодно. А код, который должен выполниться в случае отсутствия ошибок, оформляется как *блок* `else`.

Как только в *блоке* `try` возникает ошибка, Python, как мы уже говорили ранее, генерирует исключение. И сразу же начинает просматривать имена *классов исключений*, указанных после ключевых слов `except`. Найдя нужный класс, он выполняет указанный за ним *блок* `except`, после чего начинает обрабатывать код, следующий за блоком обработки исключений.

Если же в *блоке* `try` не возникнет ошибок, будет выполнен *блок* `else`, и, опять же, начнет выполняться следующий за блоком обработки исключений код.

Список некоторых классов исключений, с которыми мы столкнемся в своей программистской практике, приведены в табл. 2.7.

***Таблица 2.7.** Классы часто наиболее встречающихся исключений*

Класс исключения	Описание
`AttributeError`	Ошибка обращения к свойству или методу класса
`IndentationError`	Некорректный отступ
`IndexError`	Список не имеет элемента с таким индексом
`KeyError`	Словарь не имеет элемента с таким ключом
`NameError`	Переменная, функция или класс с таким именем не существует

Таблица 2.7 (окончание)

Класс исключения	Описание
NotImplementedError	Данный метод в классе не реализован
OverflowError	Слишком большое число
RuntimeError	Нераспознанная ошибка времени выполнения
SyntaxError	Ошибка в записи кода (*синтаксическая ошибка*)
TabError	Отступ задан табуляциями, а не пробелами
TypeError	Неверный тип параметра функции или метода; свойство или метод не поддерживается классом
UnicodeError	Возникла проблема при раскодировании строки, заданной в кодировке UTF-8
ValueError	Неверное значение аргумента оператора, параметра функции или метода
ZeroDivisionError	Деление на ноль

Вот пример:

```
n = 0
try:
  m = 10 / n
except ZeroDivisionError:
  m = 0
```

Здесь в *блоке try* в любом случае возникнет ошибка деления на ноль и будет сгенерировано исключение класса ZeroDivisionError. После чего выполнится соответствующий ей *блок except*.

Иногда возникает необходимость в случае возникновения какого-либо исключения ничего не делать, тем самым *подавив* это исключение. Для таких случаев удобно применять так называемый *пустой оператор* pass, который ничего не делает:

```
. . .
except ZeroDivisionError:
  pass
```

В одном *блоке except* можно указывать несколько классов исключений. Они заключаются в круглые скобки и разделяются запятыми:

```
. . .
except (IndexError, KeyError):
  . . .
```

Если какое-то исключение не было обработано в блоке, оно поступит на обработку во внешний блок обработки исключения или, если такового нет, в саму исполняющую среду Python. В последнем случае будет выведено соответствующее сообщение.

Осталось выяснить, как самим генерировать исключения. Для этого мы применим *оператор генерирования исключения* `raise`, за которым указывается выражение создания объекта нужного исключения:

```
raise IndexError("В списке нет элемента с таким индексом!")
```

Комментарии

Очень редко программист может разобраться в написанном им же самим коде спустя достаточно продолжительное время. Поэтому хороший стиль программирования требует документирования кода, для чего применяются *комментарии*. Они не обрабатываются Python и предназначаются исключительно для программиста.

Комментарии в Python должны предваряться символом решетки `#`:

```
n = 1
# Присваиваем переменной n число 1
m = 4 # Присваиваем переменной m число 4
```

Комментарии часто применяются при отладке программ. Так, если мы подозреваем, что какой-то фрагмент кода вызывает ошибку, мы можем превратить его в комментарий (закомментировать) и посмотреть, как программа работает без него. Потом, разобравшись, в чем проблема, мы его раскомментируем.

Модули. Импорт. Библиотека

Да этого момента мы занимались исключительно тем, что набирали код в окне интерактивного интерпретатора Python (см. рис. 2.1) и смотрели, что получится в результате. Но в реальном программировании нам придется сохранять весь набранный программный код в особых файлах, которые и составят наше приложение, и часто обращаться к другим файлам, где хранится код стандартной библиотеки.

Поэтому сейчас самое время поговорить о модулях, пакетах, операциях импорта и упомянуть о стандартной библиотеке.

Модули и пакеты

Модулем называется текстовый файл, в котором хранится программный код, написанный на языке Python. Этот код может объявлять переменные, функции и классы, которые могут быть использованы как внутри этого модуля, так и в других модулях. Размер модуля и количество объявленных в нем переменных, функций и классов не ограничены.

Содержимое файла модуля должно быть сохранено в кодировке UTF-8. Создать такой файл можно в любом текстовом редакторе, поддерживающем данную кодировку, например, в хорошо знакомом нам Блокноте.

Файл модуля должен иметь расширение py. Его имя без расширения станет именем самого модуля. Так, мы можем сохранить код, объявляющий классы `Square` и `Cube`, в файле square.py. Тем самым мы создадим модуль `square`.

Обычно в модуль сводят код, выполняющий сходные задачи. Так, в один модуль можно свести функции, выполняющие вывод списка товаров и сведения об одном товаре, поскольку обе они занимаются выводом товаров.

Каждый модуль — это вещь в себе. Все объявленные в нем сущности: переменные, функции и классы — изначально недоступны для других модулей, составляющих приложение. Чтобы получить из одного модуля доступ, скажем, к функции, объявленной в другом модуле, нам придется выполнить операцию импорта (о которой речь пойдет позже).

Если модулей в приложении много, имеет смысл структурировать их с применением *пакетов*. Обычно пакет включает модули, выполняющие схожие функции или разработанные одним программистом.

Пакет Python представляет собой обычную папку операционной системы, в которой хранятся файлы модулей. Имя такой папки станет именем пакета. Помимо этого, чтобы указать Python, что эта папка является пакетом, обязательно следует поместить в нее пустой файл с именем __init__.py.

Имеется возможность создавать и вложенные папки, формируя тем самым вложенные пакеты. Количество пакетов, равно как и степень их вложенности друг в друга, не ограничены.

Например, мы можем поместить наш модуль `square` в пакет `figures`, который, в свою очередь, вложить в пакет `geometry`, создав следующую структуру папок:

```
GEOMETRY
  __init__.py
  FIGURES
    __init__.py
    square.py
```

Здесь прописными буквами набраны имена папок, а строчными — имя модуля. И не забываем о служебных файлах __init__.py — они говорят Python, что данная папка является пакетом.

При первом выполнении модуля Python *компилирует* его, преобразуя в компактное внутреннее представление и удаляя весь необрабатываемый код, в частности, комментарии. Если мы исправим модуль, он будет автоматически перекомпилирован. Откомпилированные файлы модулей имеют расширение вида cpython-<*номер версии Python*>.pyc (у автора книги такие файлы имеют расширение cpython-33.pyc) и хранятся в папке __pycache__, вложенной в папку модуля.

Импорт

Ранее говорилось, что любой модуль Python — это вещь в себе, и ни один другой модуль не сможет использовать объявленную в нем сущность, пока не выполнит операцию импорта.

Операция *импорта* явно указывает, что мы хотим использовать в данном модуле сущность, объявленную в другом модуле. Выполняется она с помощью *выражений*

импорта. Такое выражение содержит *оператор подключения модуля* `import`. Имена импортируемых модулей записываются после этого оператора и разделяются запятыми.

При обращении к сущности, объявленной в другом модуле, мы запишем имя этого модуля, поставим точку и уже после нее укажем само имя нужной сущности. То есть напишем ее *полное имя*.

Скажем, импортировать классы `Square` и `Cube` из модуля `square` в другой модуль мы сможем, написав выражение:

```
import square
```

А для обращения к классу `Square` мы укажем его полное имя:

```
s = square.Square(100, 40)
```

Если же нужный модуль находится в пакете, то полное имя сущности будет включать имена всех пакетов, в которые последовательно вложен модуль, имя этого модуля и имя самой сущности, разделенные точками.

Например, если наш модуль `square` последовательно вложен в пакеты `geometry` и `figures`, нам для использования класса `Square` в другом модуле потребуется написать такие выражения:

```
import geometry.figures.square
s = geometry.figures.square.Square(100, 40)
```

Чтобы каждый раз не писать полные имена сущностей (которые могут быть очень длинными), мы можем выполнить операцию *импорта с присоединением*, указав выражение формата:

```
from <имя модуля> import <имена присоединяемых сущностей>|*
```

Имена присоединяемых сущностей разделяются запятыми. Если нужно присоединить сразу все сущности, вместо их списка мы укажем символ звездочки `*`.

Так мы присоединим и используем класс `Square`:

```
from geometry.figures.square import Square
s = Square(100, 40)
```

А так — оба класса и вообще все, что объявлено в модуле `square`:

```
from geometry.figures.square import *
s = Square(100, 40)
c = Cube(200, 20, 8)
```

Стандартная библиотека. Сторонние библиотеки

Ранее мы несколько раз упоминали о стандартной библиотеке Python. Самое время сказать о ней несколько слов.

Стандартная библиотека состоит из множества модулей, объединенных в пакеты, и поставляется в составе Python. Она включает большое количество функций и

классов, выполняющих различные типовые задачи программирования. Там мы можем найти инструменты для сложной обработки строк, объявления новых типов данных, средства для обмена данными по сети, шифрования и дешифрования, работы с файлами, разработки многопоточных приложений и многое другое.

Давайте для примера возьмем модуль `datetime`. В нем объявлен класс `date`, позволяющий хранить и обрабатывать значения даты:

```
from datetime import date
```

Импортируем класс `date` из модуля `datetime`:

```
now = date.today()
```

Метод класса `today` возвращает объект класса `date`, хранящий текущее значение даты:

```
birthday = date(1970, 10, 27)
```

Создаем еще один объект класса `date`, хранящий дату 27 октября 1970 года:

```
delta = now - birthday
days = delta.days
```

Получаем количество дней, прошедших между этими датами. (Его хранит свойство `days` класса `date`.)

Также в стандартной библиотеке объявлен класс `Fraction`, который мы рассмотрели ранее.

Стандартная библиотека хранится в папке Lib папки, в которую установлен Python. Полные имена сущностей, объявленных в стандартной библиотеке, формируются относительно этой папки.

Стандартной библиотекой Python мы более подробно займемся потом, когда начнем разрабатывать Web-приложения. Сейчас же нам нужно знать лишь то, что она есть и всегда придет к нам на помощь.

Помимо стандартной библиотеки, поставляемой в составе Python, мы можем загрузить и установить любое количество *сторонних библиотек*. Они разрабатываются силами сторонних программистов (отчего и получили свое название), а существует их столько, что мы можем без труда найти ту, что нам нужна.

Все сторонние библиотеки устанавливаются в папку site-packages, что автоматически создается в упомянутой ранее папке Lib. Полные имена сущностей, объявленных в сторонних библиотеках, формируются относительно этой папки.

Кстати, Django, которой мы займемся уже в следующей главе, — это тоже сторонняя библиотека Python.

Текстовый редактор Notepad++

Ранее говорилось, что для написания Python-кода можно использовать любой текстовый редактор, даже стандартный Блокнот. На практике же Блокнот для этого не очень удобен — ведь нам самим придется ставить отступы, формирующие блоки.

К счастью, существуют специальные текстовые редакторы для программистов, которые, помимо автоматического выставления отступов, реализуют также синтаксическую подсветку кода, проверку на ошибки в синтаксисе и многие другие полезные вещи. Автор книги предпочитает пользоваться бесплатным редактором Notepad++ (рис. 2.2), который можно найти на сайте **http://notepad-plus-plus.org**.

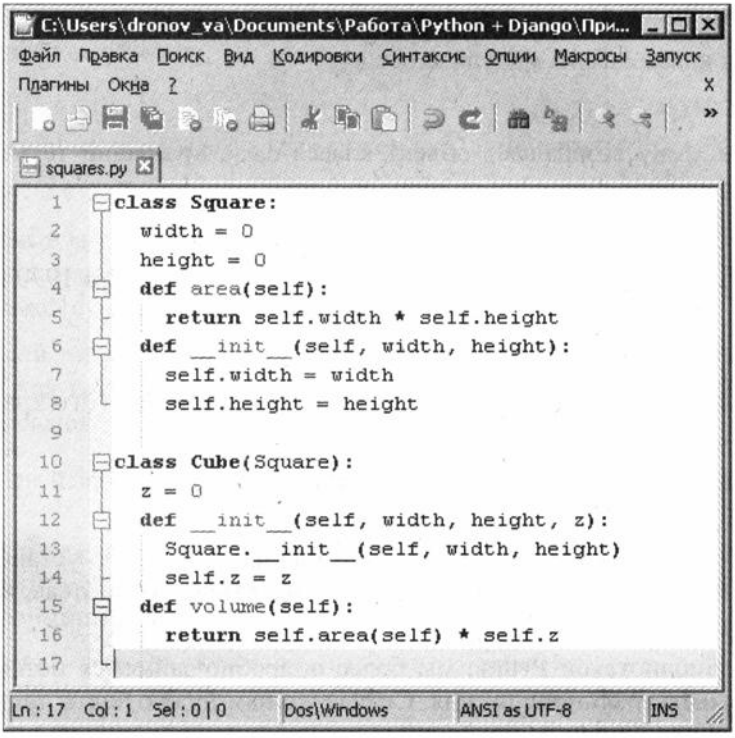


Рис. 2.2. Окно редактора Notepad++

По умолчанию этот редактор создает отступы с применением табуляции, в то время как Python требует лишь пробелы. Поэтому нам придется соответственно его настроить. Выберем в меню **Опции** (Settings) пункт **Настройки** (Preferences) и в левом списке открывшегося окна **Настройки** (Preferences) выберем пункт **Табуляция** (Tab Settings). Мы увидим то, что показано на рис. 2.3.

Проверим, выбран ли в списке **Настройка табуляции** (Tab Settings) пункт **[Default]**. Найдем флажок **Заменить пробелом** (Replace by space) и установим его. И не забудем нажать кнопку **Закрыть** (Close).

Чтобы сохранить набранный код в виде файла модуля Python, мы выберем в списке типов файлов диалогового окна сохранения файла пункт **Python file (*.py; *.pyw)**.

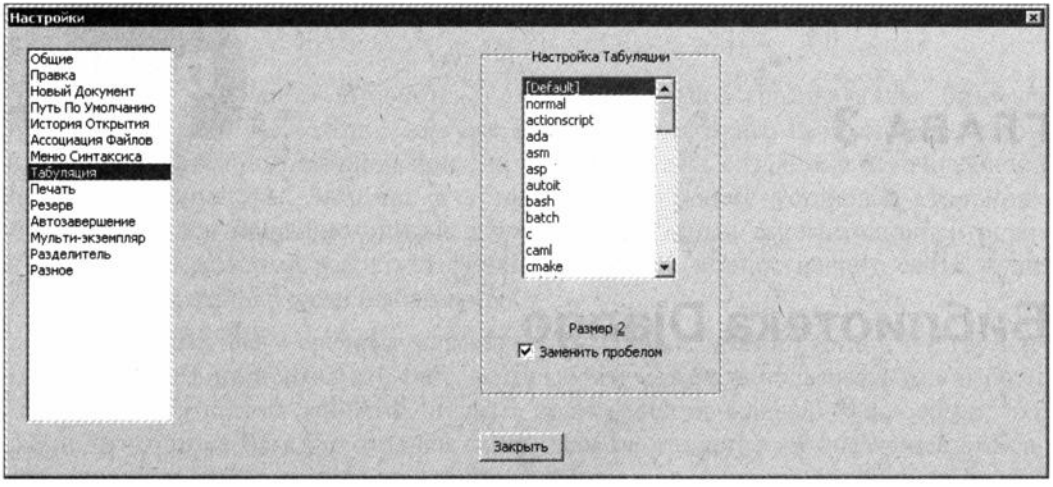


Рис. 2.3. Диалоговое окно **Настройки**; выбран пункт **Табуляция** левого списка

Что дальше?

Эта глава целиком посвящена языку программирования Python. Мы рассмотрели его основные понятия, типы данных и операции с ними, управляющие выражения, функции, классы и модули. Наконец, мы познакомились с текстовым редактором Notepad++, в котором и будем писать наш программный код.

Следующая глава посвящена библиотеке Django. Мы выясним, зачем она вообще нужна, что она дает нам как программистам, и выучим еще несколько терминов, которыми станем пользоваться в дальнейшем.

ГЛАВА 3

Библиотека Django

В предыдущей главе мы изучили язык программирования Python, на котором собираемся создавать свои Web-сайты. Как выяснилось, язык этот не так уж и сложен, но при этом предоставляет нам все нужные программисту инструменты.

Но для разработки сайтов нам понадобится кое-что еще. А именно, сторонняя библиотека Python, носящая звучное название Django. Вот о ней-то и пойдет сейчас разговор.

Библиотека Django — зачем она нужна?

Из *главы 1* мы знаем, что любой Web-сайт, построенный на основе Web-приложений, состоит из программных модулей трех разновидностей: моделей, контроллеров и шаблонов. Следовательно, нам достаточно написать все модели, контроллеры и шаблоны — и сайт готов. Так ли это?

Так, да не совсем.

Модели, контроллеры и шаблоны в любом случае не заработают сами по себе. Нам придется создать еще и необходимую программную инфраструктуру, которая выступит как костяк сайта, скрепит воедино все написанные нами модули, будет в нужный момент времени запускать их и поможет нам реализовать всяческие второстепенные задачи: разграничение доступа, кэширование и пр. Эта инфраструктура включает в себя:

- программное ядро, в частности, диспетчер (см. *главу 1*);
- шаблонизатор (автор специально выделил его из состава ядра, т. к. написать хороший шаблонизатор — крайне трудоемкая задача);
- базовую функциональность моделей и контроллеров;
- модули, реализующие второстепенные задачи Web-программирования — например, разграничение доступа.

Чтобы написать все это, как говорится, с нуля, потребуется много времени и сил. Времени и сил, которые мы можем употребить на что-либо более полезное, чем на

разработку того, что уже сделали за нас. Сделали и собрали в программный пакет с названием Django.

Django — это сторонняя библиотека для языка Python, реализующая базовую функциональность Web-сайта. Она уже включает в себя все необходимое, чтобы мы смогли начать программирование сайта, не занимаясь созданием перечисленной ранее инфраструктуры. Загрузив и установив ее (процесс установки сторонних библиотек Python подробно описан в *приложении 1*), мы сможем сконцентрироваться на коде, который реализует функциональность именно нашего сайта, — на его моделях, контроллерах и шаблонах.

Django содержит также средства для взаимодействия с базами данных, инструменты отладки, административный сайт, который мы можем использовать для работы с хранящимися в базе данными, и даже полнофункциональный Web-сервер, что крайне пригодится нам для отладки сайта. Нам не придется ни писать все это самим, ни прибегать к дополнительным программам.

Python и Django — вот «дуэт», что делает разработку даже сложных Web-сайтов такой простой!

Основные термины и принципы Django-программирования

Чтобы успешно использовать Django, нам нужно знать несколько новых принципов и терминов. Их немного, и они совсем несложны.

Проект

В терминологии Django *проектом* называется совокупность Web-приложений, составляющих один Web-сайт. Другими словами, проект — это сам Web-сайт, разработанный с применением Django.

Физически проект представляет собой обычную папку операционной системы. Имя этой папки станет именем проекта.

Структура папок и файлов проекта, формируемая при его создании самой Django, такова:

```
<ИМЯ ПРОЕКТА>
  manage.py
  <ИМЯ ПРОЕКТА>
      __init__.py
    settings.py
    urls.py
    wsgi.py
```

Прописными буквами набраны имена папок, а строчными — имена файлов.

Как видим, во внешней папке, где хранятся файлы проекта (*папка проекта*), находится файл manage.py. Этот файл хранит код утилиты, которая позволит нам выпол-

нять различные действия над проектом, в частности, создавать в нем приложения — мы будем запускать этот файл из командной строки Windows.

В папке проекта присутствует еще одна папка с таким же именем. Это *пакет проекта* — он содержит модули, относящиеся к самому проекту. О том, что это именно пакет, говорит файл __init__.py, хранящийся в этой папке (о пакетах Python рассказывалось в *главе 2*).

В пакете проекта мы видим следующие модули:

- `settings` — хранит настройки проекта в виде набора переменных;
- `urls` — хранит сведения о привязке приложений к интернет-адресам;
- `wsgi` — служебный модуль, выступающий «посредником» между Web-сервером и проектом. Этот модуль потребуется нам в *главе 35*, когда мы будем заниматься публикацией нашего сайта.

В настройках проекта указываются сведения об используемой в приложениях проекта базе данных (или базах данных — мы можем указать их несколько), список активных приложений проекта, языковые параметры и некоторые другие данные. Отметим, что все эти настройки разделяются всеми приложениями, что входят в проект.

Папка проекта может находиться в любом месте файловой системы компьютера. Так что мы можем создать проект там, где нам удобно.

Приложение

Приложение Python входит в состав проекта и реализует функциональность одного из разделов сайта и всех его подразделов. Количество приложений в проекте не ограничено.

Физически приложение представляет собой пакет, папка которого находится в папке проекта (не в пакете проекта!). Имя этого пакета станет именем приложения, а сам пакет называется *пакетом приложения*.

Пакет приложения формируется самой Django при создании приложения. Изначально он содержит следующие модули:

- `models` — хранит код моделей, входящих в состав приложения;
- `views` — хранит код контроллеров, входящих в состав приложения;
- `admin` — хранит код, задающий параметры административного приложения, что входит в состав Django;
- `tests` — тестовый модуль.

Разумеется, мы можем, если возникнет такая необходимость, создать в пакете приложения другие модули, хранящие код моделей, контроллеров или дополнительных функций и классов, что мы задействуем в коде сайта. Django в этом плане никак нас не ограничивает.

Вот только все шаблоны, применяемые в сайте, нам в любом случае придется создать вручную. Django этого за нас не сделает.

Даже если приложение входит в состав проекта, это еще не говорит о том, что оно будет задействовано в сайте. Чтобы оно успешно работало, следует, во-первых, выполнить его привязку к интернет-адресу (об этом мы скоро поговорим), а во-вторых, указать его в списке активных приложений, что находится в модуле `settings` пакета проекта (см. ранее). Только после этого приложение станет *активным*.

Нужно сказать, что часть вспомогательных модулей Django реализована также в виде приложений (*встроенные приложения*). Таким встроенным приложением является, в частности, подсистема, реализующая разграничение доступа (разговор о ней пойдет в *главе 14*). Административный сайт (о нем мы поговорим в *главе 4*), с помощью которого мы можем работать с хранящимися в базе данными, также является приложением подобного рода.

Встроенные приложения либо также привязываются к интернет-адресу и, тем самым, формируют новый раздел сайта, либо работают постоянно, обеспечивая вспомогательную функциональность. В любом случае их также нужно указать в списке активных приложений, иначе они не будут задействованы.

> ***Указание имен пакетов и модулей***
>
> Все полные имена пакетов и модулей, входящих в состав пакета проекта и отдельных приложений, указываются относительно папки проекта. Так, чтобы сослаться на модуль `views` пакета приложения `page`, следует указать полное имя `views.page`.

Привязка интернет-адресов

В *главе 1* мы узнали, что каждое приложение, входящее в состав сайта, запускается в ответ на обращение к определенному интернет-адресу, к которому оно было привязано. Единственное исключение здесь — встроенные приложения, с которыми мы только что познакомились и которые работают постоянно и не требуют такой привязки.

Каждому приложению ставится в соответствие своего рода виртуальная папка (об этом также говорилось в *главе 1*). Скажем, папку goods можно поставить в соответствие приложению списка товаров, выводящему список товаров, а папку guestbook — приложению гостевой книги, который выводит ее записи. Тогда посетитель, набрав интернет-адрес вида **http://www.somesite.ru/goods/**, попадет на список товаров, а, набрав **http://www.somesite.ru/guestbook/**, увидит гостевую книгу.

В библиотеке Django привязка интернет-адреса к приложению выполняется в модуле `urls` пакета проекта (см. раздел, посвященный проекту). Или, говоря другими словами, привязка адресов к приложениям выполняется на уровне проекта.

Но в реальности одно приложение может выполнять сразу несколько действий. Скажем, приложение списка товаров может выводить как и сам этот список, так и сведения о выбранном товаре, а приложение гостевой книги — как выводить гостевую книгу, так и добавлять в нее новую запись. Как это реализовать? Очень просто. Отдельным контроллерам приложений ставятся в соответствие виртуальные подпапки упомянутых ранее папок, формирующие подразделы данных разделов сайта.

В нашем случае мы привяжем подпапку / к контроллеру, выводящему список товаров, а подпапку good — к контроллеру, что выводит сведения об отдельном товаре.

Привязка подпапок, соответствующих подразделам сайта, к контроллерам приложения выполняется в модуле `urls` пакета приложения — уже на уровне приложения. (К сожалению, Django не создает этот модуль при формировании нового приложения, и нам придется сделать это самим.)

При привязке интернет-адреса к контроллеру мы можем указать, что последний должен принимать какие-либо данные. Эти данные будут переданы в составе интернет-адреса с применением метода GET. Например, поскольку мы собираемся реализовать вывод сведений о выбранном товаре, нам придется передавать соответствующему контроллеру идентификатор этого товара.

Структура Django-сайта

Как мы уже знаем, приложение Django реализует функциональность одного раздела сайта и всех его подразделов. Это достигается тем, что к интернет-адресу привязывается не само приложение, а его контроллер.

В связи с этим можно сформулировать правила структурирования сайтов, написанных на Django:

- один раздел сайта реализуется одним приложением. В число таких разделов входит и главная страница сайта;
- один подраздел раздела сайта реализуется одним контроллером, входящим в состав приложения;
- по возможности одно приложение не должно реализовывать функциональность, относящуюся к другому разделу сайта. Или, говоря другими словами, одно приложение не должно вторгаться в дела другого;
- однако административные задачи (например, наполнение сайта), по возможности, должны выполняться отдельным приложением, носящим название *административного*.

Соблюдать эти правила несложно, благо сам Python с его независимостью отдельных модулей и пакетов друг от друга подталкивает нас к этому. Вместе с тем, если нам понадобится в одном приложении использовать, скажем, модуль из другого приложения, мы всегда сможем это сделать, выполнив операцию импорта (см. *главу 2*).

Поддерживаемые форматы баз данных

Поскольку Django-сайт представляет собой набор приложений, все его данные хранятся в отдельной базе. Какой формат баз данных нам выбрать?

Django для хранения данных позволяет использовать базы разных форматов. Официально, самими разработчиками этой библиотеки заявлена поддержка PostgreSQL,

MySQL и Oracle, доступны также и сторонние библиотеки, разработанные третьими программистами и обеспечивающие поддержку других форматов баз данных.

Помимо этого, Django поддерживает формат баз данных SQLite. СУБД, работающая с такими базами данных, встроена в сам Python, так что никаких дополнительных программ и библиотек для реализации ее работы нам устанавливать не придется. А для нужд разработки сайтов ее возможностей вполне достаточно — если же потребуется, мы без труда сможем перенастроить сайт на работу с базой другого формата, того же MySQL.

Базы данных SQLite имеют и другое преимущество — нам не придется создавать их самим. Нужно будет лишь занести сведения о базе данных в настройки сайта, создать модели и выполнить простую команду синхронизации, после чего Django сама создаст базу, а в ней — все необходимые таблицы, индексы и связи.

Отладочный Web-сервер Django

В процессе разработки Web-сайта нам придется неоднократно запускать его, чтобы проверить, как он работает и работает ли вообще. Понятно, что для этого нужен Web-сервер, например, популярнейший Apache. Разработчики сайтов, применяющие другие языки и платформы (PHP, Zend, Yii и др.), просто вынуждены установить его на компьютер и соответственно настроить.

Но, поскольку мы используем язык Python и библиотеку Django, нам не нужен отдельный Web-сервер. В состав самой этой библиотеки входит *отладочный Web-сервер*, который устанавливается вместе с ней и не требует никакой настройки. Запустить его можно выполнением очень простой команды — мы без труда ее запомним.

Отладочный Web-сервер Django по умолчанию привязан к TCP-порту под номером 8000. Поэтому, чтобы запустить сайт на выполнение, мы должны набрать в Web-обозревателе интернет-адрес вида **http://localhost:8000/** или **http://localhost:8000/goods/**. (Впрочем, имеется возможность при запуске отладочного Web-сервера указать, чтобы он работал через другой порт.)

> ***Публикация готового сайта***
>
> Разработчики Django настоятельно не рекомендуют использовать отладочный Web-сервер для публикации готового сайта вследствие проблем с безопасностью. Поэтому, чтобы опубликовать Django-сайт, следует применить полнофункциональный Web-сервер, скажем, Apache.

Что дальше?

В этой главе мы рассмотрели теорию Django-программирования и построения Django-сайтов. Мы познакомились с проектами и приложениями, узнали о привязке интернет-адресов к приложениям, поддерживаемых форматах баз данных и отладочном Web-сервере, который пригодится нам при отладке. Можно начинать создание нашего первого Django-сайта!

ГЛАВА 4

Создание проекта и приложения Django

В предыдущей главе мы рассмотрели теоретические вопросы Django-программирования: что такое проект и приложение Django, как приложения соотносятся с интернет-адресами, какие форматы баз мы можем использовать для хранения данных сайта и что можем задействовать для его отладки. Поскольку нам не терпелось поскорее приняться за практику, мы особо на них не задержались (тем более что там и задерживаться-то было особо не на чем).

А сейчас настала пора практики. Мы узнаем, как создать и настроить проект Django, как указать сведения об используемой базе данных и как создать приложение. Мы также выясним, как запустить и остановить отладочный Web-сервер Django и как пользоваться встроенным в эту библиотеку административным Web-сайтом.

Создание проекта Django

Перед созданием проекта Django нам нужно выбрать место, где будет храниться его папка. Это может быть любая папка в файловой системе компьютера.

Запустим командную строку Windows. В ней, пользуясь соответствующими командами (они описаны в интерактивной справке к операционной системе), выполним переход в папку, где будет создан проект. И наберем команду следующего формата:

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

не забыв нажать клавишу <Enter>. Через несколько секунд проект будет создан.

Файл django-admin.py хранит код административной утилиты Django, которая, в числе прочих действий, выполняет создание нового проекта. Командный ключ `startproject` выполняет создание нового проекта с именем, указанным за ним.

> ***Путь к файлу django-admin.py***
>
> Файл *django-admin.py* находится по пути *<папка, где установлен Python>\Lib\site-packages\django\bin*. Возможно, нам потребуется указать путь к нему, если мы не занесли его в список путей операционной системы (как это сделать, описано в *приложении 1*).

Хорошая новость: проект Django никоим образом не привязывается к конкретному пути операционной системы компьютера. Это значит, что мы не только можем создать его где угодно, но и имеем возможность для продолжения работы или публикации в Интернете переместить его в любую другую папку.

Запуск и останов отладочного Web-сервера

Создав проект, мы можем проверить, был ли он создан. Для этого мы запустим отладочный Web-сервер и попытаемся выйти на соответствующий проскту сайт.

Запускать отладочный Web-сервер удобнее из отдельного окна командной строки. Откроем его, выполним в нем переход в папку только что созданного проекта и наберем такую команду:

```
manage.py runserver
```

Как мы уже знаем из *главы 3*, файл manage.py хранит код служебной утилиты, предназначенной для работы непосредственно с проектом, которому она принадлежит. В частности, именно эта утилита запускает отладочный Web-сервер Django, для чего нам достаточно указать командный ключ `runserver`.

Как только мы нажмем клавишу <Enter>, в окне командной строки появится вот такой текст:

```
Validating models...

0 errors found
February 18, 2014 - 14:33:09
Django version 1.6.2, using settings 'sample.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Первая его строка говорит о том, что Django проверяет созданные в проекте модели приложений (которых у нас пока нет). По окончании проверки будет выведена пустая строка, а за ней — строка с количеством найденных ошибок (у нас — 0).

Если в результате проверки моделей не было найдено ошибок, в последующих строках появятся:

- сегодняшние дата и время;
- версия библиотеки Django и полное имя модуля настроек, относящегося к данному проекту (у автора это модуль `sample.settings`, т. к. проект носит имя `sample`);
- сообщение о запуске отладочного Web-сервера и его интернет-адрес (**http://127.0.0.1:8000**);
- сообщение о том, что для завершения отладочного Web-сервера следует нажать комбинацию клавиш <Ctrl>+<Break>.

Если мы теперь наберем в Web-обозревателе интернет-адрес отладочного Web-сервера, то увидим следующую страничку (рис. 4.1).

Рис. 4.1. Сообщение отладочного Web-сервера при пустом проекте

Если совсем коротко, она поздравляет нас с успешно установленной Django и приглашает создать первое приложение.

Завершим работу отладочного Web-сервера, переключившись на окно командной строки, где он был запущен, и нажав комбинацию клавиш <Ctrl>+<Break>.

Настройка проекта Django

Подождем пока создавать приложения в нашем новом проекте. Давайте сначала укажем для него необходимые настройки. В частности, нам потребуется задать сведения об используемой базе данных и параметры локализации (используемый на сайте язык и часовой пояс), а также выяснить на будущее, где хранится список активных приложений.

Откроем папку проекта. Найдем в ней папку, соответствующую пакету проекта, а в ней — модуль `settings`, где в виде набора переменных хранятся настройки сайта. И откроем файл этого модуля в текстовом редакторе Notepad++.

> ***Сохранение модуля settings***
>
> После внесения любых правок в настройки сайта следует сохранить модуль `settings`. Измененные настройки будут применены к сайту немедленно.

Сведения о базе данных

Как говорилось в *главе 3*, Django поддерживает хранение данных сайта в базах самых разных форматов. Мы для разработки сайта выбрали формат SQLite как самый простой в использовании и не требующий установки никаких дополнительных программ.

Параметры используемых баз данных хранит переменная `DATABASES`. Вот код, создающий значение этой переменной и сформированный Django при создании проекта:

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

Как видим, переменная DATABASES хранит словарь Python, каждый элемент которого также представляет собой аналогичный словарь.

Один элемент «внешнего» словаря задает сведения об одной базе данных. Ключ этого элемента фактически определит имя базы.

Если в нашем проекте задействована всего одна база данных, она должна называться default (как в представленном ранее коде). Если же наш проект оперирует несколькими базами, одна из них все равно должна иметь имя default — она станет базой по умолчанию, и к ней будет выполняться обращение, если имя базы не было указано явно.

Каждый элемент вложенного словаря определяет различные параметры, необходимые для подключения к базе данных. К таким параметрам относятся полное имя модуля, реализующего взаимодействие с базой, имя ее файла, адрес обрабатывающего эту базу сервера данных, имя и пароль, что будут использоваться для подключения к ней, и др.

Таких параметров очень много. Но в случае базы SQLite имеют смысл лишь два: полное имя модуля, реализующего взаимодействие с базой, и имя хранящего ее файла. Эти параметры задает сама Django при создании нового проекта.

Элемент внутреннего списка с ключом ENGINE задает полное имя модуля, выступающего как посредник между Django и базой данных. Для баз данных формата SQLite таким «посредником» выступает модуль django.db.backends.sqlite3.

Элемент внутреннего списка с ключом NAME задаст путь к файлу с базой данных. Мы можем указать его явно, в виде строки, или как результат, возвращенный функцией.

Кстати, по умолчанию он так и задан. Пусть к файлу базы вычисляется функцией join из модуля os.path. Эта функция добавляет к пути, заданному первым параметром, имя файла, заданное вторым параметром, и возвращает полученный результат — полное имя файла. Оба параметра задаются в виде строк, и результат возвращается как строка. В нашем случае первым параметром этой функции является объявленная ранее в модуле settings переменная BASE_DIR, хранящая полученный в результате вычислений текущий путь к папке проекта, а вторым — имя файла db.sqlite3. Так что база данных сайта по умолчанию будет храниться в файле db.sqlite3, расположенном прямо в папке проекта.

Как правило, нет большого смысла указывать для файла базы другой путь — держа его в папке проекта, мы сохраним и данные, и код в одном месте, что удобно при разработке. А вот имя файла поменять смысл имеет, причем мы можем задать ему любое имя и даже расширение — скажем, назвать его site.dat или database.bin.

СУБД SQLite поддерживает и другие параметры, управляющие различными нюансами ее работы. Однако они очень специфичны, и их значения следует менять лишь в особых случаях.

Параметры локализации

Параметры локализации включают обозначение языка и временной зоны. Эти сведения используются, в частности, при сортировке записей в базах данных и выборе языка для вывода интерфейса встроенного административного Web-сайта Django (о нем будет рассказано в конце этой главы).

Параметры локализации задаются в переменных `LANGUAGE_CODE` и `TIME_ZONE` в виде строк.

```
LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'UTC'
```

Переменная `LANGUAGE_CODE` задает код языка сайта. В табл. 4.1 перечислены коды языков, с которыми мы будем иметь дело в большинстве случаев.

Таблица 4.1. *Коды некоторых языков*

Код	Язык
be-by	Белорусский (Беларусь)
en-gb	Английский (Великобритания)
en-us	Английский (США)
kk-kz	Казахский (Казахстан)
ru-ru	Русский (Россия)
uk-ua	Украинский (Украина)

Более полный список кодов языков можно найти по интернет-адресу **http://www.i18nguy.com/unicode/language-identifiers.html**.

Понятно, что, скорее всего, мы укажем для сайта русский язык, задав переменной `LANGUAGE_CODE` значение `ru-ru`.

Переменная `TIME_ZONE` служит для задания временной зоны. Различные ее значения для некоторых временных зон представлены в табл. 4.2.

Таблица 4.2. *Обозначения некоторых временных зон*

Временная зона	Город
Asia/Almaty	Алматы
Asia/Anadyr	Анадырь
Asia/Irkutsk	Иркутск
Asia/Kamchatka	Камчатка

Таблица 4.2 (окончание)

Временная зона	Город
Asia/Krasnoyarsk	Красноярск
Asia/Magadan	Магадан
Asia/Novokuznetsk	Новокузнецк
Asia/Novosibirsk	Новосибирск
Asia/Omsk	Омск
Asia/Sakhalin	Сахалин
Asia/Vladivostok	Владивосток
Asia/Yakutsk	Якутск
Asia/Yekaterinburg	Екатеринбург
Europe/Kaliningrad	Калининград
Europe/Kiev	Киев
Europe/Minsk	Минск
Europe/Moscow	Москва
Europe/Samara	Самара
Europe/Simferopol	Симферополь
Europe/Uzhgorod	Ужгород
Europe/Volgograd	Волгоград

Также можно задать значение `UTC`, обозначающее универсальное время. (Кстати, именно это значение задано в настройках сайта по умолчанию.)

Более полный список временных зон можно посмотреть на странице с интернет-адресом **http://en.wikipedia.org/wiki/List_of_tz_database_time_zones**.

Список активных приложений

Список активных приложений указывается в переменной `INSTALLED_APPS` в виде кортежа, состоящего из строк, значениями которых являются полные имена модулей этих приложений.

```
INSTALLED_APPS = (
  'django.contrib.admin',
  'django.contrib.auth',
  'django.contrib.contenttypes',
  'django.contrib.sessions',
  'django.contrib.messages',
  'django.contrib.staticfiles',
)
```

Изначально здесь перечислены несколько встроенных приложений Django. Все они либо нужны для нормальной работы библиотеки, либо применяются в большинстве сайтов и поэтому включаются в список сразу при создании проекта.

- `django.contrib.admin` — реализует работу встроенного административного Web-сайта Django;
- `django.contrib.auth` — реализует работу подсистемы разграничения доступа (см. *главу 14*);
- `django.contrib.contenttypes` — обеспечивает автоматическую обработку моделей, например, при синхронизации (о синхронизации моделей с базой данных мы поговорим чуть позже);
- `django.contrib.sessions` — обеспечивает обработку сессий и необходимо для успешной работы подсистем разграничения доступа и хранения данных на уровне клиента (см. *главу 12*);
- `django.contrib.messages` — реализует работу подсистемы сообщений Django (см. *главу 12*);
- `django.contrib.staticfiles` — реализует функционирование подсистемы обработки статических файлов (см. *главу 8*).

Мы будем добавлять в этот кортеж новые приложения, как созданные нами, так и встроенными в Django.

Остальные настройки, присутствующие в модуле `settings`, пригодятся лишь в очень специфических случаях. Мы не будем их здесь рассматривать.

Синхронизация с базой данных

Мы уже знаем, что сразу при создании проекта некоторые встроенные приложения Django уже оказываются включенными в список активных приложений. Они нужны для нормальной работы сайта.

А для нормальной работы этих встроенных приложений им требуется база, в таблицах которой они будут хранить свои данные.

Проблема в том, что изначально у нас нет никакой базы данных, а даже если она и есть, то в ней отсутствуют нужные встроенным приложениям таблицы, индексы и связи. (Вспомним: мы ведь не создали базу, а лишь задали ее параметры в настройках проекта.) Поэтому нам прямо сейчас потребуется выполнить операцию синхронизации.

Операция *синхронизации с базой данных* осуществляет действия:

- создает базу данных (выполняется не для всех форматов баз);
- создает в базе необходимые структуры — таблицы, индексы и связи;
- возможно, заносит в таблицы некие изначальные данные.

Необходимые структуры данных

Создание необходимых структур данных выполняется всего один раз. Впоследствии, если таковые структуры уже присутствуют в базе, они создаваться не будут.

В качестве основы для создания самой базы и необходимых структур данных в ней используются, во-первых, заданные нами ранее настройки базы, а во-вторых, модели, являющиеся частью всех активных приложений проекта.

Выполнить синхронизацию очень просто. Откроем командную строку Windows, выполним переход в папку проекта и наберем такую команду:

```
manage.py syncdb
```

Командный ключ `syncdb` утилиты manage.py как раз выполняет синхронизацию.

Мы сразу же увидим многочисленные сообщения о создании в базе данных новых таблиц и индексов.

А теперь — внимание! Утилита manage.py выведет предупреждение о том, что в списке пользователей подсистемы разграничения доступа Django нет ни одного пользователя с административными полномочиями, и предложит нам создать такого пользователя. Настоятельно рекомендуется согласиться, введя в ответ на предупреждение строку `yes` и нажав клавишу <Enter>. (Если же мы не создадим администратора, то впоследствии не сможем войти на встроенный административный сайт Django.)

Далее нас попросят ввести имя пользователя-администратора, его адрес электронной почты и пароль, причем пароль нам потребуется ввести дважды — для надежности. Введем все это, не забывая нажимать клавишу <Enter>.

В конце концов мы увидим сообщение об успешном создании пользователя-администратора и еще несколько сообщений о ходе работы. И синхронизация завершится.

Создание приложения Django

Задав необходимые настройки только что созданного проекта и выполнив первую синхронизацию, мы можем создать наше первое приложение. Откроем командную строку Windows, перейдем в папку проекта и введем команду вида:

```
manage.py startapp <имя приложения>
```

Командный ключ `startapp` предписывает утилите manage.py создать в проекте приложение с именем, указанным после этого ключа.

Новое приложение создается быстро — не пройдет и секунды, как оно будет сформировано.

Теперь добавим его в список активных приложений. Откроем модуль `settings` из пакета проекта, найдем объявление переменной `INSTALLED_APPS` и изменим его код, добавив в кортеж пакет приложения.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'page'
)
```

Для примера автор создал приложение `page`, после чего добавил его в список активных.

Встроенный административный сайт Django

При разработке сайта нам часто понадобится заносить в базу какие-либо данные, необходимые для отладки. Разработчикам, использующим другие решения, потребуется привлекать для этого сторонние программы. Но нам этого делать не придется.

Библиотека Django включает в свой состав полнофункциональный *встроенный административный Web-сайт*, позволяющий просматривать содержимое созданных в составе приложений моделей, заносить в них новые данные и править уже существующие. Этот сайт активен по умолчанию, и чтобы запустить его, нам не надо будет выполнять никаких дополнительных действий.

Встроенный административный сайт привязан к виртуальной папке admin. Поэтому, чтобы войти в него, нам достаточно набрать в Web-обозревателе интернет-адрес **http://localhost:8000/admin/**.

> ***Локализация интерфейса административного сайта***
>
> Здесь описан русский интерфейс административного сайта. Чтобы его включить, следует указать в настройках проекта русский язык (как это сделать, было рассказано ранее).

Первое, что мы увидим, — страницу с формой входа (рис. 4.2). Введем в поле ввода **Имя пользователя** имя пользователя-администратора, а в поле ввода **Пароль** — его пароль и нажмем кнопку **Войти**. (Имя администратора и пароль мы указали при выполнении первой синхронизации с базой данных.)

При успешном входе мы попадем на страницу списка приложений (рис. 4.3). Она содержит набор таблиц, каждая из которых соответствует одному из активных приложений сайта: имя приложения указано в заголовке соответствующей таблицы, а строки таблицы соответствуют созданным в приложении моделям.

Так, на рис. 4.3 мы видим таблицу, соответствующую встроенному приложению `Auth`, которое обеспечивает работу подсистемы разграничения доступа. В этой таблице присутствуют строки **Группы** и **Пользователи**, представляющие две имеющиеся в данном приложении модели: первая модель хранит список групп пользователей, а вторая — список самих пользователей.

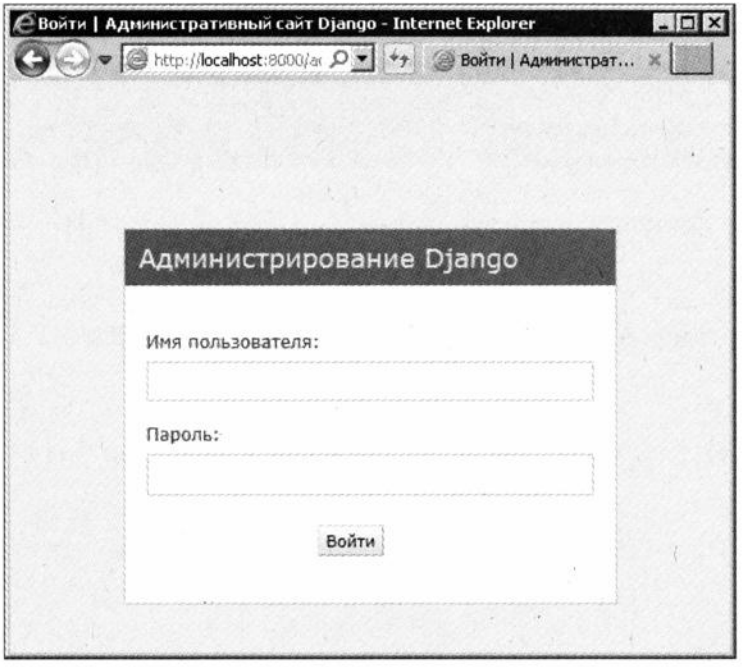


Рис. 4.2. Страница входа на административный сайт

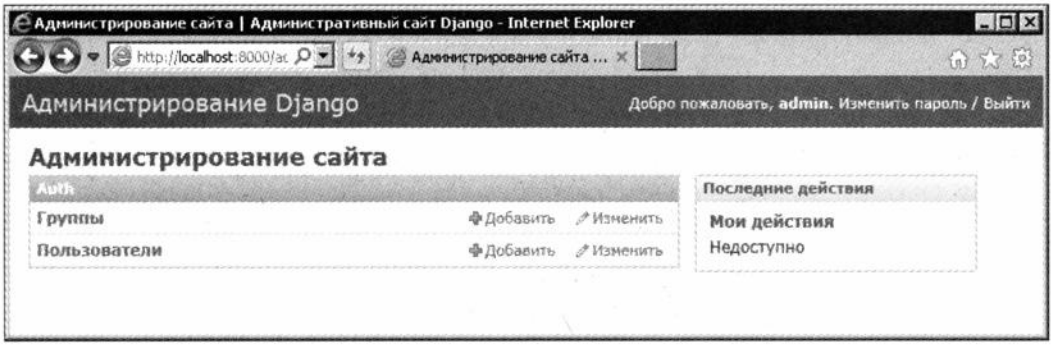


Рис. 4.3. Страница списка приложений

Чтобы просмотреть содержимое какой-либо модели, следует щелкнуть мышью на ее названии — это название представляет собой гиперссылку. Административный сайт в ответ выведет нам страницу содержимого модели (рис. 4.4).

Здесь мы видим таблицу, перечисляющую все записи выбранной нами модели. В нашем случае запись всего одна — она представляет пользователя-администратора, что мы создали при синхронизации (у автора этот пользователь носит имя `admin`).

Добавить в модель новую запись можно, нажав расположенную в верхнем правом углу кнопку **Добавить** ***<наименование сущности, хранящейся в модели>***. На экране появится страница добавления записи (рис. 4.5).

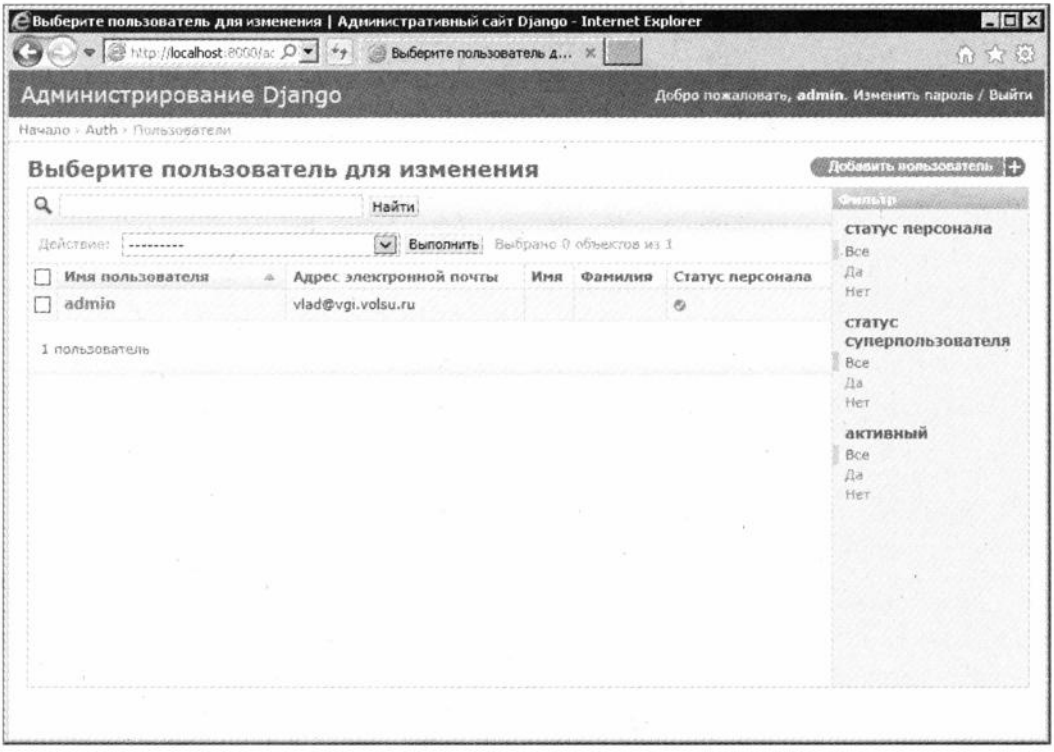


Рис. 4.4. Страница содержимого модели

Рис. 4.5. Страница добавления записи в модель

Отметим сразу, что, если модель содержит большое количество полей, таких страниц добавления записи будет несколько, и эти страницы станут выводиться последовательно, одна за другой. К моделям такого рода относится и модель **Пользователи**, с которой мы сейчас работаем.

На страницах добавления записи все просто — вводим в элементы управления нужные значения и нажимаем одну из находящихся внизу кнопок:

- **Сохранить** — либо перейти на следующую страницу добавления записи, либо, если это последняя страница, сохранить новую запись и вернуться на страницу содержимого модели;
- **Сохранить и добавить новый объект** — либо перейти на следующую страницу добавления записи, либо, если это последняя страница, сохранить новую запись и подготовить сайт для добавления еще одной записи;
- **Сохранить и продолжить редактирование** — либо перейти на следующую страницу добавления записи, либо, если это последняя страница, сохранить новую запись и подготовить сайт для ее правки.

Исправить уже хранящуюся в модели запись тоже несложно. В таблице на странице содержимого модели (см. рис. 4.4) имеется столбец, содержимое ячеек которого представляет собой гиперссылки (в нашем случае — это строка в столбце **Имя пользователя**). Нам нужно всего лишь щелкнуть на той гиперссылке, что принадлежит нуждающейся в исправлении записи.

В результате мы перейдем на набор страниц изменения записи, похожих на уже знакомые нам страницы их добавления (см. рис. 4.5). В элементы управления этих страниц будут подставлены значения соответствующих полей выбранной нами записи. Внесем нужные правки и нажмем одну из перечисленных ранее кнопок.

Удалить ненужную запись несколько сложнее. На странице содержимого модели найдем запись, которую хотим удалить, выделим ее, установив расположенный в самом левом столбце флажок (кстати, так мы можем выделить для удаления сразу несколько записей), выберем в списке **Действие** пункт **Удалить выбранные *<имя модели>*** и нажмем кнопку **Выполнить**. Django выведет страницу подтверждения, где присутствует список удаляемых записей и кнопка **Да, я уверен**. Которую мы и нажмем.

Выйти из административного сайта можно, щелкнув расположенную в верхнем левом углу гиперссылку **Выйти**. На очередной странице мы увидим довольно неуклюжее сообщение, говорящее об успешном выходе, и гиперссылку **Войти снова**.

Что дальше?

В этой главе мы создавали и настраивали наш первый проект, выполняли первую синхронизацию с базой данных и генерировали первое приложение Django. Попутно мы познакомились со встроенным административным Web-сайтом, который поможет нам наполнить модели отладочными данными.

Модели, модели, модели — только мы и слышим... Не пора ли заняться ими вплотную и создать, наконец, хоть одну из них?

ЧАСТЬ II

Вывод данных

ГЛАВА 5

Модели Django

Предыдущая часть книги носила, по большей части, вводный характер. Мы изучили основные принципы серверного Web-программирования, язык Python и библиотеку Django, с применением которых мы будем программировать свои сайты, и научились создавать проекты и приложения Django.

В этой части мы займемся исключительно практикой. И начнем с чего попроще — с реализации вывода хранящихся в базе данных.

Создание моделей

Но перед тем как выводить данные из базы, нам следует описать их структуру. Другими словами, нам нужно создать модели, описывающие эти данные.

Как создается модель

При добавлении в проект нового приложения Django сама создает в пакете приложения модуль с «говорящим» названием `models`. Его-то мы и используем для хранения кода моделей.

Если мы откроем этот модуль в текстовом редакторе, то увидим, что он содержит всего одно выражение (комментарии не в счет):

```
from django.db import models
```

Оно выполняет импорт с присоединением модуля `models` из пакета `django.db`. В этом модуле объявлены базовый класс модели `Model` и классы полей, предназначенных для хранения данных различных типов.

Каждая наша модель будет представлять собой класс — потомок класса `Model`. Он будет содержать свойства, соответствующие полям модели и хранящие объекты классов, представляющих поля определенных типов данных.

```
class Category(models.Model):
  name = models.CharField(max_length = 30, unique = True)
  description = models.TextField()
```

Здесь мы создали модель `Category`, хранящую список категорий товаров. Она содержит поле `name`, имеющее строковый тип, максимальную длину хранимого значения в 30 символов и являющееся уникальным, и поле `description` типа memo. Первое поле будет хранить название категории, второе — ее описание.

Как видим, конструкторы классов полей принимают набор именованных параметров, бóльшая часть которых является необязательными. С помощью этих параметров мы указали, в частности, что поле `name` должно хранить строковые значения максимальной длиной в 30 символов и быть уникальным.

Классы полей для различных типов данных

Классы полей Django соответствуют типам данных, поддерживаемым базами данных различных форматов. Так, существуют классы поля строкового типа, поля типа memo, целочисленного поля, поля, хранящего число с плавающей точкой, поля для хранения значения даты и времени, логического поля и пр. Такие типы полей мы можем назвать *простыми*.

Помимо этого, имеются классы полей, представляющие типы, которые не поддерживаются базами данных непосредственно, и для хранения таких данных в таблице создается поле какого-либо из простых типов — например, строкового. Это классы поля, хранящего в себе целый файл, особого поля, предназначенного для хранения графического изображения, поля, хранящего путь к файлу, и др. Назовем такие типы данных *производными*.

Давайте рассмотрим все доступные в Django классы полей и соответствующие им типы данных, сначала — простые, а потом — производные.

Классы полей для простых типов данных

Доступные нам классы полей, соответствующие простым типам данных, перечислены в табл. 5.1.

***Таблица 5.1.** Классы полей для простых типов данных*

Класс поля	Тип данных
`AutoField`	Счетчик
`BigIntegerField`	64-разрядное (длинное) целое число; позволяет хранить значения от -9223372036854775808 до 9223372036854775807
`BooleanField`	Логический
`CharField`	Строковый
`DateField`	Дата
`DateTimeField`	Дата и время
`FloatField`	Число с плавающей точкой
`IntegerField`	32-разрядное (обычное) целое число; позволяет хранить значения от -2147483648 до 2147483647
`PositiveIntegerField`	32-разрядное положительное целое число; позволяет хранить значения от 0 до 2147483647

Таблица 5.1 (окончание)

Класс поля	Тип данных
PositiveSmallIntegerField	16-разрядное (короткое) положительное целое число; позволяет хранить значения от 0 до 32767
SlugField	Короткий заголовок или название, включающее только символы латиницы, цифры, дефисы и символы подчеркивания. Обычно применяются в так называемых SEO-дружественных интернет-адресах
SmallIntegerField	16-разрядное (короткое) целое число; позволяет хранить значения от -32768 до 32767
TextField	Memo
TimeField	Время

Классы полей `PositiveIntegerField` и `PositiveSmallIntegerField` могут хранить лишь положительные значения чисел. Если им присвоить отрицательное значение, оно может быть потеряно.

Класс поля `DateField` позволяет хранить только значение даты, класс `TimeField` — только значение времени, а класс `DateTimeField` — одновременно значения даты и времени.

Классы полей для производных типов данных

Что касается производных типов данных, то им соответствуют классы полей, перечисленные в табл. 5.2.

Таблица 5.2. Классы полей для производных типов данных

Класс поля	Тип данных
EmailField	Адрес электронной почты
IPAddressField	IP-адрес протокола IPv4
GenericIPAddressField	IP-адрес протокола IPv4 или IPv6
URLField	Интернет-адрес

Параметры полей

Как мы уже знаем, класс поля задает тип хранящихся в нем данных. А дополнительные параметры поля мы можем указать, передав их именованными параметрами конструктору данного класса. Настала пора узнать, что это за параметры.

Параметры, применимые для всех типов данных

Многие параметры полей применимы для полей всех типов данных. Они перечислены в табл. 5.3 и являются необязательными.

Таблица 5.3. Параметры, применимые для всех типов данных

Параметр	Описание	Значение по умолчанию
null	Если True, поле может хранить значение NULL (соответствует значению None Python)	False
blank	Если True, поле может хранить пустое значение	False
choices	Указывает список значений, которые может хранить поле; значения вне этого списка ввести в такое поле нельзя	None
db_column	Задает имя поля таблицы, которое будет соответствовать данному полю модели	None
db_index	Если True, на основе этого поля будет создан индекс	False
default	Задает значение поля по умолчанию	None
editable	Если False, данное поле не будет выводиться на страницах добавления и правки записи встроенного административного сайта (см. *главу 4*)	True
help_text	Задает текст дополнительного описания, которое будет выводиться под соответствующим элементом управления	""
primary_key	Если True, данное поле станет ключевым	False
unique	Если True, данное поле станет уникальным	False
unique_for_date	Задает имя поля класса DateField или DateTimeField, после чего данное поле станет уникальным в пределах даты, чье значение будет взято из указанного поля	None
unique_for_month	Задает имя поля класса DateField или DateTimeField, после чего данное поле станет уникальным в пределах определенного месяца даты, чье значение будет взято из указанного поля	None
unique_for_year	Задает имя поля класса DateField или DateTimeField, после чего данное поле станет уникальным в пределах определенного года даты, чье значение будет взято из указанного поля	None
verbose_name	Задает имя поля, которое будет отображаться на страницах встроенного административного сайта	None

Вот пример:

```
class Good(models.Model):
  name = models.CharField(max_length = 50, unique = True, ↳
  verbose_name = "Название")
  . . .
  in_stock = models.BooleanField(default = True, db_index = True, ↳
  verbose_name = "В наличии")
```

Здесь мы создаем модель Good для хранения списка товаров. Поле name, хранящее наименование товара, станет уникальным, а поле in_stock, где записывается признак того, есть ли товар в наличии, получит значение по умолчанию, равное True, и на его основе будет создан индекс. Также мы задаем для обоих полей имена, под которыми они будут выводиться на страницах встроенного административного сайта.

Некоторые параметры требуют особого пояснения.

Так, параметры null и blank обычно устанавливаются в True оба:

```
class Good(models.Model):
  . . .
  category = models.ForeignKey(Category, null = True, blank = True)
```

После чего поле category получит возможность хранить значение None.

Параметр choices позволяет создать поле, в котором можно будет сохранить лишь значение из определенного нами перечня. Такое поле может пригодиться для хранения, например, категории товара, если список доступных категорий задан раз и навсегда. (В противном случае для хранения списка категорий лучше создать модель.)

В качестве значения параметра choices указывается список или кортеж, содержащий перечень доступных значений. Каждый элемент этого списка указывает одно из доступных для занесения в поле значений и, в свою очередь, представляет собой кортеж, первый элемент которого задает идентификатор доступного значения, а второй — само это значение. Отметим при этом, что в поле будет занесен именно идентификатор значения (первый элемент вложенного кортежа), а не само значение.

Вот еще пример:

```
CATEGORIES = (
  (1, "Метлы"),
  (2, "Веники"),
  (3, "Щетки")
)
class Good(models.Model):
  . . .
  category = models.IntegerField(choices = CATEGORIES, default = 1, ↳
  db_index = True)
  . . .
```

Здесь мы создаем список категорий товаров и указываем его в качестве перечня доступных значений для целочисленного поля category модели Good.

Иногда нам может понадобиться сделать так, чтобы какое-то поле имело уникальные значения в течение определенной даты, месяца или года. Скажем, мы решим, что все статьи блога, опубликованные в течение одного дня, должны иметь уникальные заголовки. В таких случаях нам помогут параметры unique_for_date, unique_for_month и unique_for_year:

```
class BlogArticle(models.Model):
  title = models.CharField(unique_for_date = "pubdate")
  pubdate = models.DateField()
  . . .
```

Фактически мы укажем, что комбинации значений полей `title` и `pubdate`, т. е. заголовок статьи и дата ее публикации, должны быть уникальными. И пользователь не сможет в один и тот же день опубликовать в блоге статьи с одинаковыми заголовками — Django этого не допустит.

```
class BlogArticle(models.Model):
  title = models.CharField(unique_for_month = "pubdate")
  pubdate = models.DateField()
  . . .
```

А теперь уникальными значениями должны быть комбинации заголовка статьи и месяца, взятого из даты ее публикации.

Параметры, специфичные для определенных типов данных

Многие из классов полей, поддерживаемых Django, позволяют задать для поля свои собственные, специфичные лишь для него параметры. Их немного.

Класс строкового поля `CharField` поддерживает необязательный параметр `max_length`. Он задает максимальную длину строки, которую можно поместить в поле, эта длина задается в символах. Значение параметра `max_length` по умолчанию — 100.

Классы полей `DateField`, `DateTimeField` и `TimeField` поддерживают два необязательных параметра, которые могут очень нам пригодиться. Эти параметры перечислены в табл. 5.4.

***Таблица 5.4.** Параметры, специфичные для классов полей `DateField`, `DateTimeField` и `TimeField`*

Параметр	Описание	Значение по умолчанию
`auto_now`	Если `True`, в поле будет автоматически заноситься значение сегодняшних даты и (или) времени при добавлении и при каждом сохранении записи	`False`
`auto_now_add`	Если `True`, в поле будет автоматически заноситься значение сегодняшних даты и (или) времени при добавлении записи	`False`

Вот пример:

```
class BlogArticle(models.Model):
  . . .
  updated = models.DateTimeField(auto_now = True)
```

Здесь мы создали в модели статьи блога поле, которое будет хранить дату и время последнего изменения статьи.

Классы полей EmailField, SlugField и URLField также поддерживают параметр max_length. Но в первом случае его значение по умолчанию равно 75 (что, надо сказать, маловато для иных почтовых адресов...), во втором — 50, а в третьем — 200.

Создание связей

Реализовывать связи между моделями (и соответствующими им таблицами базы данных) в Django очень просто. Это делается путем создания полей особых классов.

Для создания связи вида *один-ко-многим* мы объявим во вторичной модели поле класса ForeignKey. В качестве единственного обязательного параметра конструктора такого поля мы укажем класс первичной модели:

```
class Category(models.Model):
  name = models.CharField(max_length = 30, unique = True)

class Good(models.Model):
  name = models.CharField(max_length = 50, unique = True)
  description = models.TextField()
  in_stock = models.BooleanField(default = True, db_index = True)
  category = models.ForeignKey(Category)
```

В модели Good мы создали поле category, которое сформирует связь записи этой модели с соответствующей записью модели Category. Модель Good при этом станет вторичной моделью, а модель Category — первичной.

Класс поля ForeignKey поддерживает очень полезный необязательный параметр on_delete. Он позволит нам указать, что следует предпринять используемой нами СУБД, если запись первичной таблицы, на которую ссылаются записи вторичной таблицы, будет удалена.

В качестве значения данного параметра указывается одна из переменных, объявленных в модуле django.db.models. Таких переменных и соответствующих им доступных значений пять:

- CASCADE — также удалить связанные записи вторичной таблицы (значение параметра on_delete по умолчанию);
- PROTECT — не удалять запись первичной таблицы и сгенерировать исключение (об исключениях базы данных мы поговорим в *главе 11*);
- SET_NULL — записать в соответствующее поле класса ForeignKey связанных записей значение NULL;
- SET_DEFAULT — записать в соответствующее поле класса ForeignKey связанных записей указанное в его параметрах значение по умолчанию.

Значения CASCADE и PROTECT пригодятся нам, если мы создаем так называемую *жесткую связь*, при которой любая запись вторичной таблицы обязательно должна ссылаться на запись первичной. А значения SET_NULL и SET_DEFAULT помогут нам

создать *мягкую связь*, когда запись вторичной таблицы может ссылаться на запись первичной, а может и не ссылаться.

Создаем мягкую связь:

```
class Good(models.Model):
  . . .
  category = models.ForeignKey(Category, null = True, blank = True, ↳
  on_delete = models.SET_NULL)
```

Для создания связи *один-к-одному* применяется класс поля `OneToOneField`. В качестве единственного обязательного параметра он также принимает имя первичной модели:

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

class Child(models.Model):
  parent = models.OneToOneField(Parent)
```

Методы модели

Поскольку модель Django — это обычный класс Python, она может включать в свой состав не только свойства, но и методы. Эти методы могут манипулировать хранящимися в модели данными, скажем, возвращать результат каких-либо вычислений, выполненных на их основе:

```
class Good(models.Model):
  . . .
  in_stock = models.BooleanField(default = True, db_index = True)
  . . .
  def get_is_stock(self):
    if self.in_stock:
      return "+"
    else:
      return ""
```

Здесь мы добавили в модель товара метод, возвращающий строку с символом плюса, если товар есть в наличии, и возвращающий пустую строку в противном случае.

Базовый класс модели `Model` поддерживает метод `__str__`, возвращающий строковое представление модели. Оно выглядит как строка с маловразумительным содержимым вида *<имя модели>* `object` и никак не дает понять, какие данные хранятся в модели.

Мы можем переобъявить метод `__str__`, чтобы он выводил, скажем, название категории или товара с указанием, есть ли этот товар в наличии:

```
class Category(models.Model):
  name = models.CharField(max_length = 30, unique = True)
  def __str__(self):
    return self.name
```

```
class Good(models.Model):
  name = models.CharField(max_length = 50, unique = True)
  description = models.TextField()
  category = models.ForeignKey(Category)
  in_stock = models.BooleanField(default = True, db_index = True)
  def __str__(self):
    s = self.name
    if not self.in_stock:
      s = s + " (нет в наличии)"
    return s
```

Класс `Model` поддерживает также методы `save` и `delete`: первый метод выполняет сохранение записи, а второй ее удаляет. Если нам нужно выполнить какие-либо дополнительные действия при сохранении или удалении записи, мы переопределим нужный метод в классе нашей модели:

```
class Good(models.Model):
  . . .
  def save(self, *args, **kwargs):
    # Выполняем дополнительные действия перед сохранением записи

    # Обязательно вызываем метод save родителя,
    # который, собственно, выполняет сохранение записи.
    # Если мы этого не сделаем, запись не будет сохранена
    super(Good, self).save(*args, **kwargs)

    # Выполняем какие-либо действия после сохранения записи
```

Приведенный код можно использовать как шаблон для написания своих реализаций метода `save`.

А вот и шаблон для переопределения метода `delete`:

```
class Good(models.Model):
  . . .
  def delete(self, *args, **kwargs):
    # Выполняем дополнительные действия перед удалением записи

    # Обязательно вызываем метод delete родителя,
    # иначе запись не будет удалена
    super(Good, self).delete(*args, **kwargs)

    # Выполняем какие-либо действия после удаления записи
```

Надо сказать, что в большинстве случаев необходимости в переопределении методов `save` и `delete` не возникает. Однако если мы собираемся хранить в модели файлы (чем мы и займемся в *главе 13*), скорее всего, без этого не обойтись.

Следует отметить и метод `get_absolute_url`. Он не принимает параметров и возвращает в качестве результата строку с уникальным интернет-адресом данной

записи. Так, для модели `Good` мы можем реализовать в этом методе формирование интернет-адреса страницы со сведениями о товаре.

Прежде чем писать метод `get_absolute_url`, следует задать привязки, чем мы займемся лишь в *главе 6*. Поэтому давайте отложим рассмотрение примера реализации данного метода до *главы 15*, посвященной комментариям Django.

Метаданные модели

Помимо свойств-полей и методов, модели могут содержать так называемые *метаданные* — дополнительные параметры, меняющие внешнее представление и поведение модели. Так, в метаданных мы можем указать порядок сортировки записей в модели по умолчанию и имя, под которым модель будет отображаться на страницах встроенного административного сайта Django.

Метаданные представляют собой класс с именем `Meta`, который объявляется прямо внутри класса модели (*вложенный класс*). Его свойства и задают дополнительные параметры модели.

Полезные для нас свойства класса `Meta` перечислены в табл. 5.5. Их не очень много.

***Таблица 5.5.** Свойства класса `Meta`*

Свойство	Описание
`db_table`	Задает в виде строки имя таблицы, соответствующей модели
`ordering`	Задает поля, по которым будет выполняться сортировка данных
`unique_together`	Задает список полей, значения которых в совокупности должны быть уникальны
`verbose_name`	Задает в виде строки имя, под которым модель будет выводиться на страницах административного сайта
`verbose_name_plural`	То же самое, что `verbose_name`, но во множественном числе

Поля, по которым должна выполняться сортировка, задаются в свойстве `ordering` в виде списка, элементами которого станут имена полей в строковом виде. Записи будут сортироваться сначала по первому из заданных нами полей, потом — по второму и т. д. По умолчанию записи сортируются по возрастанию значения поля, а сортировка по убыванию задается указанием символ минус (-) перед именем поля.

Поля, значения которых в совокупности должны быть уникальны, указываются в свойстве `unique_together` как кортеж. Каждый элемент этого кортежа должен представлять собой строку с именем поля:

```
class Good(models.Model):
  . . .
  class Meta:
    ordering = ["-price", "name"]
```

```
unique_together = ("category", "name", "price")
verbose_name = "товар"
verbose_name_plural = "товары"
```

Записи модели `Good` будут по умолчанию отсортированы по полям `price` (по убыванию значения) и `name`. А комбинация значений полей `category`, `name` и `price` в каждой записи должна быть уникальной.

Структуры, создаваемые Django в базе данных

После создания моделей мы выполним синхронизацию с базой данных (см. *главу 4*). В результате Django создаст в базе все необходимые для хранения данных структуры: таблицы, индексы и связи.

- Для каждой модели будет создана таблица, имя которой:
  - либо задастся значением свойства `db_table` метаданных модели,
  - либо, если это свойство не объявлено, получит имя вида *<имя приложения>_<имя модели>*.
- Для каждого поля модели будет создано соответствующее ему поле таблицы. Имя этого поля:
  - либо будет задано значением параметра `db_column`,
  - либо, если такого параметра нет, совпадет с именем поля модели.
- В каждой таблице сформируется ключевое поле.
  - Либо это будет поле, для параметра `primary_key` которого мы указали значение `True`,
  - либо, если мы не указали ни одного поля модели в качестве ключевого, Django сама создаст ключевое поле `id` типа счетчика.
- Для каждого поля, параметру `db_index` которого мы указали значение `True`, будет создан индекс. Для полей типа `SlugField` индекс создается автоматически.
- Для поля модели, реализующего связь между таблицами, будет создано поле таблицы с именем вида *<имя первичной модели>*`_id` (если, конечно, мы не указали другое его имя с помощью параметра `db_column`). Это поле получит целочисленный тип, и по нему будет создан индекс.

Кстати, мы можем посмотреть описание всех структур данных, что Django создаст в базе, выполненное на языке *SQL* (Structured Query Language, язык структурированных запросов). Для этого достаточно запустить командную строку Windows, перейти в папку проекта и набрать команду вида:

```
manage.py sql <имя приложения>
```

Синхронизация с базой данных: некоторые нюансы

Что ж, наши модели готовы. Проверим, присутствует ли наше приложение в списке активных (за подробностями — к *главе 4*), — в противном случае Django не «уви-

дит» созданные в нем модели. И обязательно сохраним все исправленные файлы. После чего выполним синхронизацию с базой данных.

Повторная синхронизация моделей выполняется так же, как и первая, описанная в *главе 4*. Однако здесь есть некоторые нюансы, способные доставить немало проблем.

Предположим, мы создали какую-либо модель и выполнили синхронизацию, после чего в базе были созданы соответствующие таблицы, индексы и связи. Далее мы добавили в эту модель новое поле и выполнили синхронизацию еще раз. Казалось бы, в таблицу должно было добавиться новое поле, представляющее созданное нами поле модели. Но — увы! — это не так.

При повторной синхронизации Django лишь проверяет, существует ли в базе соответствующая модели таблица, но не просматривает список ее полей на предмет изменений их состава. Следовательно, новое поле в таблице создано не будет, и, обратившись к нему в программном коде, мы получим сообщение об ошибке.

В таких случаях нам самим придется вносить изменения в соответствующие таблицы базы данных: добавлять поля, править их и удалять. Для этого можно использовать любую программу для работы с базами SQLite. Автор предпочитает пользоваться программой Database Master, которую можно найти на Web-сайте **http://www.nucleonsoftware.com/**.

Работа с моделью во встроенном административном Web-сайте

В *главе 4* мы работали со списками групп и пользователей, которые ведутся встроенным приложением разграничения доступа административного Web-сайта Django. Этот удобный инструмент позволяет нам просматривать содержимое моделей, добавлять в них новые записи, а также править и удалять уже существующие.

Однако вновь созданные нами модели в этом сайте по умолчанию не выводятся. Нам придется явно «попросить» административный сайт вывести их в списке присутствующих в проекте моделей.

В пакете приложения находится модуль `admin`, в котором пишется код, управляющий административным сайтом. В том числе там указывается, какие модели приложения должны выводиться на этом сайте.

Откроем модуль `admin` в текстовом редакторе. Там мы увидим одно-единственное выражение, импортирующее с присоединением модуль `django.contrib.admin`:

```
from django.contrib import admin
```

Добавим ниже выражение, которое импортирует созданные нами модели:

```
from page.models import Category, Good
```

где `page` — это имя приложения, созданного автором в тестовых целях.

После чего вызовем метод `django.contrib.admin.site.register`, передав ему класс модели, которую нужно вывести на административном сайте. Отметим, что это именно класс, а не его имя в виде строки:

```
admin.site.register(Category)
admin.site.register(Good)
```

Сохраним файл и обновим страницу списка моделей административного сайта. Вуаля — теперь наши модели здесь есть!

Извлечение данных из моделей

Итак, мы создали все модели приложения и наполнили их отладочными данными. Чтобы благополучно написать контроллер (а мы займемся этим уже в *главе 6*), который выведет эти данные на Web-страницу, нам нужно выяснить, как извлечь их из модели.

Доступ ко всем записям модели

Получить все записи, что хранятся в модели, очень просто. Сначала мы обращаемся к свойству `objects`, которое поддерживается всеми моделями (поскольку объявлено в базовом классе `Model`) и хранит особый объект — *диспетчер записей*. У этого диспетчера мы вызываем не принимающий параметров метод `all` — и он вернет нам *список записей* в виде объекта класса `QuerySet`:

```
from page.models import Category, Good
qs = Good.objects.all()
```

Мы можем получить любой элемент списка записей таким же способом, который применяли в *главе 2* для получения элементов обычного списка Python:

```
q1 = qs[0]
```

Мы можем выполнить обработку списка записей в знакомом нам цикле по списку:

```
s = ""
for q in qs:
  if s != "":
    s = s + ", "
  s = s + q.name
```

Мы можем получить подмножество записей из списка:

```
qq = ds[0:5]
```

Здесь мы получаем новый список, содержащий первые пять записей из модели `Good`.

Объект-диспетчер также поддерживает метод `count`. Он не принимает параметров и возвращает количество записей в модели:

```
n = Good.objects.count()
```

Доступ к полям записи

Значения отдельных полей записи доступны через соответствующие свойства модели:

```
name = q1.name
desc = q1.description
```

Здесь мы получаем название и описание первого товара из списка.

Поле, ответственное за создание связи с первичной моделью, хранит объект связанной записи этой модели. Мы можем получить доступ к полям этой записи тем же способом:

```
category = q1.category
cat_name = category.name
```

Здесь мы получаем категорию первого товара и ее название.

Если нам потребуется из записи первичной модели получить доступ ко всем связанным записям вторичной модели, мы обратимся к особому свойству этой записи, что имеет имя вида *<имя вторичной модели>*_set. Оно хранит объект — диспетчер связанных записей вторичной модели:

```
goods = category.good_set.all()
good1 = goods[0]
good1_name = good1.name
```

Еще одно полезное свойство поддерживается всеми моделями. Оно имеет имя pk и хранит целочисленный идентификатор записи — по сути это значение ключевого поля соответствующей таблицы:

```
id = q1.pk
```

Если мы явно не указали ключевого поля в объявлении класса модели, Django сама создаст ключевое поле с именем id. Так что для получения идентификатора записи мы можем обратиться напрямую к этому полю:

```
id = q1.id
```

Программисты со стажем часто так и поступают.

Впоследствии мы можем использовать значение идентификатора, чтобы сразу извлечь из модели нужную нам запись. Как это делается, мы узнаем ближе к концу главы.

Фильтрация записей

Искать нужную запись путем перебора списка всех хранящихся в модели записей — крайне долгая процедура. Поэтому Django предоставляет нам развитые средства для фильтрации записей по различным критериям.

Прежде всего, это метод filter, поддерживаемый классом диспетчера записей. Он принимает набор именованных параметров, с помощью которых указывается кри-

терий фильтрации, и возвращает уже знакомый нам объект класса `QuerySet` — список найденных записей:

```
goods_in_stock = Good.objects.filter(in_stock = True)
```

Здесь мы выбираем все записи модели `Good`, в поле `in_stock` которых хранится значение `True`, т. е. все товары, что имеются в наличии.

```
good1_name = goods_in_stock[0].name
```

Здесь мы получаем название первого товара из имеющихся в наличии.

Как задаются критерии фильтрации, в общем, понятно. Мы присваиваем параметру, чье имя совпадает с именем поля модели, значение и в результате получаем все записи, в которых это поле хранит указанное нами значение.

Мы можем провести фильтрацию записей вторичной модели по значению поля связанной с ней первичной модели. Имя параметра будет иметь такой вид: *<имя поля вторичной модели, используемое для создания связи>*__*<имя поля первичной модели, по которому выполняется фильтрация>*:

```
goods = Good.objects.filter(category__name = "Метлы")
```

Здесь мы получаем все товары, относящиеся к категории «Метлы».

В первичной модели мы можем выбрать лишь те записи, с которыми связаны записи вторичной модели, или, наоборот, записи, с которыми не связана ни одна запись. Для этого мы применим параметр вида *<имя вторичной модели, набранное прописными буквами>*__`null` и присвоим ему значение `False` или `True` соответственно:

```
empty_cats = Category.objects.filter(good__isnull = True)
```

Здесь мы выбираем категории, к которым не принадлежит ни один товар.

По умолчанию при фильтрации будут выбраны записи, значение поля которых равно указанной нами величине. Однако мы можем изменить это поведение, использовав параметры с именами вида *<имя параметра>*__*<модификатор>*, где *модификатор* как раз и предписывает вариант поведения Django при фильтрации записей:

```
supergoods = Good.objects.filter(name__startswith = "Супер")
```

Здесь мы отбираем товары, чье название начинается со строки `Супер`.

Все доступные в Django модификаторы перечислены в табл. 5.6.

Таблица 5.6. Модификаторы фильтрации

Модификатор	Описание	Тип данных
`exact`	Значение поля должно быть равно указанному (это поведение по умолчанию, так что данный модификатор можно не указывать)	Любой
`in`	Значение поля должно быть равно одному из указанных. Сравниваемые значения задаются в виде списка	
`isnull`	Поле должно (`True`) или не должно (`False`) содержать какое-либо значение	

Таблица 5.6 (окончание)

Модификатор	Описание	Тип данных
iexact	То же самое, что exact, но без учета регистра символов	Строка
contains	Значение поля должно содержать указанное нами	
icontains	То же самое, что contains, но без учета регистра символов	
startswith	Значение поля должно начинаться с указанного нами	
istartswith	То же самое, что startswith, но без учета регистра символов	
endswith	Значение поля должно заканчиваться указанным нами	
iendswith	То же самое, что endswith, но без учета регистра символов	
gt	Значение поля должно быть больше указанного	Число
gte	Значение поля должно быть больше или равно указанному	
lt	Значение поля должно быть меньше указанного	
lte	Значение поля должно быть меньше или равно указанному	
range	Значение поля должно укладываться в указанный нами диапазон, который задается в виде кортежа	Число, дата, время, дата и время
year	Значение поля должно принадлежать к указанному нами году	Дата, дата и время
month	Значение поля должно принадлежать к указанному нами месяцу	
day	Значение поля должно принадлежать к указанному нами числу	
week_day	Значение поля должно принадлежать к указанному нами дню недели, который задается числом от 1 (воскресенье) до 7 (суббота)	
hour	Значение поля должно принадлежать к указанному нами часу	Время, дата и время
minute	Значение поля должно принадлежать к указанной нами минуте	
second	Значение поля должно принадлежать к указанной нами секунде	

Вот примеры:

```
cheap_goods = Good.objects.filter(price__lte = 100)
```

Здесь мы выбираем товары, которые стоят 100 руб. и дешевле.

```
from datetime import date
blog_articles = BlogArticle.objects.filter(pubdate__range = ↵
(date(2014, 1, 1), date(2014, 1, 31)))
```

А здесь выбираем статьи блога, опубликованные в промежутке с 1 по 31 января 2014 года. (Класс `date`, хранящий значения даты, объявлен в модуле `datetime`, который мы должны предварительно импортировать.)

```
blog_atricles = BlogArticle.objects.filter(pubdate__year = 2013)
```

Здесь мы выбираем статьи блога, опубликованные в 2013 году.

```
goods = Good.objects.filter(category__name__in = ["Метлы", "Щетки"])
```

А здесь — все метлы и щетки.

Если мы укажем в методе `filter` сразу несколько параметров-критериев, Django будет отбирать только те записи, что будут удовлетворять им всем. Программисты в таких случаях говорят, что критерии фильтрации объединяются по правилам *логического И*:

```
goods = Good.objects.filter(category__name__in = ["Метлы", "Щетки"], ↪
in_stock = True)
```

Здесь мы выбираем только те метлы и щетки, что есть в наличии.

Мы можем сравнить значение одного поля со значением другого. Для этого мы используем класс `django.db.models.F`:

```
from django.db.models import F
```

В качестве значения поля в критерии фильтрации мы укажем конструктор этого класса, которому передадим в качестве единственного параметра строку с именем сравниваемого поля:

```
blog_articles = BlogArticle.objects.filter(pubdate__lt = F("savedate"))
```

Здесь мы отбираем статьи блога, значение поля `savedate` которых больше значения поля `pubdate`, т. е. статьи, которые правились после публикации.

Класс диспетчера записей поддерживает также метод `exclude`. Он принимает те же параметры и возвращает тот же результат, что метод `filter`, но выполняет противоположную задачу — исключает из изначального набора записи, удовлетворяющие заданному критерию:

```
blog_articles = BlogArticle.objects.exclude(pubdate__lt = F("savedate"))
```

Здесь мы исключаем из набора статьи блога, что правились после публикации.

```
cheap_goods = Good.objects.filter(price__lte = 100).exclude(in_stock = ↪
False)
```

А здесь выбираем товары, которые стоят 100 руб. и дешевле, после чего исключаем те из них, что отсутствуют в наличии.

Однако часто приходится искать по более сложным критериям. Скажем, мы можем захотеть найти все статьи блога, опубликованные за 2014 год или имеющие в заголовке слово «Django». (Кстати, в таких случаях говорят, что критерии фильтрации объединяются по правилам *логического ИЛИ*.)

В таких случаях нам понадобится класс `Q`, объявленный в модуле `django.db.models`:

```
from django.db.models import Q
```

Мы передадим методу `filter` или `exclude` набор критериев фильтрации, представляющих собой конструкторы объектов данного класса, которые связаны между со-

бой логическими операторами. В качестве единственного параметра каждого такого конструктора мы укажем критерий фильтрации в уже знакомом нам виде. Что касается логических операторов, то они аналогичны операторам Python (см. табл. 2.6), но записываются по-другому: `&` аналогичен оператору `and`, `|` — `or`, а `~` — `not`. Для группировки критериев мы можем использовать круглые скобки:

```
blog_articles = BlogArticle.objects.filter(~Q(pubdate__range = ↵
(date(2014, 1, 1), date(2014, 1, 31))))
```

Здесь мы выбираем статьи блога, дата публикации которых не укладывается в промежуток с 1 по 31 января 2014 года.

```
blog_atricles = BlogArticle.objects.filter(Q(pubdate__year = 2014) | ↵
Q(title__contains = "Django"))
```

А здесь — статьи блога, либо опубликованные в 2014 году, либо содержащие в заголовке слово «Django».

Сортировка записей

Для сортировки записей мы можем использовать метод `order_by`, поддерживаемый классом диспетчера записей. Он принимает произвольное количество параметров — строк с именами полей, по которым должна выполняться сортировка. Записи будут сортироваться сначала по первому из указанных полей, потом — по второму и т. д. По умолчанию записи сортируются по возрастанию значения поля, а чтобы указать сортировку по убыванию, нужно перед именем поля поставить символ минус (-).

```
goods = Good.objects.order_by("name")
```

Здесь мы сортируем товары по их названиям.

```
goods = Good.objects.order_by("category__name", "-price", "name")
```

А здесь — товары сначала по названию категории, потом по убыванию цены и, наконец, по названию самого товара.

Не принимающий параметров метод `reverse` диспетчера записей возвращает список записей, порядок сортировки которых противоположен по отношению к изначальному:

```
goods = goods.reverse()
```

Агрегатные функции

Агрегатные функции Django позволяют выполнить какие-либо математические операции над значениями указанного нами поля всех записей списка. Мы можем подсчитать сумму значений какого-либо поля, их арифметическое среднее и пр.

Для вызова агрегатных функций применяется метод `aggregate`, поддерживаемый как диспетчером записей, так и списком записей `QuerySet`. Он принимает произвольное количество параметров, которыми должны быть объекты особых классов,

собственно, и реализующие агрегатные функции. Эти классы и форматы вызовов их конструкторов описаны в табл. 5.7. Все они объявлены в модуле `django.db.models`.

***Таблица 5.7.** Классы агрегатных функций*

Класс	Описание
`Count(`*<поле>*`, `*<только уникальные значения>*`)`	Количество значений параметра *поле* (фактически количество записей). Если необязательный параметр *только уникальные значения* равен `True`, учитываются только уникальные значения (значение по умолчанию — `False`)
`Avg(`*<поле>*`)`	Среднее арифметическое значений параметра *поле*
`Sum(`*<поле>*`)`	Сумма значений параметра *поле*
`Max(`*<поле>*`)`	Максимальное из значений параметра *поле*
`Min(`*<поле>*`)`	Минимальное из значений параметра *поле*
`StdDev(`*<поле>*`, `*<отклонение выборки>*`)`	Стандартное отклонение значений параметра *поле*. Если необязательный параметр *отклонение выборки* равен `True`, рассчитывается стандартное отклонение выборки (значение по умолчанию — `False`)
`Variance(`*<поле>*`, `*<дисперсия выборки>*`)`	Дисперсия значений параметра *поле*. Если необязательный параметр *отклонение выборки* равен `True`, рассчитывается дисперсия выборки (значение по умолчанию — `False`)

Метод `aggregate` возвращает словарь, каждый элемент которого хранит значение, полученное в результате выполнения одной из переданных ему агрегатных функций. Имена ключей этого словаря формируются в формате *<имя поля, по которому выполнялось вычисление>__<имя класса агрегатной функции, набранное прописными буквами>*, а их значения являются собственно результатом вычисления этих функций:

```
ad = Good.objects.aggregate(models.Avg("price"), models.Min("price"))
avg_price = ad["price__avg"]
min_price = ad["price__min"]
```

Здесь мы получаем среднюю и минимальную цену товара.

Поиск нужной записи

Обрабатывать целые списки записей мы будем далеко не всегда. Часто нам придется выбирать из списка какую-то одну интересующую нас запись. Конечно, это можно сделать с применением метода `filter`, указав соответствующие критерии, но есть способ лучше. Это метод `get`. Как и `filter`, он принимает в качестве параметров критерии фильтрации и возвращает объект самой найденной записи:

```
good = Good.objects.get(name = "Суперметла")
```

Здесь мы получаем товар «Суперметла».

Если мы имеем уникальный идентификатор записи, то можем выполнить поиск по знакомому нам полю `pk`, чтобы получить эту запись:

```
good = Good.objects.get(pk = 2)
```

Как правило, так мы и будем делать.

Вместо поля `pk` мы можем напрямую обратиться к полю `id`. Как говорилось ранее, это поле создается самой Django, если мы не указали ключевое поле в объявлении класса модели явно:

```
good = Good.objects.get(id = 2)
```

Если записи, удовлетворяющей указанному критерию, в модели нет, будет сгенерировано исключение `DoesNotExist`. Если же будет найдено сразу несколько подходящих записей, генерируется исключение `MultipleObjectsReturned`. Оба этих исключения унаследованы классом модели от базового класса `Model`:

```
try:
  good = Good.objects.get(pk = 2)
except Good.DoesNotExist:
  # Записи с таким идентификатором нет
except Good.MultipleObjectsReturned:
  # Есть несколько записей с таким идентификатором
  # Что странно...
else:
  # Запись с таким идентификатором найдена
```

Прочие возможности по выборке записей из моделей

Осталось рассмотреть еще несколько методов, поддерживаемых списком записей `QuerySet`, которые могут нам пригодиться. Все они собраны в табл. 5.8.

***Таблица 5.8.** Полезные методы класса `QuerySet`*

Метод	Описание
`exists()`	Возвращает `True`, если набор содержит записи, и `False` в противном случае
`first()`	Возвращает объект первой записи списка или `None`, если список пуст
`last()`	Возвращает объект последней записи списка или `None`, если список пуст
`earliest(`*<поле>*`)`	Возвращает объект записи с наименьшим значением параметра *поле*, которое должно иметь тип даты или даты и времени
`latest(`*<поле>*`)`	Возвращает объект записи с наибольшим значением параметра *поле*, которое должно иметь тип даты или даты и времени
`distinct(`[*<список имен полей, разделенных запятыми>*]`)`	Возвращает набор, включающий лишь уникальные записи. Если не указан *список полей*, в набор включаются записи, в которых значения всех полей являются уникальными. Если список полей присутствует, в набор включаются записи, в которых уникальными являются лишь значения указанных нами полей

Вот примеры:

```
if Good.objects.filter(pk = 2).exists():
  good = Good.objects.get(pk = 2)
```

Здесь мы проверяем, есть ли в модели запись с заданным нами идентификатором, и, если есть, извлекаем ее.

```
first_good = Good.objects.filter(in_stock = False).first()
```

Здесь мы получаем первый товар из тех, что отсутствуют в наличии.

```
blogs = Blog.objects.all().distinct()
```

А здесь — набор уникальных статей блога.

```
blogs = Blog.objects.all().distinct("pub_date", "title")
```

Здесь мы получаем набор статей блога, в которых уникальными являются лишь дата публикации и заголовок.

> ***Случаи дублирования записей***
>
> Вообще-то, каждый набор записей уже включает в себя лишь уникальные записи. Однако в некоторых случаях, в частности при выполнении поиска по тегам (об это будет рассказано в *главе 18*), записи, включенные в набор, могут дублироваться. В таких-то случаях нам и пригодится метод `distinct`.

Что дальше?

В этой главе мы учились создавать модели и извлекать из них данные с применением фильтрации и сортировки. Возможности Django в этом плане впечатляют, не так ли?

Теперь можно приступать к написанию контроллеров, которые будут извлекать и выводить данные. Чем мы и займемся в следующей главе.

ГЛАВА 6

Контроллеры Django. Регулярные выражения

В предыдущей главе мы занимались созданием моделей, которые будут хранить данные нашего сайта, и изучали инструменты Django для извлечения из моделей нужных нам записей, их фильтрации и сортировки. В общем, готовились к написанию нашего первого контроллера.

В этой главе мы приступим к его написанию. А заодно рассмотрим средства для привязки контроллера к интернет-адресу и, по ходу дела, регулярные выражения.

Регулярные выражения

Точнее, начнем мы с регулярных выражений. Поскольку в таком деле, как указание интернет-адресов и, особенно, передаваемых контроллерам параметров, без них буквально никуда.

Регулярное выражение — это правило, задающее критерии поиска нужного фрагмента в исходной строке и, возможно, манипуляции с найденным фрагментом. Оно включает собственно искомые символы и некоторые знаки и их последовательности, имеющие специальное значение и называемые *литералами*. Литералы регулярных выражений либо обозначают какое-либо подмножество искомых символов, либо действие, которое нужно выполнить над соседними указанными в выражении символами.

Все доступные в регулярных выражениях Python литералы перечислены в табл. 6.1.

Таблица 6.1. *Литералы регулярных выражений*

Литерал	Описание
.	Любой символ
\w	Буква, цифра или подчеркивание
\W	Не буква, не цифра и не подчеркивание
\d	Цифра

Таблица 6.1 (окончание)

Литерал	Описание
\D	Не цифра
\s	Пробельный символ (пробел, табуляция, возврат каретки или перевод строки)
\S	Не пробельный символ
\b	Начало или конец слова
\B	Не начало и не конец слова
^ и \A	Начало строки
$ и \Z	Конец строки
[*<набор>*]	Любой символ из *набора*
<A>|*<B>*	Либо символ *A*, либо символ *B*
*	Предыдущий символ может присутствовать произвольное количество раз, а может и не присутствовать вообще
+	Предыдущий символ должен присутствовать в строке как минимум один раз
?	Предыдущий символ должен либо не присутствовать в строке, либо присутствовать один раз
{*<n>*}	Предыдущий символ должен присутствовать в строке строго *n* раз
{*<m>*,*<n>*}	Предыдущий символ должен присутствовать в строке от *m* до *n* раз

Если нам потребуется найти в строке символ, совпадающий с литералом, мы включим его в регулярное выражение, предварив обратным слешем (\). Например, регулярное выражение \? найдет все символы вопросительного знака.

В табл. 6.2 приведены несколько примеров регулярных выражений.

***Таблица 6.2.** Примеры регулярных выражений*

Выражение	Совпадает со строками	Не совпадает со строками
^good$	good	goods, goodies, verygood
[abc]+	abc, b, cdabc	hgt, o, abcde
\bPy\B	Python, Pyth, Pyt	Py, thonPy
\?|&id=\d{1,3}	?id=123, &id=4	id=89, id=uyt, m=89

В регулярных выражениях мы можем создавать группы символов. Они обрабатываются как единое целое, а совпадающие с ними фрагменты строки впоследствии могут быть извлечены для обработки. Форматы описания различных групп регулярных выражений можно увидеть в табл. 6.3.

Таблица 6.3. Форматы описания групп

Формат группы	Описание
(<*подстрока*>)	Обычная группа, которая должна совпадать с параметром *подстрока*. Совпадающая с этой группой подстрока может быть извлечена обращением по порядковому номеру группы
(P<<*имя*>><*подстрока*>)	Обычная группа, но совпадающее значение может быть извлечено путем обращения к *имени* группы
(?P=<*имя*>)	Задает последовательность, совпадающую со значением группы с указанным параметром *имя*
(?=<*подстрока*>)	Группа, чьим значением станет фрагмент, за которым следует *подстрока*
(?!<*подстрока*>)	Группа, чьим значением станет фрагмент, за которым не следует *подстрока*
(?<=<*подстрока*>)	Группа, чьим значением станет фрагмент, перед которым следует *подстрока*
(?<!<*подстрока*>)	Группа, чьим значением станет фрагмент, перед которым не следует *подстрока*
(?:<*подстрока*>)	Совпадающее с этой группой значение не может быть извлечено. Применяется лишь для формирования подстроки и обрабатывается быстрее обычных групп

Давайте рассмотрим несколько примеров использования групп, чтобы лучше понять механику их работы. Эти примеры приведены в табл. 6.4.

Таблица 6.4. Примеры использования групп

Выражение	Исходная строка	Значение группы
`id=(\d+)`	id=345	345
	id=uyt	None
`(?<=id=)\d+`	id=345	345
	id=uyt	None
	id=345	None
`(?<!cat)=\d+`	cat=345	None
	cat=uyt	None
	id=345	=345

В Python регулярные выражения записываются в виде строк, предваряемых буквой `r`. Отметим, что эта буква ставится перед открывающей кавычкой.

```
re = r"id=(\d+)"
```

Онлайновые тестировщики регулярных выражений

Написать работающее регулярное выражение, особенно сложное, сходу очень трудно. Нам придется многократно тестировать регулярные выражения, для чего можно применять онлайновые сервисы-тестировщики — например, популярный сервис RegExr, доступный по адресу **http://www.regexr.com/**.

Привязка интернет-адресов

Прежде чем приступить к созданию контроллеров, нам сначала следует узнать, как и где выполняется привязка интернет-адресов к приложениям и их отдельным контроллерам. По-хорошему, именно с этого нужно начинать разработку любого контроллера.

Привязка к приложениям

Еще в *главе 3* говорилось, что привязка интернет-адресов к приложениям выполняется на уровне проекта — в модуле `urls` пакета проекта. Поэтому сразу же откроем этот модуль.

Вот первые три выражения, что мы здесь увидим:

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
admin.autodiscover()
```

Они импортируют служебные функции `patterns`, `include` и `url` из модуля `django.conf.urls` и пакет приложения встроенного административного сайта `admin` из пакета `django.contrib` и инициализируют административный сайт. Нам они не очень интересны.

А вот следующее выражение как раз и задает привязку интернет-адресов к приложениям (комментарии удалены):

```
urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
)
```

Здесь мы видим переменную `urlpatterns`, чьим значением является результат, возвращаемый служебной функцией `patterns`. А единственным параметром этой функции служит кортеж, элементы которого, за исключением первого (который должен содержать пустую строку), задают отдельные привязки.

Каждый элемент кортежа представляет собой результат, возвращенный служебной функцией `url`. А параметры у этой функции следующие:

- регулярное выражение, указывающее виртуальную папку, к которой привязывается приложение;
- результат, возвращенный служебной функцией `include`. Этой функции передается строка с полным именем модуля соответствующего приложения, в котором указывается привязка интернет-адресов на уровне приложения (т. е. к отдельным его контроллерам).

Сейчас кортеж включает всего одну привязку (один элемент) — встроенного приложения admin, реализующего работу встроенного административного сайта Django, к виртуальной папке admin. Имя модуля этого встроенного приложения хранится в переменной admin.site.urls.

Давайте внесем в перечень еще одну привязку — виртуальной папки goods к созданному ранее приложению списка товаров (у автора оно имеет имя page). Сделаем это по аналогии с уже имеющейся привязкой.

```
urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^goods/', include("page.urls")),
)
```

Теперь при обращении к интернет-адресу **goods/** будет задействовано приложение page.

Не забываем после каждой значительной правки сохранять измененные файлы. В противном случае наши исправления не возымеют эффект.

Если мы теперь попытаемся проверить наш сайт в работе, даже если попробуем зайти на встроенный административный сайт, то получим сообщение об ошибке. Оно и понятно — ведь мы еще не создали модуль urls приложения.

Привязка к контроллерам приложения

В той же *главе 3* говорилось, что привязка интернет-адресов к контроллерам приложения выполняется на уровне приложения — в модуле urls его пакета. Он не формируется автоматически, так что нам придется создать его самим. Создадим этот модуль, но пока пустой, не содержащий никакого кода.

Привязка простых интернет-адресов

Начнем с рассмотрения привязки к контроллерам простых, не содержащих параметров, интернет-адресов.

Запишем в только что созданный модуль urls пакета приложения следующий код:

```
from django.conf.urls import patterns, url
from page import views
urlpatterns = patterns('',
    url(r'^$', views.index, name = "index"),
    url(r'^good/$', views.good, name = "good"),
)
```

Здесь мы сначала импортируем из модуля django.conf.urls уже знакомые нам служебные функции patterns и url. Далее, поскольку мы будем ссылаться на функции, которые объявлены в модуле views и, собственно, являются контроллерами приложения, импортируем и этот модуль.

Привязки интернет-адресов к контроллерам приложения, как мы видим, задаются так же, как привязки адресов к самим приложениям. Только формат вызова функции `url` здесь другой:

- первым параметром передается регулярное выражение, обозначающее виртуальную подпапку папки, что мы ранее привязали к данному приложению;
- вторым параметром передается функция-контроллер, которая будет вызвана при обращении к этой подпапке приложения;
- необязательным параметром `name` указывается имя, под которым на данную привязку можно будет сослаться из шаблона, — имя привязки (о шаблонах мы будем говорить в *главах 7* и *8*).

> ***Имена привязок***
> Имена привязок должны быть уникальными в пределах всего проекта.

Сначала мы привязали к функции-контроллеру `index` из модуля `views` приложения `page` интернет-адрес, заданный регулярным выражением `^$`. Так мы обозначили, что данная функция должна выполниться при обращении к папке, что была привязана на уровне проекта к самому приложению, — к папке goods. А сама функция `views.index` будет выводить на экран список товаров.

А второй элемент кортежа привязывает к функции-контроллеру `views.good` интернет-адрес, заданный регулярным выражением `^good/$`, — папку good, что находится в папке goods. Данная функция выведет на экран сведения о выбранном посетителем товаре.

Указание в интернет-адресах параметров, передаваемых контроллеру

Обычно товары в списках отображаются по категориям. Посетитель заходит в раздел списка товаров, выбирает категорию и смотрит, какие товары в ней есть. Если же категория не была выбрана явно, выводится какая-либо из категорий на усмотрение разработчиков сайта — например, первая в списке.

Чтобы вывести содержимое выбранной посетителем категории, нам потребуется каким-то образом передать ее идентификатор соответствующему контроллеру. Кроме того, чтобы показать посетителю подробные сведения о выбранном в списке товаре, нам нужно передать в контроллер идентификатор этого товара. Как все это сделать?

Проще всего передать параметры методом GET, добавив их в конец интернет-адреса. Также можно включить их в состав самого интернет-адреса, сделав идентификатор категории или товара его частью, своего рода виртуальной подпапкой папки, привязанной к контроллеру. Мы рассмотрим оба способа.

Метод GET требует, чтобы параметры передавались в виде пар формата *<имя параметра>=<значение параметра>*. При этом параметры ставятся в конце интернет-адреса, отделяются от него вопросительным знаком (`?`) и разделяются символом амперсанда (`&`).

/goods/?id=2&sort=name

Здесь мы передали контроллеру параметры `id` и `sort` со значениями `2` и `"name"` соответственно.

Чтобы контроллер успешно принял параметры, мы должны указать их в привязке. При этом часть регулярного выражения, которая соответствует значению параметра, мы поместим в группу — чтобы контроллер потом смог получить это значение.

Исправим в модуле `urls` пакета приложения код, описывающий привязку, следующим образом:

```
urlpatterns = patterns('',
  url(r'^(?:\?id=(?P<id>\d+))?$', views.index, name = "index"),
  url(r'^good/\?id=(?P<id>\d+)$', views.good, name = "good"),
)
```

После этого контроллер `views.index` получит параметр `id`, задающий идентификатор выбранной посетителем категории, а контроллер `views.good` — параметр `id`, чьим значением будет идентификатор товара. Тогда, набрав интернет-адрес **goods/?id=2**, мы получим список товаров из категории с идентификатором 2, а набрав **goods/good/?id=45** — описание товара с идентификатором 45.

Отметим, что в первой привязке мы указали группу, извлекающую значение параметра `id` (идентификатора категории), как необязательную, т. е. этот параметр может как присутствовать в интернет-адресе, так и отсутствовать в нем. Впоследствии в контроллере мы сделаем так, чтобы посетитель, набрав **goods/** (без указания данного параметра), получил список товаров, относящихся к первой категории в списке.

> ***Именованные группы***
>
> Здесь мы задействовали группы регулярных выражений с указанием их имен. Однако ничто не мешает нам использовать обычные группы, где имена не указываются, а для доступа к извлеченным с помощью групп значениям служат порядковые номера. Просто с именованными группами проще.

В последнее время популярность приобретает второй способ передачи параметров — включение их в состав самого интернет-адреса. Здесь параметр указывается в виде виртуальной подпапки, вложенной в папку, что была привязана к контроллеру. Например, сформированный таким способом интернет-адрес **goods/2/** ссылается на категорию с идентификатором 2, а интернет-адрес **goods/good/45/** — на товар с идентификатором 45.

Реализуем такой способ передачи параметров. Для чего снова исправим описывающий привязку код, находящийся в модуле `urls` пакета приложения.

```
urlpatterns = patterns('',
  url(r'^(?:(?P<id>\d+)/)?$', views.index, name = "index"),
  url(r'^good/(?P<id>\d+)/$', views.good, name = "good"),
)
```

И здесь мы в первой привязке указали, что идентификатор категории является необязательным.

Какой способ передачи параметров контроллеру выбрать, решаем мы сами. Первый несколько привычнее для программистов со стажем, зато второй порождает более короткие и легкие для понимания интернет-адреса.

Давайте примем за основу второй способ. Мы уже написали реализующий его код привязки — не переделывать же его!

Создание контроллеров

Разобравшись с регулярными выражениями и привязкой интернет-адресов к контроллерам, мы можем приступить собственно к реализации контроллеров.

Если контроллеров не слишком много, их код можно поместить в модуль `views` пакета приложения, что любезно создала для нас Django. В противном случае лучше разнести контроллеры по нескольким разным модулям.

Откроем модуль `views` пакета приложения. Если не принимать во внимание комментарии, хранящийся в нем код исчерпывается единственным выражением:

```
from django.shortcuts import render
```

Оно импортирует функцию `render` из модуля `django.shortcuts`. Функция `render` пригодится нам потом, в *главах 7* и *8*, когда мы начнем разрабатывать и выводить на экран шаблоны, сейчас же она нам совершенно бесполезна.

Контроллер Django представляет собой обычную функцию Python (*функция-контроллер*). Эта функция должна принимать один обязательный параметр — объект класса `HttpRequest`, хранящий сведения о полученном запросе.

Среди свойств класса `HttpRequest` можно выделить следующие:

- `GET` — список параметров, полученных методом GET;
- `POST` — список параметров, полученных методом POST;
- `REQUEST` — список параметров, полученных методами POST и GET;
- `method` — строка `"POST"`, если были получены данные, переданные методом POST, или строка `"GET"`, если были получены данные с применением метода GET или был выполнен обычный запрос;
- `path` — строка, хранящая интернет-адрес данной страницы без указания домена и списка GET-параметров.

Первые три свойства хранят объекты класса `QueryDict`, который можно рассматривать как несколько более мощную разновидность словаря Python (см. *главу 2*).

```
def index(request):
  cat_id = request.GET["id"]
  . . .
```

Здесь мы получаем значение параметра `id`, переданного методом GET.

Получить значения переданных контроллеру параметров можно и другим, более удобным способом. Мы можем передать параметры, указанные нами в привязке,

непосредственно функции-контроллеру, причем перечислять их нужно в том порядке, в котором они присутствовали в самой привязке:

```
def index(request, cat_id):
  if cat_id == None:
    cat = Category.objects.first()
  else:
    cat = Category.objects.get(pk = cat_id)
  . . .
```

Чем занимается контроллер, мы узнали еще в *главе 1*. Он извлекает данные из модели, обрабатывает их и выводит результат обработки с применением шаблона.

Поскольку шаблонами мы начнем заниматься только в *главе 7*, а проверить работу контроллеров надо уже сейчас, мы можем использовать для отладочных целей класс `HttpResponse`, объявленный в модуле `django.http`. Конструктору этого класса передается в качестве параметра строка, которую следует вывести в Web-обозревателе, а сам созданный объект этого класса должен возвращаться функцией-контроллером в качестве результата.

Давайте напишем наш первый контроллер. Им будет функция `index`, выводящая список товаров:

```
from django.http import HttpResponse
from page.models import Category, Good

def index(request, cat_id):
  if cat_id == None:
    cat = Category.objects.first()
  else:
    cat = Category.objects.get(pk = cat_id)
  goods = Good.objects.filter(category = cat).order_by("name")
  s = "Категория: " + cat.name + "<br><br>"
  for good in goods:
    s = s + "(" + str(good.pk) + ") " + good.name + "<br>"
  return HttpResponse(s)
```

Чтобы вставить разрыв строки, мы используем HTML-тег `<br>`. Если применить для этого специальный символ переноса строки `\n`, Web-обозреватель его проигнорирует.

Для простоты мы пока не будем проверять, присутствует ли в модели категория с указанным в параметре идентификатором. В приложениях, предназначенных для эксплуатации в реальных условиях, это обязательно нужно сделать.

Объявим также второй контроллер — функцию `good`. Если этого не сделать, приложение не заработает. А сейчас эта функция будет просто возвращать пустую строку:

```
def good(request):
  return ""
```

Запустим отладочный Web-сервер Django, откроем Web-обозреватель и наберем в нем интернет-адрес **http://localhost:8000/goods/**. И тут же увидим список товаров, относящихся к самой первой категории (рис. 6.1).

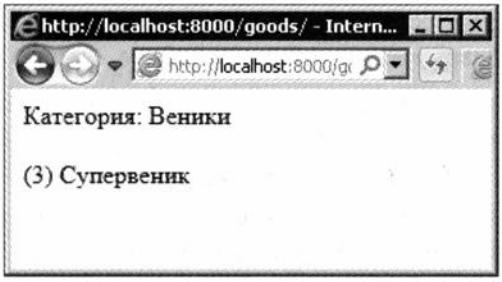


Рис. 6.1. Список товаров

Наберем интернет-адрес **http://localhost:8000/goods/2/** — и получим список товаров из другой категории, имеющей идентификатор 2. Подставим в интернет-адрес другой идентификатор и посмотрим, что имеется в соответствующей категории.

Наигравшись вдоволь с нашим первым контроллером, продолжим работу. На очереди — второй контроллер, который будет выводить подробные сведения о выбранном посетителем товаре. Мы уже его объявили — это «пустая» функция `good`. Перепишем ее код таким образом:

```
def good(request, good_id):
  good = Good.objects.get(pk = good_id)
  s = good.name + "<br><br>" + good.category.name + "<br><br>" + ↳
  good.description
  if not good.in_stock:
    s = s + "<br><br>" + "Нет в наличии!"
  return HttpResponse(s)
```

Наберем в Web-обозревателе интернет-адрес **http://localhost:8000/goods/good/1/**. И посмотрим сведения о товаре с идентификатором 1 (рис. 6.2). И повторим ту же операцию для товаров с другими идентификаторами — чтобы убедиться, что контроллер работает.

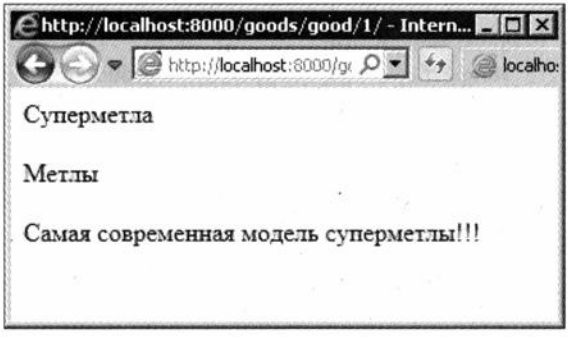


Рис. 6.2. Сведения о товаре

Обработка «ошибки 404»

Если Web-сервер получит запрос к несуществующему файлу, он генерирует так называемую *ошибку 404* и выдает страницу, предупреждающую об отсутствующем файле. Было бы неплохо, если бы и наш сайт при попытке обратиться, скажем, к несуществующему товару выдавал нечто подобное.

В этом нам поможет исключение `Http404`, объявленное в модуле `django.http`. Нам нужно просто сгенерировать его в нужном месте кода. Django сама обработает это исключение и выведет встроенную страницу с соответствующим сообщением.

Перепишем функцию-контроллер `good`, чтобы в случае отсутствия в модели товара с указанным идентификатором она выводила встроенную страницу «ошибки 404» Django:

```
def good(request, good_id):
  try:
    good = Good.objects.get(pk = good_id)
  except Good.DoesNotExist:
    raise Http404
  s = good.name + "<br><br>" + good.category.name + "<br><br>" + ↳
  good.description
  if not good.in_stock:
    s = s + "<br><br>" + "Нет в наличии!"
  return HttpResponse(s)
```

Сама встроенная «страница 404» представлена на рис. 6.3. Это отладочная версия такой страницы. В сайтах, предназначенных для публикации, будет выводиться другая версия — рабочая. Кроме того, мы можем сами указать шаблон страницы для «страницы 404», но займемся этим в самом конце работы над сайтом, в *главе 35*.

Рис. 6.3. Встроенная страница «ошибки 404» Django

В качестве домашнего задания исправьте функцию-контроллер `index` так, чтобы она генерировала «ошибку 404» при указании несуществующей категории. Можете также написать еще один контроллер, который будет выводить список категорий.

Что дальше?

В этой главе мы изучали регулярные выражения, что активно применяются при указании интернет-адресов, привязываемых к контроллерам. И, наконец, привязывали адреса к контроллерам и писали сами контроллеры.

Следующим нашим шагом в овладении Django-программированием будет создание шаблонов, на основе которых станут генерироваться результирующие Web-страницы. В следующей главе мы изучим основные принципы их создания.

ГЛАВА 7

Простые шаблоны Django

В предыдущей главе мы создавали контроллеры, которые выполняли обработку хранящихся в моделях данных и выводили результаты на экран. Правда, выводимые ими результаты, по правде говоря, не впечатляли, поскольку представляли собой обычный текст.

В этой главе мы примемся за написание шаблонов, на основе которых станут генерироваться полноценные Web-страницы. Мы ведь Web-сайт разрабатываем, не так ли?

Что такое шаблон Django?

Как уже говорилось в *главе 1*, шаблон Django — это обычная Web-страница, включающая как HTML-код, задающий статичные элементы и разметку, так и особые команды, помещающие в нужное место шаблона какие-либо значения либо управляющие формированием кода. Эти команды обрабатываются шаблонизатором, в то время как обычный HTML-код такой обработке не подвергается.

Давайте рассмотрим вот такой код шаблона:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>{{ category.name }}</title>
  </head>
  <body>
    <h1>Список товаров</h1>
    <h2>Категории:</h2>
    <ul>
      {% for cat in cats %}
        <li><a href="/goods/{{ cat.id }}/">{{ cat.name }}</a></li>
      {% endfor %}
    </ul>
```

```
    <h2>Товары:</h2>
    <table>
      <tr>
        <th>Название</th>
        <th>Есть в наличии</th>
      </tr>
      {% for good in goods %}
        <tr>
          <td><a href="/goods/good/{{ good.id }}/">
          {{ good.name }}</a></td>
          <td>{% if good.in_stock %}+{% endif %}</td>
        </tr>
      {% endfor %}
    </table>
  </body>
</html>
```

Он выведет на экран страницу, содержащую списки категорий и товаров, относящихся к выбранной категории. Здесь среди обычного HTML-кода хорошо видны особые команды шаблонизатора Django — они заключены в фигурные скобки.

По умолчанию файлы шаблонов Django будет искать в папке templates, которая находится в папке приложения. Создадим эту папку.

> ***Уникальные имена шаблонов***
>
> Django будет искать файлы шаблонов в папках *templates* папок всех приложений, что входят в состав проекта. Поэтому мы должны давать файлам шаблонов имена, уникальные в пределах всего проекта, — в противном случае Django может загрузить шаблон, имеющий то же имя, но относящийся к другому приложению.

Теперь запустим текстовый редактор Notepad++, наберем в нем представленный ранее код шаблона и сохраним его в файле index.html. Только не забудем в диалоговом окне сохранения указать тип файла: **Hyper Text Markup Language file (*.html, *.htm, *.shtml, *.shtm, *.xhtml, *.hta)**.

Команды шаблонизатора

Что ж, команды шаблонизатора мы в коде шаблона увидели. Но что они делают? Самое время об этом поговорить. И разговор этот будет довольно долгим, поскольку таких команд очень много.

Все команды шаблонизатора можно разбить на три категории: переменные, теги и фильтры шаблона. Рассмотрим их по очереди.

Переменные шаблона

Переменные шаблона хранят выводимые в этом шаблоне данные. Такая переменная выглядит как имя, заключенное в двойные фигурные скобки:

```
<h2>{{ category_name }}</h2>
```

Здесь мы выводим значение переменной `category_name` внутри тега `<h2>`.

> **Обращение к несуществующей переменной**
>
> Если мы попытаемся обратиться к несуществующей переменной шаблона, то получим пустое значение. Никакого сообщения об ошибке при этом шаблонизатор нам не выведет. Так что при отладке шаблонов будем особенно внимательны!

Если переменная хранит объект, для доступа к его свойствам мы можем использовать уже знакомую нам запись с точкой:

```
<h2>{{ category.name }}</h2>
```

Здесь мы выводим значение свойства `name` объекта, хранящегося в переменной `category`.

Таким же образом мы можем вызывать методы объектов, но только те, что не принимают параметров. Круглые скобки после имени метода в этом случае не ставятся:

```
<p>Всего категорий: {{ cats.count }}</p>
```

Здесь мы вызываем метод `count` набора записей `cats` и выводим его результат.

Для доступа к элементам списка в шаблонах применяется иной синтаксис — индекс элемента ставится после имени списка и отделяется от него точкой:

```
<h2>{{ category_names.0 }}</h2>
```

Здесь мы выводим значение первого элемента списка `category_names`.

```
<h2>{{ cats.0.name }}</h2>
```

А здесь — значение свойства `name` объекта, хранящегося в первом элементе списка `cats`.

Теги шаблона

Теги шаблона выполняют какие-либо действия над фрагментами кода шаблона. В отличие от HTML-тегов, они обрабатываются шаблонизатором, а не самим Web-обозревателем. Теги шаблона записываются в формате `{% <тег> %}`.

Тегов шаблонов, поддерживаемых Django, довольно много. Для удобства рассмотрения мы также разобьем их на категории.

Теги условных выражений

Часто бывает необходимо вывести на страницу определенный фрагмент кода в зависимости от выполнения или невыполнения какого-либо условия. То есть сформировать в шаблоне условное выражение.

Для этого нам понадобится тег шаблона `if`, формат записи которого следующий:

```
{% if <условие 1> %}
  <блок 1>
[{% elif <условие 2> %}
  <блок 2>
[<прочие теги elif>]]
```

```
[{% else %}
  <блок else>]
{% endif %}
```

Обрабатывается эта конструкция так же, как аналогичное условное выражение языка Python (см. *главу 2*).

В *условии* мы можем использовать операторы сравнения, перечисленные в табл. 2.4 и 2.5, и логические операторы из табл. 2.6:

```
{% if cats.count > 0 %}
  <h2>Категории:</h2>
  <ul>
    {% for cat in cats %}
      <li><a href="/goods/{{ cat.id }}/">{{ cat.name }}</a></li>
    {% endfor %}
  </ul>
{% else %}
  <p>Список категорий пуст.</p>
{% endif %}
```

Если в списке `cats` есть категории, формируем их список, в противном случае выводим вместо него соответствующее сообщение:

```
<td>{% if good.in_stock %}+{% endif %}</td>
```

Если значение свойства `in_stock` переменной шаблона равно `True`, выводим в ячейке таблицы символ плюса.

Если нам нужно просто сравнить два значения и на основании их равенства или неравенства выполнить какое-либо действие, мы можем использовать теги шаблона `ifequal` и `ifnotequal`. Они обрабатываются шаблонизатором быстрее, чем тег `if` с соответствующим условием:

```
{% ifequal|ifnotequal <значение 1> <значение 2> %}
  <блок>
{% endifequal|endifnotequal %}
```

Здесь тег шаблона `ifequal` выводит *блок*, если указанные *значения* равны, а тег шаблона `ifnotequal` — если они не равны.

```
<td>{% ifnoteqial good.discount 0 %}Скидка!!!{% endifnotequal %}</td>
```

Если значение свойства `discount` объекта `good` не равно нулю, выводим в ячейке таблицы надпись "Скидка!!!"

Тег цикла

Но еще чаще, чем логические выражения, в шаблонах применяются циклы по спискам. Они аналогичны знакомым нам циклам Python и обрабатываются так же.

Для формирования цикла по списку служит тег шаблона `for`. Формат его записи таков:

```
{% for <переменная элемента списка> in <список> %}
  <блок тела цикла>
[{% empty %}
  <блок empty>]
{% endfor %}
```

Здесь *блок empty* (если, конечно, он присутствует) будет выполнен, если указанный *список* не содержит элементов.

```
{% for cat in cats %}
  <li><a href="/goods/{{ cat.id }}/">{{ cat.name }}</a></li>
{% endfor %}
```

А здесь мы выводим элементы списка `cats`.

В параметре *блок тела цикла* мы можем использовать особые переменные, хранящие различные относящиеся к выполнению цикла значения (*переменные цикла*). Они доступны только в теле цикла и перечислены в табл. 7.1.

***Таблица 7.1.** Переменные цикла*

Переменная цикла	Описание
`forloop.counter`	Номер текущего прохода цикла, начиная с 1
`forloop.counter0`	Номер текущего прохода цикла, начиная с 0
`forloop.revcounter`	Номер текущего прохода цикла, отсчитанный с его конца, начиная с 1
`forloop.revcounter0`	Номер текущего прохода цикла, отсчитанный с его конца, начиная с 0
`forloop.first`	`True`, если это первый проход цикла
`forloop.last`	`True`, если это последний проход цикла
`forloop.parentloop`	Ссылается на внешний цикл для доступа к значениям его переменных (для вложенных циклов)

Вот пример:

```
{% for cat in cats %}
  {% if forloop.first %}
    <h2>Категории:</h2>
    <ul>
  {% endif %}
  <li><a href="/goods/{{ cat.id }}/">{{ forloop.counter }}
  {{ cat.name }}</a></li>
  {% if forloop.last %}
    </ul>
  {% endif %}
{% empty %}
  <p>Список категорий пуст.</p>
{% endfor %}
```

В теле цикла нам может оказаться полезным тег шаблона `cycle`:

```
{% cycle <список значений, разделенных пробелами> %}
```

При первом проходе цикла этот тег вернет первое значение из перечисленных в нем, при втором — второе значение, при третьем — третье и т. д. Когда значения закончатся, их перебор начнется с начала, т. е. при очередном проходе цикла снова будет возвращено первое значение, при втором — второе и т. д.

```
{% for good in goods %}
  <tr class="{% cycle 'normal' 'alternate'">
    . . .
  </tr>
{% endfor %}
```

Здесь при нечетных проходах цикла к строке таблицы будет привязан стилевой класс `normal`, а при четных — `alternate`.

Теги, управляющие выводом

По умолчанию Django проверяет все выводимые в шаблонах значения на предмет наличия в них символов, используемых в HTML для особых целей. К таким символам относятся, в частности, знак «меньше» (<) и двойные кавычки.

Все эти символы перед выводом автоматически заменяются на соответствующие им литералы HTML:

- символ < заменяется на литерал `<`;
- > — на `>`;
- ' (амперсанд) — на `&#39;`;
- " (двойные кавычки) — на `&quot;`;
- `&` — на `&`.

Так что нам не придется делать это самим.

Однако временами может понадобиться вывести содержимое какой-либо переменной как есть, без такого преобразования. Для этого служит тег шаблона `autoescape`:

```
{% autoescape on|off %}
  <код, выводящий значения>
{% endautoescape %}
```

Параметр `on` этого тега включает автозамену специальных символов HTML, а значение `off` отключает ее:

```
{% autoescape off %}
  <p>{{ good.description }}</p>
{% endautoescape %}
```

Тег `firstof` выводит первое непустое и ненулевое значение из указанного в нем перечня:

```
{% firstof <список переменных, разделенных пробелами> %}
```

Предположим, что переменная шаблона `var1` содержит пустую строку, `var2` — какое-либо число, отличное от нуля, а `var3` — непустую строку. Тогда тег:

```
{% firstof var1 var2 var3 %}
```

выведет значение переменной `var2`, т. к. это первая в списке переменная с непустым и ненулевым значением.

Тег `verbatim` запрещает шаблонизатору Django обрабатывать его содержимое:

```
{% verbatim %}
  <код, который не должен быть обработан шаблонизатором>
{% endverbatim %}
```

Например, код:

```
{% verbatim %}
  <h2>{{ category_name }}</h2>
{% endverbatim %}
```

выведет на экран строку `{{ category_name }}`, являющуюся заголовком второго уровня.

Тег `with` пригодится в случаях, если мы собираемся многократно вывести значение какой-либо сложной структуры данных, скажем, объекта, вложенного в другой объект:

```
{% with <имя формальной переменной>=<значение> %}
  <код, выводящий значение формальной переменной>
{% endwith %}
```

Например:

```
{% with cat=good.category %}
  <p><a href="/goods/{{ cat.id }}/">{{ cat.name }}</a></p>
{% endwith %}
```

Шаблонизатор Django поддерживает еще несколько тегов, предназначенных для обработки вложенных шаблонов и формирования интернет-адресов. Мы рассмотрим их в *главе 8*.

Комментарии

Язык Python поддерживает комментарии, которые не обрабатываются интерпретатором, а служат лишь для сведений разработчика и его коллег. Чем же язык шаблонов Django хуже?

Для указаний комментариев в коде шаблонов применяется следующий синтаксис:

```
{# <текст комментария> #}
```

Например:

```
{# Не забыть вставить сюда вывод списка категорий #}
```

Фильтры шаблона

Фильтры шаблона либо выполняют какие-либо действия над значением переданной им переменной шаблона, либо вычисляют на основе этого значения какой-либо результат. Для указания фильтров применяется следующий формат:

```
<переменная шаблона>|<фильтр 1>[:<значение фильтра 1>]
[|<фильтр 2>[:<значение фильтра 2>]] . . .
```

Как видим, мы можем указать для значения одной переменной сразу несколько фильтров. Значение фильтра, если оно не является переменной шаблона, всегда заключается в двойные кавычки, даже если это число.

```
<p>{{ good.description|linebreaksbr }}</p>
```

Здесь мы заменяем в значении свойства `description` объекта `good` все символы перевода строки тегами `<br>` HTML.

```
<p>{{ good.price|floatformat:"2" }}</p>
```

А здесь выводим значение свойства `price` объекта `good` с округлением до двух цифр после запятой.

Шаблонизатор Django поддерживает очень много фильтров. Самые полезные из них приведены в табл. 7.2.

***Таблица 7.2.** Фильтры шаблона*

Фильтр шаблона	Описание
`add:<число>`	Складывает числовое значение и *число*
`capfirst`	Переводит первую букву в строковом значении в верхний регистр
`cut:<строка>`	Удаляет из строкового значения все вхождения параметра *строка*
`date:<формат>`	Форматирует дату согласно указанному *формату*
`default:<значение>`	Если переменная хранит пустое и нулевое значение, возвращает указанное *значение*
`default_if_none:<значение>`	Если переменная хранит значение `None`, возвращает указанное *значение*
`divisibleby:<делитель>`	Возвращает `True`, если числовое значение делится на указанный *делитель* без остатка
`escape`	Заменяет в строковом значении все недопустимые символы HTML на литералы. Применяется внутри тега шаблона `autoescape`, отключающего такую автозамену (см. ранее)
`escapejs`	Преобразует строковое значение к виду, пригодному к использованию в Web-сценариях, которые написаны на языке JavaScript
`first`	Возвращает первый элемент списка

Таблица 7.2 (окончание)

Фильтр шаблона	Описание
`floatformat:[`*<количество значащих цифр после точки>*`]`	Округляет числовое значение
`last`	Возвращает последний элемент списка
`length`	Возвращает размер списка
`linebreaks`	Помещает строку в теги `<p>` и заменяет все переводы строки тегами ` `
`linebreaksbr`	Заменяет все переводы строки тегами ` `
`lower`	Преобразует все буквы строки в нижний регистр
`random`	Возвращает случайно выбранный из списка элемент
`safe`	Отключает для данного строкового значения преобразование недопустимых в HTML символов в литералы
`striptags`	Удаляет из строки все теги HTML
`time:`*<формат>*	Форматирует время согласно указанному параметру *формат*
`title`	Преобразует первую букву каждого слова строки в верхний регистр
`truncatechars:`*<количество символов>*	Урезает строку до указанного в параметре *количество символов* и добавляет в ее конце многоточие
`truncatewords:`*<количество слов>*	Урезает строку до указанного в параметре *количество слов* и добавляет в ее конце многоточие
`truncatewords_html:`*<количество слов>*	То же, что `truncatewords`, но с закрытием всех открытых тегов HTML
`upper`	Урезает строку до размера, указанного в параметре *количество символов*, и добавляет в ее конце многоточие
`urlencode`	Урезает строку до размера, указанного в параметре *количество слов*, и добавляет в ее конце многоточие
`wordcount`	Возвращает количество слов в строке
`yesno:<"список строк, перечисленных через запятую">`	Если `True`, выводит первую строку из *списка*, если `False` — вторую, если `None` — третью

Назначение и принцип работы большей части фильтров понятны из их описаний. Но некоторые фильтры требуют особых пояснений.

Фильтр `floatformat`, округляющий числа, принимает в качестве параметра целое число, указывающее количество цифр после запятой в результирующем числе. Параметр этот может быть указан как:

- положительное число — тогда результат всегда будет выводиться с указанным количеством цифр после запятой, даже если число фактически является целым;
- ноль — тогда число будет округлено до целой части;

- отрицательное число — тогда результат будет включать дробную часть только в том случае, если он не является целым числом.

Если не указать параметр, то этот фильтр будет работать так, будто задано значение параметра `-1`.

В табл. 7.3 перечислены несколько примеров использования фильтра `floatformat` с разными значениями параметра.

***Таблица 7.3.** Примеры использования фильтра `floatformat`*

Значение параметра	Исходное число	Результат
0	34,56789	35
3	34,56789	34.568
	12,34	12.340
	12	12.000
-3	34,56789	34.568
	12,34	12.340
	12	12
Не указан	34,56789	34.6
	12,34	12.3
	12	12

Вот пример:

```
<p>Цена: {{ good.price|floatformat: "2"}}</p>
```

Здесь мы выводим цену товара, округлив ее до двух цифр после запятой.

Фильтры `date` и `time` требуют указания формата, в котором будут выводиться значения даты и времени соответственно. Этот формат представляет собой строку, содержащую особые литералы, которые указывают, какое именно значение должно быть подставлено на их место. Наиболее полезные из них приведены в табл. 7.4.

***Таблица 7.4.** Литералы, применяемые в описаниях формата даты и времени*

Литерал	Описание
`j`	Число без начального нуля
`d`	Число с начальным нулем
`w`	Номер дня недели от 0 (воскресенье) до 6 (суббота)
`m`	Месяц с начальным нулем
`n`	Месяц без начального нуля
`E`	Название месяца на языке, определенном языковыми настройками
`Y`	Год в виде четырехзначного числа

***Таблица 7.4** (окончание)*

Литерал	Описание
y	Год в виде двухзначного числа
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
a	Обозначение периода времени: "a.m." или "p.m."
A	Обозначение периода времени: "AM" или "PM"
i	Минуты
s	Секунды с начальным нулем
I	"1", если в данный момент действует летнее время, и "0" в противном случае
L	`True`, если текущий год является високосным, и `False` в противном случае
t	Количество дней в текущем месяце даты
z	Номер дня в году
u	Микросекунды
F	Название месяца на английском языке
b	Трехбуквенное обозначение месяца на английском языке со строчной первой буквой: "jan", "feb" и т. д.
M	Трехбуквенное обозначение месяца на английском языке с прописной первой буквой: "Jan", "Feb" и т. д.
l	Название дня недели на английском языке
D	Трехбуквенное обозначение дня недели на английском языке: "Mon", "Tue" и т. д.
r	Дата, отформатированная согласно требованиям документа RFC 2822
T	Обозначение временной зоны

Вот примеры:

```
<p>{{ blog_article.pubdate|date:"j.d.Y" }}</p>
```

Здесь мы выводим дату публикации статьи блога в привычном нам формате *<число>.<месяц>.<год>*.

```
<p>{{ blog_article.pubdate|date:"j.E.Y" }}</p>
```

А этот код выведет дату в более развернутом формате: *<число>.<название месяца>.<год>*.

```
<p>{{ blog_article.pubdate|time:"G:i" }}</p>
```

Здесь мы выводим время публикации статьи в формате *<часы>:<минуты>*.

```
{% if blog_article.pubdate|date:"L" %}
  <p>Високосный год.</p>
{% endif %}
```

А здесь проверяем, является ли год, когда была опубликована статья, високосным, и, если является, выводим соответствующее сообщение.

Кстати, если нам нужно применить один и тот же фильтр или группу фильтров сразу к нескольким значениям, мы можем воспользоваться тегом шаблона `filter`:

```
{% filter <список фильтров> %}
  <код>
{% endfilter %}
```

Например:

```
{% filter linebreaksbr|striptags %}
  <p>{{ good.name }}</p>
  <p>{{ good.description }}</p>
{% endfilter %}
```

В результате фильтры `linebreaksbr` и `striptags` будут применены к значениям свойств `name` и `description` объекта `good`.

Рендеринг шаблона

Итак, шаблон мы создали. Осталось узнать, как вывести с его помощью на экран результат работы контроллера, или, как говорят Django-программисты, выполнить *рендеринг* шаблона.

Проще всего для этого вызвать функцию `render`, объявленную в модуле `django.shortcuts`. Благо импортировать ее нам не придется — сама Django поместила в начало модуля `views` выражение, выполняющее эту операцию (см. *главу 6*).

Функция `render` принимает следующие параметры:

- объект класса `HttpRequest`, хранящий сведения о полученном запросе. Этот объект принимает первым параметром любая функция-контроллер (за подробностями — к *главе 6*);
- путь к файлу шаблона, заданный в виде строки относительно папки templates, что находится в папке пакета приложения;
- *контекст данных* шаблона, т. е. хранилище данных, которые будут выводиться с его помощью. Он представляет собой обычный словарь Python: ключи его элементов послужат именами переменных шаблона, а их значения станут значениями соответствующих переменных.

Результат, возвращаемый функцией `render`, — а этим результатом будет строка с кодом сформированной Web-страницы, — должен быть возвращен функцией-контроллером в качестве результата.

Давайте перепишем контроллер `views.index` таким образом, чтобы он выводил список товаров в виде страницы с применением созданного нами ранее шаблона index.html (его код приведен в начале этой главы):

```
def index(request, cat_id):
  cats = Category.objects.all().order_by("name")
  if cat_id == None:
    cat = Category.objects.first()
  else:
    cat = Category.objects.get(pk = cat_id)
  goods = Good.objects.filter(category = cat).order_by("name")
  return render(request, "index.html", { "category": cat, "cats": cats,↵
  "goods": goods})
```

Поскольку мы собираемся выводить, помимо списка товаров из выбранной категории, также и список самих категорий, нам придется сформировать и его. В остальном здесь все нам уже знакомо.

Запустим отладочный Web-сервер Django, откроем Web-обозреватель и наберем в нем интернет-адрес **http://localhost/goods/**. Если мы не допустили в коде ошибок, то увидим страницу списка товаров (рис. 7.1).

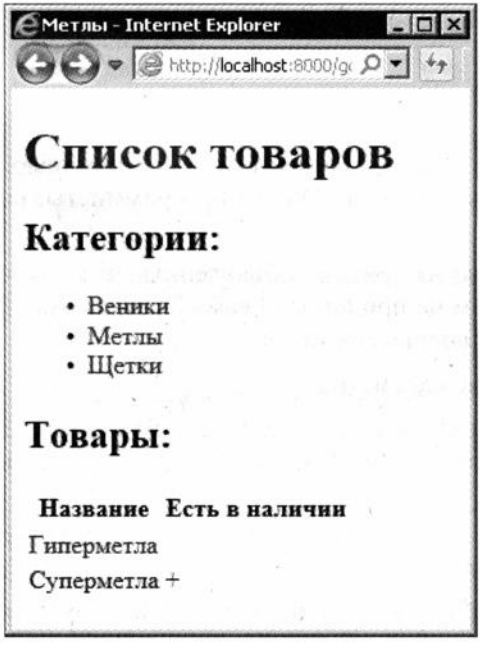


Рис. 7.1. Список товаров, сформированный с применением шаблона

Конечно, сейчас наша первая программно сгенерированная страничка выглядит на редкость непрезентабельно. Но уже в *главе 8* мы вплотную займемся ее дизайном и оформлением.

Пощелкаем на гиперссылках в списке категорий и посмотрим, какие товары содержатся в соответствующих категориях. И, убедившись, что переписанный контроллер работает, приступим к работе над вторым контролером — тем, что выводит сведения о выбранном товаре.

Это функция `views.good`. Давайте перепишем ее так, чтобы она выводила данные с помощью шаблона good.html (мы создадим его чуть позже):

```
def good(request, good_id):
  try:
    good = Good.objects.get(pk = good_id)
  except Good.DoesNotExist:
    raise Http404
  return render(request, "good.html", {"good": good})
```

Осталось создать шаблон good.html. Сохраним его в той же папке templates папки пакета приложения:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>{{ good.name }}</title>
  </head>
  <body>
    <h1>{{ good.name }}</h1>
    <p>Категория: {{ good.category.name }}</p>
    <p>{{ good.description|linebreaksbr }}</p>
    <p>{% if not good.in_stock %}Нет в наличии!{% endif %}</p>
    <p><a href="/goods/{{ good.category.id }}/">Назад</a></p>
  </body>
</html>
```

Здесь мы указали, что в описании товара все переводы строк должны быть заменены HTML-тегами `<br>`, применив фильтр `linebreaksbr`. В результате мы сможем форматировать текст описаний товаров, разбивая его на строки.

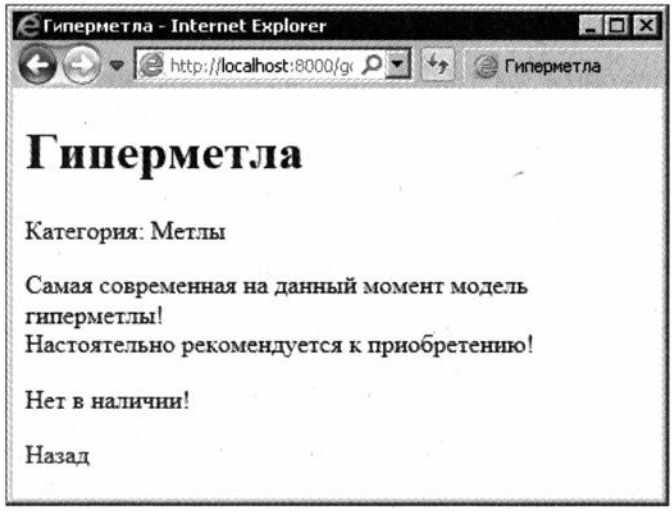


Рис. 7.2. Описание товара, сформированное с применением шаблона

Также не забудем гиперссылку, ведущую на страницу списка товаров соответствующей категории. Этого требует хороший стиль Web-дизайна.

Закончив, проверим сайт в работе. Щелкнем на названии товара — и попадем на страницу с его описанием (рис. 7.2). Просмотрев его, щелкнем на гиперссылке **Назад**, чтобы вернуться на список товаров.

Что дальше?

В этой главе мы учились писать шаблоны для вывода результатов работы контроллеров. И написали два — пока еще совсем простых.

В следующей главе мы продолжим разговор о шаблонах. Мы займемся их оформлением с помощью каскадных таблиц стилей, узнаем о статичных файлах и их обработке и вложенных шаблонах, а также познакомимся с инструментами для формирования интернет-адресов, ссылающихся на контроллеры. В общем, приступим к рассмотрению более развитых возможностей шаблонизатора Django.

ГЛАВА 8

Более сложные шаблоны Django

В предыдущей главе мы изучили инструменты для создания шаблонов Django — переменные, теги и фильтры шаблонов, а также средства для вывода шаблонов на экран. И, разумеется, опробовали их в действии.

В этой главе мы продолжим работу над шаблонами. Мы сделаем их более похожими на современные Web-страницы с помощью каскадных таблиц стилей. Мы исключим из них повторяющийся код, выделив его в родительский шаблон. И, наконец, мы воспользуемся средствами Django для формирования интернет-адресов на основе привязки, чтобы не писать их каждый раз вручную.

Оформление и верстка шаблонов

Первые наши шаблоны не могли похвастаться богатством оформления. Да они вообще не содержали никакого оформления — одно лишь содержимое, один лишь HTML-код. Настала пора вплотную заняться их внешним видом. Мы ведь хотим создать профессионально выглядящий Web-сайт!

Для оформления шаблонов мы можем задействовать все возможности, предлагаемые каскадными таблицами стилей CSS. В этом шаблонизатор Django нас совершенно не ограничивает.

Давайте перепишем код обоих шаблонов так, чтобы они больше походили на современные Web-страницы. После чего создадим таблицу стилей, которая задаст для них оформление.

Код обновленного шаблона index.html будет выглядеть так:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link type="text/css" href="main.css" rel="stylesheet">
    <title>{{ category.name }} :: Все для уборки</title>
  </head>
```

```
  <body>
    <div id="header">
      <h1>Все для уборки</h1>
    </div>
    <div id="leftmenu">
      <ul>
        {% for cat in cats %}
          <li><a href="/goods/{{ cat.id }}/">{{ cat.name }}</a></li>
        {% endfor %}
      </ul>
    </div>
    <div id="main">
      <h2>{{ category.name }}</h2>
      <table>
        <tr>
          <th>Название</th>
          <th>Есть в наличии</th>
        </tr>
        {% for good in goods %}
          <tr>
            <td><a href="/goods/good/{{ good.id }}/">
            {{ good.name }}</a></td>
            <td class="centered">{% if good.in_stock %}+{% endif %}</td>
          </tr>
        {% endfor %}
      </table>
    </div>
    <div id="footer">
      <p>Все права принадлежат разработчикам сайта.</p>
    </div>
  </body>
</html>
```

Здесь мы сформировали «шапку» и «поддон» страницы, включающие, соответственно, название сайта (и гипотетической фирмы, для которой мы его делаем) и сведения о правах его разработчиков. «Шапку» и «поддон» мы поместили в блоки (теги `<div>`) `header` и `footer`.

Список категорий мы поместили в блок `leftmenu`, который будет выводиться у левого края страницы. А основное содержимое (в данном случае — список товаров) теперь будет находиться в блоке `main`, что займет остальное ее пространство.

А вот код обновленного шаблона good.html:

```
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link type="text/css" href="main.css" rel="stylesheet">
```

```
    <title>{{ good.name }} :: {{ good.category.name }} ::
    Все для уборки</title>
  </head>
  <body>
    <div id="header">
      <h1>Все для уборки</h1>
    </div>
    <div id="leftmenu">
      <ul>
        {% for cat in cats %}
          <li><a href="/goods/{{ cat.id }}/">{{ cat.name }}</a></li>
        {% endfor %}
      </ul>
    </div>
    <div id="main">
      <h2>{{ good.name }}</h2>
      <p class="category">Категория: {{ good.category.name }}</p>
      <p>{{ good.description|linebreaksbr }}</p>
      <p>{% if not good.in_stock %}Нет в наличии!{% endif %}</p>
      <p class="not-in-stock"><a href="/goods/{{ good.category.id }}/">
      Назад</a></p>
    </div>
    <div id="footer">
      <p>Все права принадлежат разработчикам сайта.</p>
    </div>
  </body>
</html>
```

По большей части, здесь все то же самое, за исключением того, что в блоке `main` будут выводиться сведения о выбранном товаре.

Осталось создать таблицу стилей. Вот ее код:

```
@charset "utf-8";
body {
  margin: 0px;
  padding: 0px;
}
#header {
  text-align: center;
  border-bottom: 1px #cccccc solid;
}
#header h1 {
  margin: 20px 0px 20px 0px;
  letter-spacing: 0.2em;
}
#leftmenu {
  float: left;
```

```
  width: 100px;
  padding: 10px;
}
#leftmenu ul {
  margin: 0px;
  padding: 0px;
}
#leftmenu ul li {
  list-style-type: none;
  margin-bottom: 5px;
}
#main {
  margin-left: 110px;
  border-left: 1px #cccccc solid;
  padding: 10px;
}
#main h2 {
  margin: 0px 0px 10px 0px;
}
#main table {
  width: 100%;
  border-collapse: collapse;
}
#main table td, #main table th {
  border: 1px #666666 solid;
  padding: 5px;
}
#main table td.centered {
  text-align: center;
}
#main p.category {
  font-weight: bold;
}
#main p.not-in-stock {
  color: #FF0000;
}
#footer {
  text-align: right;
  border-top: 1px #cccccc solid;
  padding-right: 10px;
}
#footer p {
  font-size: smaller;
  font-style: italic;
}
```

Наберем его в Notepad++ и сохраним в кодировке UTF-8 в файле main.css, который поместим в ту же папку templates. В качестве типа файла в диалоговом окне сохранения укажем: **Cascade Style Sheets File (*.css)**.

В очередной раз модифицируем функцию-контроллер `views.good`, добавив в нее выражение для получения списка категорий — ведь нам придется выводить его в обновленном шаблоне good.html:

```
def good(request, good_id):
  cats = Category.objects.all().order_by("name")
  try:
    good = Good.objects.get(pk = good_id)
  except Good.DoesNotExist:
    raise Http404
  return render(request, "good.html", {"cats": cats, "good": good})
```

Запустим отладочный Web-сервер, откроем Web-обозреватель и наберем в нем интернет-адрес **http://localhost:8000/goods/**. По идее, должна бы открыться новая, красиво оформленная Web-страница. Но нет — мы видим практически то же, что было показано на рис. 7.1 (за исключением добавившихся «шапки» и «поддона»).

Судя по всему, Web-обозреватель почему-то не загрузил созданную нами таблицу стилей main.css. Но почему? Давайте разберемся.

Статичные файлы и их обработка

Все дело в том, что Django не «знает», где хранится файл main.css. И, следовательно, не может отправить его Web-обозревателю.

Любые файлы, которые не генерируются программно контроллерами и шаблонами, а берутся непосредственно с жесткого диска компьютера, в терминологии Django носят название *статичных*. К таким файлам относятся таблицы стилей, графические изображения, Web-сценарии и пр. Наша таблица стилей main.css тоже является статичным файлом.

По умолчанию Django ищет все статичные файлы в папке static, вложенной в папку пакета приложения. Создадим эту папку и переместим файл main.css в нее.

Далее нам нужно задать кое-какие настройки, чтобы Django смогла успешно сформировать интернет-адреса статичных файлов. Так что откроем модуль `settings` пакета проекта. Нам следует задать префикс, который будет добавляться к интернет-адресу каждого статичного файла. Он указывается в переменной `STATIC_URL` в виде строки:

```
STATIC_URL = '/static/'
```

В результате для файла нашей таблицы стилей main.css, хранящегося непосредственно в папке static, будет сгенерирован интернет-адрес **/static/main.css**. Не забудем сохранить модуль `settings` после сделанных изменений.

Теперь нам нужно указать шаблонизатору Django, что файл таблицы стилей является статичным. Откроем шаблон index.html. Сначала нам следует загрузить модуль шаблонизатора, реализующий поддержку статичных файлов. Для загрузки дополнительных модулей шаблонизатора применяется тег шаблона `load`, после которого указывается имя модуля, — этот тег должен присутствовать в самом начале кода шаблона. Осталось лишь сказать, что за обработку статичных файлов «отвечает» модуль `staticfiles`.

Исходя из всего этого, вставим в самое начало кода нашего шаблона строку:

```
{% load staticfiles %}
```

Осталось пометить файл таблицы стилей как статичный — тогда Django будет «знать», где он хранится. Для этого применяется тег шаблона `static`, который указывается вместо интернет-адреса файла:

```
{% static <путь к файлу относительно папки static> %}
```

Найдем в коде шаблона тег `<link>`, загружающий таблицу стилей, и изменим его следующим образом:

```
<link type="text/css" href="{% static "main.css" %}" rel="stylesheet">
```

Если мы теперь попытаемся вызвать список товаров, то перед нами откроется окно, показанное на рис. 8.1. Конечно, с точки зрения современного сложного Web-дизайна это не сильно впечатляет, но мы ведь занимаемся здесь не дизайном, а программированием.

Откроем шаблон good.html и внесем в него те же самые правки. И посмотрим на результат (рис. 8.2).

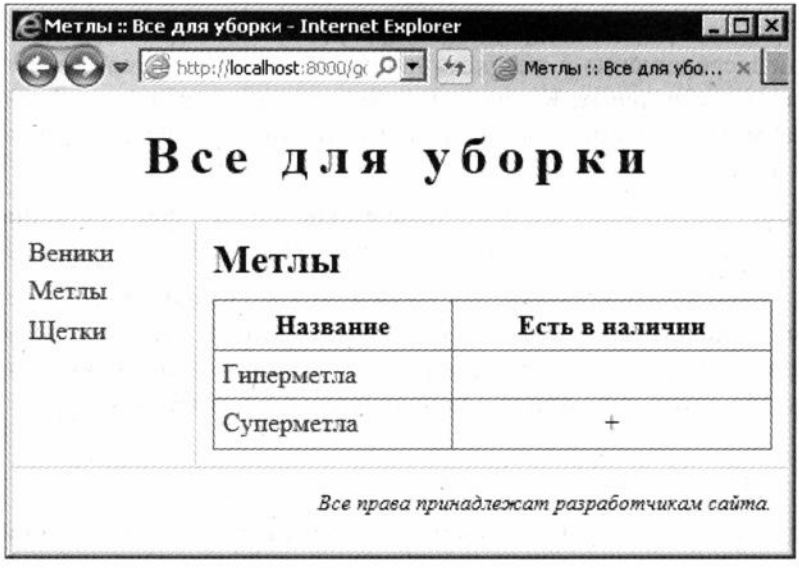


Рис. 8.1. Список товаров, оформленный с помощью каскадной таблицы стилей

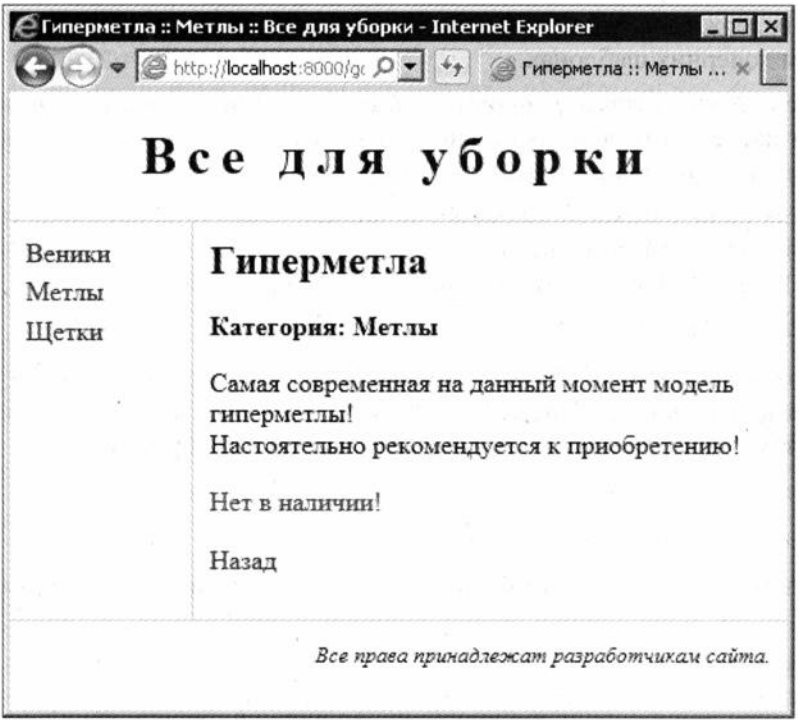


Рис. 8.2. Сведения о товаре, оформленные с помощью каскадной таблицы стилей

Устранение дублирования кода в шаблонах

Пока еще немалая (если не бóльшая) часть кода в обоих наших шаблонах практически одинакова. В самом деле, оба шаблона имеют одинаковую «шапку», одинаковый список категорий и одинаковый «поддон». Да и инфраструктурные теги, структурирующие код (`<html>`, `<head>` и `<body>`), задающие кодировку и привязывающие таблицу стилей, в обоих случаях одни и те же. Уникальными для каждой страницы являются лишь содержимое блока `main` и название страницы, задаваемое тегом `<title>`.

Можно ли как-то исключить дублирование повторяющегося кода? Можно. И сделать это совсем не трудно.

Наследование шаблонов

В *главе 2*, изучая классы Python, мы узнали, что они могут наследоваться друг от друга. Родительский класс при этом реализует какую-либо базовую функциональность, а порожденные от него классы добавляют в него свои, уникальные возможности.

Нечто подобное справедливо и для шаблонов Django. Мы можем задействовать механизм *наследования шаблонов* Django, вынеся повторяющийся код в родительский

шаблон, а содержимое, уникальное для каждой страницы, реализовать в соответствующих им дочерних шаблонах.

Сначала нам нужно создать *родительский шаблон*, или *шаблон-родитель*, от которого будут наследоваться все остальные шаблоны сайта. Он должен включать в себя код, одинаковый для всех страниц сайта.

В тех местах родительского шаблона, где должен присутствовать код, уникальный для каждой страницы, мы определим так называемые *блоки шаблона* (не путать с блоками Python!). В этом нам поможет тег шаблона `block`:

```
{% block <имя блока> %}{% endblock %}
```

Давайте создадим копию шаблона index.html и назовем ее base.html. Это и будет наш родительский шаблон. Откроем его. Отыщем код, создающий содержимое тега `<title>` (название страницы), и изменим его следующим образом:

```
<title>{% block title %}{% endblock %} :: Все для уборки</title>
```

Здесь мы создали блок шаблона `title`.

Далее найдем код, формирующий блок `main` и его содержимое, и заменим его таким кодом:

```
<div id="main">
  {% block main %}
  {% endblock %}
</div>
```

Этот блок шаблона получил имя `main`.

Теперь создадим *дочерние шаблоны*, или *шаблоны-потомки*, которые будут наследовать содержимое родителя. Они будут хранить лишь уникальный для данных страниц код, заключенный в соответствующие блоки.

Сначала мы укажем, что этот шаблон является потомком. Для чего поставим в самом начале его кода тег шаблона `extends`:

```
{% extends <путь к файлу родительского шаблона относительно папки templates> %}
```

Параметр *путь к файлу родительского шаблона* указывается в виде строки. Также он может быть значением какой-либо переменной.

Чтобы задать содержимое блока шаблона, используется все тот же тег `block`:

```
{% block <имя блока> %}
  <содержимое блока>
{% endblock %}
```

Это содержимое и будет подставлено в то место родительского шаблона, где объявлен блок с таким же именем.

Вооружившись новыми знаниями, приступим к работе. Откроем шаблон index.html и заменим его код следующим:

```
{% extends "base.html" %}
{% block title %}{{ category.name }}{% endblock %}
```

```
{% block main %}
  <h2>{{ category.name }}</h2>
  <table>
    <tr>
      <th>Название</th>
      <th>Есть в наличии</th>
    </tr>
    {% for good in goods %}
      <tr>
        <td><a href="/goods/good/{{ good.id }}/">{{ good.name }}</a></td>
        <td class="centered">{% if good.in_stock %}+{% endif %}</td>
      </tr>
    {% endfor %}
  </table>
{% endblock %}
```

Откроем шаблон good.html и введем в него такой код:

```
{% extends "base.html" %}
{% block title %}{{ good.name }} :: {{ good.category.name }}
{% endblock %}
{% block main %}
  <h2>{{ good.name }}</h2>
  <p class="category">Категория: {{ good.category.name }}</p>
  <p>{{ good.description|linebreaksbr }}</p>
  <p class="not-in-stock">{% if not good.in_stock %}Нет в наличии!
  {% endif %}</p>
  <p><a href="/goods/{{ good.category.id }}/">Назад</a></p>
{% endblock %}
```

Обратите внимание, насколько уменьшился объем кода в новых шаблонах. Нет, положительно, наследование шаблонов — отличная штука!

Осталось только проверить сайт в работе. И, если мы не сделали ошибок, работать он будет.

Подгружаемые шаблоны

Если некоторые страницы нашего сайта включают в себя один и тот же фрагмент, мы можем вынести его в отдельный шаблон и поставить в коде шаблонов, соответствующих этим страницам, особые теги, которые будут его подгружать и выводить на экран. Подобного рода шаблон называют *подгружаемым*.

Для подгрузки шаблона применяется тег `include`:

```
{% include <путь к файлу подгружаемого шаблона относительно папки
templates> [with <набор пар вида "имя переменной шаблона"="значение",
разделенных пробелами> [only]]
```

Параметр *путь к файлу подгружаемого шаблона* указывается в виде строки. Также он может быть значением какой-либо переменной.

В подгружаемый шаблон автоматически перейдут все переменные текущего шаблона, и значения их будут теми же.

```
{% include "/includes/left_menu.html" %}
```

Здесь мы подгружаем шаблон left_menu.html, хранящийся в подпапке includes папки templates, и выводим его в данном месте текущего шаблона.

Мы можем указать новое значение для какой-либо переменной, что перейдет в подгружаемый шаблон (в текущем шаблоне эта переменная сохранит старое значение):

```
{% include "/includes/left_menu.html" with category="1" %}
```

Здесь мы задаем новое значение для переменной `category` подгружаемого шаблона includes\left_menu.html. Все остальные переменные, полученные подгружаемым шаблоном от текущего, сохранят старые значения.

Обратите внимание, что между именем переменной, знаком равенства и значением не должно быть пробелов. В противном случае шаблонизатор не сможет обработать этот код.

Мы можем даже указать, чтобы подгружаемый шаблон вообще не получил из текущего шаблона никаких переменных, кроме тех, что мы зададим явно:

```
{% include "/includes/left_menu.html" with category="1" only %}
```

Теперь подгружаемый шаблон includes\left_menu.html получит всего одну переменную — `category`.

Шаблоны и статичные файлы уровня проекта

Все шаблоны и статичные файлы, что мы создали к этому моменту, хранятся в подпапках templates и static папки проекта приложения. Следовательно, они принадлежат данному приложению, отчего и носят название *шаблонов* и *статичных файлов уровня приложения*.

Проблема в том, что, как правило, все шаблоны всех приложений проекта имеют одинаковый дизайн, включают одни и те же изображения и используют одни и те же таблицы стилей и файлы Web-сценариев. Поэтому нам придется либо многократно дублировать родительские шаблоны и статичные файлы во всех приложениях проекта (что непродуктивно и приводит к перерасходу дискового пространства), либо искать иной выход.

Таким выходом может стать перенос всех этих файлов с уровня приложения на уровень проекта (*шаблоны* и *статичные файлы уровня проекта*). Тогда они станут «собственностью» самого проекта и, стало быть, могут быть использованы любым входящим в него приложением.

Сначала мы создадим в папке проекта подпапки templates и static и перенесем в них, соответственно, базовые шаблоны и статичные файлы из всех одноименных папок приложений. После чего зададим в настройках проекта пути к этим папкам.

Из *главы 4* мы помним, что настройки проекта задаются в модуле `settings` пакета проекта. Откроем его.

Список путей к папкам, в которых Django будет искать файлы шаблонов, в дополнение к подпапке templates папки пакета приложения, указывается в переменной `TEMPLATE_DIRS`. Значением этой переменной должен быть кортеж (подойдет и обычный список), каждый элемент которого хранит отдельный путь к папке в виде строки.

Добавим в конце модуля `settings` такой код:

```
TEMPLATE_DIRS = (os.path.join(BASE_DIR, 'templates'),)
```

Он задает кортеж с единственным элементом — полным путем к только что созданной нами подпапке templates папки проекта.

Сразу же усвоим, что в путях к файлам следует применять символ прямого слеша (/), а не обратного (\):

```
TEMPLATE_DIRS = (os.path.join(BASE_DIR, 'others/templates'),)
```

В противном случае мы получим сообщение об ошибке.

Переменная `BASE_DIR` объявляется ранее в том же модуле и хранит полученный в результате вычислений текущий путь к папке проекта. А функция `os.path.join` добавляет к пути, заданному первым параметром, имя папки, заданное вторым параметром, и возвращает полученный результат — полный путь к папке.

А список путей к папкам, где Django будет искать статичные файлы, опять же, в дополнение к подпапке static папки пакета приложения, задается как значение переменной `STATICFILES_DIRS`. Это значение должно быть кортежем (или обычным списком), каждый элемент которого хранит путь к отдельной папке, заданный в виде строки.

Добавим в код модуля `settings` такое выражение:

```
STATICFILES_DIRS = (os.path.join(BASE_DIR, 'static'),)
```

Тем самым мы укажем, что собираемся хранить статичные файлы в подпапке static папки проекта.

> ***ПОРЯДОК ПЕРЕБОРА ПРИЛОЖЕНИЙ***
>
> Django ищет файлы шаблонов и статичные файлы сначала в папках, указанных в переменных `TEMPLATE_DIRS` и `STATICFILES_DIRS`, а потом — в папках *templates* и *static* папок всех приложений, входящих в состав проекта. Приложения при этом будут перебираться в том порядке, в котором они указаны в списке активных приложений (в переменной `INSTALLED_APPS`). Не забываем об этом.

Проверим сайт в работе. Если мы не допустили ошибок, проверка обязательно увенчается успехом.

Формирование интернет-адресов средствами Django

В обоих шаблонах мы создали гиперссылки, указывающие на списки товаров отдельных категорий и на отдельные товары. Так, гиперссылка с интернет-адресом вида **/goods/<*идентификатор категории*>/** укажет на категорию товаров, а гиперссылка с интернет-адресом вида **/goods/good/<*идентификатор товара*>/** — на сам товар.

Но предположим, что мы решили изменить имя виртуальной папки, привязанной к данному приложению, с goods, скажем, на catalog. Тогда нам придется внести соответствующие правки не только в код самой привязки, но и в код шаблонов, формирующий гиперссылки. А это может быть весьма трудоемко...

Но что это мы все пишем вручную! Ведь Django сама может формировать интернет-адреса, основываясь на заданных нами привязках!

И предоставляет нам для этого особый тег шаблона `url`.

```
{% url <имя привязки> [<список параметров, разделенных пробелами>]
[as <переменная шаблона>] %}
```

Параметр *имя привязки* было указано нами ранее — в коде привязки контроллеров к интернет-адресам как значение необязательного параметра `name` функции `url` (см. *главу 6*). Здесь же оно должно быть задано строкой или являться значением переменной шаблона.

Параметры, которые должны быть помещены в состав интернет-адреса, перечисляются после параметра *имя привязки*. Они должны быть указаны в том порядке, в котором в коде привязки, а именно в интернет-адресе — первом параметре функции `url`, были проставлены группы регулярных выражений, что должны извлекать значения этих параметров.

Давайте вспомним написанный ранее код привязки:

```
urlpatterns = patterns('',
  url(r'^(?:(?P<cat_id>\d+)/)?$', views.index, name = "index"),
  url(r'^good/(?P<good_id>\d+)/$', views.good, name = "good"),
)
```

И, держа его перед глазами, соответственно исправим код, создающий гиперссылки в шаблонах.

Начнем с гиперссылок, указывающих на категории, в шаблоне base.html:

```
<a href="{% url "index" cat.id %}">{{ cat.name }}</a>
```

Формируем интернет-адрес, ссылающийся на привязку с именем `index` (это привязка к контроллеру `views.index`), и указываем для него в качестве параметра значение переменной шаблона `cat.id` (идентификатор категории). В результате мы получим интернет-адрес вида **/goods/<*идентификатор категории*>/**.

Ранее в коде привязки, в первом параметре функции `url`, для получения значений параметров мы использовали группы регулярных выражений с именами. Следова-

тельно, для указания значения параметров в теге url мы можем использовать эти имена:

```
<a href="{% url "index" cat_id=cat.id %}">{{ cat.name }}</a>
```

Группа регулярного выражения в привязке имеет у нас имя cat_id. Так что мы использовали для указания значения формируемого интернет-адреса именованный параметр с таким же именем.

Нужно лишь помнить, что между именем параметра, знаком равенства и значением не должно быть пробелов, иначе Django не сможет обработать этот тег.

Если нам потребуется указать в шаблоне один и тот же интернет-адрес несколько раз, мы можем сохранить его в отдельной переменной шаблона:

```
{% url "index" cat_id=cat.id as cat_url %}
```

Затем сохраняем готовый интернет-адрес в переменной шаблона cat_url:

```
<a href="{{ cat_url }}">{{ cat.name }}</a>
```

После чего вставляем его в код гиперссылки.

Закончив, откроем шаблон index.html, найдем код, создающий список товаров, и исправим теги гиперссылок, что указывают на сведения об этих товарах:

```
<a href="{% url "good" good_id=good.id %}">{{ good.name }}</a>
```

Осталось ввести аналогичные правки в код шаблона good.html. Сделаем это самостоятельно. И вновь проверим, работает ли сайт как надо.

Что дальше?

В этой главе мы изучали более сложные инструменты Django по работе с шаблонами: статичные файлы, наследование шаблонов и формирование интернет-адресов на основе сведений о привязке. Мы также занимались оформлением шаблонов, сделав их более похожими на современные Web-страницы. Теперь мы знаем о шаблонах все, что нам может потребоваться.

Следующая глава будет посвящена пагинатору Django — средству для создания многостраничных списков с эффектом «листания». Он очень пригодится нам, если перечень товаров у нас вдруг разрастется до слишком больших размеров.

ГЛАВА 9

Постраничный вывод данных. Пагинатор Django

В предыдущей главе мы закончили разговор о шаблонах, рассмотрев возможности их оформления и верстки, обработку статичных файлов, механизм наследования шаблонов и средства для формирования интернет-адресов на основании привязок. Так что о выводе данных мы теперь знаем практически все, что нам нужно.

Многие современные Web-страницы выводят списки каких-либо позиций: товаров, статей, файлов, электронных писем и др. И, если таких позиций слишком много, список может оказаться весьма большим и неудобным для просмотра.

Выходом из положения может оказаться реализация *постраничного просмотра*. Для таких случаев Django предоставляет нам встроенный инструмент, предназначенный для разбиения больших списков на отдельные страницы, — *пагинатор*.

Инициализация пагинатора

Первым делом его надлежит инициализировать. Для этого нужно импортировать модуль `Paginator` из модуля `django.core.paginator` и создать объект класса `Paginator`:

```
from django.core.paginator import Paginator
```

Конструктор класса `Paginator` принимает следующие обязательные параметры:

- перечень позиций, которые будут разбиваться на страницы. Им может стать любой список Python или объект класса `QuerySet` Django, представляющий список записей модели (подробнее об этом см. в *главе 5*);
- максимальное количество позиций на странице в виде целого числа.

Помимо этого, мы можем передать конструктору еще два необязательных именованных параметра:

- `orphans` — минимальное количество позиций, допустимое на последней странице. Если окажется так, что на последней странице присутствует меньшее количество позиций, чем предписано данным параметром, эти позиции будут размещены на предпоследней странице, которая, таким образом, станет больше. Зна-

чение по умолчанию — 0 (т. е. на последней странице может присутствовать сколько угодно позиций, даже одна);

- `allow_empty_first_page` — имеет смысл, когда переданный конструктору список позиций пуст. Если значение данного параметра равно `True`, при обращении к первой странице мы получим эту страницу, разумеется, пустую. Если же значение параметра равно `False`, при обращении к первой странице будет сгенерировано исключение `EmptyPage` (о нем мы поговорим чуть позже).

Создадим пагинатор, который будет обрабатывать список товаров `goods`, разбивая его на страницы по 10 позиций на каждой и с минимально допустимым количеством позиций на последней странице, равным двум:

```
pag = Paginator(goods, 10, orphans = 2)
```

Класс `Paginator` поддерживает следующие полезные нам свойства:

- `count` — возвращает общее количество позиций в списке:
- `num_pages` — возвращает количество страниц;
- `page_range` — возвращает список, содержащий номера полученных страниц, начиная с единицы (вида [1, 2, 3...]).

Например:

```
page_count = pag.num_pages
```

Кроме того, класс `Paginator` поддерживает метод `page`, о котором сейчас пойдет речь.

Получение заданной страницы списка

Чтобы получить какую-либо страницу списка, нам следует передать контроллеру номер этой страницы. Сделать это можно привычным нам способом — методом GET. Скажем, обращение по интернет-адресу **/goods/2/?page=2** выведет на экран вторую страницу списка товаров, относящихся к категории с идентификатором 2.

Указывать ли параметр номера страницы в привязке — в первом параметре функции `url` (см. *главу 6*), — на взгляд автора, дело вкуса. Автор предпочитает его не указывать. Все равно любой контроллер получает первым параметром объект класса `HttpRequest`, хранящий сведения о запросе, и мы можем получить значение любого переданного методом GET параметра, обратившись к свойству `GET` этого класса.

Как мы помним из *главы 6*, это свойство хранит словарь с элементами, содержащими значения всех параметров, переданных тому или иному контроллеру методом GET. Получить значение нужного параметра мы можем по его ключу, чье имя совпадает с именем GET-параметра:

```
def index(request, cat_id):
  try:
    page_num = request.GET["page"]
```

```
  except KeyError:
    page_num = 1
  pag = Paginator(goods, 10, orphans = 2)
```

Поскольку параметр `page` может отсутствовать, нам следует обрабатывать исключение `KeyError`, возникающее, если словарь не содержит элемента с указанным ключом (подробнее об этом исключении говорилось в *главе 2*).

Ранее мы упоминали о методе `page` класса `Paginator`. Этот метод принимает в качестве единственного параметра номер страницы (нумерация страниц начинается с единицы) в виде целого числа и возвращает объект класса `Page`, который, в свою очередь, представляет список позиций, находящихся на странице с указанным номером:

```
goods = pag.page(page_num)
```

Мы можем манипулировать содержимым этого списка теми же способами, что применяли в случае знакомого нам по *главе 5* списка записей `QuerySet`. В том числе, можно передать его методу `render`:

```
return render(request, "index.html", {"category": cat, "cats": cats,↵
"goods": goods})
```

Метод `page` может генерировать три исключения, которые нам придется обрабатывать. Все эти исключения объявлены в модуле `django.core.paginator`:

- `InvalidPage` — возникает в случае некорректного задания номера страницы (например, если он представляет собой число с плавающей точкой или строку) или при обращении к несуществующей странице (типичный случай — список разбит на десять страниц, а выполняется обращение к одиннадцатой);
- `PageNotAnInteger` — возникает в случае некорректного задания номера страницы;
- `EmptyPage` — возникает в случае обращения к несуществующей странице.

Классы исключений `PageNotAnInteger` и `EmptyPage` являются потомками класса `InvalidPage`. Так что мы можем обрабатывать либо `InvalidPage`, либо `PageNotAnInteger` и `EmptyPage`:

```
from django.core.paginator import InvalidPage
. . .
try:
  goods = pag.page(page_num)
except InvalidPage:
  goods = pag.page(1)
```

Класс `Page` поддерживает два полезных свойства:

- `number` — возвращает номер данной страницы;
- `paginator` — возвращает объект класса `Paginator`, с применением которого была создана эта страница.

Он также поддерживает ряд методов, перечисленных в табл. 9.1. Все эти методы не принимают параметров.

Таблица 9.1. *Методы класса* `Page`

Метод	Описание
`has_next`	Возвращает `True`, если это не последняя страница
`has_previous`	Возвращает `True`, если это не первая страница
`has_other_pages`	Возвращает `True`, если это не единственная страница
`next_page_number`	Возвращает номер следующей страницы. Если это последняя страница, генерируется исключение `InvalidPage`
`previous_page_number`	Возвращает номер предыдущей страницы. Если это первая страница, генерируется исключение `InvalidPage`
`start_index`	Возвращает индекс первой позиции списка из присутствующих на странице
`end_index`	Возвращает индекс последней позиции списка из присутствующих на странице

Эти методы пригодятся нам, в основном, в шаблонах, в контроллерах же они применяются довольно редко.

Давайте перепишем код контроллера `views.index` так, чтобы список товаров выводился постранично. Зададим количество элементов на странице равным 10:

```
from django.core.paginator import Paginator, InvalidPage
def index(request, cat_id):
  try:
    page_num = request.GET["page"]
  except KeyError:
    page_num = 1
  cats = Category.objects.all().order_by("name")
  if cat_id == None:
    cat = Category.objects.first()
  else:
    cat = Category.objects.get(pk = cat_id)
  paginator = Paginator(Good.objects.filter(category = cat).↵
  order_by("name"), 10)
  try:
    goods = paginator.page(page_num)
  except InvalidPage:
    goods = paginator.page(1)
  return render(request, "index.html", {"category": cat, "cats": cats,↵
  "goods": goods})
```

Формирование гиперссылок для перехода между страницами

Осталось сделать так, чтобы на результирующих Web-страницах выводились гиперссылки для переходов между страницами. Это:

- гиперссылка для перехода на предыдущую страницу;
- набор гиперссылок для перехода на конкретные страницы списка (первую, вторую, третью и т. д.);
- гиперссылка для перехода на следующую страницу.

Вывести их не проблема, ведь мы уже знаем, как это делается. Однако здесь нужно учесть ряд условий:

- упомянутый ранее набор гиперссылок следует выводить только в том случае, если список содержит больше одной страницы (в противном случае этот набор просто не нужен);
- гиперссылку для перехода на предыдущую страницу следует выводить только в том случае, если это не первая страница списка;
- гиперссылку для перехода на следующую страницу следует выводить только в том случае, если это не последняя страница списка.

Исходя из всего этого, давайте исправим код шаблона index.html для реализации постраничного вывода списка товаров — отыщем закрывающий тег `<table>`, что формирует таблицу списка товаров, и вставим после него вот такой довольно объемистый код (выделен полужирным шрифтом):

```
</table>
{% if goods.paginator.num_pages > 1 %}
  <div id="pagination">
    {% if goods.has_previous %}
      <div id="previous-page"><a href="{% url "index"↵
      cat_id=category.pk %}?page={{ goods.previous_page_number }}">↵
      &lt;</a></div>
    {% endif %}
    {% if goods.has_next %}
      <div id="next-page"><a href="{% url "index" cat_id=category.pk%}↵
      ?page={{ goods.next_page_number }}">&gt;</a></div>
    {% endif %}
    <div id="num-pages">
      {% for pn in goods.paginator.page_range %}
        {% if not forloop.first %}
            |  
        {% endif %}
        {% if goods.number != pn %}
          <a href="{% url "index" cat_id=category.pk %}?page={{ pn }}">
        {% endif %}
```

```
          {{ pn }}
          {% if goods.number != pn %}
            </a>
          {% endif %}
        {% endfor %}
      </div>
    </div>
  {% endif %}
{% endblock %}
```

Обратите внимание, как выполняется проверка на необходимость вывода всего набора гиперссылок и отдельных гиперссылок для перехода на предыдущую и последующую страницы и как формируются интернет-адреса отдельных страниц.

Теперь добавим к таблице стилей main.css следующий код:

```
#pagination {
  margin-top: 20px;
}
#pagination #previous-page {
  float: left;
}
#pagination #num-pages {
  text-align: center;
}
#pagination #next-page {
  float: right;
}
```

Он задаст оформление для набора гиперссылок.

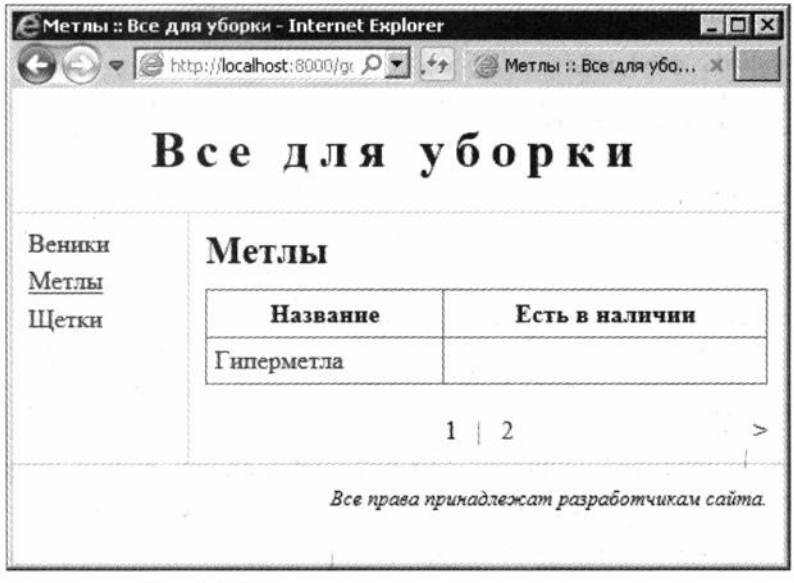


Рис. 9.1. Список товаров с постраничным выводом

Сохраним все файлы и, как обычно, проверим сайт в работе. Поскольку у нас список товаров невелик, временно зададим в целях отладки количество позиций на странице равным одному. Страница со списком товаров будет выглядеть так, как показано на рис. 9.1.

Готово? Еще нет. Нам осталось решить одну небольшую проблему...

Возврат на корректную страницу списка

Давайте перейдем на вторую страницу списка товаров и попробуем просмотреть сведения о товаре. А затем, находясь на странице сведений о товаре, щелкнем гиперссылку **Назад**, чтобы вернуться на список товаров.

Проблема в том, что мы вернемся на первую страницу этого списка. Мы ведь не задали в интернет-адресе гиперссылки **Назад** параметр с номером нужной страницы, так что контроллер `views.index` выведет нас на первую страницу списка. Собственно, так мы его и запрограммировали...

Но как реализовать возврат на ту страницу списка товаров, с которой мы перешли на страницу подробных сведений? Ответ напрашивается сам собой: передать контроллеру `views.good` номер нужной страницы, чтобы он сформировал в гиперссылке **Назад** шаблона good.html интернет-адрес, указывающий на эту страницу.

Сказано — сделано! Снова откроем шаблон index.html и найдем код, формирующий таблицу со списком товаров, а в нем — открывающий тег `<a>`, что создает гиперссылку с указанием на подробные сведения о товаре. Исправим его следующим образом:

```
<a href="{% url "good" good_id=good.id %}?page={{ goods.number }}">
```

Далее перепишем код контроллера `views.good`, чтобы он получал номер страницы и передавал его в шаблон good.html:

```
def good(request, good_id):
  try:
    page_num = request.GET["page"]
  except KeyError:
    page_num = 1
  cats = Category.objects.all().order_by("name")
  try:
    good = Good.objects.get(pk = good_id)
  except Good.DoesNotExist:
    raise Http404
  return render(request, "good.html", {"cats": cats, "good": good,↩
  "pn": page_num})
```

Осталось соответственно исправить код шаблона good.html, точнее, открывающего тега `<a>`, что создает гиперссылку возврата на список товаров:

```
<a href="{% url "index" cat_id=good.category.id %}?page={{ pn }}">
```

Вот теперь все действительно готово!

Что дальше?

В этой главе мы познакомились с пагинатором Django — замечательным инструментом, который поможет нам, если мы захотим выводить большие списки каких-либо позиций разбитыми на отдельные страницы.

В следующей главе мы познакомимся с другой разновидностью контроллеров, поддерживаемой Django, — классами-контроллерами. Во многих случаях они много удобнее знакомых нам функций-контроллеров.

ГЛАВА 10

Вывод на основе классов. Классы-контроллеры Django

В предыдущей главе мы изучали пагинатор Django, который поможет нам разбить слишком длинные списки на страницы. Полезная штука, не правда ли?

К этому моменту мы написали два контроллера. Оба представляют собой обычные функции Python. Оба получают переданные в составе интернет-адреса параметры, извлекают из модели нужные данные, возможно, разбивают их на страницы с помощью пагинатора и выводят их через соответствующий шаблон. Казалось, чего еще желать?

Можно пожелать большего. Скажем, переложить труд по выборке нужной записи из модели и разбиению списка записей на страницы на плечи самой Django. Фантастика? Отнюдь. С так называемыми *классами-контроллерами* Django — это реальность.

Введение в классы-контроллеры

Помимо давно знакомых нам функций-контроллеров, Django позволяет нам создать другой тип контроллеров, представляющих собой классы (*классы-контроллеры*). Мы можем объявить их в модуле `views` (если контроллеров немного) или в отдельном модуле или модулях.

В коде привязки класс-контроллер также указывается вторым параметром функции `url` (см. *главу 6*). Однако для этого применяется следующий формат:

<имя класса-контроллера>`.as_view([`*<параметры>*`])`

Фактически мы вызываем метод класса-контроллера `as_view`. При этом мы, возможно, передаем этому методу параметры, которые укажут значения свойств создаваемого объекта контроллера, — эти параметры напрямую повлияют на его работу.

```
urlpatterns = patterns('',
  url(r'^(?:(?P<cat_id>\d+)/)?$', GoodListView.as_view↵
  (template_name = "index.html"), name = "index"),
```

```
  url(r'^good/(?P<good_id>\d+)/$', GoodDetailView.as_view↵
  (template_name = "good.html"), name = "good"),
)
```

Здесь мы привязываем классы-контроллеры `GoodListView` и `GoodDetailView`, передавая им параметр `template_name` — имя файла шаблона, с применением которого будут выводиться данные.

В остальном привязка классов-контроллеров выполняется так же, как и привязка функций-контроллеров.

Django предоставляет нам несколько готовых классов-контроллеров. Мы можем использовать их как есть, а можем создать на их основе классы-потомки. Последний вариант во многих случаях предпочтительнее.

Класс-контроллер *TemplateView*

Класс `TemplateView` — самый простой из классов-контроллеров Django. Он объявлен в модуле `django.views.generic.base`:

```
from django.views.generic.base import TemplateView
```

Свойств этот класс поддерживает немного. Прежде всего, это уже знакомое нам свойство `template_name`, задающее файл шаблона. Его значением должна быть строка:

```
url(r'^about$', TemplateView.as_view(template_name =↵
"/others/about.html"), name = "about"),
```

Здесь мы привязываем к интернет-адресу **about/** класс-контроллер `TemplateView` и указываем для него шаблон **others/about.html**.

```
class AboutView(TemplateView):
  template_name = "/others/about.html"
```

А здесь порождаем на основе класса `TemplateView` потомок с именем `AboutView`, в котором задаем имя файла шаблона. (Хотя, конечно, ради этого объявлять новый класс все же не стоит...)

```
url(r'^about$', AboutView.as_view(), name = "about"),
```

Здесь мы привязываем вновь объявленный класс-контроллер к интернет-адресу **about/**.

Далее, свойство `request`. Оно хранит уже знакомый нам по *главе 6* объект класса `HttpRequest`, представляющий сведения о запросе:

```
try:
  page_num = self.request.GET["page"]
except KeyError:
  page_num = 1
```

Здесь мы получаем значение GET-параметра `page` — номер страницы списка. (Безусловно, этот код будет работать лишь в теле какого-либо метода класса-потомка, поскольку использует для ссылки на сам объект переменную `self`.)

Свойство `args` хранит список, представляющий все параметры, что были указаны в привязке с применением обычных, «безымянных», групп регулярных выражений. Эти параметры присутствуют в списке в том порядке, в котором в привязке были объявлены соответствующие им группы:

```
url(r'^good/(\d+)/$', GoodDetailView.as_view(), name = "good"),
. . .
good_id = self.args[0]
```

Здесь в методе класса `GoodDetailView`, порожденного от `TemplateView`, мы получаем значение идентификатора товара.

А свойство `kwargs` хранит список, представляющий все параметры, что были указаны в привязке с применением именованных групп регулярных выражений:

```
url(r'^good/(?P<good_id>\d+)/$', GoodDetailView.as_view(),↵
name = "good"),
. . .
good_id = self.kwargs["good_id"]
```

Все это, конечно, замечательно. Но как нам сформировать для шаблона контекст данных? Без этого толку от классов-контроллеров немного.

Очень просто. Для этого мы переопределим в классе-потомке метод `get context data`, который как раз и создает контекст данных. Этот метод принимает в качестве параметра словарь со всеми именованными параметрами, соответствующими переданным контроллеру данным, так что нам не придется пользоваться упомянутым ранее свойством `kwargs`. Отметим, что все эти параметры переносятся в контекст данных, и мы можем использовать их в шаблоне.

При переопределении метода `get context data` нам следует помнить две вещи. Во-первых, сначала нам обязательно следует вызвать метод класса-родителя, который, собственно, сформирует сам контекст данных и заполнит его изначальными данными — в частности, значениями полученных контроллером параметров. Во-вторых, созданный контекст данных должен быть возвращен из метода в качестве результата:

```
url(r'^good/(?P<good_id>\d+)/$', GoodDetailView.as_view(),↵
name = "good"),
. . .
class GoodDetailView(TemplateView):
  template_name = "good.html"
  def get_context_data(self, **kwargs):
    context = super(GoodDetailView, self).get_context_data(**kwargs)
    context["good"] = Goods.objects.get(pk = good_id)
    return context
```

Здесь метод `get_context_data` унаследован от класса `ContextMixin`, свойство `template_name` — от класса `TemplateResponseMixin`, а остальная функциональность класса `TemplateView` — от класса `View`. Все эти классы также объявлены в модуле `django.views.generic.base`.

Настала пора испытать класс-контроллер `TemplateView` в деле. Откроем модуль `urls` пакета приложения, закомментируем уже имеющийся там код привязки и добавим новый, вот такой:

```
from page.twviews import GoodListView, GoodDetailView
urlpatterns = patterns('',
  url(r'^(?:(?P<cat_id>\d+)/)?$', GoodListView.as_view(),↳
  name = "index"),
  url(r'^good/(?P<good_id>\d+)/$', GoodDetailView.as_view(),↳
  name = "good"),
)
```

Мы привязали интернет-адреса к классам-контроллерам `GoodListView` и `GoodDetailView`, которые вскоре объявим в новом модуле `twviews` пакета приложения.

Создадим этот модуль и наберем в нем такой код:

```
from django.views.generic.base import TemplateView
from django.core.paginator import Paginator, InvalidPage
from page.models import Category, Good

class GoodListView(TemplateView):
  template_name = "index.html"
  def get_context_data(self, **kwargs):
    context = super(GoodListView, self).get_context_data(**kwargs)
    try:
      page_num = self.request.GET["page"]
    except KeyError:
      page_num = 1
    context["cats"] = Category.objects.order_by("name")
    if kwargs["cat_id"] == None:
      context["category"] = Category.objects.first()
    else:
      context["category"] = Category.objects.get(pk = kwargs["cat_id"])
    paginator = Paginator(Good.objects.filter(category =↳
    context["category"]).order_by("name"), 1)
    try:
      context["goods"] = paginator.page(page_num)
    except InvalidPage:
      context["goods"] = paginator.page(1)
    return context

class GoodDetailView(TemplateView):
  template_name = "good.html"
  def get_context_data(self, **kwargs):
    context = super(GoodDetailView, self).get_context_data(**kwargs)
    try:
      context["pn"] = self.request.GET["page"]
```

```
    except KeyError:
      context["pn"] = 1
    context["cats"] = Category.objects.order_by("name")
    try:
      context["good"] = Good.objects.get(pk = kwargs["good_id"])
    except Good.DoesNotExist:
      raise Http404
    return context
```

Проверим сайт. Все должно работать.

Пока еще мы не ощутили большой пользы от классов-контроллеров. В самом деле, объем кода практически не уменьшился — мы лишь перенесли код из функции-контроллера в тело метода `get_context_data` соответствующего класса с минимальными изменениями. Так что выбор между функциями-контроллерами и классами — потомками `TemplateView` — по большей части, дело вкуса.

Но все изменится, когда мы применим в работе более специализированные классы-контроллеры — потомки класса `TemplateView`. С ними разработка контроллеров станет заметно проще.

Класс-контроллер списка *ListView*

Класс-контроллер `ListView`, объявленный в модуле `django.views.generic.list`, предназначен для вывода списков каких-либо позиций — например, товаров. По сравнению со своим родителем `TemplateView` он предоставляет следующую дополнительную функциональность:

- формирование в контексте данных переменной, хранящей список записей из указанной модели;
- встроенную поддержку пагинации.

Неплохое подспорье для нас, разработчиков!

Список свойств, поддерживаемых этим классом, приведен в табл. 10.1. Все они унаследованы от класса `MultipleObjectMixin`, также объявленного в модуле `django.views.generic.list`.

***Таблица 10.1.** Свойства класса `ListView`*

Свойство	Описание
`model`	Класс модели, из которой будут браться записи для вывода
`queryset`	Список записей для вывода в виде объекта класса `QuerySet` (см. *главу 5*). Значение этого свойства имеет приоритет перед значением свойства `model`
`context_object_name`	Имя переменной контекста данных, в которой будет храниться список записей. Если не указано, будет создана переменная `object_list`
`paginate_by`	Количество позиций на странице при задействованной пагинации. Значение по умолчанию — `None` (т. е. пагинация не выполняется)

Таблица 10.1 (окончание)

Свойство	Описание
paginate_orphans	Минимальное количество позиций, допустимое на последней странице (за подробностями — к *главе 9*). Значение по умолчанию — 0
allow_empty	Имеет смысл лишь в том случае, когда список записей пуст. Если значение этого свойства равно True, будет выведена одна пустая страница, в противном случае будет выполнен переход на страницу «ошибки 404». Значение по умолчанию — True
page_kwarg	Строка с именем параметра, которым в интернет-адресе передается номер страницы. Значение по умолчанию — page

Пару слов о поддержке пагинации. Номер страницы передается в интернет-адресе с параметром, чье имя, как мы знаем, указывается в свойстве page_kwarg. Этот параметр может как входить в состав самого интернет-адреса (мы так передаем идентификаторы категории и товара — в таком случае следует указать в интернет-адресе в привязке соответствующую группу регулярного выражения), так и указываться в GET-параметре (мы так его и передаем):

```
from django.views.generic.list import ListView
class GoodListView(ListView):
  template_name = "index.html"
  queryset = Good.objects.order_by("name")
  paginate_by = 10
```

Здесь мы порождаем от класса ListView потомок GoodListView.

Не принимающий параметров метод get_queryset также унаследован от класса MultipleObjectMixin. Он возвращает список записей, которые будут выводиться на экран. Мы можем переопределить этот метод в классе-потомке, если нам требуется фильтровать или сортировать записи.

Метод get_context_data, переопределенный в классе MultipleObjectMixin, формирует контекст данных со следующими переменными:

- переменная, хранящая список записей для вывода. Ее имя задается в свойстве context_object_name (по умолчанию — object_list);
- *<имя класса модели, набранное прописными буквами>*_list — то же самое, что предыдущая;
- is_paginated — True, если пагинация задействована, и False в противном случае;
- paginator — объект класса Paginator (см. *главу 9*), с применением которого выполняется пагинация;
- page_obj — объект класса Page, представляющий текущую страницу;
- и, разумеется, переменные, хранящие значения всех полученных контроллером параметров, для которых были созданы именованные группы регулярных выражений.

Мы можем переопределить этот метод в потомке, чтобы добавить в контекст дополнительные данные.

Метод `get`, унаследованный от другого класса — `BaseListView`, но объявленного в том же модуле `django.views.generic.list`, присваивает переменной контекста данных, в которой должен храниться список записей, этот самый список. Он выполняется раньше перечисленных ранее методов и сам вызывает сначала метод `get_context_data`, а потом — метод `get_queryset`.

Метод `get` принимает следующие параметры:

- объект класса `HttpRequest`, хранящий сведения о запросе (см. *главу 6*);
- список со всеми неименованными параметрами, соответствующими переданным контроллеру данным;
- словарь со всеми именованными параметрами.

Этот метод, разумеется, переопределенный в потомке, — идеальное место для выполнения кода, формирующего данные, которые будут использоваться в остальных методах:

```
from django.views.generic.list import ListView
class GoodListView(ListView):
  template_name = "index.html"
  paginate_by = 10
  cat = None

  def get(self, request, *args, **kwargs):
    if self.kwargs["cat_id"] == None:
      self.cat = Category.objects.first()
    else:
      self.cat = Category.objects.get(pk = self.kwargs["cat_id"])
    return super(GoodListView, self).get(request, *args, **kwargs)

  def get_context_data(self, **kwargs):
    context = super(GoodListView, self).get_context_data(**kwargs)
    context["cats"] = Category.objects.order_by("name")
    context["category"] = self.cat
    return context

  def get_queryset(self):
    return Good.objects.filter(category = self.cat).order_by("name")
```

Мы объявили в классе `GoodListView` свойство `cat`, в котором будет храниться выбранная категория, присвоили ему значение в теле переопределенного метода `get`, а потом использовали это значение в методах `get_context_data` и `get_queryset`.

Что ж, полный код нового класса-контроллера `GoodListView` у нас есть. Откроем модуль `twviews` и вставим этот код туда, не забыв закомментировать уже имеющееся там объявление одноименного старого класса.

Осталось открыть шаблон index.html, найти блок шаблона `main` и исправить его содержимое таким образом:

```
{% block main %}
  <h2>{{ category.name }}</h2>
  <table>
    <tr>
      <th>Название</th>
      <th>Есть в наличии</th>
    </tr>
    {% for good in object_list %}
      <tr>
        <td><a href="{% url "good" good_id=good.id %}?
        page={{ page_obj.number }}">{{ good.name }}</a></td>
        <td class="centered">{% if good.in_stock %}+{% endif %}</td>
      </tr>
    {% endfor %}
  </table>
  {% if paginator.num_pages > 1 %}
    <div id="pagination">
      {% if page_obj.has_previous %}
        <div id="previous-page"><a href="{% url "index"
        cat_id=category.pk %}?page={{ page_obj.previous_page_number }}">
        &lt;</a></div>
      {% endif %}
      {% if page_obj.has_next %}
        <div id="next-page"><a href="{% url "index"
        cat_id=category.pk %}?page={{ page_obj.next_page_number }}">
        &gt;</a></div>
      {% endif %}
      <div id="num-pages">
        {% for pn in paginator.page_range %}
          {% if not forloop.first %}
              |  
          {% endif %}
          {% if page_obj.number != pn %}
            <a href="{% url "index" cat_id=category.pk %}?page={{ pn }}">
          {% endif %}
          {{ pn }}
          {% if page_obj.number != pn %}
            </a>
          {% endif %}
        {% endfor %}
      </div>
    </div>
  {% endif %}
{% endblock %}
```

Да, в результате перехода на новый класс-контроллер объем кода уменьшился ненамного. Но зато нам теперь не придется беспокоиться о создании пагинации, поскольку этим за нас займется класс `ListView`.

Класс-контроллер подробных сведений *DetailView*

Класс-контроллер `DetailView`, что объявлен в модуле `django.views.generic.detail`, наилучшим образом подходит для вывода подробных сведений о какой-либо позиции — например, о товаре. Он также является потомком класса `TemplateView` и, по сравнению с ним, может:

- извлечь из интернет-адреса параметр — идентификатор выводимой позиции;
- выбрать из модели соответствующую ему запись;
- создать в контексте данных шаблона переменную, которая будет хранить эту запись.

Как видим, этот класс возьмет на себя бóльшую часть работы по выводу сведений о товаре.

Класс предоставляет нам несколько полезных свойств, перечисленных в табл. 10.2. Эти свойства унаследованы от класса `SingleObjectMixin` из модуля `django.views.generic.detail`.

***Таблица 10.2.** Свойства класса `DetailView`*

Свойство	Описание
`model`	Класс модели, из которой будет выбрана запись для вывода
`queryset`	Список записей, из которого будет выбрана запись для вывода, в виде объекта класса `QuerySet`. Значение этого свойства имеет приоритет перед значением свойства `model`
`context_object_name`	Имя переменной контекста данных, в которой будет храниться выбранная запись. Если не указано, будет создана переменная `object`
`pk_url_kwarg`	Строка с именем параметра, которым передается идентификатор записи. Значение по умолчанию — `pk`
`slug_field`	Строка с именем поля модели, в котором хранится короткий заголовок или название, применяемый в SEO-дружественных интернет-адресах (обычно оно имеет тип `SlugField` — подробнее см. в *главе 5*). Если не указан, будет использовано поле модели с именем `slug`
`slug_url_kwarg`	Строка с именем параметра, которым передается короткий заголовок или название. Значение по умолчанию — `slug`

Сначала класс-контроллер проверяет, был ли ему передан параметр с именем, заданным в свойстве `slug_url_kwarg`, и, если он был передан, найдет запись, в кото-

рой значение поля, чье имя указано свойством slug_field, совпадает со значением этого параметра. Если же такого параметра нет, контроллер получит значение параметра с именем, заданным свойством pk_url_kwarg, и будет искать запись с идентификатором, совпадающим с этим значением:

```
from django.views.generic.detail import DetailView
class GoodDetailView(DetailView):
  template_name = "good.html"
  model = Good
  pk_url_kwarg = "good_id"
```

Здесь мы порождаем от класса DetailView потомок GoodDetailView.

Не принимающий параметров метод get_queryset возвращает список записей, из которых будет выбираться нужная. Мы можем переопределить этот метод в классе-потомке, если нам требуется выполнять с этим списком какие-либо дополнительные действия.

Метод get_object возвращает выбранную из списка запись. Мы можем передать в качестве необязательного параметра queryset набор записей, из которого будет выбираться запись, вместо указанного в свойствах model или queryset. Если мы хотим выбирать запись согласно более сложному критерию, нежели идентификатор или короткий заголовок, мы переопределим этот метод в потомке.

Метод get_context_data формирует в контексте данных шаблона следующие переменные:

- переменная, хранящая выбранную запись. Ее имя задается в свойстве context_object_name (по умолчанию — object);
- *<имя класса модели, набранное прописными буквами>* — то же самое, что предыдущая.

Переопределив этот метод в потомке, мы сможем вставить в него код, добавляющий в контекст другие данные.

Все перечисленные ранее три метода также унаследованы от класса SingleObjectMixin.

Метод get, унаследованный от класса BaseDetailView из модуля django.views.generic.detail, присваивает выбранную запись предназначенной для нее переменной контекста данных. Он выполняется раньше перечисленных уже методов и сам вызывает сначала метод get_context_data, а потом — методы get_object и get_queryset. Мы можем этим воспользоваться, если хотим сформировать данные, что будут использоваться в других методах, переопределив этот метод в потомке:

```
from django.views.generic.detail import DetailView
class GoodDetailView(DetailView):
  template_name = "good.html"
  model = Good
  pk_url_kwarg = "good_id"
```

```
def get_context_data(self, **kwargs):
  context = super(GoodDetailView, self).get_context_data(**kwargs)
  try:
    context["pn"] = self.request.GET["page"]
  except KeyError:
    context["pn"] = "1"
  context["cats"] = Category.objects.order_by("name")
  return context
```

Это полный код нового класса-контроллера `GoodDetailView`. Так что мы можем открыть модуль `twviews` и вставить его туда, предварительно закомментировав код объявления старого класса с тем же именем.

Поскольку в старом классе-контроллере в контексте данных для хранения выбранного товара мы формировали переменную шаблона `good`, а класс `DetailView` сам формирует для этого переменную, чье имя совпадает с именем класса модели, набранным прописными буквами, нам не придется переделывать шаблон good.html. Так что можно сразу же приступить к проверке работоспособности сайта.

Как видим, код нового класса-контроллера получился несколько компактнее. И это притом, что он выполняет довольно сложную работу по выборке списка категорий и получению номера страницы. В более простых случаях выигрыш от использования классов-контроллеров может быть еще более заметным.

Вынос общей функциональности в другие классы

Оба наших класса-контроллера при работе выполняют одно и то же действие — получают список категорий и помещают его в переменную шаблона `cats`. Писать код, выполняющий это действие, в каждом классе-контроллере непродуктивно, особенно если сайт достаточно сложен и включает множество приложений и, соответственно, контроллеров. Нельзя ли вынести этот код в другой класс?

Разумеется, можно. Только сначала следует решить, в какой именно.

Классы-контроллеры `TemplateView`, `ListView` и `DetailView` имеют несколько родителей (благо Python допускает множественное наследование — см. *главу 2*). Прежде всего, это классы `View`, `BasicListView` и `BasicDetailView`, реализующие базовую функциональность по обработке запросов и занесению в контекст данных выводимой информации. Нам они не очень интересны..

Перечисленные ранее классы также имеют в списке родителей классы `TemplateResponseMixin`, `MultipleObjectTemplateResponseMixin` (модуль `django.views.generic.list`) и `SingleObjectTemplateResponseMixin` (модуль `django.views.generic.detail`). Эти классы реализуют выборку нужного шаблона и собственно его рендеринг — они также не очень нам интересны.

Гораздо больший интерес представляют другие классы, занимающиеся подготовкой контекста данных и выборкой предназначенной для вывода информации.

Давайте их рассмотрим:

- класс `ContextMixin`, реализующий создание контекста данных. От него наследуется класс `TemplateView`;
- класс `MultipleObjectMixin`, реализующий выборку из модели и помещение в контекст данных списка выводимых записей. Является потомком класса `ContextMixin`. От него наследуется класс `ListView`;
- класс `SingleObjectMixin`, реализующий выборку записи с указанным идентификатором или коротким заголовком и помещение ее в контекст данных. Также является потомком класса `ContextMixin`. От него наследуется класс `DetailView`.

А теперь подумаем. Мы хотим вынести в отдельный класс код, который будет добавлять в контекст данных переменную со списком категорий. Следовательно, его родитель уже должен реализовывать функциональность по созданию контекста данных и при этом являться общим родителем для классов `ListView` и `DetailView`. В таком случае наилучшим выбором на роль родителя нашего нового класса будет `ContextMixin`.

Вставим в модуль `twviews` перед объявлениями обоих классов-контроллеров вот такой код:

```
from django.views.generic.base import ContextMixin
class CategoryListMixin(ContextMixin):
  def get_context_data(self, **kwargs):
    context = super(CategoryListMixin, self).get_context_data(**kwargs)
    context["cats"] = Category.objects.order_by("name")
    return context
```

Здесь все нам уже знакомо. Мы переопределяем метод `get_context_data`, в котором и реализуем дополнение контекста данных списком категорий.

Осталось исправить объявления классов-контроллеров, добавив в список их родителей вновь объявленный класс и закомментировав код, получающий список категорий (исправленный код выделен полужирным шрифтом).

```
class GoodListView(ListView, CategoryListMixin):
  . . .
  def get_context_data(self, **kwargs):
    . . .
    # context["cats"] = Category.objects.order_by("name")
. . .
class GoodDetailView(DetailView, CategoryListMixin):
  . . .
  def get_context_data(self, **kwargs):
    . . .
    # context["cats"] = Category.objects.order_by("name")
```

Проверим сайт в работе. После чего удалим закомментированный код — он совершенно не нужен, а лишь увеличивает размер модуля и вносит путаницу.

Классы-контроллеры для вывода по датам

Часто приходится выводить какие-либо данные, основываясь на хранящихся в них значениях даты (вывод по датам). Например, если мы реализуем на сайте список новостей, нам обязательно понадобится вывести их список отсортированным по убыванию значения даты, когда новость была создана. Также может оказаться полезным вывести, скажем, новости, созданные в течение определенного года или месяца.

Конечно, мы можем сами написать нужные классы-контроллеры, взяв в качестве родителя описанный ранее класс `ListView`. Но делать это совершенно не обязательно, т. к. Django уже включает в свой состав несколько классов-контроллеров, специально предназначенных для вывода данных по датам. И сейчас мы о них поговорим.

Начнем с того, что все эти классы являются потомками класса `MultipleObjectMixin` и, стало быть, поддерживают все свойства, что перечислены в табл. 10.1. Да-да, в том числе, они уже имеют встроенную поддержку пагинации, что для нас большое подспорье. И, разумеется, они поддерживают свойство `template_name`, в котором задается файл шаблона.

Далее отметим, что свойство `allow_empty`, описанное в табл. 10.1, для этих классов имеет значение по умолчанию `False`, и это нужно иметь в виду.

Все классы, что мы сейчас рассмотрим, объявлены в модуле `django.views.generic.dates`.

Класс-контроллер архива *ArchiveIndexView*

Класс-контроллер `ArchiveIndexView` предназначен для вывода списка записей, отсортированных по убыванию значения даты («архива»). Он отлично подходит для вывода архива новостей, списка статей блога и других подобных задач.

Одним из родителей этого класса является `DateMixin`, поддерживающий следующие свойства:

- `date_field` — задает в виде строки имя поля модели, на основе которого будет выполняться сортировка записей. Это поле должно иметь тип `DateField` или `DateTimeField`;
- `allow_future` — если `True`, в списке будут присутствовать записи со значениями дат, относящимися к будущему. Если `False`, такие записи не будут включаться в список. Значение по умолчанию — `False` (и правильно — кому нужны новости из будущего?).

> ***Записи, помеченные сегодняшней датой***
>
> Если для свойства `allow_future` задано значение `True`, то в список не попадут и записи, помеченные сегодняшней датой. Это досадный недостаток Django, который нужно всегда учитывать в работе.

Рассмотрим пример:

```
class New(models.Model):
  title = models.CharField(max_length = 100, db_index = True)
  description = models.TextField()
  content = models.TextField()
  pub_date = models.DateField(db_index = True, auto_now_add = True)
```

Здесь мы создаем модель `New`, в которой будут храниться новости. Задаем для нее поля `title` (заголовок новости), `description` (краткое описание, которое будет выводиться в списке новостей наряду с заголовком), `content` (содержание новости) и `pub_date` (дата ее создания). Для последнего поля задаем автоматическое занесение в него значения при создании записи:

```
from django.views.generic.dates import ArchiveIndexView
from news.models import New
class NewsArchiveView(ArchiveIndexView):
  model = New
  date_field = "pub_date"
  template_name = "news_archive.html"
```

Затем мы объявляем класс-контроллер `NewsArchiveView`, который будет выводить список новостей (в тестовых целях автор создал новое приложение с именем `news`, предназначенное для работы с новостями):

```
from news.views import NewsArchiveView
urlpatterns = patterns('',
  url(r'^$', NewsArchiveView.as_view(), name = "news_archive"),
)
```

И не забываем выполнить его привязку.

Поскольку наш класс-контроллер не содержит методов, мы можем вообще его не объявлять, а использовать непосредственно класс `ArchiveIndexView`:

```
from django.views.generic.dates import ArchiveIndexView
from news.models import New
urlpatterns = patterns('',
  url(r'^$', ArchiveIndexView.as_view(model = New,↵
  date_field = "pub_date", template_name = "index.html"),↵
  name = "news_archive"),
)
```

Класс-контроллер `ArchiveIndexView` создает в контексте данных шаблона две переменные:

- `latest` — список записей для вывода;
- `date_list` — список всех годов, взятых из указанного нами ранее в свойстве `date_field` поля модели. Они представляются в виде значений даты, в котором нас интересует лишь год.

Помимо этого, контекст данных включит все переменные, относящиеся к пагинации (см. раздел, посвященный классу `ListView`).

```
{% for new in latest %}
  <h3>{{ new.title }}</h3>
  <p>{{ new.pub_date }}</p>
  <p>{{ new.description }}</p>
{% endfor %}
```

Здесь мы выводим в шаблоне список новостей.

```
<ul>
  {% for date in date_list %}
    <li>{{ date|date:"Y" }}</li>
  {% endfor %}
</ul>
```

А здесь — список всех годов, в которые была создана хотя бы одна новость.

Если нам следует вывести в шаблоне какие-либо сторонние данные — например, список категорий товаров, мы переопределим в классе-контроллере метод `get_context_data`, как делали это ранее:

```
from pages.models import Category
class NewsArchiveView(ArchiveIndexView):
  . . .
  def get_context_data(self, **kwargs):
    context = super(NewsArchiveView, self).get_context_data(**kwargs)
    context["cats"] = Category.objects.order_by("name")
    return context
```

Класс-контроллер вывода по годам *YearArchiveView*

Класс-контроллер `YearArchiveView` предназначен для вывода списка записей, относящихся к указанному году (вывод по годам).

Этот класс унаследован от класса `DateMixin` и поддерживает свойства `date_field` и `allow_future` (см. предыдущий раздел). Также он является потомком класса `YearMixin` и поддерживает два свойства этого класса:

- `year` — задает год в виде строки. Значение по умолчанию — `None`;
- `year_format` — задает строку с форматом, согласно которому будет распознаваться полученное значение года. Литералы, используемые для указания такого формата, представлены в табл. 7.4, а в этом случае их следует предварить символом процента (`%`). Формат, используемый по умолчанию, — `%Y`, т. е. год из четырех цифр.

Значение года, за который следует вывести записи из указанной модели, будет взято:

- либо из свойства `year`;
- либо, если оно не указано, из группы регулярного выражения с именем `year`, созданной в интернет-адресе в привязке;
- либо, если такой группы нет, из GET-параметра с именем `year`.

Если мы создаем свой класс-контроллер на базе класса `YearArchiveView`, нам обязательно следует задать для свойства `make_object_list` значение `True`. Так мы укажем классу-контроллеру сразу же сформировать список записей. Если же мы используем класс-контроллер `YearArchiveView` непосредственно, указав его в привязке (как поступили ранее с классом `ArchiveIndexView`), данное свойство задавать не нужно:

```
from django.views.generic.dates import YearArchiveView
from news.models import New
class YearNewsArchiveView(YearArchiveView):
  model = New
  date_field = "pub_date"
  template_name = "year_archive.html"
  make_object_list = True
```

Здесь мы объявляем класс-контроллер, который будет выводить новости за определенный год.

```
. . .
from news.views import YearNewsArchiveView
urlpatterns = patterns('',
  . . .
  url(r'^(?P<year>\d{4})/$', YearNewsArchiveView.as_view(),↪
  name="year_archive"),
)
```

И выполняем привязку этого класса-контроллера. Отметим, что в интернет-адресе мы создали группу регулярного выражения с именем `year`, которая будет извлекать значение года. Тогда, чтобы получить все новости за 2014 год, мы обратимся по интернет-адресу **/news/2014/**.

Класс-контроллер `YearArchiveView` создает в контексте данных шаблона, помимо всего прочего, вот такие переменные:

- `date_list` — список всех месяцев указанного года, взятых из поля модели, что задано в свойстве `date_field`. Они представляются в виде значений даты, в которых нас интересует лишь собственно месяц;
- `year` — указанный год, представленный в виде значения даты;
- `next_year` — первый день следующего года в виде значения даты;
- `previous year` — первый день предыдущего года в виде значения даты.

Вот пример:

```
<h2>Архив новостей за {{ year|date:"Y" }} год</h2>
```

Здесь мы выводим значение заданного года.

```
{% for new in object_list %}
  <h3>{{ new.title }}</h3>
  <p>{{ new.pub_date }}</p>
  <p>{{ new.description }}</p>
{% endfor %}
```

А здесь — список новостей за этот год.

```
<ul>
  {% for date in date_list %}
    <li>{{ date|date:"E" }}</li>
  {% endfor %}
</ul>
```

И список месяцев этого года, за которые была создана хотя бы одна новость.

Класс-контроллер вывода по месяцам *MonthArchiveView*

Класс-контроллер `MonthArchiveView` формирует список записей, относящихся к указанным нами году и месяцу (вывод по месяцам).

Этот класс является потомком классов `DateMixin` и `YearMixin` и поддерживает все их свойства (см. ранее). Также он наследуется от класса `MonthMixin` и получает от него два следующих свойства:

- `month` — задает месяц в виде строки. Значение по умолчанию — `None`;
- `month_format` — задает строку с форматом, согласно которому будет распознаваться значение месяца. Формат, используемый по умолчанию, — `%b`, т. е. трехбуквенное обозначение месяца на английском языке (что не всегда удобно).

Значение месяца, за который следует отобрать записи для включения в список, будет извлекаться:

- либо из свойства `month`;
- либо, если оно не указано, из группы регулярного выражения с именем `month`, созданной в интернет-адресе привязки;
- либо, если такой группы нет, из GET-параметра с именем `month`.

Вот пример:

```
class MonthNewsArchiveView(MonthArchiveView):
  model = New
  date_field = "pub_date"
  template_name = "month_archive.html"
  make_object_list = True
  month_format = "%m"
```

Здесь мы объявляем класс-контроллер для вывода новостей, датированных указанным месяцем указанного года. Задаем для распознавания значения месяца формат `%m` — номер месяца из двух цифр с начальным нулем.

```
. . .
from news.views import MonthNewsArchiveView
urlpatterns = patterns('',
  . . .
  url(r'^(?P<year>\d{4})/(?P<month>\d+)/$',↵
  MonthNewsArchiveView.as_view(), name="month_archive"),
)
```

И выполняем его привязку. Здесь в интернет-адресе мы создали две группы регулярных выражений: `year`, которая будет извлекать значение года, и `month`, получающая номер месяца. Тогда, чтобы получить все новости за апрель 2014 года, мы обратимся по интернет-адресу **/news/2014/4/**.

Класс-контроллер `MonthArchiveView` создаст в контексте данных шаблона следующие дополнительные переменные:

- `date_list` — список всех чисел указанных месяца и года, взятых из поля модели, что задано в свойстве `date_field`. Они представлены в виде значений даты, в которых нас интересует лишь собственно число;
- `month` — указанные месяц и год в виде значения даты;
- `next_month` — первый день следующего месяца в виде значения даты;
- `previous_month` — первый день предыдущего месяца в виде значения даты.

Вот пример:

```
<h2>Архив новостей {{ month|date:"E" }} {{ month date:"Y" }} года</h2>
```

Здесь мы выводим указанные месяц и год.

```
<ul>
  {% for date in date_list %}
    <li>{{ date|date:"j" }}</li>
  {% endfor %}
</ul>
```

А здесь — список чисел заданного месяца, в которые была создана хотя бы одна новость.

Ну, а как выводить список новостей, мы уже знаем.

Класс-контроллер вывода по дням *DayArchiveView*

Класс-контроллер `DayArchiveView` отбирает записи, относящиеся к указанным году, месяцу и числу (вывод по дням).

Он является потомком классов `DateMixin`, `YearMixin` и `MonthMixin` и поддерживает все их свойства (см. ранее). Также он наследуется от класса `DayMixin`, от которого получает два следующих свойства:

- `day` — задает число в виде строки. Значение по умолчанию — `None`;
- `day_format` — задает строку с форматом, согласно которому будет распознаваться значение числа. Формат по умолчанию — `%b`, т. е. число из двух цифр с начальным нулем.

Число, за которое следует отобрать записи в список, будет извлекаться:

- либо из свойства `day`;
- либо, если оно не указано, из группы регулярного выражения с именем `day`, созданной в привязке;
- либо, если такой группы нет, из GET-параметра с именем `day`.

Вот пример:

```
from django.views.generic.dates import DayArchiveView
class DayNewsArchiveView(DayArchiveView):
  model = New
  date_field = "pub_date"
  template_name = "day_archive.html"
  make_object_list = True
  month_format = "%m"
```

Здесь мы объявляем класс-контроллер для вывода новостей, датированных указанным числом за указанные месяц и год. И не забудем задать для распознавания значения месяца формат `%m`.

```
. . .
from news.views import DayNewsArchiveView
urlpatterns = patterns('',
  . . .
  url(r'^(?P<year>\d{4})/(?P<month>\d+)/(?P<day>\d+)/$',↳
  DayNewsArchiveView.as_view(), name="day_archive"),
)
```

А здесь привязываем его к интернет-адресу, где созданы три группы регулярных выражений: `year`, которая будет извлекать значение года, `month`, получающая номер месяца, и `day` — для получения числа. Тогда, чтобы получить все новости за 2 апреля 2014 года, нужно будет обратиться по интернет-адресу **/news/2014/4/2/**.

Класс-контроллер `DayArchiveView` создаст в контексте данных шаблона вот такие дополнительные переменные:

- `day` — указанные число, месяц и год, представленные в виде значения даты;
- `next_day` — следующее число в виде значения даты;
- `previous_day` — предыдущее число в виде значения даты;
- `next_month` — первый день следующего месяца в виде значения даты;
- `previous_month` — первый день предыдущего месяца в виде значения даты.

Вот пример:

```
<h2>Архив новостей за {{ day|date:"j" }} {{ day|date:"E" }}
{{ day|date:"Y" }} года</h2>
```

Здесь мы выводим переданные контроллеру число, месяц и год.

Класс-контроллер вывода по текущей дате *TodayArchiveView*

Класс-контроллер `TodayArchiveView` отбирает записи, относящиеся к текущей дате. Он аналогичен рассмотренному нами ранее классу `DayArchiveView`, но по понятным причинам не требует передачи ему значений числа, месяца и дня:

```
. . .
from django.views.generic.dates import TodayArchiveView
from news.models import New
urlpatterns = patterns('',
  . . .
  url(r'^today/$', TodayArchiveView.as_view(model = New,↵
  date_field = "pub_date", template_name = "day_archive.html"),↵
  name = "today_archive"),
)
```

Здесь мы привязываем вызов класса-контроллера `TodayArchiveView` к интернет-адресу **/today/**.

В остальном этот класс используется так же, как и подобный ему класс `DayArchiveView`. (В приведенном ранее примере кода мы даже использовали тот же шаблон, что и ранее для вывода новостей за указанное число.)

Что дальше?

В этой главе мы изучали классы-контроллеры. При умелом использовании они позволят заметно упростить нашу работу, взяв на себя выборку предназначенных для вывода данных и пагинацию.

Теперь мы, пожалуй, можем сказать, что знаем о выводе данных средствами Django все. Настала пора заняться их вводом.

ЧАСТЬ III

Ввод и правка данных

ГЛАВА 11

Простые формы Django

В предыдущей части книги мы реализовывали вывод данных. Мы научились создавать модели, функции-контроллеры, шаблоны и использовать пагинатор. И, наконец, мы познакомились с классами-контроллерами, которые могут взять немалую часть по подготовке выводимых данных на себя.

Эта часть книги будет посвящена средствам Django, реализующим ввод и правку данных. Они включают специализированные классы-контроллеры, формы, связанные с моделями, обычные формы, средства для добавления, правки и удаления записей и обработки файлов, загруженных на сайт посетителем.

Начнем мы с самого простого — с высокоуровневых классов-контроллеров, реализующих ввод и правку данных, и всевозможных форм, как связанных с моделями, так и не связанных с ними.

Высокоуровневые классы-контроллеры для добавления, правки и удаления записей

Самый простой способ дать посетителю возможность добавлять и править данные, опубликованные на сайте (в нашем случае — товары), — применить специализированные классы-контроллеры высокого уровня. Они сами выведут на Web-страницу соответствующую форму, если нужно, сами заполнят ее данными, взятыми из указанной нами записи, и сами выполнят операцию добавления, правки или удаления этой записи.

Это классы `CreateView`, `UpdateView` и `DeleteView`, объявленные в модуле `django.views.generic.edit` и предназначенные, соответственно, для добавления, правки и удаления записей. Они поддерживают все свойства и методы, унаследованные от класса `SingleObjectMixin`, и свойство `template_name`, унаследованное от класса `TemplateResponseMixin` (см. *главу 10*).

Кроме того, классы `CreateView` и `UpdateView` поддерживают еще несколько полезных для нас свойств, перечисленных в табл. 11.1, а класс `DeleteView` поддерживает лишь свойство `success_url`.

Таблица 11.1. *Свойства классов* `CreateView` *и* `UpdateView`

Свойство	Описание
`fields`	Список имен полей модели, которые должны присутствовать в выводимой на экран форме. Если не указан, будут выведены все поля модели
`initial`	Словарь, задающий изначальные данные для подстановки в соответствующие элементы управления в формах. Ключи элементов словаря задают имена полей, а значения и станут этими изначальными данными
`success_url`	Интернет-адрес, на который будет выполнено перенаправление после успешного добавления или правки записи

Вот пример:

```
from django.views.generic.edit import CreateView
class GoodCreate(CreateView):
  model = Good
  template_name = "good_add.html"
  success_url = "/"
```

Здесь мы создаем класс-потомок `GoodCreate`, предназначенный для добавления нового товара. После успешного добавления товара будет выполнен возврат на страницу списка товаров.

Один и тот же класс-контроллер используется как для вывода формы, так и для ее обработки: проверки введенных данных, добавления, правки или удаления записи, если эти данные корректны, или повторного вывода формы на экран, если в данных была допущена ошибка.

Следовательно, объект каждого класса-контроллера, используемого для ввода и правки данных, создается дважды: первый раз — при выводе Web-страницы с формой, второй — когда пользователь, закончив ввод или правку, нажимает расположенную в форме кнопку отправки данных.

При первом создании объекта этих классов в нем выполняется уже знакомый нам по *главе 10* метод `get`, но на этот раз унаследованный от класса `ProcessFormView` из модуля `django.views.generic.edit`. Переопределив этот метод в потомке, мы можем выполнить в нем какие-либо подготовительные действия, — например, занесение в форму изначальных данных.

При втором создании объекта вызывается метод `post`, что унаследован от того же класса-родителя и принимает тот же набор параметров, что и метод `get`. Этот метод, разумеется, переопределенный в потомке, можно использовать для выполнения завершающих операций, к которым, в частности, относится и формирование интернет-адреса возврата.

В формировании этого интернет-адреса нам поможет функция `reverse`, объявленная в модуле `django.core.urlresolvers`. Она выполняет ту же задачу, что и тег шаблона `url`, знакомый нам по *главе 8*. В качестве единственного дополнительного параметра эта функция принимает имя, указанное параметром `name` функции `url`

в привязке интернет-адреса (см. *главу 6*). Она также может принимать следующие именованные необязательные параметры:

- `args` — задает в виде обычного списка Python параметры, которые будут подставлены в интернет-адрес на место неименованных групп регулярных выражений. Эти параметры должны указываться в списке в том порядке, в каком в коде привязки были перечислены соответствующие группы;
- `kwargs` — задает в виде словаря Python параметры, что будут подставлены в интернет-адрес на место именованных групп регулярных выражений. Ключи этих параметров должны соответствовать именам групп.

Вот пример использования функции `reverse`:

```
self.success_url = reverse("index", kwargs =↵
{"cat_id": Category.objects.get(pk = self.kwargs["cat_id"]).id})
```

Как правило, в случае успешного добавления, исправления или удаления какой-либо позиции производится возврат на Web-страницу списка этих позиций. Если позиция была добавлена, возврат производится в начало списка, а после исправления или удаления — на ту страницу, где она находится или находилась.

Разумеется, мы можем указать имя привязки в качестве значения свойства `success_url` и непосредственно в классе. Однако в этом случае следует использовать функцию `reverse_lazy`, объявленную в том же модуле `django.core.urlresolvers`. Вызывается она точно так же, как и функция `reverse`:

```
class GoodCreate(CreateView):
  . . .
  success_url = reverse_lazy("index")
```

Метод `get_context_data` классов-контроллеров `CreateView` и `UpdateView` сформирует в контексте данных переменную `form`. Она будет хранить созданную форму в виде объекта класса `ModelForm`, объявленного в модуле `django.forms`. Об этом классе мы поговорим позже, когда начнем создавать формы, связанные с моделью.

Кроме этого, метод `get_context_data` классов-контроллеров `UpdateView` и `DeleteView` сформирует в контексте данных следующие переменные:

- `object` — запись, которая в данный момент правится или удаляется;
- *<имя класса модели, набранное прописными буквами>* — то же самое, что и предыдущая.

Так что мы сразу же сможем вывести на страницу содержимое полей этой записи.

Выяснив все, что нужно, мы можем объявить классы-контроллеры `GoodCreate`, `GoodUpdate` и `GoodDelete` и два вспомогательных класса, реализующих общую функциональность:

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.views.generic.edit import ProcessFormView
from django.core.urlresolvers import reverse
```

```
class GoodEditMixin(CategoryListMixin):
  def get_context_data(self, **kwargs):
    context = super(GoodEditMixin, self).get_context_data(**kwargs)
    try:
      context["pn"] = self.request.GET["page"]
    except KeyError:
      context["pn"] = "1"
    return context

class GoodEditView(ProcessFormView):
  def post(self, request, *args, **kwargs):
    try:
      pn = request.GET["page"]
    except KeyError:
      pn = "1"
    self.success_url = self.success_url + "?page=" + pn
    return super(GoodEditView, self).post(request, *args, **kwargs)

class GoodCreate(CreateView, GoodEditMixin):
  model = Good
  template_name = "good_add.html"
  def get(self, request, *args, **kwargs):
    if self.kwargs["cat_id"] != None:
      self.initial["category"] = Category.objects.get(pk =↳
      self.kwargs["cat_id"])
    return super(GoodCreate, self).get(request, *args, **kwargs)
  def post(self, request, *args, **kwargs):
    self.success_url = reverse("index", kwargs =↳
    {"cat_id": Category.objects.get(pk = self.kwargs["cat_id"]).id})
    return super(GoodCreate, self).post(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(GoodCreate, self).get_context_data(**kwargs)
    context["category"] = Category.objects.get(pk =↳
    self.kwargs["cat_id"])
    return context

class GoodUpdate(UpdateView, GoodEditMixin, GoodEditView):
  model = Good
  template_name = "good_edit.html"
  pk_url_kwarg = "good_id"
  def post(self, request, *args, **kwargs):
    self.success_url = reverse("index", kwargs =↳
    {"cat_id": Good.objects.get(pk = kwargs["good_id"]).category.id})
    return super(GoodUpdate, self).post(request, *args, **kwargs)

class GoodDelete(DeleteView, GoodEditMixin, GoodEditView):
  model = Good
```

```
  template_name = "good_delete.html"
  pk_url_kwarg = "good_id"
  def post(self, request, *args, **kwargs):
    self.success_url = reverse("index", kwargs =↵
    {"cat_id": Good.objects.get(pk = kwargs["good_id"]).category.id})
    return super(GoodDelete, self).post(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(GoodDelete, self).get_context_data(**kwargs)
    context["good"] = Good.objects.get(pk = kwargs["good_id"])
    return context
```

Класс `GoodEditMixin`, порожденный от объявленного нами ранее класса `CategoryListMixin`, будет получать переданный контроллеру номер страницы и добавлять его в контекст данных в качестве отдельной переменной. Это позволит нам сформировать гиперссылку возврата на список товаров.

Класс `GoodEditView` — потомок класса `ProcessFormView`, общего предка классов `GoodCreate`, `GoodUpdate` и `GoodDelete`, — также получит номер страницы и добавит его к интернет-адресу возврата (значению свойства `success_url`) в качестве GET-параметра.

В классе `GoodCreate` не забудем добавить в контекст данных переменную, хранящую категорию, чтобы получить возможность вывести в шаблоне название этой категории и гиперссылку возврата.

Вставим приведенный ранее код в модуль `twviews`.

Напоследок откроем модуль `urls` и вставим в код привязки следующие выражения (выделены полужирным шрифтом):

```
from page.twviews import GoodCreate, GoodUpdate, GoodDelete
urlpatterns = patterns('',
  url(r'^(?:(?P<cat_id>\d+)/)?$', GoodListView.as_view(),↵
  name = "index"),
  url(r'^good/(?P<good_id>\d+)/$', GoodDetailView.as_view(),↵
  name = "good"),
  url(r'^(?P<cat_id>\d+)/add/$', GoodCreate.as_view(),↵
  name = "good_add"),
  url(r'^good/(?P<good_id>\d+)/edit/$', GoodUpdate.as_view(),↵
  name = "good_edit"),
  url(r'^good/(?P<good_id>\d+)/delete/$', GoodDelete.as_view(),↵
  name = "good_delete"),
)
```

Так мы привяжем интернет-адреса добавления, правки и удаления товаров к вновь объявленным классам-контроллерам.

Нам может пригодиться и метод `form_valid`, поддерживаемый классами `CreateView` и `UpdateView`. Он выполняет собственно сохранение занесенных в форму данных. Переопределив его в классе-потомке, мы можем выполнить в нем занесение в поля

создаваемой или исправляемой записи каких-либо значений, не взятых из формы, а сгенерированных программно.

Метод `form_valid` принимает в качестве единственного параметра тот же объект класса `ModelForm`, представляющий форму. А класс `ModelForm` поддерживает свойство `instance`, хранящее саму создаваемую или исправляемую запись:

```
class BlogCreate(CreateView):
  model = Blog
  . . .
  def form_valid(self, form):
    form.instance.user = self.request.user
    return super(BlogCreate, self).form_valid(form)
```

Здесь мы заносим в поле `user` записи модели `Blog` пользователя, который в данный момент выполнил вход на сайт. (Реализация разграничения доступа и, в частности, получение текущего пользователя и сведений о нем будут рассмотрены в *главе 14*.)

Создание шаблонов форм

Как мы уже выяснили, метод `get_context_data` классов-контроллеров `CreateView` и `UpdateView` сформирует в контексте данных переменную `form`, которая будет хранить созданную форму в виде объекта класса `ModelForm`. Этот класс поддерживает три полезных нам в данном случае метода (все они не принимают параметров и, стало быть, могут быть вызваны из шаблонов):

- `as_p` — возвращает HTML-код, создающий форму, где каждый элемент управления находится в отдельном абзаце:

  ```
  {{ form.as_p }}
  ```

- `as_ul` — возвращает HTML-код, создающий форму, где каждый элемент управления находится в отдельном пункте списка. Отметим, что теги, создающие сам список, возвращенный код не содержит, и нам придется создать их самим:

  ```
  <ul>
    {{ form.as_ul }}
  </ul>
  ```

- `as_table` — возвращает HTML-код, создающий форму, где каждый элемент управления находится в отдельной строке таблицы. Тега `<table>` возвращенный код не содержит, и нам также придется создать его самим:

  ```
  <table>
    {{ form.as_table }}
  </table>
  ```

Кроме того, HTML-код, сгенерированный этими тремя методами, не включает код, формирующий кнопку отправки данных, и тег `<form>`, создающий саму форму. Эти теги мы вставим в код шаблона сами.

Внутрь формы настоятельно рекомендуется поместить тег шаблона `csrf_token`. Он указывает шаблонизатору Django сгенерировать код, предотвращающий выполнение межсайтовых сценариев, что должно повысить защищенность нашего сайта.

Вот типичный код, создающий форму:

```
<form action="" method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="Сохранить">
</form>
```

Значение атрибута `action` тега `<form>` следует оставить пустым. Это предпишет Web-обозревателю отправить введенные в форму данные по тому же интернет-адресу, с которого была загружена исходная страница. (Как мы помним из сказанного ранее, для вывода формы и ее обработки будет использован один и тот же контроллер.)

Напишем код шаблона good_edit.html:

```
{% extends "base.html" %}
{% block title %}Правка :: {{ good.name }} :: {{ good.category.name }}↵
{% endblock %}
{% block main %}
  <form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Сохранить">
  </form>
  <p><a href="{% url "index" cat_id=good.category.id %}↵
  ?page={{ pn }}">Назад</a></p>
{% endblock %}
```

Код шаблона good_add.html будет практически таким же (отличия выделены полужирным шрифтом):

```
. . .
{% block title %}Добавление товара :: {{ category.name }}{% endblock %}
{% block main %}
    . . .
    <input type="submit" value="Добавить">
  </form>
  <p><a href="{% url "index" cat_id=category.id %}↵
  ?page={{ pn }}">Назад</a></p>
{% endblock %}
```

А шаблон good_delete.html можно сделать на основе шаблона good.html, добавив в него код, создающий форму:

```
{% extends "base.html" %}
{% block title %}Удаление :: {{ good.name }} :: ↵
{{ good.category.name }}{% endblock %}
```

```
{% block main %}
  <h2>{{ good.name }}</h2>
  <p class="category">Категория: {{ good.category.name }}</p>
  <p>{{ good.description|linebreaksbr }}</p>
  <p class="not-in-stock">{% if not good.in_stock %}Нет в наличии!
  {% endif %}</p>
  <form action="" method="post">
    {% csrf_token %}
    <input type="submit" value="Удалить">
  </form>
  <p><a href="{% url "index" cat_id=good.category.id %}
  ?page={{ pn }}">Назад</a></p>
{% endblock %}
```

Напоследок откроем шаблон index.html и добавим в него гиперссылки, ведущие на страницы добавления, правки и удаления товара. Добавленный код выделен полужирным шрифтом:

```
{% block main %}
  <h2>{{ category.name }}</h2>
  <p><a href="{% url "good_add" cat_id=category.id %}
  ?page={{ page_obj.number }}">Добавить товар</a></p>
  <table>
    <tr>
      <th>Название</th>
      <th>Есть в наличии</th>
      <th> </th>
      <th> </th>
    </tr>
    {% for good in object_list %}
      <tr>
        <td><a href="{% url "good" good_id=good.id %}
        ?page={{ page_obj.number }}">{{ good.name }}</a></td>
        <td class="centered">{% if good.in_stock %}+{% endif %}</td>
        <td class="centered"><a href="{% url "good_edit"
        good_id=good.id %}?page={{ page_obj.number }}">Изменить</a></td>
        <td class="centered"><a href="{% url "good_delete"
        good_id=good.id %}?page={{ page_obj.number }}">Удалить</a></td>
      </tr>
    {% endfor %}
  </table>
  . . .
{% endblock %}
```

Закончив, проверим сайт в действии.

Интерфейс для добавления, правки и удаления записей

Посмотрим на страницу добавления нового товара нашего сайта (рис. 11.1). Она содержит форму со всеми необходимыми элементами управления для ввода сведений о товаре: наименования, описания, категории и признака того, есть ли товар в наличии. Причем для поля каждого типа генерируется наиболее подходящий для него элемент управления: поле ввода для строкового поля, область редактирования для поля memo, раскрывающийся список для поля связи с другой моделью и флажок для логического поля.

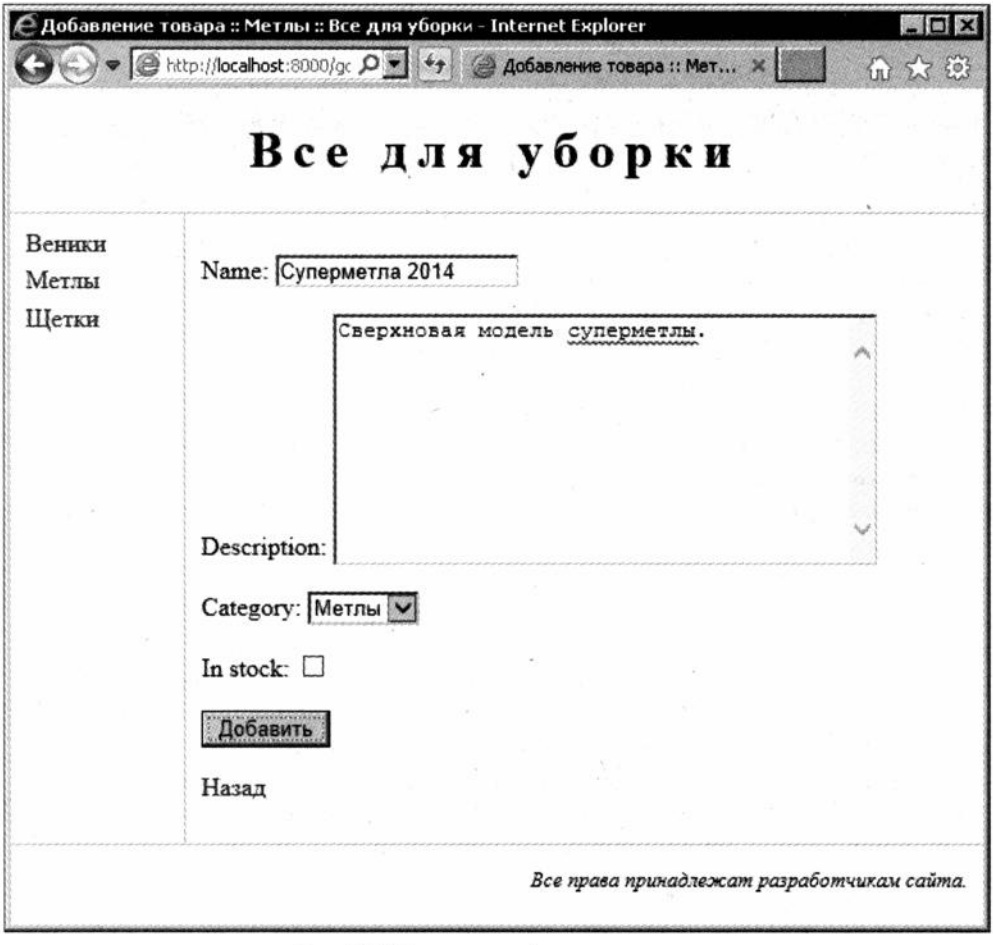


Рис. 11.1. Страница добавления товара

В форме будут присутствовать поля, указанные в свойстве `fields` классов-контроллеров, описанных ранее, и порядок указания полей в этом свойстве задаст порядок перечисления полей в форме. Если значение свойства `fields` не задано, форма включит все поля модели, кроме указанных, как недоступные для редакти-

рования (сделать это можно с помощью параметра `editable` конструктора поля — см. *главу 5*).

В качестве надписей для элементов управления будут взяты названия полей, заданные в параметре конструкторов `verbose_name` (см. *главу 5*). Если это название не указано, будет взято имя поля (как и показано на рис. 11.1).

Перед добавлением или сохранением записи Django обязательно проверяет корректность значений ее полей. Так, в нашем случае, поскольку поле названия товара у нас помечено как уникальное, ввод в него названия, которое уже присутствует в модели, вызовет повторный вывод той же формы с сообщением об ошибке (рис. 11.2).

При желании мы можем предложить Django сгенерировать формы по-другому. Как это сделать, будет рассказано в *главе 12*.

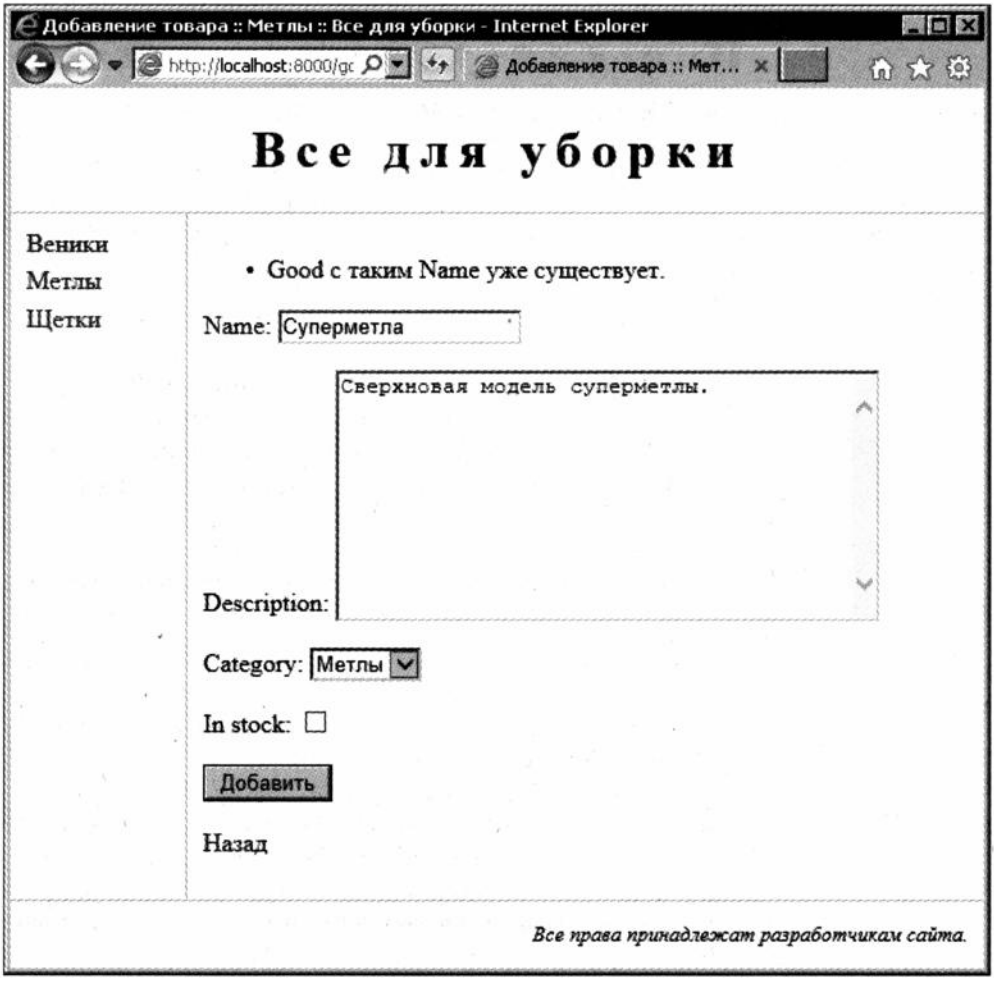


Рис. 11.2. Страница добавления товара с сообщением об ошибке

Формы Django, связанные с моделями

Высокоуровневые классы-контроллеры при работе неявно формируют объект класса `ModelForm`, что объявлен в модуле `django.forms`. Этот объект и выполняет всю работу по получению данных из указанной записи, выводу соответствующего интерфейса, проверке корректности введенных данных и собственно добавлению, исправлению или удалению записи — в зависимости от используемого класса-контроллера.

Класс `ModelForm` берет из модели все сведения, необходимые для создания формы и обработки введенных в нее данных. Они включают перечень полей, их параметры (длина, критерий уникальности значения, название, которое будет выводиться на экран в качестве надписи к элементу управления, и пр.) и порядок их следования.

Класс `ModelForm` всегда привязывается к определенной модели, из которой берет нужные для его работы сведения. Поэтому такие формы называют *связанными с моделями*.

Создание формы, связанной с моделью

Однако мы имеем возможность сами объявить потомок класса `ModelForm`, в котором указать нужные нам параметры создаваемой формы, включая названия полей, их список и пр. Это может оказаться полезным, если мы хотим поменять интерфейс для ввода данных в модель.

Простой способ

Когда нам требуется всего лишь немного изменить поведение такой формы, мы объявим в классе-потомке `ModelForm` вложенный класс `Meta` (схожий с одноименным классом, в котором задаются метаданные модели — см. *главу 5*) и зададим в нем нужные свойства. Все полезные нам свойства класса `Meta` перечислены в табл. 11.2.

***Таблица 11.2.** Свойства вложенного класса `Meta` класса формы, связанной с моделью*

Свойство	Описание
`model`	Модель, к которой будет привязана форма
`fields`	Список имен полей модели, которые должны присутствовать в форме. Если не указан, форма включит все поля модели
`exclude`	Список имен полей модели, которые не должны присутствовать в форме. Если указанное в этом списке поле также перечислено в списке, заданном в свойстве `fields`, оно не будет присутствовать в форме
`labels`	Надписи для элементов управления в виде словаря, ключи элементов которого должны совпадать с именами полей, а их значения зададут текст надписей
`help_texts`	Задает текст дополнительного описания, которое будет выводиться под соответствующим элементом управления

Вот пример:

```
from django.forms import ModelForm
from page.models import Good
class GoodForm(ModelForm):
  class Meta:
    model = Good
    fields = ["name", "description", "in_stock"]
    labels = {"name": "Название",
              "description": "Описание",
              "in_stock": "Есть в наличии"}
    help_texts = {"name": "Должно быть уникальным"}
```

Здесь мы создаем форму с элементами управления для ввода названия товара, его описания и признака, есть ли товар в наличии. Как видим, в этом случае посетитель не сможет указать категорию товара, поскольку форма не включит в свой состав соответствующего поля модели.

Сложный способ

Если же мы хотим более радикально изменить внешний вид и поведение формы, связанной с моделью, мы можем указать список полей модели, которые должны в ней присутствовать, и их параметры явно.

Форма в этом случае создается так же, как модель (см. *главу 5*). Мы объявляем класс формы, задаем во вложенном классе `Meta` модель, к которой она будет привязана, и указываем список полей.

> ***Свойство для вложенного класса Meta***
>
> В этом случае во вложенном классе `Meta` допустимо лишь указание свойства `model`. Прочие свойства, перечисленные в табл. 11.2, для него указывать не следует.

Имена полей формы должны совпадать с соответствующими им именами полей модели. А в качестве их значений указываются объекты классов, представляющих поля формы различных типов. В табл. 11.3 перечислены эти классы, равно как и соответствующие им классы полей формы.

Таблица 11.3. Классы полей модели и соответствующие им классы полей формы

Класс поля модели	Класс поля формы
`BigIntegerField`	`IntegerField`
`BooleanField`	`BooleanField`
`CharField`	`CharField`
`DateField`	`DateField`
`DateTimeField`	`DateTimeField`
`EmailField`	`EmailField`
`FileField`	`FileField`

Таблица 11.3 (окончание)

Класс поля модели	Класс поля формы
FilePathField	FilePathField
FloatField	FloatField
ForeignKey	ModelChoiceField
GenericIPAddressField	GenericIPAddressField
ImageField	ImageField
IntegerField	IntegerField
IPAddressField	IPAddressField
PositiveIntegerField	IntegerField
PositiveSmallIntegerField	IntegerField
SlugField	SlugField
SmallIntegerField	IntegerField
TextField	CharField
TimeField	TimeField
URLField	URLField

Класс поля модели `AutoField` не имеет соответствующего класса поля формы, поскольку такие поля в формах никак не представляются.

Помимо этого, мы можем использовать в формах класс поля `ChoiceField`. Такое поле позволяет выбрать значение из списка указанных нами и аналогично полю моделей с заданным параметром `choices` (см. *главу 5*).

Классы полей форм объявлены в модуле `django.forms`. В конструкторах классов полей форм мы можем указывать необязательные именованные параметры, приведенные в табл. 11.4.

***Таблица 11.4.** Параметры, поддерживаемые конструкторами классов полей форм, и классы полей, для которых они применимы*

Параметр	Описание	Класс полей формы
required	Если `True`, то поле должно содержать уникальное значение	Все
label	Текст надписи для соответствующего элемента управления	
initial	Изначальное значение для поля	
help_text	Текст дополнительных сведений	
min_length	Минимальная длина вводимого значения в символах	`CharField`, `EmailField`, `URLField`
max_length	Максимальная длина вводимого значения в символах	

Таблица 11.4 (окончание)

Параметр	Описание	Класс полей формы
choices	Список доступных для выбора значений. Указывается в том же формате, что аналогичный список, задаваемый в параметре choices модели (см. *главу 5*)	ChoiceField
input_formats	Список форматов указания значений, которые может принимать поле. Задается в виде списка, каждый элемент которого должен представлять собой строку с форматом. Для задания форматов применяются символы литералов, приведенные в табл. 7.4, перед которыми в этом случае ставится символ процента (%)	DateField, DateTimeField, TimeField
min_value	Минимальное значение для ввода	FloatField, IntegerField
max_value	Максимальное значение для ввода	
queryset	Набор записей, из которого будут взяты записи для выбора	ModelChoiceField
empty_label	Текст, представляющий пустое поле. Если None, задать пустое значение для поля будет нельзя	

Все параметры, которые мы не указали, форма возьмет из связанной с ней модели.

Вот примеры:

```
from django import forms
from page.models import Category, Good
class GoodForm(forms.ModelForm):
  class Meta:
    model = Good
  name = forms.CharField(label = "Название",↳
  help_text = "Должно быть уникальным")
  description = forms.CharField(widget = forms.Textarea, ↳
  label = "Описание")
  category = forms.ModelChoiceField(queryset = Category.objects.all(),↳
  label = "Категория", empty_label = None)
  in_stock = forms.BooleanField(initial = True, label = "Есть в наличии")
```

Здесь мы создаем форму для ввода товаров. Для поля description указываем в качестве выводимого элемента управления область редактирования (подробнее о задании элементов управления для полей мы поговорим в *главе 12*).

```
class BlogArticleForm(forms.ModelForm):
  . . .
  pubdate = forms.DateField(input_formats = ["%j.%d.%y"])
```

А здесь задаем для поля даты публикации в модели статей блога формат записи даты *<число>.<месяц>.<год>*.

Использование формы, связанной с моделью

Итак, форму, связанную с моделью, мы создали. Осталось выяснить, как ее использовать.

А использовать ее можно как в уже знакомых нам классах-контроллерах высокого уровня, так и в обычном классе-контроллере, — например, потомке описанного в *главе 10* класса `ListView` (если мы хотим поместить форму на той же странице, где у нас выводится список каких-либо позиций). Разумеется, ничто не мешает нам применить такую форму и в функциях-контроллерах (см. *главу 6*).

Использование формы в классах-контроллерах, предназначенных для добавления и правки записей

Чтобы использовать форму, связанную с моделью, в потомках классов-контроллеров высокого уровня `CreateView` и `UpdateView` достаточно указать ее класс в свойстве `form_class`:

```
class GoodCreate(CreateView):
  form_class = GoodForm
  . . .

class GoodUpdate(UpdateView):
  form_class = GoodForm
  . . .
```

Использование формы в классах-контроллерах, предназначенных для вывода данных

Несколько сложнее использовать форму в классах-контроллерах, предназначенных для вывода данных. Здесь нам самим придется создавать объект класса формы, если нужно, заполнять его данными, проверять их корректность и, в конце концов, сохранять в модели.

Первое, что нам потребуется сделать, — создать форму и поместить ее в контекст данных шаблона. Сделать это лучше всего в методах `get` и `get_context_data` класса-контроллера соответственно.

В классе-контроллере, предназначенном для добавления записи, мы укажем в конструкторе класса формы один параметр. Он носит имя `initial`, является необязательным и получает в качестве значения словарь с изначальными данными, которые должны быть подставлены в форму. Формат этого словаря нам уже знаком: ключи элементов носят те же имена, что и поля формы, а значения и станут изначальными данными для них.

Рассмотрим фрагмент кода такого класса-контроллера:

```
from django.views.generic.base import TemplateView
from page.models import Good, Category
class GoodCreate(TemplateView):
```

```
form = None
template_name = "good_add.html"
def get(self, request, *args, **kwargs):
  if self.kwargs["cat_id"] == None:
    cat = Category.objects.first()
  else:
    cat = Category.objects.get(pk = self.kwargs["cat_id"])
  self.form = GoodForm(initial = {"category": cat})
  return super(GoodCreate, self).get(request, *args, **kwargs)
def get_context_data(self, **kwargs):
  context = super(GoodCreate, self).get_context_data(**kwargs)
  context["form"] = self.form
  return context
```

После того как посетитель введет в форму сведения о новом товаре и нажмет кнопку отправки данных, этот контроллер будет вызван повторно. И нам потребуется обработать введенные данные — проверить их на корректность и сохранить в модели.

Делать это нужно в методе `post`, который мы объявим в классе-контроллере. Этот метод должен принимать те же параметры, что и метод `get`.

Здесь мы вновь создадим объект формы, но передадим его конструктору другой параметр, не имеющий имени. Его значением станут данные, введенные пользователем и переданные этому контроллеру методом POST. Они хранятся в свойстве `POST` объекта класса `HttpRequest`, представляющего запрос и получаемого методом `post` первым параметром. Поскольку данные, передаваемые методом POST, представляют собой словарь, мы можем передать конструктору формы значение свойства `POST` без всяких модификаций.

Класс формы `ModelForm` поддерживает не принимающий параметров метод `is_valid`. Он возвращает `True`, если введенные в форму данные корректны, т. е. удовлетворяют всем заданным в модели и самой форме условиям.

Если данные введены правильно, мы вызовем у формы не принимающий параметров метод `save`. Он выполняет сохранение данных формы в запись модели. (Поскольку мы не указали в конструкторе формы запись, куда следует сохранять эти данные, форма создаст новую запись.)

Чтобы после успешного сохранения записи перенаправить посетителя на другую страницу, мы можем использовать функцию `redirect`, объявленную в модуле `django.shortcuts`. Единственным обязательным параметром она принимает либо интернет-адрес, либо имя привязки, указанное параметром `name` в функции `url` (см. *главу 6*). Во втором случае мы можем передать функции `redirect` параметры, необходимые для формирования интернет-адреса.

Результат, возвращенный функцией `redirect`, мы должны вернуть из метода `post` в качестве результата.

Если же данные формы оказались некорректными, мы выведем ее на экран повторно. Для этого достаточно вызвать метод `get` класса-родителя и вернуть возвращен-

ный им результат. Отметим, что вызывать нужно именно метод родителя — если же мы вызовем метод, объявленный в текущем классе, он вновь сгенерирует пустую форму для ввода нового товара.

Чтобы лучше понять сказанное, посмотрим на код, объявляющий в нашем классе-контроллере метод post:

```
from django.shortcuts import redirect
class GoodCreate(TemplateView):
  . . .
  def post(self, request, *args, **kwargs):
    if self.kwargs["cat_id"] == None:
      cat = Category.objects.first()
    else:
      cat = Category.objects.get(pk = self.kwargs["cat_id"])
    self.form = GoodForm(request.POST)
    if self.form.is_valid():
      self.form.save()
      return redirect("index", cat_id = cat.id)
    else:
      return super(GoodCreate, self).get(request, *args, **kwargs)
```

Теперь займемся классом-контроллером для правки записи. В этом классе мы также передадим конструктору класса формы один параметр. Он называется instance, является в данном случае обязательным и получит в качестве значения запись, значения из которой будут подставлены в поля формы, — эти данные потом и будет править посетитель:

```
class GoodUpdate(TemplateView):
  form = None
  template_name = "good_edit.html"
  def get(self, request, *args, **kwargs):
    self.form = GoodForm(instance = Good.objects.get(pk =↪
    self.kwargs["good_id"]))
    return super(GoodUpdate, self).get(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(GoodUpdate, self).get_context_data(**kwargs)
    context["good"] = Good.objects.get(pk = self.kwargs["good_id"])
    context["form"] = self.form
    return context
```

Что касается метода post этого класса, то в нем конструктор формы получит, помимо уже знакомого нам неименованного параметра с введенными посетителем данными, еще и параметр instance, который укажет исправляемую запись. В нее-то и будут записаны данные формы, как только мы вызовем ее метод save:

```
  def post(self, request, *args, **kwargs):
    good = Good.objects.get(pk = self.kwargs["good_id"])
    self.form = GoodForm(request.POST, instance = good)
```

```
    if self.form.is_valid():
      self.form.save()
      return redirect("index", cat_id = good.category.id)
    else:
      return super(GoodUpdate, self).get(request, *args, **kwargs)
```

Как уже говорилось, класс `ModelForm` поддерживает свойство `instance`, хранящее создаваемую или исправляемую запись модели, к которой была связана форма. Мы можем использовать это свойство для доступа к записи и ее полям — например, для занесения в них каких-то значений, сформированных программно. Идеальное место для этого — переопределенный в потомке метод `form_valid` (см. приведенный ранее пример).

Использование формы в функциях-контроллерах

Использовать формы, связанные с моделями, можно и в обычных функциях-контроллерах, изученных нами еще в *главе 6*. Здесь нам пригодится свойство `method` класса `HttpRequest` — с его помощью мы можем проверить, нужно ли вывести форму, или посетитель уже ввел в нее данные, которые нужно проверить и сохранить.

Вот пример кода функции-контроллера, добавляющей новый товар:

```
def good_create(request, cat_id):
  if cat_id == None:
    cat = Category.objects.first()
  else:
    cat = Category.objects.get(pk = cat_id)
  if request.method == "POST":
    form = GoodForm(request.POST)
    if form.is_valid():
      form.save()
      return redirect("index", cat_id = cat.id)
    else:
      return render(request, "good_add.html", {"category": cat, "form": form})
  else:
    form = GoodForm(initial = {"category": cat})
    return render(request, "good_add.html", {"category": cat, "form": form})
```

Обычные формы Django

Помимо форм, связанных с моделью, Django позволяет создавать и *обычные формы*, ни с чем не связанные. Они тоже могут нам пригодиться.

Создание обычных форм

Обычные формы создаются так же, как и формы, связанные с моделью. За двумя исключениями: они являются потомками класса `Form`, объявленного в модуле `django.forms`, и в них не допускается указание вложенного класса `Meta`:

```
from django import forms
from page.models import Category
class GoodForm(forms.Form):
  name = forms.CharField(label = "Название",↵
  help_text = "Должно быть уникальным")
  description = forms.CharField(widget = forms.Textarea, ↵
  label = "Описание")
  category = forms.ModelChoiceField(queryset = Category.objects.all(),↵
  label = "Категория", empty_label = None)
  in_stock = forms.BooleanField(initial = True, label = "Есть в наличии")
```

Здесь мы объявляем класс обычной формы для ввода товаров.

Обработка обычных форм

Конструктор класса `Form` принимает всего один необязательный параметр. Это словарь с данными, которые следует подставить в поля формы. Он должен иметь такой же формат, как и аналогичный словарь, принимаемый конструктором класса `ModelForm` (см. ранее):

```
. . .
def get(self, request, *args, **kwargs):
  good = Good.objects.get(pk = self.kwargs["good_id"])
  self.form = GoodForm({"name": good.name,↵
  "description": good.description, "category": good.category,↵
  "in_stock": good.in_stock})
  . . .
```

Класс `Form` также поддерживает метод `is_valid`, выполняющий проверку введенных в форму данных:

```
def post(self, request, *args, **kwargs):
  . . .
  self.form = GoodForm(request.POST)
  if self.form.is_valid():
    . . .
```

Но на метод `save` мы рассчитывать уже не можем, поскольку такая форма не привязана к модели и, следовательно, не «знает», куда и как сохранять введенные в нее данные.

Однако класс `Form` поддерживает свойство `cleaned_data`. Его значение представляет собой словарь, ключами элементов которого являются имена полей формы, а их значениями — введенные в эти поля значения. Через это свойство мы сможем получить данные формы:

```
name = self.form.cleaned_data["name"]
description = self.form.cleaned_data["description"]
category = self.form.cleaned_data["category"]
in_stock = self.form.cleaned_data["in_stock"]
```

Инструменты модели для добавления, правки и удаления записей

Раз обычная форма не может сохранить данные в модель, нам придется сделать это самим. Соответственно, нам понадобятся инструменты для добавления, правки и удаления записей, предоставляемые моделями.

Создать в модели новую запись мы можем путем создания объекта ее класса и присвоения его свойствам нужных значений:

```
good = Good()
good.name = "Суперметла"
good.description = "Новая модель суперметлы!!!"
good.category = Category.objects.get(name = "Метлы")
```

Мы можем задать значения для полей модели во время создания записи, передав их конструктору в качестве именованных параметров:

```
good = Good(name = "Суперметла",↳
description = "Новая модель суперметлы!!!",↳
category = Category.objects.get(name = "Метлы"))
```

Созданная нами таким образом запись существует только в памяти компьютера. Чтобы сохранить ее на диск, в саму базу данных, мы вызовем у нее не принимающий параметров метод `save`:

```
good.save()
```

Вот полный код метода `post` класса-контроллера, выполняющего добавление нового товара:

```
class GoodCreate(TemplateView):
  . . .
  def post(self, request, *args, **kwargs):
    if self.kwargs["cat_id"] == None:
      cat = Category.objects.first()
    else:
      cat = Category.objects.get(pk = self.kwargs["cat_id"])
    self.form = GoodForm(request.POST)
    if self.form.is_valid():
      good = Good(name = self.form.cleaned_data["name"],↳
      description = self.form.cleaned_data["description"],↳
      category = self.form.cleaned_data["category"],↳
      in_stock = self.form.cleaned_data["in_stock"])
      good.save()
      return redirect("index", cat_id = cat.id)
    else:
      return super(GoodCreate, self).get(request, *args, **kwargs)
```

Метод `save` возвращает результат — объект, представляющий сохраненную запись. Мы сможем воспользоваться этим в *главах 12* и *25*, когда начнем работать с наборами форм.

Создание новой записи и ее сохранение можно выполнить за один раз. Для этого мы воспользуемся методом `create` класса-диспетчера записей (о диспетчерах записей говорилось в *главе 5*). Этот метод принимает набор именованных параметров, задающих значения для полей создаваемой записи:

```
. . .
if self.form.is_valid():
  Good.objects.create(name = self.form.cleaned_data["name"],↳
  description = self.form.cleaned_data["description"],↳
  category = self.form.cleaned_data["category"],↳
  in_stock = self.form.cleaned_data["in_stock"])
  . . .
```

Исправить хранящиеся в записи данные мы можем, занеся в ее свойства новые значения и также вызвав метод `save`. Этот метод объявлен в классе `Model` и, следовательно, поддерживается всеми моделями:

```
class GoodUpdate(TemplateView):
  . . .
  def post(self, request, *args, **kwargs):
    self.form = GoodForm(request.POST)
    if self.form.is_valid():
      good = Good.objects.get(pk = self.kwargs["good_id"])
      good.name = self.form.cleaned_data["name"]
      good.description = self.form.cleaned_data["description"]
      good.category = self.form.cleaned_data["category"]
      good.in_stock = self.form.cleaned_data["in_stock"]
      good.save()
      return redirect("index", cat_id = good.category.id)
    else:
      return super(GoodUpdate, self).get(request, *args, **kwargs)
```

Для удаления записи мы используем не принимающий параметров метод `delete`, который также объявлен в классе `Model`:

```
class GoodDelete(TemplateView):
  . . .
  def post(self, request, *args, **kwargs):
    self.form = GoodForm(request.POST)
    if self.form.is_valid():
      good = Good.objects.get(pk = self.kwargs["good_id"])
      good.delete()
      return redirect("index", cat_id = good.category.id)
    else:
      return super(GoodDelete, self).get(request, *args, **kwargs)
```

Конечно, для работы с моделями удобнее применять связанные с ними формы. Но в некоторых случаях нам может потребоваться сохранять в модели не сами данные, введенные в форму, а результат каких-либо основанных на них вычислений. Кроме

того, обычные формы могут применяться для указания данных, вообще не предназначенных для сохранения в модели, — например, подстрок для поиска.

Если нам понадобится внести какие-либо исправления сразу в несколько записей, отобранных по какому-либо критерию (скажем, указать, что все метлы имеются в наличии), мы воспользуемся методом `update` класса-диспетчера записей. Этот метод принимает именованные параметры, задающие новые значения для полей записей набора:

```
brooms = Good.objects.filter(category__name = "Метлы")
brooms.update(in_stock = True)
```

Наконец, для удаления сразу нескольких записей, выбранных по определенному критерию, служит не принимающий параметров метод `delete` диспетчера записей:

```
Good.objects.filter(in_stock = False).delete()
```

Здесь мы удаляем все товары, отсутствующие в наличии.

Что дальше?

В этой главе мы начали знакомство со средствами Django, предназначенными для ввода данных. Мы узнали о специализированных классах-контроллерах, выполняющих добавление, правку и удаление записи, формах, связанных с моделями, и обычных формах. И заодно выяснили, какие средства предоставляются моделями для добавления, правки и удаления записей.

В следующей главе мы рассмотрим средства валидации введенных в форму данных и расширенные возможности по их выводу на экран. Разговор пойдет и о системе сообщений Django и хранении данных в сессиях — все это может нам пригодиться в дальнейшем. И, наконец, мы поговорим о наборах форм — замечательном инструменте, который позволит нам работать сразу над всеми записями, хранящимися в модели.

ГЛАВА 12

Более сложные формы Django

В предыдущей главе мы реализовывали ввод данных средствами Django. Эти средства включали высокоуровневые классы-контроллеры для добавления, правки и удаления записей, формы, связанные с моделями, и обычные формы.

В этой главе мы продолжим разговор о формах. Мы научимся выполнять дополнительную проверку введенных в них данных, радикально менять их внешний вид, применять сообщения Django, чтобы уведомить посетителя об успешном добавлении или удалении записи, и сохранять данные в сессиях. А еще мы познакомимся с *наборами форм* — замечательным инструментом, которым может похвастаться далеко не каждая библиотека — конкурент Django.

Сообщения об ошибках и проверка данных

Перед тем как сохранить в модели данные формы, Django выполняет их проверку́, или, как говорят программисты, *валидацию*. Валидация включает в себя проверку факта ввода данных в обязательное поле, правильности типа введенного в поле значения (например, действительно ли в поле `FloatField` введено число с плавающей точкой) и прочих параметров введенного значения (например, укладывается ли строка в поле заданной длины).

Основная валидация выполняется самой формой — классами `Form` или `ModelForm`. При этом, если данные введены некорректно, сама форма выводит соответствующее сообщение об ошибке.

Мы можем задать для формы свое собственное сообщение об ошибке, более подходящее к ситуации, а также и реализовать в ней дополнительные правила валидации данных. И сейчас мы узнаем, как это сделать.

Задание сообщений об ошибках

В зависимости от того, каким способом была создана форма, связанная с моделью, — простым или сложным (см. *главу 11*), — мы используем для задания сообщений об ошибках разные инструменты:

- если форма была создана простым способом — путем указания нужных параметров во вложенном классе `Meta`, — мы зададим сообщения об ошибках в свойстве `error_messages` этого класса. Значение этого свойства должно представлять собой словарь, ключи элементов которого будут именами полей, а значения — вложенными словарями. Во вложенном словаре ключ каждого элемента станет кодом ошибки, а значение — самим сообщением;
- если форма была создана сложным способом — непосредственным заданием списка полей и их параметров, — сообщения об ошибках должны быть указаны в параметре `error_messages` конструктора поля. Значением этого параметра должен быть словарь, ключи элементов которого будут кодами ошибок, а значения — сообщениями.

Второй способ также применяется для указания сообщений об ошибках в обычных, не связанных с моделями формах. (Собственно, других вариантов тут нет.)

Коды ошибок, поддерживаемые различными классами полей, и их описания перечислены в табл. 12.1.

***Таблица 12.1.** Коды ошибок*

Код ошибки	Описание ошибки	Классы полей
`required`	В обязательное поле не было введено значение	Все
`min_length`	Введенная строка имеет длину меньшую, чем указана в параметре `min_lenght`	`CharField`
`max_length`	Введенная строка имеет длину большую, чем указана в параметре `max_lenght`	
`invalid_choice`	Введено значение, отсутствующее в списке	`ChoiceField`, `ModelChoiceField`
`invalid`	Введенное значение имеет неверный формат	`DateField`, `DateTimeField`, `EmailField`, `FloatField`, `IntegerField`, `IPAddressField`, `GenericIPAddressField`, `SlugField`, `TimeField`, `URLField`
`min_value`	Введено значение меньшее, чем указано в параметре `min_value`	`FloatField`, `IntegerField`
`max_value`	Введено значение большее, чем указано в параметре `max_value`	

Вот примеры:

```
class GoodForm(ModelForm):
  class Meta:
    model = Good
    . . .
```

```
    error_messages = {"name": {"required": "Укажите название товара",↳
    "min_length": "Слишком короткое название",↳
    "max_length": "Слишком длинное название"}}
```

Здесь мы задаем список сообщений об ошибках для поля `name` формы, созданной простым способом.

```
NAME_ERROR_LIST = {"required": "Укажите название товара",↳
"min_length": "Слишком короткое название",↳
"max_length": "Слишком длинное название"}
class GoodForm(forms.ModelForm):
  class Meta:
    model = Good
  name = forms.CharField(label = "Название",↳
  help_text = "Должно быть уникальным", error_messages = NAME_ERROR_LIST)
  . . .
```

А здесь делаем то же самое для формы, созданной сложным способом.

Валидаторы и их написание

Для валидации данных Django использует особые функции, называемые *валидаторами*. Такая функция принимает один параметр — проверяемое значение поля — и генерирует объявленное в модуле `django.core.exceptions` исключение `ValidationError`, если это значение не проходит проверку.

Django уже включает в свой состав набор валидаторов для проверки корректности значений различных типов: целых чисел, адресов электронной почты и пр. Мы можем написать свои валидаторы, если хотим выполнять дополнительную проверку заносимых в форму значений. Это совсем несложно.

Вот объявление простого валидатора, который проверяет, не является ли заданное значение цены товара отрицательным:

```
from django.core.exceptions import ValidationError
def validate_positive(value):
  if value < 0:
    raise ValidationError("Значение цены должно быть положительным!",↳
    code = "invalid")
```

Как видим, конструктор класса `ValidationError` принимает обязательный параметр с описанием возникшей ошибки и необязательный (но настоятельно рекомендуемый к заданию) параметр `code`, задающий код ошибки (см. табл. 12.1).

Конструкторы классов полей в формах поддерживают свойство `validators`. Его значением должен быть список функций-валидаторов, которые будут дополнительно применены к данному полю:

```
class GoodForm(forms.ModelForm):
  class Meta:
    model = Good
  . . .
```

```
  price = forms.FloatField(label = "Цена",↳
  validators = [validate_positive])
  . . .
```

К сожалению, указать валидаторы во вложенном классе `Meta` мы не можем.

Проверка данных на уровне формы

Валидаторы Django — достаточно универсальные инструменты, и их можно применять для проверки данных в произвольном количестве полей. Но это их достоинство одновременно является и недостатком — в теле валидатора мы не можем получить доступ к другим полям формы, чтобы использовать их в проверке.

Класс `Form`, от которого так или иначе наследуются все классы форм, поддерживает не принимающий параметров метод `clean`. Он преобразует введенные в форму данные к нужному типу, заполняет преобразованными данными словарь, что хранится в свойстве `cleaned_data` (см. *главу 11*), и возвращает его в качестве результата.

Переопределенный в потомке метод `clean` — идеальное место для выполнения проверки, которая затрагивает сразу несколько полей формы:

```
class GoodForm(forms.ModelForm):
  . . .
  def clean(self):
    cleaned_data = super(GoodForm, self).clean()
    if cleaned_data["price"] == cleaned_data["new_price"]:
      raise ValidationError("Цена с учетом скидки должна быть меньше!",↳
      code = "invalid")
    return cleaned_data
```

Здесь мы сначала вызываем унаследованный от родителя метод `clean`, чтобы получить словарь с уже преобразованными данными. Далее мы проверяем корректность этих данных и, если нужно, генерируем исключение `ValidationError`. И не забываем вернуть словарь с данными в случае успешной проверки.

Управление выводом форм на экран

При выводе формы на экран Django сама подбирает для каждого поля наиболее подходящий с ее точки зрения элемент управления и генерирует вполне оптимальный HTML-код. Однако в некоторых случаях нам может понадобиться указать для какого-либо поля формы другой элемент управления, изменить оформление формы или даже полностью изменить ее формат. Настала пора выяснить, как это выполняется.

Назначение полям формы элементов управления

Функциональность элементов управления, или, в терминологии Django, *виджетов* (widgets), реализуется особыми классами, объявленными в модуле `django.forms`.

Указать для поля формы элемент управления мы можем либо в свойстве `widgets` вложенного класса `Meta`, либо в параметре `widget` конструктора класса поля.

- Свойство `widgets` вложенного класса `Meta` должно принимать в качестве значения словарь, ключами элементов которого станут имена полей, а значениями — имена классов элементов управления:

```
from django import forms
class GoodForm(forms.ModelForm):
  class Meta:
    model = Good
    widgets = {"description": forms.Textarea,↳
    "category": forms.RadioSelect}
    . . .
```

- Параметр `widget` конструктора класса поля должен принимать как значение класс элемента управления:

```
from django import forms
class GoodForm(forms.ModelForm):
  class Meta:
    model = Good
  . . .
  description = forms.CharField(widget = forms.Textarea)
  category = forms.ModelChoiceField↳
  (queryset = Category.objects.all(), widget = forms.RadieSelect)
  . . .
```

Здесь мы задаем для поля `description` в качестве элемента управления область редактирования, а для поля `category` — набор переключателей.

Все полезные для нас элементы управления, поддерживаемые Django, перечислены в табл. 12.2, равно как и классы полей формы, для которых они используются по умолчанию.

***Таблица 12.2.** Классы элементов управления*

Класс элемента управления	Описание	Класс поля
`CheckboxInput`	Флажок	`BooleanField`
`DateInput`	Поле для ввода даты	`DateField`
`DateTimeInput`	Поле для ввода значения даты и времени	`DateTimeField`
`EmailInput`	Поле для ввода адреса электронной почты	`EmailField`
`HiddenInput`	Скрытое поле. Используется для служебных целей	
`NumberInput`	Поле для ввода числа	`FloatField`, `IntegerField`
`PasswordInput`	Поле для ввода пароля	

Таблица 12.2 (окончание)

Класс элемента управления	Описание	Класс поля
RadioSelect	Набор переключателей	
Select	Список	ChoiceField, ModelChoiceField
Textarea	Область редактирования	
TextInput	Обычное поле ввода	CharField, IPAddressField, GenericIPAddressField, SlugField
TimeInput	Поле для ввода времени	TimeField
URLInput	Поле для ввода интернет-адреса	URLField

Мы можем указать для элемента управления дополнительные характеристики. Они задаются в виде именованных параметров в круглых скобках после имени класса элемента управления. Если параметров нет, круглые скобки ставить не нужно.

Наиболее полезные для нас параметры и классы элементов управления, которые их поддерживают, приведены в табл. 12.3.

Таблица 12.3. Параметры элементов управления и классы

Параметр	Описание	Класс элемента управления
attrs	Атрибуты формирующего элемент HTML-тега. Указываются в виде словаря, ключи элементов которого совпадают с именами атрибутов, а значения этих элементов станут значениями соответствующих атрибутов	Все
render_value	Имеет смысл лишь при повторном выводе формы на экран в результате ввода ошибочных данных. Если True, введенное ранее значение будет присутствовать в элементе управления. Значение по умолчанию — False	PasswordInput
format	Формат для вывода изначального значения данного элемента управления. Для задания формата применяются символы литерапов, приведенные в табл. 7.4, перед которыми в этом случае ставится символ процента (%)	DateInput, DateTimeInput, TimeInput

Вот пример:

```
class GoodForm(forms.ModelForm):
  class Meta:
    . . .
    widgets = {"description": forms.Textarea(attrs =↳
    {"rows": 80, "cols": 10})}
```

Управление генерированием HTML-кода формы

Django предоставляет нам довольно много возможностей для управления генерированием HTML-кода, который создает форму. Мы можем использовать другой тег для вывода надписей или вывести их не левее элементов управления, а правее их. Мы можем переделать по своему усмотрению список ошибок. Мы даже можем создать форму с использованием не набора абзацев, списка или таблицы, как делали это в *главе 11*, а, например, блоков.

Контекст данных шаблона, где присутствует форма, включает в себя переменную, хранящую эту форму. Эта переменная обычно носит имя `form`.

Но ее значение можно рассматривать и как список полей, включенных в состав формы. Следовательно, мы можем обработать этот список в теге шаблона `for` (см. *главу 7*):

```
<form action="" method="post">
  {% for field in form %}
    . . .
  {% endfor %}
  <p><input type="submit" value="Сохранить"></p>
</form>
```

Отдельные поля формы представляются объектами класса `BoundField`. Этот класс поддерживает следующие свойства:

- `label` — текст надписи;
- `label_tag` — HTML-код, создающий надпись;
- `help_text` — текст дополнительных сведений;
- `is_hidden` — `True`, если данное поле является скрытым;
- `errors` — HTML-код, создающий неупорядоченный список, каждый пункт которого представляет собой описание одной из возникших ошибок;
- `name` — имя поля.

При обращении к самому объекту класса `BoundField` будет возвращен HTML-код, создающий элемент управления.

Вот так может выглядеть код шаблона, создающего форму на основе набора блоков:

```
<form action="" method="post">
  {% for field in form %}
    <div class="form-field">
      {% if field.errors|length > 0 %}
        <div class="error-list">
          {{ field.errors }}
        </div>
      {% endif %}
```

```
      <div class="label">{{ field.label }}</div>
      <div class="control">{{ field }}</div>
      {% if field.help_text %}
        <div class="help">{{ field.help_text }}</div>
      {% endif %}
    </div>
  {% endfor %}
  <div class="submit-button"><input type="submit"
  value="Сохранить"></div>
</form>
```

Здесь для получения количества элементов в списке мы используем фильтр шаблона `length`, описанный в *главе 7*.

Свойство `errors` класса `BoundField` хранит список строк, каждая из которых представляет собой описание ошибки. Мы можем воспользоваться этим, чтобы сгенерировать список ошибок на свой вкус:

```
. . .
{% if field.errors|length > 0 %}
  <div class="error-list">
    {% for error in field.errors %}
      <div class="error-description">{{ error }}</div>
    {% endfor %}
  </div>
{% endif %}
. . .
```

Класс `Form` также поддерживает не принимающие параметров методы `visible_fields` и `hidden_fields`. Они возвращают списки всех видимых и невидимых полей формы соответственно:

```
<form action="" method="post">
  {% for field in form.hidden_fields %}
    {{ field }}
  {% endfor %}
  {% for field in form.visible_fields %}
    . . .
  {% endfor %}
  <div class="submit-button"><input type="submit"
  value="Сохранить"></div>
</form>
```

Класс `Form` поддерживает и набор свойств, чьи имена совпадают с именами полей формы. Эти свойства хранят соответствующие поля — мы можем использовать их, чтобы непосредственно сослаться на то или иное поле:

```
<form action="" method="post">
  {{ form.in_stock }}
  {{ form.category }}
```

```
  {{ form.name }}
  {{ form.description }}
  <div class="submit-button"><input type="submit"
  value="Сохранить"></div>
</form>
```

Здесь мы выводим в форме для ввода нового товара поля в следующем порядке: признак, есть ли товар в наличии, категория товара, название товара и его описание.

Осталось лишь сказать, что класс `Form` поддерживает еще два полезных свойства:

- `required_css_class` — стилевой класс, который будет привязан к элементам управления, соответствующим обязательным полям формы;
- `error_css_class` — стилевой класс, который будет привязан к элементам управления, в которых были введены некорректные данные.

Вот пример:

```
class GoodForm(forms.ModelForm):
  . . .
  required_css_class = "required"
  error_css_class = "error"
  . . .
```

Сообщения Django и их использование

Правила хорошего тона Web-программирования рекомендуют каким-то образом уведомлять посетителя о том, что введенные им данные были успешно сохранены. Django предлагает для этого так называемые *сообщения*, которые представляют собой строки, записываются в особое хранилище, при необходимости извлекаются из него и удаляются сразу же после извлечения.

Каждый вновь созданный проект уже настроен таким образом, чтобы в его приложениях сразу можно было использовать сообщения. В частности, в список активных включено приложение `django.contrib.messages`, которое и реализует работу подсистемы сообщений. (О списке активных приложений проекта и его конфигурировании говорилось в *главе 4*.)

Все необходимые инструменты — функции и переменные — объявлены в модуле `django.contrib.messages`. Нам следует его импортировать:

```
from django.contrib import messages
```

Функция `add_message` добавляет в хранилище новое сообщение. Она принимает три параметра:

- сведения о запросе в виде объекта класса `HttpRequest` (за подробностями — к *главе 6*);
- уровень сообщения в виде числа;
- строка, представляющая само сообщение.

Django предлагает пять уровней сообщения. Соответствующие им числовые идентификаторы хранятся в следующих переменных:

- DEBUG — отладочное сообщение (сообщения такого уровня будут игнорироваться после запуска сайта в эксплуатацию);
- INFO — информационное сообщение общего назначения;
- SUCCESS — сообщение об успешном выполнении какой-либо операции;
- WARNING — предупреждение о какой-либо нештатной ситуации, которая, тем не менее, не может нарушить работу сайта;
- ERROR — сообщение о критической ошибке.

Вот пример:

```
class GoodCreate(TemplateView):
  . . .
  def post(self, request, *args, **kwargs):
    if self.kwargs["cat_id"] == None:
      cat = Category.objects.first()
    else:
      cat = Category.objects.get(pk = self.kwargs["cat_id"])
    self.form = GoodForm(request.POST)
    if self.form.is_valid():
      self.form.save()
      messages.add_message(request, messages.SUCCESS,↩
      "Товар успешно добавлен в список")
      return redirect("index", cat_id = cat.id)
    else:
      return super(GoodCreate, self).get(request, *args, **kwargs)
```

Здесь мы добавляем в класс-контроллер GoodCreate функцию отправки сообщения об успешном добавлении товара в список.

Если в хранилище есть сообщения, приложение django.contrib.messages само формирует в контексте данных шаблона переменную messages. Эта переменная хранит список сообщений, каждая из которого представляет собой объект класса Message. Этот класс поддерживает свойство tags, хранящее строковое представление уровня сообщения, и свойство message, где хранится сам его текст. Впрочем, для получения текста сообщения можно обратиться к самому этому объекту.

Такой код мы можем использовать для вывода сообщений на Web-странице:

```
{% if messages %}
  <ul class="message-list">
    {% for message in messages %}
      <li class="{{ message.tags }}">{{ message }}</li>
    {% endfor %}
  </ul>
{% endif %}
```

Кстати, если мы используем высокоуровневые классы-контроллеры `CreateView` и `UpdateView`, предназначенные для добавления и изменения записи соответственно, то можем переложить работу по отправке сообщений об успехе на плечи такого контроллера. Для этого мы добавим в его список родителей класс `SuccessMessageMixin`, объявленный в модуле `django.contrib.messages.views`, и зададим текст сообщения в унаследованном от этого класса свойстве `success_message`. Отметим, что класс `SuccessMessageMixin` должен быть первым в списке родителей, иначе сообщения формироваться не будут:

```
from django.contrib.messages.views import SuccessMessageMixin
class GoodCreate(SuccessMessageMixin, TemplateView):
  . . .
  success_message = "Товар успешно добавлен в список"
  . . .
```

Это еще один довод использовать для рутинных операций классы-контроллеры высокого уровня.

К сожалению, с классом-контроллером `DeleteView`, выполняющим удаление записи, такой номер не пройдет. Нам придется переопределять в его потомке метод `post`, в котором и выполнять отправку сообщения.

Данные сессии

Многие современные Web-сайты имеют форму, где зарегистрированные пользователи вводят свои имя и пароль, чтобы совершить вход. И зачастую при повторном посещении такого сайта в поле ввода имени пользователя этой формы уже подставлено введенное ранее значение. Можем ли мы реализовать нечто подобное на своем сайте?

Конечно, можем! Мы сохраним введенное значение в составе так называемых *данных сессии*, которые, с одной стороны, привязываются к конкретному IP-адресу компьютера, с которого посетитель зашел на сайт (со всех прочих IP-адресов они недоступны), а с другой — хранятся в течение определенного времени, что позволяет извлечь их во время следующего посещения сайта.

Как и в случае сообщений, для работы с данными сессии нам не нужно ничего менять в настройках сайта. Все необходимые параметры там уже заданы, в том числе, в списке активных приложений присутствует приложение `django.contrib.sessions`, которое и реализует работу подсистемы данных сессии Django...

Это приложение, в числе прочего, создает в свойстве `session` объекта сведений о запросе словарь. Ключи элементов этого словаря представляют собой имена хранящихся в составе сессии данных, а значения элементов — сами эти данные.

Давайте дополним наш многострадальный класс-контроллер `GoodCreate` возможностью хранить в составе данных сессии указанный посетителем признак того, есть ли товар в наличии. В качестве имени этого значения выберем имя соответствующего поля модели — `in_stock`:

```
class GoodCreate(TemplateView):
  . . .
  def get(self, request, *args, **kwargs):
    if self.kwargs["cat_id"] == None:
      cat = Category.objects.first()
    else:
      cat = Category.objects.get(pk = self.kwargs["cat_id"])
    try:
      in_stock = request.session["in_stock"]
    except:
      in_stock = True
    self.form = GoodForm(initial =↪
    {"category": cat, "in_stock": in_stock})
    return super(GoodCreate, self).get(request, *args, **kwargs)
  . . .
```

В методе `get` класса-контроллера мы извлекаем из данных сессии значение с именем `in_stock` и задаем его в составе изначальных данных для формы создания нового товара.

При этом нужно сказать, что значение свойства `session` объекта сведений о запросе — не простой словарь, а объект класса `SessionBase`. Помимо обычной функциональности словаря Python, он поддерживает метод `get`, принимающий два параметра:

- обязательный — строка с именем сохраненного в составе данных сессии значения;
- необязательный `default` — значение, которое будет возвращено методом, если такого имени в составе данных сессии не было обнаружено. Если не указан, будет возвращено `None`.

В качестве результата метод `get` возвращает либо значение с указанным именем, хранящееся в данных сессии, либо величину, указанную параметром `default`.

Так что приведенный ранее код можно записать немного по-другому:

```
class GoodCreate(TemplateView):
  . . .
  def get(self, request, *args, **kwargs):
    . . .
    self.form = GoodForm(initial =↪
    {"category": cat, "in_stock": request.session.get("in_stock", True)})
    return super(GoodCreate, self).get(request, *args, **kwargs)
  . . .
```

Получившийся код стал заметно короче.

А в методе `post` мы сохраняем значение этого признака в составе данных сессии:

```
class GoodCreate(TemplateView):
  . . .
  def post(self, request, *args, **kwargs):
    . . .
```

```
    self.form = GoodForm(request.POST)
    if self.form.is_valid():
      request.session["in_stock"] = self.form.cleaned_data["in_stock"]
      self.form.save()
      return redirect("index", cat_id = cat.id)
    else:
      return super(GoodCreate, self).get(request, *args, **kwargs)
```

Наборы форм

Обычная форма позволяет создать только одну запись модели. Если нам потребуется создать еще одну запись, нам придется вывести эту форму повторно. То же самое и с правкой записи — нам потребуется последовательно выводить одну и ту же форму для каждой записи, которую нужно изменить.

Но есть и другой вариант — вывести на экран сразу несколько форм, каждая из которых «отвечает» за отдельную запись модели. Тогда мы можем внести исправления сразу в несколько моделей и даже создать новую запись, указав данные для нее в специально предусмотренной для этого «пустой» форме. А потом сохранить все разом.

Django позволяет сделать нечто подобное с помощью так называемых *наборов форм* (formsets). Каждый такой набор сам создает на экране нужное количество форм, сам заполняет их данными, взятыми из записей модели (конечно, если модель содержит хоть одну запись), и сам же выполняет сохранение введенных в формы данных, при необходимости создавая новые записи.

Применив набор форм, мы можем ввести сразу несколько категорий и товаров, относящихся к определенной категории. А применив связанный набор форм, мы можем создать сразу несколько товаров, относящихся к какой-либо категории.

Как правило, на практике используются наборы форм, связанные с моделями, и вложенные наборы форм. Их-то мы здесь и рассмотрим.

Наборы форм, связанные с моделями

Как и обычные связанные с моделями формы, наборы форм такого рода привязываются к определенной модели. Это позволяет реализовать автоматическое сохранение введенных в них данных.

Создание наборов форм

Создать набор форм, связанный с моделью, можно вызовом функции `modelformset_factory`, объявленной в модуле `django.forms.models`. В качестве единственного обязательного параметра ей передается класс нужной модели:

```
from django.forms.models import modelformset_factory
from page.models import Category
CategoryFormset = modelformset_factory(Category)
```

Так мы создаем набор форм для работы со списком категорий.

Помимо обязательного параметра, функция `modelformset_factory` принимает ряд именованных необязательных. Они включают в себя параметры `fields`, `exclude`, `widgets`, `labels`, `help_texts` и `error_messages`, соответствующие одноименным свойствам класса `Meta` форм, связанных с моделями (см. эту главу и *главу 12*):

```
CategoryFormset = modelformset_factory(Category,↳
labels = {"name": "Название"},↳
help_texts = {"name": "Должно быть уникальным"})
```

Здесь мы задаем для поля названия категории надпись и справочный текст.

Можно также указать несколько параметров, специфичных именно для наборов форм (табл. 12.4).

Таблица 12.4. *Параметры функции* `modelformset_factory`*, специфичные для наборов форм*

Параметр	Описание
`form`	Задает класс формы, которая будет использована для ввода данных. Значение по умолчанию — `None` (т. е. форма будет генерироваться автоматически)
`max_num`	Задает максимальное количество форм в наборе. Значение по умолчанию — `None` (1000 форм, чего хватит в большинстве случаев)
`extra`	Задает максимальное количество выводимых пустых форм для создания новых записей. Значение по умолчанию — 1
`validate_max`	Если `True`, превышение указанного в параметре `max_num` количества форм вызывает ошибку ввода. Значение по умолчанию — `False`

Вот пример:

```
class CategoryForm(forms.ModelForm):
  class Meta:
    model = Category
  name = forms.CharField(label = "Название",↳
  help_text = "Должно быть уникальным")
```

Здесь мы создаем форму для ввода категорий.

```
CategoryFormset = modelformset_factory(Category, form = CategoryForm,↳
max_num = 10, validate_max = True)
```

А здесь — набор форм, указав для него только что созданную форму и ограничив число выводимых форм десятью.

Но что же возвращает функция `modelformset_factory` в качестве результата? Нет, не объект набора форм, а представляющий его класс. На основе которого нам самим придется создать объект:

```
formset = CategoryFormset()
```

Класс набора форм является потомком класса `BaseModelFormSet`, объявленного в том же модуле `django.forms.models`. Конструктор этого класса поддерживает необяза-

тельный параметр queryset, с помощью которого мы можем указать набор записей, из которого будут взяты данные для заполнения форм. Этот набор записей будет использован вместо сформированного на основе указанной нами в вызове функции modelformset_factory модели:

```
formset = CategoryFormset(queryset = Category.objects.order_by("-name"))
```

Здесь мы задаем в качестве набора записей список категорий, отсортированный по их названиям в обратном порядке.

Вывод наборов форм

Итак, мы создали набор форм. Теперь нужно как-то вывести его на экран, чтобы посетитель смог ввести в него данные.

Базовый класс набора форм BaseModelFormSet поддерживает знакомые нам по *главе 11* методы as_table, as_p и as_ul. Так что в самом простом случае мы можем просто вызвать один из этих методов в коде шаблона.

Но здесь есть один нюанс. Набор форм должен быть помещен внутри единого для всех входящих в него форм тега <form> (это значит, что на Web-странице набор форм представляет собой одну большую форму):

```
<form method="post" action="">
  {% csrf_token %}
  {{ formset.as_p }}
  <input type="submit" value="Сохранить">
</form>
```

Так мы выводим набор форм, отформатированный в виде набора абзацев. (Здесь предполагается, что набор форм хранится в переменной контекста formset.)

Набор форм можно рассматривать как список входящих в его состав форм. Эти формы мы можем обработать и вывести по очереди, в цикле.

Однако здесь возникает другой нюанс. Если мы выводим входящие в состав набора формы по отдельности, то в обязательном порядке должны поместить в тег <form> еще и особую служебную форму набора. Эта служебная форма включает в свой состав скрытые поля HTML, содержащие служебные данные, и хранится в свойстве management_form класса BaseModelFormSet:

```
<form method="post" action="">
  {% csrf_token %}
  {{ formset.management_form }}
  {% for form in formset %}
    <div class="form">{{ form.as_p }}</div>
  {{ endfor }}
  <input type="submit" value="Сохранить">
</form>
```

Здесь мы заключаем каждую форму в блок.

```
<form action="" method="post">
  {% csrf_token %}
  {{ formset.management_form }}
  {% for form in formset %}
    {% for field in form.hidden_fields %}
      {{ field }}
    {% endfor %}
    {% for field in form.visible_fields %}
      <div class="form-field">
        {% if field.errors.count > 0 %}
          <div class="error-list">
            {{ field.errors }}
          </div>
        {% endif %}
        <div class="label">{{ field.label }}</div>
        <div class="control">{{ field }}</div>
        {% if field.help_text %}
          <div class="help">{{ field.help_text }}</div>
        {% endif %}
      </div>
    {% endfor %}
  {% endfor %}
  <div class="submit-button"><input type="submit"
  value="Сохранить"></div>
</form>
```

А здесь заключаем в блок каждый элемент управления формы.

Сохранение введенных в набор форм данных

Хорошо! Посетитель ввел в набор форм данные и нажал кнопку сохранения. Что нам следует сделать, чтобы не разочаровать его?

Конструктор базового класса `BaseModelFormSet` принимает два необязательных неименованных параметра, которыми ему передаются, соответственно, данные и файлы для вставки в формы. В этом он схож с конструктором обычной формы:

```
formset = CategoryFormset(request.POST, request.FILES)
```

Этот класс поддерживает также знакомые нам по *главе 11* методы `is_valid` и `save`:

```
if formset.is_valid():
  formset.save()
```

Следовательно, для сохранения данных из набора форм мы можем использовать те же приемы, что и для сохранения данных, занесенных в обычную связанную с моделью форму.

Далее приведен полный код класса-контроллера, который можно использовать для вывода набора форм и сохранения введенных в него данных. Этот класс предназначен для работы с категориями товаров.

```
from django.forms.models import modelformset_factory
from django.views.generic.base import TemplateView
from page.models import Category

CategoryFormset = modelformset_factory(Category,↳
labels = {"name": "Название"})

class CategoryListView(TemplateView):
  template_name = "cats.html"
  formset = None
  def get(self, request, *args, **kwargs):
    self.formset = CategoryFormset()
    return super(CategoryListView, self).get(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(CategoryListView, self).get_context_data(**kwargs)
    context["formset"] = self.formset
    return context
  def post(self, request, *args, **kwargs):
    self.formset = CategoryFormset(request.POST)
    if self.formset.is_valid():
      self.formset.save()
      # Перенаправляем посетителя на какую-либо страницу
    else:
      return super(CategoryListView, self).get(request, *args, **kwargs)
```

Отметим, что родителем этого класса выступает наиболее простой из классов-контроллеров `TemplateView`. Он подходит здесь наилучшим образом, т. к. нам не требуется ни выводить список записей, ни выбирать из набора записей нужную нам по ее идентификатору, т. е. использовать возможности, предоставляемые его более сложными потомками.

Реализация переупорядочения и удаления записей посредством набора форм

Набор форм, что мы только что создали, позволяет посетителю править данные в уже существующих записях и создавать новые. Но наборы форм Django могут также выполнять переупорядочение и удаление записей. И все, что нам нужно сделать, чтобы задействовать эту возможность, — добавить в контроллер чуть-чуть кода.

Функция `modelformset_factory` поддерживает еще два необязательных параметра, которые мы еще не рассматривали:

- `can_order` — если `True`, пользователь сможет переупорядочивать записи;
- `can_delete` — если `True`, пользователь сможет удалять записи.

Значения по умолчанию обоих этих параметров — `False`.

Вот пример:

```
CategoryFormset = modelformset_factory(Category,↳
labels = {"name": "Название"}, can_order = True, can_delete = True)
```

Здесь мы даем посетителю возможность менять порядок категорий и удалять их средствами набора форм.

Более никаких действий для этого предпринимать не нужно. Для указания порядка следования записей и их пометки с целью удаления базовый класс `BaseModelFormSet` сам сформирует дополнительные элементы управления, а при сохранении сам поменяет записи местами и удалит их.

Мы можем управлять выводом упомянутых ранее дополнительных элементов управления. Класс `BaseModelFormSet` поддерживает свойства `can_order` и `can_delete`, которые возвращают `True`, если в наборе форм активизированы средства для переупорядочивания и удаления записей соответственно. Также он поддерживает свойства `ORDER` и `DELETE`, которые хранят элементы управления, предназначенные для переупорядочивания записей и их удаления.

Вот такой код можно использовать в шаблоне для вывода набора форм с возможностью переупорядочивания и удаления записей:

```
<form action="" method="post">
  {% csrf_token %}
  {{ formset.management_form }}
  {% for form in formset %}
    {% for field in form.hidden_fields %}
      {{ field }}
    {% endfor %}
    {% for field in form.visible_fields %}
      {% if field.label != "Порядок" and field.label != "Удалить" %}
        <div class="form-field">
          {% if field.errors.count > 0 %}
            <div class="error-list">
              {{ field.errors }}
            </div>
          {% endif %}
          <div class="label">{{ field.label }}</div>
          <div class="control">{{ field }}</div>
          {% if field.help_text %}
            <div class="help">{{ field.help_text }}</div>
          {% endif %}
        </div>
      {% endif %}
    {% endfor %}
    {% if formset.can_order %}
      <div>Порядок: {{ form.ORDER }}</div>
    {% endif %}
    {% if formset.can_delete %}
      <div>{{ form.DELETE }} Удалить</div>
    {% endif %}
  {% endfor %}
```

```
  <div class="submit-button"><input type="submit"
  value="Сохранить"></div>
</form>
```

Здесь мы исключаем из основного набора элементов управления те, что имеют надписи «Порядок» и «Удалить», — элементы с такими надписями служат для указания порядка записей и их пометки для удаления. Эти элементы мы выводим отдельно.

Как набор форм выводится на экран?

А теперь посмотрим, как набор форм выглядит на экране.

Начнем с обычного набора, без возможности переупорядочивания и удаления записей. Он показан на рис. 12.1 (для удобства автор разделил отдельные формы горизонтальными линиями).

Видно, что этот набор представляет собой просто совокупность форм, каждая из которых отображает данные, хранящиеся в отдельной записи модели. Одна из форм пуста — она предназначена для создания новой записи (в нашем наборе только одна форма такого рода).

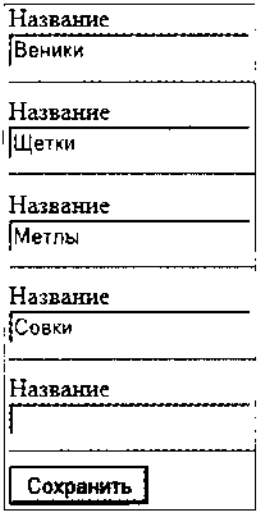


Рис. 12.1. Набор форм для работы с категориями

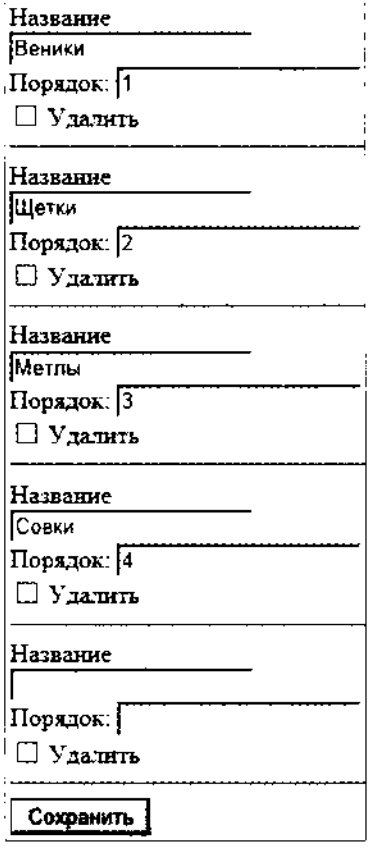


Рис. 12.2. Набор форм с возможностью переупорядочивания и удаления записей

После нажатия кнопки сохранения набор форм автоматически сохранит все внесенные изменения, занеся их в записи моделей и, при необходимости, создав новые записи.

Если мы дадим набору форм возможность переупорядочивания и удаления записей, он станет выглядеть так, как показано на рис. 12.2.

Здесь в каждую форму добавлены еще два элемента управления:

- поле ввода для указания порядкового номера записи. Изменяя порядковые номера, мы можем менять записи местами;
- флажок, который при установке пометит запись на удаление. При сохранении введенных в набор форм данных такие записи будут стерты из модели.

Как видим, наборы форм предлагают вполне интуитивно понятные средства для управления записями. Так что обязательно надо взять их на заметку...

Вложенные наборы форм

Вложенные наборы форм (inline formsets) почти во всем аналогичны только что рассмотренным нами наборам, связанным с моделями. За одним исключением — вложенный набор выводит только записи дочерней модели, связанной с указанной нами записью родительской модели. С помощью такого набора форм мы можем, например, вывести все товары, которые относятся к заданной категории, дополнить и исправить их перечень.

Вложенный набор форм создается вызовом функции `inlineformset_factory`, объявленной в модуле `django.forms.models`. Этой функции передаются уже два обязательных параметра: родительская модель и дочерняя модель.

```
from django.forms.models import inlineformset_factory
from page.models import Category, Good
CategoryGoodFormset = inlineformset_factory(Category, Good)
```

В этом примере мы создаем вложенный набор форм для работы со списком товаров, относящихся к определенной категории.

Функция `inlineformset_factory` также поддерживает все параметры, что доступны для функции `modelformset_factory`. Но в этом случае значением параметра `can_delete` будет `True` (так что вложенный набор форм по умолчанию станет поддерживать возможность удаления записей), а значение параметра `extra` — 3 (т. е. будет выведены сразу три «пустые» формы, предназначенные для создания новых записей).

Функция `inlineformset_factory` возвращает в качестве результата класс набора форм, являющийся потомком класса `BaseInlineFormSet` из модуля `django.forms.models`. Его конструктор принимает обязательный именованный параметр `instance`, с которым передается запись родительской модели. В таком случае вложенный набор форм будет манипулировать связанными с ней записями дочерней модели.

В остальном вложенные наборы форм ничем не отличаются от наборов форм, связанных с моделями.

Далее приведен код класса-контроллера, который будет выводить вложенный набор форм для работы со списком товаров, относящихся к указанной категории.

```
from django.forms.models import inlineformset_factory
from django.views.generic.base import TemplateView
from page.models import Category, Good

CategoryGoodFormset = inlineformset_factory(Category, Good)

class CategoryDetailView(TemplateView):
  template_name = "cat.html"
  formset = None
  cat = None
  def get(self, request, *args, **kwargs):
    self.cat = Category.objects.get(pk = self.kwargs["cat_id"])
    self.formset = CategoryGoodFormset(instance = self.cat)
    return super(CategoryDetailView, self).get(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(CategoryDetailView, self).get_context_data(**kwargs)
    context["category"] = self.cat
    context["formset"] = self.formset
    return context
  def post(self, request, *args, **kwargs):
    self.formset = CategoryGoodFormset(request.POST, request.FILES,↳
    instance = Category.objects.get(pk = self.kwargs["cat_id"]))
    if self.formset.is_valid():
      self.formset.save()
      # Перенаправляем посетителя на какую-либо страницу
    else:
      return super(CategoryDetailView, self).get(request, *args, **kwargs)
```

Здесь мы формируем в контексте данных шаблона переменную `cat`, хранящую указанную категорию. Это позволит нам вывести на странице сведения об этой категории. А идентификатор категории у нас передается через параметр `cat_id`, формируемый с помощью именованной группы регулярных выражений (см. *главу 6*).

Что дальше?

В этой главе мы продолжали заниматься формами, совершенствуя средства валидации данных, улучшая внешний вид и вводя в функциональность всяческие полезные мелочи. А еще мы рассмотрели наборы форм, которые позволят посетителю работать сразу над целым списком записей модели.

Следующая глава станет завершающей в разговоре о формах и вообще средствах ввода данных, поддерживаемых Django. Мы выясним, как реализовать выгрузку на сайт файлов, в частности файлов с графическими изображениями. Мы ведь собираемся добавить в список товаров их картинки, не так ли?

ГЛАВА 13

Выгрузка файлов на Web-сайт

В предыдущей главе мы занимались средствами валидации, управляли генерированием HTML-кода, создающего формы, работали с сообщениями Django и данными сессии и разбирались с наборами форм. Хоть эта функциональность форм и называется дополнительной, она, тем не менее, не будет лишней.

Очень многие современные сайты предоставляют посетителю возможность выгрузки файлов. Почтовые Web-сервисы позволяют прикреплять к письму произвольные файлы в виде вложений, интернет-магазины — указывать для товаров изображения, блоги — добавлять к статьям картинки.

Не удивительно, что и Django предоставляет богатые средства для выгрузки файлов на сайт и их обработки — в частности, вывода на Web-страницы.

Необходимые настройки сайта

Но перед тем как задействовать средства для выгрузки файлов, следует задать для проекта настройки, которые напрямую затрагивают работу этих средств. Поэтому сразу же откроем модуль `settings` пакета проекта.

Переменная `MEDIA_ROOT` задает полный путь к папке, где будут храниться выгруженные на сайт файлы, в виде строки:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'uploads')
```

Пусть выгруженные файлы хранятся в папке uploads, вложенной в папку проекта. (О переменной `BASE_DIR` и функции `os.path.join` рассказывалось в *главе 4*.)

> ***Значение переменной MEDIA_ROOT***
>
> Не следует указывать для переменной `MEDIA_ROOT` то же значение, что и для переменной `STATIC_ROOT` (см. *главу 8*).

И не забудем создать эту папку, поскольку Django за нас этого не сделает.

Переменная `MEDIA_URL` задает префикс для формирования интернет-адреса всех выгруженных файлов, также в виде строки:

```
MEDIA_URL = '/media/'
```

Теперь для файла good.jpg, хранящегося в папке goods\thumbnails, что находится в папке uploads, будет сгенерирован интернет-адрес **/media/goods/thumbnails/good.jpg**.

Далее нам нужно привязать заданную в переменной `MEDIA_ROOT` папку к заданному в переменной `MEDIA_URL` префиксу интернет-адресов. Это нужно для того, чтобы встроенный отладочный Web-сервер Django смог обработать запросы на загрузку выгруженных файлов, и мы получили бы возможность, скажем, увидеть выведенное на экран выгруженное графическое изображение.

Откроем модуль `urls` пакета проекта и вставим в его начало такие выражения:

```
from django.conf import settings
from django.conf.urls.static import static
```

После чего добавим в самый его конец следующий код:

```
urlpatterns += static(settings.MEDIA_URL,↳
document_root = settings.MEDIA_ROOT)
```

На этом необходимые настройки закончены.

Хранение файлов в модели

Django предлагает нам хранить файлы прямо в модели. Это позволяет, с одной стороны, связать файл или файлы непосредственно с сущностью, к которой они относятся, а с другой, указать для файла произвольные дополнительные сведения (дата и время выгрузки, описание, ключевые слова и пр.). А если потребуется, мы с легкостью получим интернет-адрес хранящегося в модели файла, чтобы вставить его в HTML-код результирующей страницы.

> ***Хранение путей к файлам***
>
> Разумеется, в поле модели хранятся не сами файлы, а их пути, заданные относительно указанной нами в переменной `MEDIA_ROOT` папки.

Классы полей для хранения файлов в модели

Нам доступны два класса полей модели, предназначенных для хранения файлов. Эти классы мы еще не рассматривали в *главе 5*.

- `FileField` — хранит файл любого типа;
- `ImageField` — хранит графический файл.

> ***Установка сторонней библиотеки Pillow***
>
> Чтобы успешно использовать в моделях поля класса `ImageFile`, следует установить стороннюю библиотеку Pillow. Установочный пакет этой библиотеки под Windows можно найти по интернет-адресу **http://www.lfd.uci.edu/~gohlke/pythonlibs/#pillow**.

Конструкторы классов `FileField` и `ImageField` принимают обязательный именованный параметр `upload_to`. Он задает папку, где физически размещаются хранящиеся в данном поле файлы, — эта папка должна быть вложена в папку, указанную нами

в переменной MEDIA_ROOT в настройках проекта (не забываем, что эту папку мы также должны создать самостоятельно):

```
class Good(models.Model):
  . . .
  thumbnail = models.ImageField(upload_to = "goods/thumbnails")
```

Пусть картинки товаров хранятся в подпапке goods\thumbnails, вложенную в папку uploads.

Мы можем использовать в путях к таким папкам литералы, с помощью которых указываются форматы значений даты и времени. Эти литералы приведены в табл. 7.4, и в данном случае их следует предварить символом процента (%):

```
class Good(models.Model):
  . . .
  thumbnail = models.ImageField(upload_to =↳
  "goods/thumbnails/%Y/%m/%d")
```

В этом случае файл изображения, загруженный, скажем, 14 марта 2014 года, будет сохранен в папке goods\thumbnails\2014\3\14.

Конструктор класса ImageField дополнительно поддерживает еще два необязательных параметра:

- width_field — имя поля модели, где будет храниться ширина изображения;
- height_field — имя поля модели, где будет храниться высота изображения.

Эти значения будут заноситься при каждом сохранении записи. Оба этих поля уже должны присутствовать в модели:

```
class Good(models.Model):
  . . .
  thumbnail_width = PositiveSmallIntegerField()
  thumbnail_height = PositiveSmallIntegerField()
  thumbnail = models.ImageField(uploaded_to ="goods/thumbnails",↳
  width_field = "thumbnail_width", height_field = "thumbnail_height")
```

Классы полей FileField и ImageField поддерживают также необязательный параметр max_length (о нем рассказывалось в *главе 5*). В этом случае значение по умолчанию указанного параметра равно 100.

Получение сведений о файлах, хранящихся в модели

Значение, хранящееся в полях классов FileField или ImageField, представляет собой объект особого класса FieldFile. Перечисленные далее свойства этого класса позволят нам получить различные сведения о хранящемся в поле файле:

- name — путь к файлу относительно папки, чей путь указан в переменной MEDIA_ROOT (см. ранее);
- size — размер файла в байтах;
- url — интернет-адрес файла;

- `width` — ширина графического изображения в пикселах;
- `height` — высота графического изображения в пикселах.

Последние два свойства поддерживаются лишь классом `ImageField`.

```
good = Good.objects.get(pk = kwargs["good_id"])
file_url = good.thumbnail.url
```

В этом примере мы получаем интернет-адрес файла, хранящего изображение товара.

```
<td>{% if good.thumbnail %}<img src="{{ good.thumbnail.url }}">↵
{% endif %}</td>
```

А здесь выводим изображение товара на Web-странице, предварительно проверив, указано ли оно.

Кроме того, класс `ImageField` поддерживает не возвращающий результата метод `delete`, который выполняет удаление выбранного файла. Этот метод принимает единственный необязательный параметр `save`:

- если значение параметра `save` равно `True` (значение по умолчанию), то после удаления файла значение поля будет очищено, а сама запись — сохранена;
- если же его значение равно `False`, то после удаления файла значение поля также будет очищено, но запись сохранена не будет. Нам придется самим вызвать у записи метод `save`, чтобы сохранить ее.

Вот пример удаления файла с параметром `save` равным `False`:

```
good = Good.objects.get(pk = kwargs["good_id"])
good.thumbnail.delete(save = False)
good.save()
```

> ***Удаление связанного файла***
>
> При удалении записи, включающей в свой состав поле класса `FileField` или `ImageField`, связанный с этим полем файл не будет удален. Нам придется удалить его самим — вызовом метода `delete`. В противном случае этот файл станет «мусорным» (о «мусорных» файлах и борьбе с ними будет рассказано в конце этой главы).
>
> ```
> good = Good.objects.get(pk = kwargs["good_id"])
> good.thumbnail.delete(save = False)
> good.delete()
> ```

Выгрузка файлов через формы

Так, полдела мы сделали — задали в моделях поля для хранения файлов. Осталось предусмотреть в формах поля для выгрузки файлов.

Поля формы, предназначенные для выгрузки файлов

Нам доступны классы полей форм `FileField` и `ImageField`. Их конструкторы поддерживают все общие для различных классов полей именованные параметры, перечисленные в табл. 11.5, а также параметр `max_length`.

```
class GoodForm(forms.ModelForm):
  class Meta:
    model = Good
  . . .
  thumbnail = forms.ImageField(label = "Есть в наличии")
```

В процессе валидации классы `FileField` и `ImageField` могут генерировать сообщения об ошибках с кодами `required`, `invalid` и `max_length`, описанными в табл. 12.1. Ошибка с кодом `invalid` возникает, как правило, в случае некорректного задания метода кодирования данных в форме. (Как задать правильный метод кодирования, мы узнаем чуть позже.)

```
class GoodForm(forms.ModelForm):
  . . .
  thumbnail = forms.ImageField(label = "Изображение",↳
  error_messages = {"required: "Укажите файл изображения"})
```

Также могут быть сгенерированы ошибки с кодами, что приведены в табл. 13.1.

Таблица 13.1. Коды ошибок классов `FileField` и `ImageField`

Код ошибки	Описание ошибки
`missing`	Файл почему-то не был выгружен (например, произошла ошибка чтения с диска)
`empty`	Выгружаемый файл пуст
`invalid_image`	Формат графического файла не поддерживается библиотекой Pillow. Генерируется лишь классом `ImageField`

Вот пример:

```
class GoodForm(forms.ModelForm):
  . . .
  thumbnail = forms.ImageField(label = "Изображение",↳
  error_messages = {"required: "Укажите файл изображения",↳
  "invalid_image": "Изображение должно быть сохранено в формате,↳
  поддерживаемом сайтом"})
```

Для обоих классов полей, предназначенных для выгрузки файла, по умолчанию создается элемент управления класса `ClearableFileInput`. Он представляет собой стандартное поле для задания выгружаемого файла, дополненное флажком, установка которого очищает это поле.

Мы можем указать для классов полей `FileField` и `ImageField` элемент управления `FileInput` — стандартное поле для указания выгружаемого файла, поддерживаемое HTML. Для обязательных полей он будет наилучшим выбором.

Настройка формы для выгрузки файлов

Ранее говорилось, что для успешной выгрузки файла нам следует задать для формы определенный метод кодирования данных перед отправкой. Он задается с помощью атрибута `enctype` тега `<form>`.

Ранее мы не указывали этот атрибут тега, и наши формы кодировали данные с применением метода `application/x-www-form-urlencoded`, используемого по умолчанию. Этот метод наилучшим образом подходит для кодирования обычных данных, но никак не отправляемых файлов.

Чтобы форма могла успешно отправить файл, нам следует явно указать для нее метод кодирования данных `multipart/form-data`:

```
<form action="" method="post" enctype="multipart/form-data">
  . . .
</form>
```

Класс `Form` поддерживает свойство `is_multipart`. Оно возвращает `True`, если форма включает в себя поля для хранения файлов и, следовательно, требует указания метода кодирования данных `multipart/form-data`:

```
<form action="" method="post"↳
{% if form.is_multipart %} enctype="multipart/form-data" {% endif %}>
  . . .
</form>
```

Обработка выгруженных файлов в контроллерах

Последнее, что мы должны сделать, — обработать выгруженные посетителем файлы в контроллере.

Мы уже знаем, что каждый контроллер, предназначенный для добавления, правки и удаления записи, вызывается дважды: при выводе формы и после того, как посетитель нажмет в этой форме кнопку отправки данных. Соответственно, объект формы также создается дважды. В первый раз ему передаются изначальные данные, которые должны присутствовать в форме, а во второй раз — данные, введенные посетителем, для их валидации и, возможно, сохранения.

Когда форма создается второй раз, данные, введенные посетителем и полученные методом POST, передаются ее конструктору первым неименованным параметром:

```
. . .
self.form = GoodForm(request.POST)
if self.form.is_valid():
  self.form.save()
  . . .
```

Однако эти данные не включают в свой состав выгруженные файлы. Последние передаются отдельно, доступны через свойство `FILES` объекта сведений о запросе и также представляют собой обычный словарь Python. Этот словарь мы укажем вторым неименованным параметром конструктора формы.

Вот так будет выглядеть фрагмент кода метода POST в классе-контроллере для создания новой записи:

```
. . .
self.form = GoodForm(request.POST, request.FILES)
if self.form.is_valid():
  self.form.save()
  . . .
```

А вот так — в классе-контроллере для правки записи:

```
. . .
self.form = GoodForm(request.POST, request.FILES, instance = good)
if self.form.is_valid():
  self.form.save()
  . . .
```

Проверка типа выгруженных файлов

Как правило, сайты в большинстве случаев допускают выгрузку только файлов определенного типа. Это позволяет, с одной стороны, упростить и стандартизировать их обработку, а с другой, обезопасить сайт от опасного ПО.

Поэтому нам нужно проверять типы всех выгруженных файлов перед тем, как сохранить введенные в форму данные, и в случае указания файла недопустимого типа немедленно уведомлять об этом посетителя. И если класс поля `ImageField` сам выполняет проверку, является ли выбранный файл графическим и поддерживается ли библиотекой Pillow, то в случае класса `FileField` нам придется делать это самим.

Каждый файл, перечисленный в словаре, что хранится в свойстве `FILES` объекта сведений о запросе, является объектом класса `UploadedFile`. Этот класс поддерживает свойство `content_type`, хранящее MIME-тип файла в виде строки.

Далее приведен фрагмент кода класса-контроллера, реализующий такую проверку. Он удостоверяется, что посетитель выгрузил через форму архивный файл формата ZIP.

```
. . .
self.form = SomeForm(request.POST, request.FILES)
if self.form.is_valid():
  filetype = request.FILES["somefile"].content_type
  if filetype == "application/zip":
    self.form.save()
    messages.add_message(request, messages.SUCCESS, "Позиция добавлена")
    return redirect("index")
  else:
    messages.add_message(request, messages.ERROR,↳
    "Допускаются только архивы ZIP!")
    return super(SomeCreate, self).get(request, *args, **kwargs)
else:
  return super(GoodCreate, self).get(request, *args, **kwargs)
```

Список MIME-типов наиболее часто встречающихся форматов файлов можно найти по интернет-адресу **http://en.wikipedia.org/wiki/Internet_media_type**.

Проблема «мусорных» файлов и ее решение

А теперь поговорим об одной проблеме, с которой мы обязательно столкнемся, как только выгрузим на наш сайт первые файлы. Эта проблема настолько важна, что автор посвятил ей отдельный раздел.

Мы уже знаем, что поля классов `FileField` и `ImageField` хранят не сам файл — это бы сильно увеличило размер базы данных и отрицательно сказалось бы на производительности, — а лишь ссылку на него. Эта ссылка представляет собой путь к файлу, указанный относительно папки, где хранятся выгруженные файлы.

Но дело в том, что при удалении записи модели, в которой присутствует поле одного из упомянутых ранее классов, Django не удаляет сам хранящийся в этом поле файл. Он так и остается на диске, лишь занимая свободное пространство. То же самое происходит, если в поле будет занесен другой файл, — тот файл, что хранился в поле ранее, и в этом случае не будет удален.

Подобные файлы, не привязанные ни к какой записи и не используемые ничем, носят в обиходе название «мусорных». Ибо они и в самом деле представляют собой мусор.

Но как избавиться от «мусорных» файлов? Точнее, как указать Django удалять такие файлы самостоятельно?

В этой главе мы узнали, что в поле классов `FileField` и `ImageField` хранится объект класса `FieldFile`, представляющий сведения о привязанном к этому полю файле. Этот класс поддерживает метод `delete`, который выполняет удаление помещенного в поле файла.

А в *главе 5* мы узнали, что все модели поддерживают унаследованные от класса-родителя `Model` методы `save` и `delete`. Переопределив эти методы в классах моделей, мы можем поместить в них код, который будет выполнять какие-либо действия при сохранении и удалении записи.

И в самом деле: методы `save` и `delete` класса модели — просто идеальное место для вставки кода, который будет удалять «мусорные» файлы! Так и сделаем.

Вот такой код переопределенного метода `save` мы можем использовать для удаления файла, который хранился в поле класса `FileField` или `ImageField` ранее, до правки:

```
class Good(models.Model):
  . . .
  thumbnail = models.ImageField(upload_to = "goods/thumbnails")
  . . .
  def save(self, *args, **kwargs):
    try:
      this_record = Good.objects.get(id = self.id)
```

```
    if this_record.thumbnail != self.thumbnail:
      this_record.thumbnail.delete(save = False)
  except:
    pass
  super(Good, self).save(*args, **kwargs)
```

Здесь мы ищем в модели Good запись, чей идентификатор совпадает с идентификатором текущей записи, т. е. ту же самую запись. Если хранящийся в ее поле thumbnail файл не совпадает с тем, что хранится в том же поле текущей записи, т. е. если в это поле было занесено новое значение — другой файл, — то старый файл (который в этом случае станет «мусорным») удаляется. На всякий случай мы поместили весь код в блок обработки исключений, который в случае возникновения исключения ничего не делает.

Отметим, что при вызове метода delete мы указали для его параметра save значение False. Если мы этого не сделаем, Django в момент удаления файла выполнит сохранение записи. При этом вновь будет вызван метод save, и приложение войдет в бесконечный цикл вызовов, который прервется лишь примерно через минуту. Такого, безусловно, не следует допускать.

А этот код переопределенного метода delete мы можем использовать, чтобы при удалении записи удалялся и хранящийся в поле файл:

```
class Good(models.Model):
  . . .
  thumbnail = models.ImageField(upload_to = "goods/thumbnails")
  . . .
  def delete(self, *args, **kwargs):
    self.thumbnail.delete(save = False)
    super(Good, self).delete(*args, **kwargs)
```

Отметим, что в качестве значения параметра save метода delete мы указали False. В противном случае запись будет удалена раньше времени, и в работе модели может возникнуть ошибка.

> ***Другие способы решения проблемы «мусорных» файлов***
>
> Существуют и другие способы решения проблемы «мусорных» файлов. В их числе — написание собственного класса файлового хранилища, который сам удаляет такие файлы, и даже установка дополнительной библиотеки, которая работает в отдельном процессе и выполняет поиск и удаление файлов, оказавшихся в числе «мусора». Однако здесь автор рассмотрел самый простой способ.

Что дальше?

В этой главе мы вели разговор об инструментах Django, призванных помочь нам в реализации выгрузки файлов на сайт. И на этом закончили рассказ о том, как дать посетителю возможность добавлять и править данные сайта.

Следующая глава будет посвящена очень важному вопросу — разграничению доступа. В самом деле, мы же не хотим, чтобы кто угодно имел доступ к внутренним данным нашего сайта!

ЧАСТЬ IV

Разграничение доступа. Комментарии. Статичные страницы

ГЛАВА 14

Разграничение доступа

Предыдущая часть книги была посвящена реализации ввода и правки внутренних данных сайта. Мы познакомились с высокоуровневыми классами-контроллерами, предназначенными для добавления, изменения и удаления записей, формами, связанными с моделью, обычными формами и инструментами моделей, которые позволят нам добавлять, править и удалять записи непосредственно. Также мы узнали о средствах валидации данных, введенных в формы, управлении генерированием HTML-кода, который создает формы, и, наконец, о возможностях Django по выгрузке файлов на сайт. Теперь мы знаем о вводе данных все, что нам нужно.

Но проблема в том, что не всем посетителям сайта следует давать возможность работы с его внутренними данными. В самом деле, не годится открывать доступ к инструментам для добавления, правки и удаления товаров любому желающему, ведь среди этих самых любых желающих вполне может оказаться интернет-хулиган, а то и настоящий злоумышленник...

Так что нам потребуется как-то ограничить доступ к разделу, предназначенному для работы с внутренними данными сайта (*административному*, или *закрытому*, *разделу*), реализовав систему разграничения доступа. Как это сделать?

Принципы разграничения доступа

Во-первых, нам следует создать еще одну модель, которая будет хранить список *зарегистрированных пользователей* сайта. Каждая запись этой модели будет хранить имя пользователя, под которым он здесь зарегистрирован («логин»), пароль и, возможно, какие-либо другие данные (реальные имя и фамилия, год рождения, адрес электронной почты, права и пр.). В этот список мы занесем всех посетителей, которые должны иметь доступ к административному разделу.

Во-вторых, мы потребуем, чтобы все такие посетители перед тем, как получить доступ к административному разделу, выполнили процедуру *входа* на сайт. Для этого мы создадим особую Web-страницу с формой, где посетитель должен вести свое имя и пароль (*страницу входа*).

В-третьих, мы сделаем так, чтобы обычные, не зарегистрированные в созданном ранее списке посетители (*гости*) не имели доступа к страницам административного раздела. Делается это с применением особых программных средств, которые мы обязательно рассмотрим.

Обычно гости, пытающиеся добраться до административного раздела, перенаправляются на страницу входа. Таким образом, они могут выполнить процедуру входа, если, конечно, занесены в список пользователей, — в противном же случае, по крайней мере, поймут, что этот раздел сайта для них закрыт.

В-четвертых, нам следует предусмотреть возможность для зарегистрированного пользователя, уже вошедшего на сайт, произвести *выход*. В результате этого сайт «забудет», что данный пользователь вошел на сайт, и будет считать его гостем со всеми вытекающими последствиями, разумеется, пока он не выполнит вход на сайт повторно. Процедура выхода выполняется особой Web-страницей, называемой *страницей выхода*.

Здесь описан простейший случай реализации разграничения доступа. В реальности же все несколько сложнее. Так, к параметрам каждого зарегистрированного пользователя добавляется список его *прав*, которые можно рассматривать как обозначения задач, которые ему разрешается выполнять. Например, пользователь может иметь отдельные права на работу со списком категорий и списком товаров, но не иметь права на работу со списком записей гостевой книги. Каждый пользователь может иметь произвольное количество различных прав.

Также пользователи часто объединяются в *группы*. Это делается для удобства управления пользователями — скажем, группе можно назначить определенные права и потом давать их пользователям, просто добавляя их в эту группу. Каждый пользователь может входить в произвольное количество групп.

Django включает в свой состав полноценную подсистему разграничения доступа, а именно: список пользователей, групп и прав, страницы для входа и выхода и программные инструменты для проверки, выполнил ли пользователь вход и имеет ли он необходимые права. Нам остается лишь ее задействовать, что совсем не сложно.

Настройка проекта для реализации разграничения доступа

Любой вновь созданный проект Django уже настроен таким образом, чтобы программист смог задействовать в нем подсистему разграничения доступа. В частности, в список активных уже включены приложения `django.contrib.auth`, собственно реализующее работу этой подсистемы, и `django.contrib.contenttypes`, обеспечивающее правильную обработку моделей. (О списке активных приложений рассказывалось в *главе 4*.)

Однако некоторые настройки нам все же лучше произвести. Они необязательны, но позволят нам впоследствии несколько уменьшить объем работ. Поэтому сразу же откроем модуль `settings` пакета проекта.

Переменная LOGIN_URL задает страницу входа либо в виде ее интернет-адреса, либо в виде имени соответствующей привязки (имя привязки указывается именованным параметром name функции url — подробнее см. в *главе 6*):

```
LOGIN_URL = "/login/"
```

Так мы задаем интернет-адрес страницы входа непосредственно.

```
LOGIN_URL = "login"
```

А здесь мы указываем этот интернет-адрес в виде имени привязки.

Переменная LOGOUT_URL задает страницу выхода также либо как ее интернет-адрес, либо как имя ее привязки:

```
LOGOUT_URL = "logout"
```

А переменная LOGIN_REDIRECT_URL указывает интернет-адрес или имя привязки страницы, на которую будет выполнено перенаправление после успешного входа, если страница перенаправления не была указана явно:

```
LOGIN_REDIRECT_URL = "/goods/"
```

Теперь после успешного входа посетитель будет перенаправлен на список товаров.

И не забудем сохранить модуль settings после всех изменений.

Список пользователей и групп

Чтобы получить доступ к списку пользователей и групп Django, мы зайдем на встроенный административный сайт, набрав интернет-адрес **http://localhost:8000/admin/**, и выполним процедуру входа. В общем, сделаем то же самое, что проделывали в *главе 4*, когда знакомились с административным сайтом.

На странице списка приложений, которая появится на экране после успешного входа, мы найдем таблицу с заголовком Auth. Эта таблица соответствует приложению django.contrib.auth, которое реализует работу подсистемы разграничения доступа. В самой таблице мы увидим гиперссылки **Группы** и **Пользователи**, ведущие на соответствующие списки.

Откроем список пользователей. В данный момент в нем присутствует лишь один пользователь admin, который создается самой Django при формировании нового проекта. Откроем сведения об этом пользователе, щелкнув на его имени.

В целом, здесь все понятно. Мы увидим поля ввода для указания имени пользователя, под которым он зарегистрирован в списке, его реального имени и фамилии, адреса электронной почты и значений даты и времени регистрации и последнего входа на сайт. И, разумеется, кнопки для сохранения сделанных изменений.

Мы также увидим три флажка, задающих состояние пользователя и его особые права (рис. 14.1):

- **Активный** — установка этого флажка делает пользователя активным. Так что, если нам потребуется временно закрыть какому-либо пользователю доступ

в административный раздел, удалять его необязательно — достаточно просто сбросить этот флажок;

- **Статус персонала** — установка его дает пользователю статус персонала сайта, т. е. возможность захода на встроенный административный сайт Django и получения рассылок о вновь добавленных комментариях (о комментировании будет рассказано в *главе 15*);
- **Статус суперпользователя** — установка его дает пользователю все доступные права без необходимости в явном их задании.

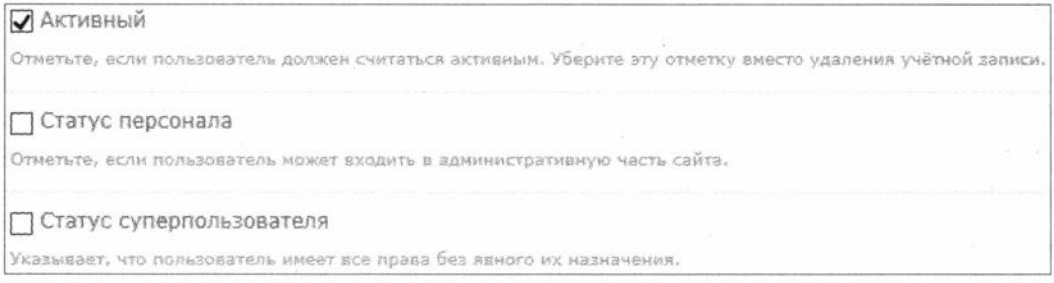


Рис. 14.1. Флажки, задающие состояние и особые права пользователя

Права суперпользователя

Права суперпользователя в оптимальном случае должен иметь лишь один пользователь из зарегистрированных на сайте. Обычно этим пользователем является разработчик или ведущий администратор сайта — только он может вносить правки во внутренние данные сайта без ограничения.

Самое интересное оставим напоследок. Это две пары списков, расположенные ниже перечисленных ранее флажков.

Начнем с нижней пары списков, надпись рядом с которыми гласит **Права пользователя** (рис. 14.2). Как уже понятно, с их помощью задаются права конкретного пользователя. В левом списке перечислены доступные права, а в правом — права, которыми обладает этот пользователь.

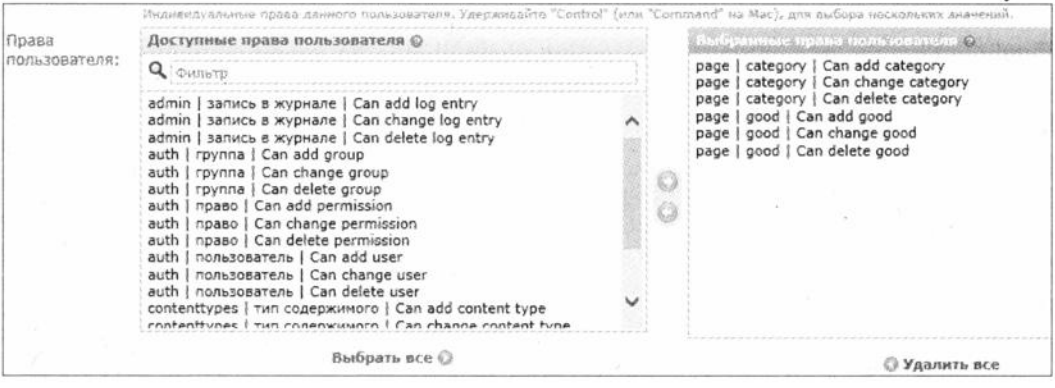


Рис. 14.2. Списки для задания прав пользователя

Дать пользователю какое-либо право несложно: выбираем в левом списке соответствующий ему пункт и нажимаем кнопку со стрелкой вправо, расположенную между списками. Чтобы выбрать сразу несколько прав, мы выберем первое, нажмем клавишу <Ctrl> и, удерживая ее, выберем остальные. А расположенная ниже кнопка **Выбрать все** позволит нам дать пользователю все права сразу.

Так же просто убрать какое-либо из данных пользователю ранее прав. Выбираем в правом списке соответствующий ему пункт (или пункты) и нажимаем расположенную между списками кнопку со стрелкой влево. А находящаяся под списком кнопка **Удалить все** удалит все права, сделав пользователя «бесправным».

Теперь посмотрим на собственно права. Django создает их сама, основываясь на наборе моделей, что присутствуют во входящих в проект приложениях. Для каждой такой модели создаются три права, позволяющие добавлять, изменять и удалять записи. Пункты, что им соответствуют, носят имена вида:

<имя приложения> | *<имя модели>* | Can *<действие>* *<имя модели>*

где действие: `add` — добавление, `change` — изменение и `delete` — удаление.

Следовательно, право с именем `page | category | Can add category` дает пользователю возможность добавления записей в модель `Category` приложения `page` (т. е. создания новых категорий товаров), а право `auth | пользователь | Can delete user` — право удаления записей из модели `User` приложения `Auth` (удаления зарегистрированных пользователей).

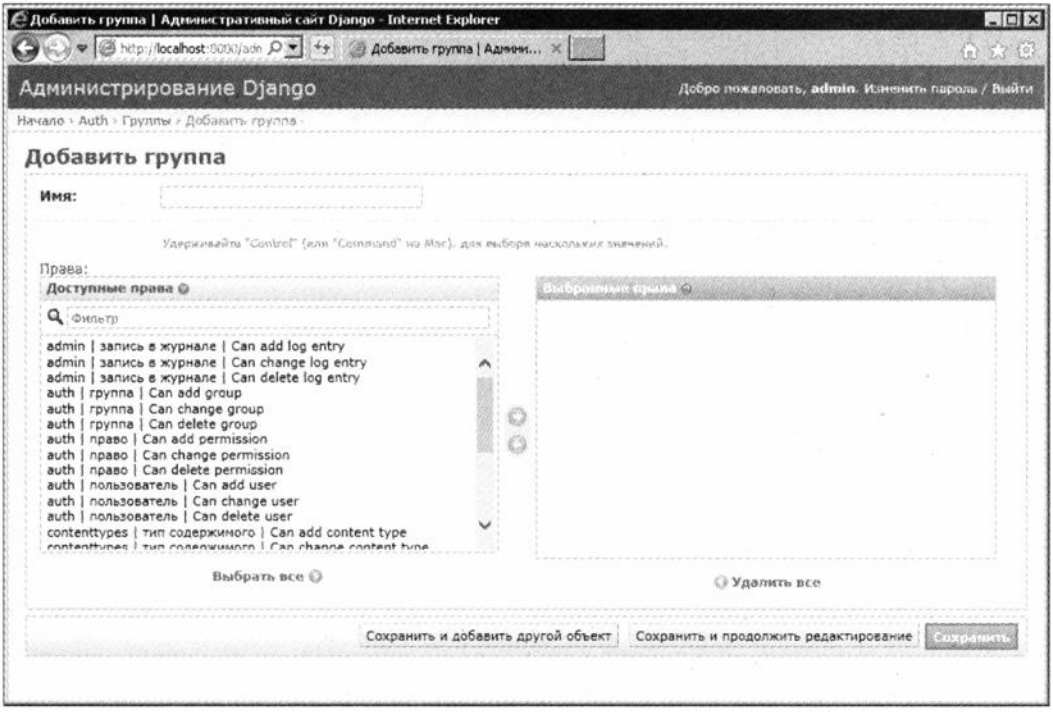


Рис. 14.3. Страница добавления группы

Мы можем достаточно тонко настроить набор действий, которые разрешено выполнять пользователю над данными, хранящимися в той или иной модели. Для этого нам достаточно всего лишь дать ему соответствующие права.

Аналогичная пара списков **Группы**, как уже понятно, служит для указания групп, в которых состоит пользователь. Изначально список групп пуст, и нам сначала придется его заполнить.

Что касается параметров группы, то их совсем немного. На рис. 14.3, где показана страница добавления группы, мы видим лишь ее имя и права.

Реализация входа на сайт

Перед тем как пользователь сможет получить доступ к административному разделу, он должен выполнить процедуру входа на сайт. Поэтому рассмотрение реализации разграничения доступа мы начнем именно с нее.

Django включает в свой состав уже готовую функцию-контроллер, реализующую вход на сайт. Эта функция сама выводит на экран страницу с формой входа, сама проверяет правильность введенных в форму данных, сама ищет в списке пользователей того, чьи имя и пароль указал пользователь, сама выполняет вход или, напротив, выводит сообщение о том, что такой пользователь в списке не найден.

Это функция `login`, объявленная в модуле `django.contrib.auth.views`. Поскольку она является полноценным контроллером, мы можем указать ее вызов прямо в списке привязок интернет-адресов. И сделать это следует в модуле `urls` пакета проекта — так мы создадим единую подсистему разграничения доступа для всего сайта:

```
. . .
urlpatterns = patterns('',
  . . .
  url(r'^login/', "django.contrib.auth.views.login", name = "login"),
  . . .
)
```

Здесь мы привязали нашу функцию-контроллер к интернет-адресу `login`. Мы задали для нее имя привязки `login` — это имя мы впоследствии используем в коде шаблонов. А имя самой функции-контроллера мы заключили в кавычки, как и все имена функций-контроллеров в этом случае (подробнее — в *главе 6*).

Функция-контроллер `django.contrib.auth.views.login` принимает два необязательных именованных параметра, которые могут быть нам полезны:

- `template_name` — задает путь к файлу шаблона страницы входа. Если он не указан, будет использован файл шаблона registration/login.html, расположенный либо в папке templates уровня приложения, либо в той же папке уровня проекта. (О шаблонах уровня приложения и проекта см. *главу 8*.)
- `extra_context` — словарь, содержащий данные, которые будут добавлены к контексту данных шаблона для страницы входа.

Указать значения необязательных параметров для функции-контроллера можно третьим неименованным параметром функции `url`, который является необязательным:

```
from page.models import Category
. . .
urlpatterns = patterns('',
  . . .
  url(r'^login/', "django.contrib.auth.views.login",↪
  {"template_name": "login.html",↪
  "extra_context": {"cats": Category.objects.all()}}, name = "login"),
  . . .
)
```

Здесь мы указываем в качестве шаблона файл login.html, находящийся непосредственно в папке шаблонов, и добавляем в его контекст данных список категорий.

Если посетитель сайта зашел непосредственно на страницу входа, набрав ее интернет-адрес, что мы задали в привязках, то после успешного входа будет выполнено перенаправление на страницу, чей интернет-адрес указан в переменной `LOGIN_REDIRECT_URL` настроек проекта (см. ранее). Если же пользователь, не выполнив процедуру входа, попытается зайти на какую-либо из закрытых страниц административного раздела, Django предложит ему войти на сайт, перенаправив на страницу входа. Когда пользователь успешно войдет на сайт, он будет автоматически перенаправлен на страницу, на которую до этого пытался попасть.

Функция-контроллер `django.contrib.auth.views.login` передает в составе контекста данных в шаблон следующие переменные, которые могут нам пригодиться:

- `form` — форма входа;
- `next` — интернет-адрес страницы административного раздела, на которую будет выполнено перенаправление после успешного входа. Если посетитель зашел непосредственно на страницу входа, данная переменная будет содержать значение `None`.

Разумеется, в контекст данных будет добавлено также содержимое словаря, указанного нами в качестве значения параметра `extra_context` в функции `login`.

В коде формы нам, помимо кнопки отправки данных, необходимо создать скрытое поле с именем `next` и дать ему в качестве значения содержимое переменной контекста шаблона `next`.

Вот полный код шаблона страницы входа, который мы можем использовать на своем сайте:

```
{% extends "base.html" %}
{% load static %}
{% block title %}Вход{% endblock %}
{% block main %}
  <h2>Вход</h2>
  <form action="" method="post">
```

```
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Войти">
    <input type="hidden" name="next" value="{{ next }}">
  </form>
{% endblock %}
```

На этом реализация входа на сайт закончена.

Реализация разграничения доступа

Теперь нам нужно реализовать проверку, выполнил ли пользователь вход на сайт и достаточно ли у него прав для открытия страницы административного раздела.

Проверка, выполнил ли пользователь вход на сайт

Во многих случаях требуется всего лишь проверить, выполнил ли пользователь успешный вход на сайт. В таком случае мы можем использовать декоратор `login_required`, объявленный в модуле `django.contrib.auth.decorators`.

Этот декоратор проверяет, выполнил ли посетитель вход на сайт, и только после этого разрешает ему зайти на соответствующую контроллеру страницу. Если же пользователь не выполнил процедуру входа, Django перенаправит его на страницу входа, чей интернет-адрес указан в переменной `LOGIN_URL` настроек проекта. При этом функции-контроллеру `login` (см. ранее) будет передан интернет-адрес страницы, на которую пытался войти посетитель (в нашем случае это страница добавления товара), для чего будет использован GET-параметр с именем `next`.

В случае использования классов-контроллеров мы укажем этот декоратор прямо в соответствующей привязке интернет-адреса — в модуле `urls`. Сам вызов декоратора ставится во втором параметре функции `url`, а вызов метода `as_view` класса-контроллера задается единственным его параметром (за подробностями о классах-контроллерах и их методе `as_view` — к *главе 10*):

```
from django.contrib.auth.decorators import login_required
. . .
urlpatterns = patterns('',
  . . .
  url(r'^(?P<cat_id>\d+)/add/$', login_required(GoodCreate.as_view()),↵
  name = "good_add"),
  . . .
)
```

Здесь мы указали, что для вызова класса-контроллера добавления нового товара в список пользователь должен выполнить вход на сайт.

Если же мы используем функции-контроллеры, то все еще проще. Мы вставим вызов декоратора `login_required` перед первой строкой объявления нужной функции:

```
from django.contrib.auth.decorators import login_required
. . .
@login_required
def good_create(request, cat_id):
  . . .
```

Проверка, имеет ли пользователь необходимые права

Несколько сложнее проверить, имеет ли пользователь права, необходимые для выполнения той или иной операции. Для этого понадобится выяснить имя соответствующего права.

Еще в начале этой главы мы узнали, что Django сама создает права, основываясь на всех входящих в состав приложений проекта моделях. Эти права имеют внутренние имена вида:

<имя приложения>.<действие>_<имя модели>

где *действие*: `add` — добавление, `change` — изменение и `delete` — удаление.

Например, право `page.add_good` дает пользователю возможность добавлять товары в список, а право `page.delete_category` — право удалять категории.

Воспользуемся декоратором `permission_required`, также объявленным в модуле `django.contrib.auth.decorators`. Он проверяет, выполнил ли пользователь вход на сайт и имеет ли он право, внутреннее имя которого указано единственным параметром этого декоратора в виде строки. В остальном он работает так же, как уже знакомый нам декоратор `login_required`.

Разработчики, применяющие классы-контроллеры, вставят вызов этого декоратора также в привязках интернет-адресов — непосредственно перед вызовом метода `as_view` класса-контроллера, а сам вызов этого метода заключат в круглые скобки:

```
from django.contrib.auth.decorators import permission_required
. . .
urlpatterns = patterns('',
  . . .
  url(r'^(?P<cat_id>\d+)/add/$', ↳
  permission_required("page.add_good")↳
  (GoodCreate.as_view()), name = "good_add"),
  url(r'^good/(?P<good_id>\d+)/edit/$',↳
  permission_required("page.change_good")↳
  (GoodUpdate.as_view()), name = "good_edit"),
  url(r'^good/(?P<good_id>\d+)/delete/$',↳
  permission_required("page.delete_good")↳
  (GoodDelete.as_view()), name = "good_delete"),
)
```

Здесь мы указываем, что для выполнения операций добавления, изменения и удаления товара пользователь должен иметь соответствующие права.

Поклонники функций-контроллеров просто вставят вызов декоратора перед первой строкой объявления функции:

```
from django.contrib.auth.decorators import permission_required
. . .
@permission_required("page.add_good")
def good_create(request, cat_id):
  . . .
```

Более сложные случаи проверки

В некоторых случаях нам требуется выполнить более сложную проверку — например, имеет ли пользователь сразу несколько прав, необходимых для выполнения какого-либо действия. Такого рода проверки следует выполнять непосредственно в коде контроллеров.

Объект класса `HttpRequest`, хранящий сведения о запросе и получаемый в качестве параметра любой функцией-контроллером и методами `get` и `post` классов-контроллеров (см. *главы 6* и *10*), поддерживает свойство `user`. Это свойство хранит текущего пользователя в виде объекта класса `User`, объявленного в модуле `django.contrib.auth.models`, и неважно, является ли пользователь зарегистрированным или простым гостем (как мы уже успели заметить, фактически класс `User` — это модель, и потом мы этим воспользуемся):

```
class GoodCreate(TemplateView):
  . . .
  def get(self, request, *args, **kwargs):
    current_user = request.user
    . . .
```

Не принимающий параметров метод `is_authenticated` класса `User` возвращает `True`, если пользователь выполнил успешный вход:

```
def get(self, request, *args, **kwargs):
  if request.user.is_authenticated():
    . . .
  else:
    return redirect("login")
```

Здесь мы проверяем, выполнил ли пользователь вход, и, если не выполнил, перенаправляем его на страницу входа.

Да-да, нам придется самим перенаправлять посетителя на страницу входа! Django за нас этого не сделает.

Чтобы передать функции-контроллеру `login` интернет-адрес страницы, на которую должно быть выполнено перенаправление после успешного входа, мы применим GET-параметр `next`. Интернет-адрес текущей страницы хранится в свойстве `path` объекта сведений о запросе:

```
def get(self, request, *args, **kwargs):
  if request.user.is_authenticated():
    . . .
  else:
    return redirect("/login/?next=" + request.path)
```

Здесь мы указываем функции-контроллеру `login` (см. ранее в этой главе) после входа выполнить возврат на нужную страницу.

Класс `User` поддерживает еще два полезных для нас метода. Они позволят нам проверить, имеет ли пользователь определенные права.

- Метод `has_perm` принимает в качестве параметра строку с внутренним именем права и возвращает `True`, если пользователь им обладает. Причем в этом случае проверяются не только права непосредственно пользователя, но и все права, которыми обладают все группы, членом которых является пользователь:

```
def get(self, request, *args, **kwargs):
  if request.user.is_authenticated():
    if request.user.has_perm("page.add_good"):
      . . .
    else:
      return redirect("/login/?next=" + request.path)
  else:
    return redirect("/login/?next=" + request.path)
```

  Здесь мы дополнительно проверяем, имеет ли пользователь, выполнивший вход на сайт, право на добавление товаров.

- Метод `has_perms` работает так же, но в качестве параметра принимает список строк с наименованиями прав и проверяет, имеет ли пользователь все эти права:

```
def get(self, request, *args, **kwargs):
  if request.user.is_authenticated():
    if request.user.has_perms(["page.add_good", "page.change_good",↵
    "page.delete_good"]):
      . . .
    else:
      return redirect("/login/?next=" + request.path)
  else:
    return redirect("/login/?next=" + request.path)
```

  Здесь мы дополнительно проверяем, имеет ли пользователь права на добавление, правку и удаление товаров.

Выполнение проверки в шаблонах

Хорошим тоном Web-программирования считается вывод элементов управления (кнопок или гиперссылок) для перехода на страницы административного раздела только в том случае, если пользователь, во-первых, выполнил вход на сайт, а во-вторых, имеет все необходимые права. Как это реализовать? Очень просто. Django

в любом случае неявно создает в контексте данных шаблона переменные user и perms, хранящие, соответственно, текущего пользователя и его права.

Для проверки, выполнил ли пользователь вход на сайт, мы применим знакомый метод is_authenticated. Поскольку он не принимает параметров, мы можем вызвать его в коде шаблона:

```
{% if user.is_authenticated %}
  <td><a href="{% url "good_edit" good_id=good.id %}">Изменить</a></td>
  <td><a href="{% url "good_delete" good_id=good.id %}">Удалить</a></td>
{% endif %}
```

Здесь мы выводим столбцы списка товаров с гиперссылками **Изменить** и **Удалить**, только если пользователь выполнил успешный вход.

Значение переменной perms представляет собой объект, поддерживающий набор свойств с именами вида:

<имя приложения>.<действие>_<имя модели>

где *действие*: add — добавление, change — изменение и delete — удаление. Каждое из этих свойств хранит значение True, если пользователь обладает соответствующим правом:

```
{% if perms.page.add_good %}
  <p><a href="{% url "good_add" cat_id=category.id %}">Добавить
  товар</a></p>
{% endif %}
```

Здесь мы выводим гиперссылку **Добавить товар** только в том случае, если пользователь имеет право на добавление товара.

Реализация выхода с сайта

По окончании работы с административным разделом сайта пользователь должен произвести выход. Реализовать операцию выхода также несложно, поскольку Django предоставляет функцию-контроллер logout, объявленную в модуле django.contrib.auth.views. Она выполняет процедуру выхода и выводит на экран соответствующую страницу.

Вызов этой функции также указывается в списке привязок в модуле urls пакета проекта (в результате на нашем сайте будет единая страница выхода):

```
. . .
urlpatterns = patterns('',
  . . .
  url(r'^logout/', "django.contrib.auth.views.logout", name = "logout"),
  . . .
)
```

Мы привязали эту функцию к интернет-адресу logout и задали для нее имя привязки logout.

Функция-контроллер `django.contrib.auth.views.logout` принимает три необязательных именованных параметра, которые могут нам пригодиться:

- `template_name` — задает путь к файлу шаблона страницы выхода. Если он не указан, будет использован файл шаблона registration/logged_out.html.
- `next_page` — задает интернет-адрес или имя привязки страницы, на которую будет выполнено перенаправление после успешного выхода. Если параметр не указан, перенаправление выполняться не будет, и посетитель останется на странице выхода;
- `extra_context` — словарь с данными, которые будут добавлены к контексту данных шаблона для страницы выхода.

Вот пример:

```
from page.models import Category
. . .
urlpatterns = patterns('',
  . . .
  url(r'^logout/', "django.contrib.auth.views.logout",↵
  {"template_name": "logout.html",↵
  "extra_context": {"cats": Category.objects.all()}}, name = "logout"),
  . . .
)
```

Здесь мы указываем в качестве шаблона файл logout.html, находящийся непосредственно в папке шаблонов, и добавляем в его контекст данных список категорий.

В контексте данных для шаблона страницы выхода будет создана переменная `title`. Она хранит строку сообщения об успешном выходе, но в реальности это сообщение вида «Не авторизован», для нас совершенно бесполезное. Также в контекст данных будет добавлено содержимое словаря, указанного нами в параметре `extra_context`.

Вот полный код шаблона, который мы можем использовать для создания более информативной страницы выхода:

```
{% extends "base.html" %}
{% load static %}
{% block title %}Выход{% endblock %}
{% block main %}
  <h2>Выход</h2>
  <p>Вы успешно выполнили процедуру выхода с сайта.</p>
  <p><a href="{% url "index" %}">На список товаров</a></p>
{% endblock %}
```

Django также предлагает функцию-контроллер `logout_then_login`, объявленную в том же модуле `django.contrib.auth.views`. Она выполняет процедуру выхода и сразу же перенаправляет посетителя на страницу входа:

```
. . .
urlpatterns = patterns('',
  . . .
```

```
  url(r'^logout/', "django.contrib.auth.views.logout_then_login",↵
  name = "logout"),
  . . .
)
```

Из дополнительных параметров, принимаемых этой функцией, нам пригодится лишь уже знакомый нам `extra_context`.

Создание дополнительных прав

Помимо прав, созданных самой Django на основе входящих в состав приложений проекта моделей, мы можем создать для какой-либо из моделей свои собственные, дополнительные права. Это может понадобиться в сложных приложениях, выполняющих специфические действия.

Права объявляются прямо в нужной модели, во вложенном классе `Meta` (за подробностями об этом классе — к *главе 5*). Для этого применяется свойство `permissions` данного класса. Его значением должен быть обычный список или кортеж Python, каждый элемент которого объявляет одно право и должен представлять собой кортеж из двух элементов-строк, задающих, соответственно, внутреннее имя права и его описание.

Правила Django-программирования требуют, чтобы наименование каждого права начиналось с символов `"can_"` и описывало сущность права максимально точно. Описание права будет выводиться в списках прав на страницах добавления и правки пользователей встроенного административного сайта (см. рис. 14.1):

```
class Blog(models.Model):
  . . .
  class Meta:
    permissions = (("can_blog", "Ведение блога"),)
```

Здесь мы создали в модели `Blog` новое право `can_blog`, которое дает пользователю возможность вести свой блог.

Объявив права, мы выполним синхронизацию моделей с базой данных (как это делается, описывалось в *главе 4*). После этого Django сформирует в таблице прав все необходимые записи.

Проверка на наличие у пользователя созданных нами прав выполняется так же, как и в случае прав, сформированных автоматически. Полное наименование дополнительного права формируется по схеме *<имя приложения>.<заданное нами внутреннее имя права>*:

```
urlpatterns = patterns('',
  . . .
  url(r'^blog/$', permission_required("blog.can_blog")↵
  (BlogView.as_view()), name = "blog"),
  . . .
)
```

Получение сведений о пользователе

Если пользователь выполнил успешный вход на сайт, мы должны как-то сообщить ему об этом. Обычно в таких случаях где-либо на Web-страницах выводится имя, под которым он был зарегистрирован на сайте, или даже реальные имя и фамилия.

Самого пользователя, точнее, представляющий его объект класса `User`, мы можем получить из свойства `user` объекта сведений о запросе. Но как получить сведения о нем? С помощью набора методов класса `User`, перечисленных в табл. 14.1. Все эти методы не принимают параметров, поэтому могут быть использованы в шаблонах.

***Таблица 14.1.** Методы класса `User`*

Метод	Описание
`get_username`	Возвращает имя, под которым пользователь был зарегистрирован на сайте («логин»)
`get_short_name`	Возвращает реальную фамилию
`get_full_name`	Возвращает строку из реальных имени и фамилии, разделенных пробелом
`is_anonymous`	Возвращает `True`, если это гость

Вот пример возвращения строки из реальных имени и фамилии:

```
{% if user.is_authenticated %}
  <p>Добро пожаловать, {{ user.get_full_name }}!</p>
{% endif %}
```

Помимо этих методов, класс `User` поддерживает четыре полезных свойства:

- `is_active` — `True`, если пользователь помечен как активный;
- `is_staff` — `True`, если пользователь помечен как входящий в состав персонала сайта, т. е. имеет возможность входа на встроенный административный сайт;
- `is_superuser` — `True`, если пользователь помечен как суперпользователь, т. е. имеющий максимальные права без явного их указания;
- `email` — адрес электронной почты пользователя.

Использование модели *User*

Ранее мы говорили, что класс `User`, объявленный в модуле `django.contrib.auth.models` и предназначенный для хранения данных о зарегистрированном пользователе, на самом деле представляет собой модель. Так что, если нам понадобится привязать какие-либо сущности к определенному пользователю, мы с легкостью сделаем это, создав хорошо знакомую нам по *главе 5* связь между моделями:

```
from django.contrib.auth.models import User
class Blog(models.Model):
  . . .
  user = models.ForeignKey(User)
```

Так мы создаем связь между моделью `Blog`, хранящей статьи блога, и моделью `User`. Благодаря этому мы теперь сможем привязать каждую статью блога к конкретному пользователю:

```
class BlogUpdate(TemplateView):
  . . .
  def get(self, request, *args, **kwargs):
    blog = Blog.objects.get(pk = self.kwargs["blog_id"])
    if blog.user == request.user:
      . . .
    else:
      return redirect("login")
```

Здесь мы проверяем, тот ли пользователь, что в данный момент вошел на сайт, создал эту статью, перед тем как запустить процесс ее правки.

Низкоуровневые средства для реализации входа и выхода

Хотя Django предусматривает очень удобные встроенные функции-контроллеры `login` и `logout`, выполняющие процедуры входа и выхода соответственно, в некоторых случаях они могут оказаться бесполезными. Это может случиться, например, когда мы хотим поместить форму для входа прямо на главной странице сайта (такое часто встречается на сайтах интернет-магазинов).

В такой ситуации мы воспользуемся низкоуровневыми средствами для выполнения входа и выхода, предлагаемыми Django. Они включают в себя три функции, объявленные в модуле `django.contrib.auth`.

Сначала мы создадим форму входа:

```
from django import forms
class LoginForm(forms.Form):
  username = forms.CharField(label = "Имя")
  password = forms.CharField(widget = forms.PasswordInput,↵
  label = "Пароль")
```

Отметим, что для поля пароля этой формы мы указали в качестве элемента управления поле для ввода пароля (`PasswordInput`). Подробно об элементах управления и назначения их полям формы говорилось в *главе 12*.

После этого можно приступать к созданию контроллера, выполняющего вход. В роли такого контроллера может выступать как класс-контроллер, так и функция-контроллер.

Сначала вызовем первую из трех функций, о которых говорилось ранее, — `authenticate`. Она принимает параметры `username` и `password`, задающие имя поль-

зователя и его пароль, и возвращает объект класса `User`, представляющий пользователя с такими именем и паролем, или `None`, если подходящего пользователя в списке нет.

Получив пользователя с указанными именем и паролем, мы проверим, является ли он активным. Для этого, как мы помним, достаточно проверить значение свойства `is_active`.

Если пользователь активен, мы вызовем функцию номер два — `login` (не путать с функцией-контроллером с тем же именем!), которая, собственно, и выполнит процедуру входа. В качестве параметров мы передадим ей объект сведений о запросе и объект, хранящий сведения о пользователе. Результата эта функция не возвращает.

Вот полный код класса-контроллера, который может быть использован для выполнения входа:

```
from django.contrib.auth import authenticate, login
from django.views.generic.base import TemplateView
from django.shortcuts import redirect
class LoginView(TemplateView):
  form = None
  template_name = "template_login.html"
  def get(self, request, *args, **kwargs):
    self.form = LoginForm()
    return super(LoginView, self).get(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(LoginView, self).get_context_data(**kwargs)
    context["form"] = self.form
    return context
  def post(self, request, *args, **kwargs):
    self.form = LoginForm(request.POST)
    if self.form.is_valid():
      user = authenticate(username = self.form.cleaned_data["username"],↵
      password = self.form.cleaned_data["password"])
      if user is not None:
        if user.is_active:
          login(request, user)
          return redirect("index")
        else:
          return super(LoginView, self).get(request, *args, **kwargs)
      else:
        return super(LoginView, self).get(request, *args, **kwargs)
    else:
      return super(LoginView, self).get(request, *args, **kwargs)
```

Что же касается выхода, то он выполняется вызовом функции номер три — `logout` (опять же, не путать ее с одноименной функцией-контроллером!). В качестве един-

ственного параметра она принимает объект сведений о запросе и не возвращает результата.

Вот полный код класса-контроллера, выполняющего выход:

```
from django.contrib.auth import logout
from django.views.generic.base import TemplateView
class LogoutView(TemplateView):
  template_name = "template_logout.html"
  def get(self, request, *args, **kwargs):
    logout(request)
    return super(LogoutView, self).get(request, *args, **kwargs)
```

Вероятно, здесь будет проще использовать функцию-контроллер. Впрочем, написать ее несложно.

Что дальше?

В этой главе мы учились реализовывать на нашем сайте разграничение доступа. Мы познакомились со встроенным списком пользователей Django, высокоуровневыми функциями-контроллерами, выполняющими вход и выход, и средствами для проверки, выполнил ли пользователь вход на сайт и имеет ли он необходимые права. Наконец, мы узнали, как можно выполнить вход и выход средствами низкого уровня, если высокоуровневые функции-контроллеры нам по ряду причин не подходят.

Следующая глава будет посвящена встроенным в Django инструментам комментирования. Эти инструменты настолько мощны и гибки, что с их помощью мы можем реализовать комментирование чего угодно: товаров, статей блога, категорий, изображений и многого другого.

ГЛАВА 15

Комментарии Django

В предыдущей главе мы занимались реализацией разграничения доступа на сайте. Мы познакомились со списком пользователей Django, функциями-контроллерами для выполнения входа и выхода и инструментами проверки факта входа пользователя на сайт и наличия у него необходимых прав. Теперь ни один злоумышленник не зайдет в административный раздел нашего сайта и не испортит его внутренние данные!

Многие сайты предоставляют посетителям возможность комментирования опубликованных на них статей, изображений, файлов, товаров и данных иного рода. Было бы неплохо реализовать нечто подобное и у нас. Что, сделаем?

Тем более, что нам почти ничего не придется делать самим. В составе Django поставляется развитая подсистема комментирования, которую нам остается лишь активизировать.

Настройка проекта для реализации комментирования

В отличие от подсистемы разграничения доступа (см. *главу 14*), подсистема комментирования изначально в проекте не активизируется. Так что нам придется внести в настройки проекта нужные правки.

Откроем модуль `settings` пакета проекта, где указываются его настройки. Сразу же найдем переменную `INSTALLED_APPS`, в которой задается список активных приложений (подробнее — в *главе 4*), и внесем в этот список приложения `django.contrib.sites` и `django.contrib.comments`:

```
INSTALLED_APPS = (
  . . .
  'django.contrib.sites',
  'django.contrib.comments',
)
```

> ***О ПРИЛОЖЕНИИ*** `DJANGO.CONTRIB.SITES`
>
> Приложение `django.contrib.sites` служит для обеспечения работы нескольких Web-сайтов на одной копии Django. В данном случае оно нам не нужно, однако требуется для успешной работы подсистемы комментирования.

Для нормальной работы приложения `django.contrib.sites` нужно также задать идентификатор текущего сайта. Он задается в виде целого числа в переменной `SITE_ID`. Идентификатор сайта может быть любым, а для единственного сайта обычно задают 1:

```
SITE_ID = 1
```

Мы можем задать максимальную длину комментария. Она указывается в переменной `COMMENT_MAX_LENGTH` в виде целого числа и измеряется в символах. Комментарии, чья длина превысит указанное нами значение, будут отвергнуты. Значение этого параметра по умолчанию — 3000 символов:

```
COMMENT_MAX_LENGTH = 1024
```

Также мы можем указать, должны ли комментарии, помеченные как удаленные, присутствовать в списке. (В этом случае нам самим придется проверять, помечен ли комментарий как удаленный, и, если так, выводить соответствующую надпись. Подробнее об этом будет рассказано далее в этой главе.) Указать этот признак мы можем с помощью переменной `COMMENTS_HIDE_REMOVED`: значение `True` предписывает не включать удаленные комментарии в список (это, кстати, поведение по умолчанию), а `False` — включать:

```
COMMENTS_HIDE_REMOVED = False
```

Задав все необходимые настройки, выполним синхронизацию с базой данных. В результате Django создаст в базе все необходимые таблицы.

Осталось лишь открыть модуль `urls` пакета проекта и вставить в список привязок следующую строку:

```
urlpatterns = patterns('',
  . . .
  url(r'^comments/', include("django.contrib.comments.urls")),
)
```

Она привязывает на уровне проекта приложение `django.contrib.comments` к виртуальной папке comments. (Разумеется, имя этой папки может быть другим.)

Набор привязок, объявленный в модуле `django.contrib.comments.urls`, включает ряд служебных страниц: страницу с сообщением об успешном добавлении комментария, страницу для удаления комментария и пр. Эти служебные страницы крайне неудобны в работе, в частности, из-за того, что мы не можем вывести на них произвольные данные (тот же список категорий), поскольку выводящие их контроллеры не предоставляют возможности добавления данных в контекст шаблона. Поэтому в дальнейшем мы либо вообще не будем пользоваться этими страницами, либо сделаем все, чтобы они вообще не выводились на экран.

Однако, к сожалению, указанная привязка необходима для нормальной работы подсистемы комментирования. Так что нам в любом случае придется ее ввести.

Как работает подсистема комментирования Django?

Настала пора немного передохнуть и ознакомиться с теорией. А именно, поговорить о том, как работает подсистема комментирования, встроенная в Django.

Сразу следует уяснить, что каждый комментарий привязывается к определенной записи модели. Модель может быть любой — как написанной нами самими, так и принадлежащей любому из встроенных приложений Django (например, пользователю или группе).

Комментарий включает в себя имя пользователя, его адрес электронной почты, интернет-адрес его сайта и собственно содержимое комментария. Интернет-адрес сайта является необязательным, остальные параметры обязательны к вводу.

> ***Служебные данные в составе комментария***
>
> Помимо данных, вводимых посетителем, в составе комментария также сохраняются некоторые служебные данные — в частности, IP-адрес компьютера, с которого был добавлен комментарий, и дата и время его создания. Эти сведения обычно не выводятся на страницах сайта и требуются лишь модераторам — так, IP-адрес может пригодиться при отслеживании спамеров и интернет-хулиганов.

Комментарии могут оставлять как зарегистрированные пользователи, уже выполнившие вход на сайт (в этом случае в комментарий подставляются его имя и адрес электронной почты), так и гости. Если мы хотим ограничить комментирование только зарегистрированными пользователями, нам придется позаботиться об этом самим.

Изначально все комментарии помечаются как видимые на странице. Однако мы можем создать класс-автомодератор, который помечает все комментарии, оставленные после определенной даты, как скрытые. (Как создавать автомодераторы, мы узнаем ближе к концу этой главы.) Скрыть комментарий вручную может и модератор-человек, хотя в таком случае удобнее будет пометить его как удаленный.

> ***Скрытые комментарии на экран не выводятся***
>
> Все комментарии, помеченные как скрытые, не будут включаться в список комментариев и, соответственно, выводиться на экран. Изменить это поведение Django мы не сможем.

Комментарий может быть помечен как удаленный. Реально он все еще останется в списке комментариев, но не будет выводиться на странице (если переменной `COMMENTS_HIDE_REMOVED` в модуле `settings` было присвоено значение `True`) или вместо него будет выводиться сообщение вида «Комментарий удален» (если для этой переменной мы указали значение `False`).

Кроме того, модератор может вообще убрать комментарий, удалив из модели, где хранится список комментариев, соответствующую ему запись. Но это крайняя

мера, прибегать к которой следует только в особых случаях (в основном, во время отладки сайта).

Базовые средства для реализации комментирования

Самое интересное, что для успешного создания системы комментирования на нашем сайте нам не придется вносить никаких изменений в код моделей и контроллеров. Достаточно лишь вставить несколько тегов в код шаблонов.

Сначала нам нужно загрузить модуль `comments` шаблонизатора, обрабатывающий теги, что выводят содержимое комментариев и форму для их ввода. Для чего достаточно вставить в код шаблона следующий тег:

```
{% load comments %}
```

Этот тег должен находиться перед любым тегом подсистемы комментирования, которые мы сейчас и рассмотрим.

Вывод стандартной формы для комментирования

Чтобы вывести в нужном месте Web-страницы стандартную форму добавления комментария, достаточно поместить в то место страницы, где она должна присутствовать, один-единственный тег `render_comment_form`:

```
{% render_comment_form for <объект> %}
```

Здесь *объект* — это переменная контекста шаблона, где хранится объект, к которому будут привязываться добавляемые комментарии (товар, статья блога и др.).

Так, чтобы реализовать возможность добавления комментариев к товару, мы можем вставить в код шаблона сведений о товаре (он хранится в файле good.html) вот такой тег (выделен полужирным шрифтом):

```
{% block main %}
  . . .
  <p><a href="{% url "index" cat_id=good.category.id %}?page={{ pn }}">↵
  Назад</a></p>
  <p> </p>
  {% render_comment_form for good %}
{% endblock %}
```

Объект, представляющий товар, сведения о котором выводятся на странице, хранится у нас в переменной контекста шаблона `good`.

Стандартная форма для добавления комментариев показана на рис. 15.1. Выглядит она довольно непрезентабельно, но мы потом это исправим.

После нажатия кнопки **Опубликовать** Django выведет стандартную страницу с сообщением об успешно добавленном комментарии. Это одна из тех стандартных страниц подсистемы комментирования, о которых мы упоминали в разделе, посвященном настройкам проекта.

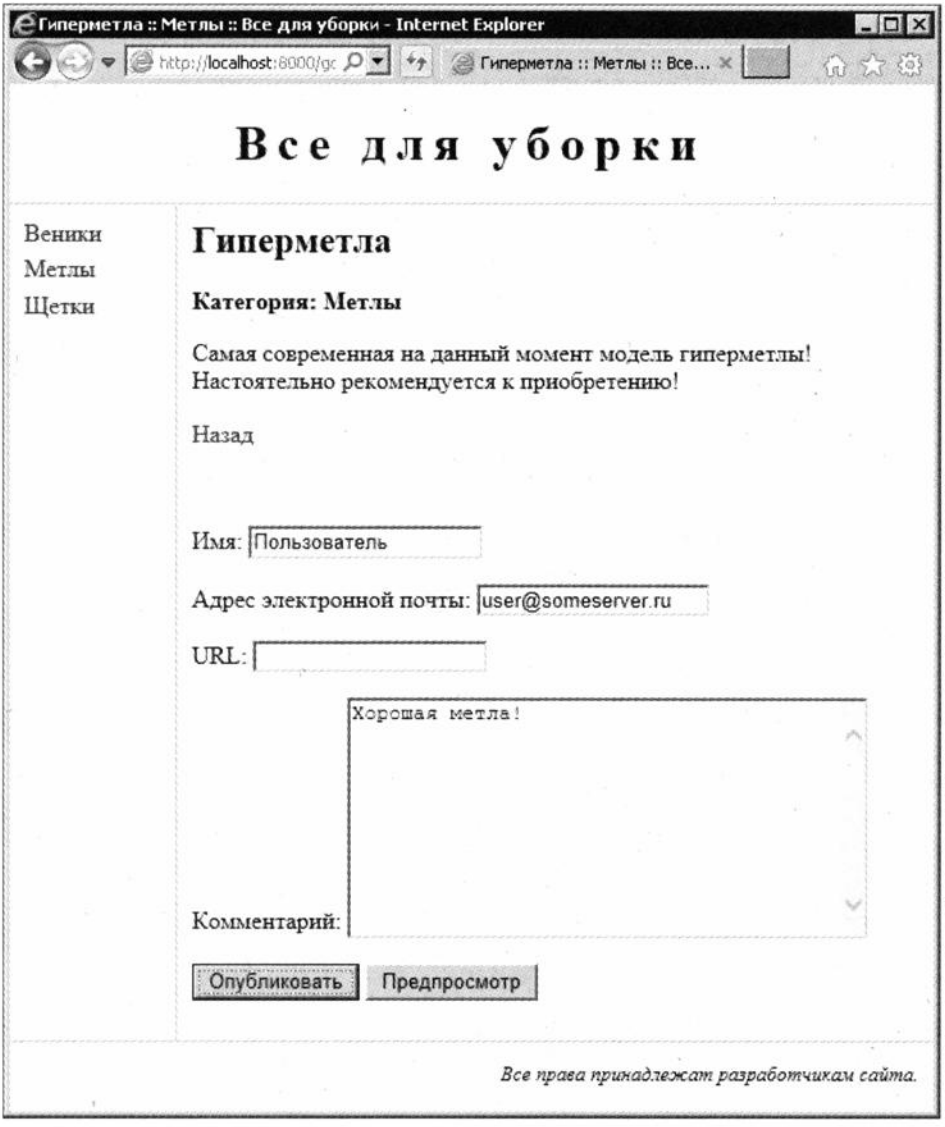


Рис. 15.1. Страница сведений
о товаре со стандартной формой для добавления комментария

Как уже говорилось, контроллер, который выводит эту страницу, не предусматривает возможности добавления в контекст данных шаблона произвольных значений. Так что вывести на ней список категорий товаров мы не можем. Кроме того, на этой странице не будет гиперссылки для возврата на предыдущую страницу, поэтому, чтобы вернуться к сведениям о товаре, нам придется нажать кнопку перехода на предыдущую страницу Web-обозревателя.

Неплохо было бы сделать так, чтобы после добавления комментария выполнялся повторный вывод страницы со сведениями о товаре. Совсем скоро мы узнаем, как это сделать.

Вывод стандартного списка комментариев

Для вывода самого списка комментариев в стандартном формате служит другой тег — `render_comment_list`:

```
{% render_comment_list for <объект> %}
```

Комментарии будут выведены в том месте страницы, где находится этот тег.

Часто бывает полезным получить количество комментариев, привязанных к тому или иному объекту, чтобы потом вывести его на странице. Для этого применяется тег `get_comment_count`:

```
{% get_comment_count for <объект> as <переменная> %}
```

Количество комментариев будет помещено в указанный нами параметр *переменная* контекста шаблона. Нам останется только вывести его значение известным нам по *главе 7* способом.

Исправим код шаблона страницы сведений о товаре таким образом, чтобы на ней выводились количество комментариев и их список (добавленный код выделен полужирным шрифтом):

```
{% block main %}
  . . .
  <p><a href="{% url "index" cat_id=good.category.id %}?page={{ pn }}">
  Назад</a></p>
  <p> </p>
  {% get_comment_count for good as good_comments_count %}
  <p>Всего комментариев: {{ good_comments_count }}.</p>
  {% render_comment_list for good %}
  <p> </p>
  {% render_comment_form for good %}
{% endblock %}
```

Результат наших действий показан на рис. 15.2. Да, как и форма, список комментариев не блещет изысками форматирования, но это лишь пока...

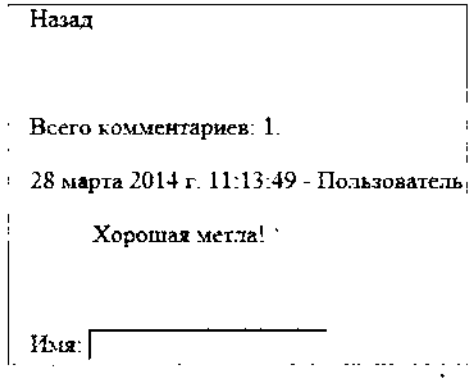


Рис. 15.2. Фрагмент страницы сведений о товаре со списком комментариев

Управление выводом списка комментариев и формы комментирования

Как мы только что убедились, и форма для добавления комментария, и список комментариев в их стандартном виде выглядят не лучшим образом и, скорее всего, не подойдут к дизайну нашего сайта. Поэтому нам следует отформатировать их надлежащим образом.

Управление выводом списка комментариев

Вывести список комментариев в том формате, который нам нужен, несложно — все, что нам для этого потребуется, мы уже знаем из *глав 7* и *8*. Осталось лишь выяснить, как получить список комментариев и отдельные составные части комментария: имя пользователя, его адрес электронной почты, само содержимое комментария и др.

Чтобы получить список комментариев, мы воспользуемся тегом `get comment list`:

```
{% get_comment_list for <объект> as <переменная> %}
```

После чего список комментариев, привязанных к *объекту*, окажется в заданном нами параметре *переменная*:

```
{% get_comment_list for good as good_comments %}
```

Так мы получаем список комментариев, привязанных к товару.

Каждый комментарий представляет собой объект класса `Comment`, объявленного в модуле `django.contrib.comments.models`. (Отметим, что этот класс представляет собой модель.) Класс `Comment` поддерживает несколько свойств, позволяющих получить сведения о комментарии:

- `content_object` — сущность, для которой был оставлен комментарий, фактически — запись соответствующей модели (например, товар или статья блога);
- `content_type` — тип этой сущности (подробности — далее);
- `user_name` — имя пользователя, оставившего комментарий;
- `user_email` — его адрес электронной почты;
- `user_url` — интернет-адрес его сайта;
- `comment` — собственно содержимое комментария;
- `submit_date` — дата отправки комментария;
- `is_removed` — `True`, если комментарий был помечен модератором как удаленный;
- `ip_address` — IP-адрес, с которого был отправлен комментарий (пригодится для отслеживания спамеров).

Что касается свойства `content type`, то оно хранит объект класса `ContentType`, описывающий сущность — запись модели, для которой был оставлен комментарий.

Этот класс поддерживает следующие свойства:

- `model` — имя класса модели, набранное строчными буквами;
- `name` — наименование модели, указанное в свойстве `verbose_name` вложенного класса `Meta` (т. е. метаданных модели, за подробностями — к *главе 5*).

```
{% if comment.content_type.model == "good" %}
  <p>Товар: {{ comment.content_object.name }}</p>
{% else %}
  <p>Статья блога: {{ comment.content_object.title }}</p>
{% endif %}
```

Как уже говорилось ранее, если переменной `COMMENTS_HIDE_REMOVED` модуля `settings` пакета проекта было присвоено значение `True`, комментарии, помеченные как удаленные, не будут включаться в список комментариев. Но если мы задали для этой переменной значение `False`, удаленные комментарии будут присутствовать в списке. И тогда нам придется проверять, не является ли такой комментарий удаленным, т. е. не хранит ли его свойство `is_removed` значение `True`.

Давайте перепишем шаблон good.html таким образом, чтобы список комментариев к товару выводился в виде набора блоков. Фрагмент кода измененного шаблона будет выглядеть так:

```
{% block main %}
  . . .
  <p> </p>
  {% get_comment_count for good as good_comments_count  %}
  <p>Всего комментариев: {{ good_comments_count }}.</p>
  {% get_comment_list for good as good_comments %}
  {% for comment in good_comments %}
    {% if comment.is_removed %}
      <p class="comment-removed">Комментарий удален.</p>
    {% else %}
      <div class="comment">
        <div class="user">{{ comment.user_name }}</div>
        <div class="content">{{ comment.comment }}</div>
        <div class="date">{{ comment.submit_date }}</div>
      </div>
    {% endif %}
    <p> </p>
  {% endfor %}
  {% render_comment_form for good %}
{% endblock %}
```

В случае, если комментарий был помечен как удаленный, мы будем выводить на странице сообщающую об этом надпись.

И вставим в нашу таблицу стилей вот такой код:

```
.comment {
  border: 1px solid #cccccc;
}
```

```
.comment div {
  padding: 4px;
}
.comment .user {
  background-color: #eeeeee;
  font-weight: bold;
}
.comment .date {
  background-color: #eeeeee;
  font-style: italic;
  text-align: right;
}
.comment-removed {
  font-weight: bold;
}
```

Результат показан на рис. 15.3. Как видим, список комментариев стал намного симпатичнее.

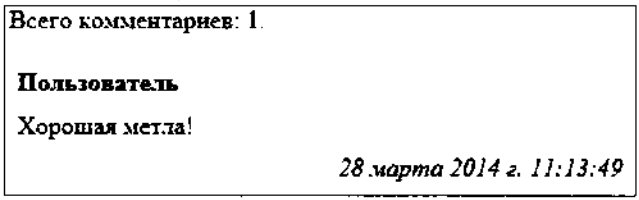


Рис. 15.3. Список комментариев к товару после форматирования

Может случиться так, что нам потребуется вывести список комментариев сразу на нескольких страницах приложения. Дублировать один и тот же код в нескольких шаблонах непродуктивно, так что мы вынесем его в отдельный шаблон.

Файл, в котором хранится шаблон списка комментариев, должен иметь имя list.html и храниться в папке comments. Сначала Django будет искать эту папку (и, соответственно, файл) в папке шаблонов уровня текущего приложения, а потом — в папке шаблонов уровня проекта. Это позволит нам использовать разные шаблоны для вывода комментариев, привязанных к разным объектам. (О шаблонах уровня приложения и проекта рассказывалось в *главе 8*.)

В контексте данных этого шаблона будет присутствовать переменная `comment_list`. Ее значением станет список комментариев в том же формате, в котором он формируется тегом `get_comment_list`.

Если мы вынесли код списка комментариев в отдельный шаблон, для вывода этого списка на страницах сайта мы используем уже знакомый нам тег шаблона `render_comment_list`.

Вот полный код шаблона comments/list.html, который мы можем использовать для вывода комментариев к товару:

```
{% for comment in comment_list %}
  {% if comment.is_removed %}
    <p class="comment-removed">Комментарий удален.</p>
  {% else %}
    <div class="comment">
      <div class="user">{{ comment.user_name }}</div>
      <div class="content">{{ comment.comment }}</div>
      <div class="date">{{ comment.submit_date }}</div>
    </div>
  {% endif %}
  <p> </p>
{% endfor %}
```

Управление выводом формы для комментирования

Так, с выводом списка комментариев нужным нам образом мы разобрались. Займемся формой для добавления комментариев.

Сначала мы получим саму форму, для чего используем тег `get_comment_form`:

```
{% get_comment_form for <объект> as <переменная> %}
```

Форма комментирования параметра *объект* будет сохранена в указанном нами параметре *переменная*.

Интернет-адрес, по которому будут отправлены введенные в форму данные, можно получить с помощью тега шаблона `comment_form_target`. Этот интернет-адрес мы укажем в атрибуте `action` тега `<form>`.

Как и все прочие формы, форма комментирования должна содержать кнопку отправки данных. Этой кнопке следует дать имя `submit`. Форма также может содержать кнопку предварительного просмотра комментария, которой мы дадим имя `preview`.

> ***Предварительный просмотр комментария***
>
> Предварительный просмотр комментария будет выполняться на служебной странице — одной из тех, о которых мы упоминали в разделе, посвященном необходимым настройкам проекта. Поскольку эти страницы очень неудобны, и мы собираемся исключить вывод их на экран, вероятно, лучше не давать посетителям возможность предварительного просмотра комментариев. (Тем более что особой пользы от этой возможности нет.)

Далее приведен код формы, который мы можем использовать в шаблоне good.html:

```
{% get_comment_form for good as form %}
<form action="{% comment_form_target %}" method="post">
  {% csrf_token %}
  {% for field in form.hidden_fields %}
    {{ field }}
  {% endfor %}
  {% for field in form.visible_fields %}
```

```
    <div class="form-field">
      {% if field.errors.count > 0 %}
        <div class="error-list">
          {{ field.errors }}
        </div>
      {% endif %}
      <div class="label">{{ field.label }}</div>
      <div class="control">{{ field }}</div>
    </div>
  {% endfor %}
  <div class="submit-button"><input type="submit" name="submit"
  value="Отправить"></div>
</form>
```

Правда, здесь возникнет одна проблема. Когда мы выведем на экран страницу сведений о товаре с обновленной формой комментирования, то увидим, что в ней присутствует текстовое поле с подписью «Если вы что-то введете в это поле, то ваш комментарий будет помечен как спам». Оно предусмотрено для защиты от программ автоматической рассылки спама, которые не «знают» об этой особенности подсистемы комментирования Django и будут стараться заполнить все поля такой формы.

Нам следует скрыть это поле, чтобы посетитель по ошибке ничего в него не ввел. Оно имеет имя `honeypot`, и для его скрытия нам достаточно использовать код вида:

```
. . .
{% if field.name == "honeypot" %}
<div class="honeypot">{{ field }}</div>
{% else %}
<div class="label">{{ field.label }}</div>
<div class="control">{{ field }}</div>
{% endif %}
. . .
```

И не забыть создать в таблице стилей стилевой класс `honeypot`:

```
.honeypot {
  display: none;
}
```

Если нам нужно выводить одну и ту же форму комментирования на разных страницах приложения, мы вынесем ее код в отдельный шаблон. Этот шаблон должен храниться в файле comments/form.html (по аналогии с шаблоном списка комментариев, с которым мы познакомились ранее). В контексте данных этого шаблона будет присутствовать переменная `form`, хранящая саму форму.

Вот такой код мы можем использовать для создания шаблона формы комментирования:

```
{% load comments %}
{% get_comment_form for good as form %}
```

```
<form action="{% comment_form_target %}" method="post">
  {% csrf_token %}
  {% for field in form.hidden_fields %}
    {{ field }}
  {% endfor %}
  {% for field in form.visible_fields %}
    <div class="form-field">
      {% if field.errors.count > 0 %}
        <div class="error-list">
          {{ field.errors }}
        </div>
      {% endif %}
      {% if field.name == "honeypot" %}
        <div style="display: none;">{{ field }}</div>
      {% else %}
        <div class="label">{{ field.label }}</div>
        <div class="control">{{ field }}</div>
      {% endif %}
    </div>
  {% endfor %}
  <div class="submit-button"><input type="submit" name="submit"
  value="Отправить"></div>
</form>
```

Для вывода формы комментирования, код которой хранится в отдельном шаблоне, мы применим в шаблонах страниц знакомый нам тег `render_comment_form`.

Перенаправление после добавления комментария

Мы уже знаем, что, как только посетитель добавит новый комментарий, Django перенаправит его на стандартную страницу с соответствующим сообщением. О недостатках этой страницы, равно как и всех служебных страниц такого рода, мы уже знаем и поэтому твердо решили от них избавиться. В частности, мы хотим сделать так, чтобы после добавления комментария страница сведений о товаре выводилась повторно — таким образом, посетитель сразу же увидит свой комментарий и поймет, что он был успешно создан.

Интернет-адрес, по которому должно быть выполнено перенаправление, задается прямо в форме комментирования. Для этого мы создадим в ней скрытое поле с именем `next` и в качестве его значения укажем нужный нам интернет-адрес:

```
<form action="{% comment_form_target %}" method="post">
  . . .
  <input type="hidden" name="next"
  value="{% url "good" good_id=good.id %}?page={{ pn }}">
  . . .
</form>
```

Здесь мы указываем перенаправление на страницу сведений о товаре, как и планировали ранее.

Комментирование только для зарегистрированных пользователей

Очень многие сайты предоставляют возможность оставлять комментарии только зарегистрированным пользователям. Как это сделать под Django?

Прежде всего, перед выводом формы комментирования нам следует проверить, выполнил ли пользователь вход на сайт. Для этого достаточно обратиться к переменной контекста шаблона `user`, где хранится текущий пользователь, и вызвать у него метод `is_authenticated`. (О разграничении доступа на Django-сайтах рассказывалось в *главе 14*.)

Если пользователь выполнил вход на сайт, его имя и адрес электронной почты будут автоматически подставлены в соответствующие поля комментария. Поэтому в форме комментирования выводить элементы управления для их ввода необязательно.

Рассмотрим следующий код шаблона формы для добавления комментария, которая доступна лишь для зарегистрированных пользователей:

```
{% load comments %}
{% if user.is_authenticated %}
  {% get_comment_form for good as form %}
  <form action="{% comment_form_target %}" method="post">
    {% csrf_token %}
    {% for field in form.hidden_fields %}
      {{ field }}
    {% endfor %}
    {% for field in form.visible_fields %}
      <div class="form-field">
        {% if field.errors.count > 0 %}
          <div class="error-list">
            {{ field.errors }}
          </div>
        {% endif %}
        {% if field.name == "honeypot" %}
          <div style="display: none;">{{ field }}</div>
        {% elif field.name == "name" or field.name == "email" or
        field.name == "url" %}
          {# Ничего не выводим #}
        {% else %}
          <div class="label">{{ field.label }}</div>
          <div class="control">{{ field }}</div>
        {% endif %}
      </div>
    {% endfor %}
```

```
    <input type="hidden" name="next"
    value="{% url "good" good_id=good.id %}?page={{ pn }}">
    <div class="submit-button"><input type="submit" name="submit"
    value="Отправить"></div>
    <div class="submit-button"><input type="submit" name="preview"
    value="Предпросмотр"></div>
  </form>
{% else %}
  <p>Чтобы добавить комментарий, выполните
  <a href="{% url "login" %}">вход на сайт</a>.</p>
{% endif %}
```

Здесь мы исключаем из состава выводимых на экран поля `name`, `email` и `url`, где указываются имя пользователя, его адрес электронной почты и интернет-адрес сайта соответственно. (Последний мало кто вводит, так что он не особо нужен.) В результате мы получим форму, содержащую лишь поле для ввода содержимого комментария.

Автомодератор Django и его использование

Комментирование опубликованных на сайте позиций — это, конечно, хорошо. Но хотелось бы несколько большей гибкости в плане управления комментированием. Так, неплохо было бы предусмотреть возможность запрета комментирования для определенных позиций и отключение комментирования по прошествии заданного времени. Также хорошо было бы предусмотреть возможность отправки по электронной почте определенным пользователям сообщений о вновь добавленных комментариях.

Возможно ли реализовать все это на Django? Разумеется! Для этого достаточно воспользоваться так называемым *автомодератором*, который выполнит все перечисленные ранее действия сам, без нашего участия.

Создание автомодератора

Автомодератор — это особый класс, являющийся потомком класса `CommentModerator` из модуля `django.contrib.comments.moderation` и привязываемый к модели, записи которой будут комментироваться. Объявляется этот класс в модуле `models` приложения — там же, где и модели.

Класс `CommentModerator` поддерживает несколько свойств, которые определяют параметры автоматического модерирования комментариев. Все эти свойства перечислены в табл. 15.1.

Таблица 15.1. *Свойства класса* `CommentModerator`

Свойство	Описание
`enable_field`	Задает имя поля модели, принадлежащего типу `BooleanField`. Если это поле хранит значение `False`, комментирование данной записи будет запрещено. Значение по умолчанию — `None`

Таблица 15.1 (окончание)

Свойство	Описание
email_notification	Если True, то при добавлении нового комментария всем пользователям, перечисленным в особом списке (об этом списке рассказано далее), будет отправлено по электронной почте соответствующее уведомление. Значение по умолчанию — False
auto_close_field	Задает имя поля модели, принадлежащего типу DateField или DateTimeField. По прошествии количества дней, указанного в свойстве close_after, после даты, хранящейся в этом поле, комментирование данной записи будет запрещено. Значение по умолчанию — None
close_after	Задает количество дней, по прошествии которых комментирование будет запрещено. Значение по умолчанию — None
auto_moderate_field	Задает имя поля модели, принадлежащего типу DateField или DateTimeField. По прошествии количества дней, указанного в свойстве moderate_after, после даты, хранящейся в этом поле, все вновь добавленные к данной записи комментарии будут делаться скрытыми. Значение по умолчанию — None
moderate_after	Задает количество дней, по прошествии которых все вновь добавленные комментарии будут помечаться как скрытые. Если указано значение 0, все комментарии будут скрываться сразу же после добавления. Значение по умолчанию — None

Вот примеры:

```
class Blog(models.Model):
  title = models.CharField(maxlength = 200)
  body = models.TextField()
  . . .
  pub_date = models.DateField(auto_now_add = True)
  enable_comments = models.BooleanField(default = True)
```

Здесь мы создаем модель Blog, в которую будут записываться статьи блога. В поле pub_date будет храниться дата публикации статьи — мы указали, что значение в это поле станет заноситься автоматически при создании новой записи. А в поле enable_comments будет храниться признак того, разрешены ли для статьи комментарии, значение по умолчанию этого поля — True. (О создании моделей говорилось в *главе 5*.)

```
from django.contrib.comments.moderation import CommentModerator
class BlogModerator(CommentModerator):
  enable_field = "enable_comments"
  email_notification = True
  auto_moderate_field = "pub_date"
  moderate_after = 30
```

А здесь создаем автомодератор для модели Blog. Активизируем автоматическую рассылку уведомлений о вновь добавленных комментариях, задаем возможность

запрета комментирования и указываем, что все комментарии, добавленные спустя 30 дней после создания статьи, должны изначально делаться скрытыми.

После объявления класса автомодератора его следует привязать к модели. Для этого мы вызовем не возвращающий результата метод `register` класса `moderator`, объявленного в том же модуле `django.contrib.comments.moderation`. Первым параметром этому методу передадим класс модели, а вторым — класс автомодератора, который к ней привязывается:

```
from django.contrib.comments.moderation import moderator
moderator.register(Blog, BlogModerator)
```

На этом создание автомодератора можно считать законченным.

Шаблон почтового сообщения

Нам также понадобится шаблон, на основе которого будут генерироваться почтовые сообщения о поступлении новых комментариев. К сожалению, Django не включает такой шаблон в комплект поставки, так что нам придется создать его самим.

Шаблон почтового сообщения может быть записан как в виде обычного текста (тогда будет отправлено обычное текстовое письмо), так и в формате HTML (что вызовет отправку письма в этом формате). В первом случае шаблон должен храниться в файле comment_notification_email.txt, а во втором — в файле comment_notification_email.html. В обоих случаях его следует поместить в подпапку comments папки templates либо уровня приложения, либо уровня проекта.

В контекст данных такого шаблона передаются две переменные:

- `comment` — объект класса `Comment`, представляющий комментарий;
- `content_object` — объект, для которого был оставлен данный комментарий. В нашем случае таким объектом будет товар.

Таким образом, для отправки обычного текстового сообщения мы можем написать шаблон со следующим кодом:

```
На сайте был оставлен новый комментарий:
  - товар: {{ content_object.name }} (http://www.somesite.ru↵
{{ content_object.get_absolute_url }}));
  - пользователь: {{ comment.user_name }};
  - e-mail: {{ comment.user_email }};
  - содержимое: {{ comment.comment }};
  - дата: {{ comment.submit_date }};
  - IP-адрес: {{ comment.ip_address }}.
```

Здесь мы подставляем после наименования товара в круглых скобках интернет-адрес, указывающий на страницу сведений о товаре. Для его вычисления мы используем метод `get_absolute_url`, возвращающий уникальный интернет-адрес, однозначно идентифицирующий данную запись модели. Об этом методе упоминалось в *главе 5* — сейчас же настало время объявить его:

```
from django.core.urlresolvers import reverse
. . .
class Good(models.Model):
  . . .
  def get_absolute_url(self):
    return reverse("good", kwargs = {"good_id": self.id})
```

Здесь для создания интернет-адреса на основе привязки мы применили знакомую нам по *главе 12* функцию `reverse`.

К сожалению, возвращенный методом `get_absolute_url` интернет-адрес не включит имя хоста. Нам придется самим вписать его в текст шаблона, как показано в приведенном ранее примере.

Настройка подсистемы отправки почты

Чтобы Django смогла успешно отправлять электронные письма с сообщениями о поступлении новых комментариев, мы должны указать в модуле `settings` пакета проекта параметры отправки почты. Этих параметров немного, и все они вместе с переменными, где задаются их значения, перечислены в табл. 15.2.

***Таблица 15.2.** Параметры отправки электронной почты*

Переменная	Описание
`MANAGERS`	Список пользователей, которые будут получать электронные письма с уведомлениями. Должен представлять собой кортеж, каждый элемент которого задает одного пользователя и также представляет собой кортеж из двух строковых элементов: имени пользователя и его адреса электронной почты. Все перечисленные в данном списке пользователи должны быть зарегистрированы на сайте
`EMAIL_HOST`	Интернет-адрес SMTP-сервера электронной почты
`EMAIL_PORT`	Номер порта, через который будет отправляться электронная почта. Значение по умолчанию — 25 (стандартный порт протокола отправки почты SMTP)
`EMAIL_HOST_USER`	Имя пользователя для подключения к SMTP-серверу
`EMAIL_HOST_PASSWORD`	Пароль для подключения к SMTP-серверу
`EMAIL_USE_TLS`	Если `True`, для подключения к SMTP-серверу будет использоваться защищенное соединение. Значение по умолчанию — `False`
`DEFAULT_FROM_EMAIL`	Адрес электронной почты отправителя. Значение по умолчанию — `"webmaster@localhost"`

Вот пример:

```
MANAGERS = (("admin", "admin@someserver.ru"),↵
("moderator", "moderator@someserver.ru"))
```

Указываем в качестве получателей уведомлений пользователей `admin` и `moderator`.

```
EMAIL_HOST = "someserver.ru"
EMAIL_HOST_USER = "someuser"
EMAIL_HOST_PASSWORD = "123456789"
EMAIL_USE_TLS = True
DEFAULT_FROM_EMAIL = "mailer@someserver.ru"
```

Здесь мы задаем собственно параметры отправки почты. (Разумеется, тут нужно подставить реальные интернет-адрес сервера и параметры подключения.)

Инструменты Django для модерирования комментариев

И рассмотрим встроенные в Django инструменты для модерирования комментариев. Они представляют собой список комментариев, доступ к которому можно получить через встроенный административный сайт.

Откроем этот сайт, набрав интернет-адрес **http://localhost:8000/admin/**, и, если нужно, выполним процедуру входа. Найдем на открывшейся в Web-обозревателе странице таблицу с заголовком **Comments**, соответствующую приложению `django.contrib.comments`, и щелкнем находящуюся в ней гиперссылку **Комментарии**.

> ***Дополнительная библиотека pytz***
>
> Для успешной работы со списком комментариев встроенного административного сайта Django следует установить дополнительную библиотеку pytz. Ее можно найти по интернет-адресу **https://pypi.python.org/pypi/pytz**.

На странице списка комментариев нам все уже знакомо. Единственное нововведение — в ее верхней части, над собственно списком комментариев, мы увидим набор гиперссылок, позволяющих быстро отфильтровать комментарии по дате их добавления.

Щелкнем на каком-либо из уже добавленных комментариев. Открывшаяся страница предоставит нам сведения об этом комментарии: имя создавшего его пользователя, адрес электронной почты и интернет-адрес сайта этого пользователя, содержимое комментария, дату и время его добавления и IP-адрес компьютера, с которого он был добавлен.

А еще в самом низу страницы мы увидим два флажка (рис. 15.4.). Флажок **Публичный** при его установке делает комментарий видимым на странице. А установленный флажок **Удален** помечает комментарий как удаленный.

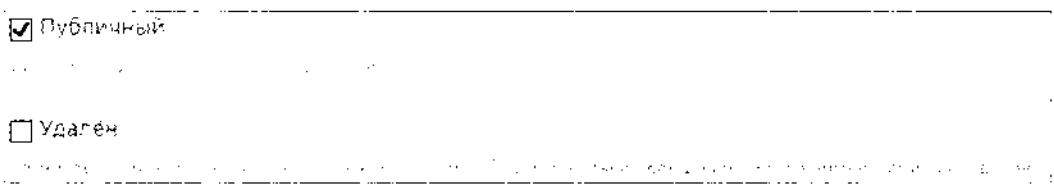


Рис. 15.4. Управляющие флажки на странице сведений о комментарии

Модерировать комментарии могут только пользователи, обладающие статусом персонала и соответствующим правом, которое носит наименование `comments | комментарий | Can change comments`. (О задании прав пользователей рассказывалось в *главе 14*.)

> ***Инструменты модерирования комментариев на сайте***
>
> Django предоставляет инструменты для модерирования комментариев прямо на сайте. Однако эти инструменты не очень удобны, поэтому автор не рекомендует их задействовать.

Что дальше?

В этой главе мы занимались системой комментирования опубликованных на сайте позиций. Мы изучили встроенные средства комментирования Django и инструменты, позволяющие настроить ее под наши нужды. Конечно, эти средства местами несколько неуклюжи, но во многих случаях могут нам пригодиться.

В следующей главе мы займемся подсистемой статичных страниц Django. Она позволит нам создать на сайте страницы с неизменяемым содержимым: сведения о разработчиках, список контактов и пр. Правда, эта подсистема также далеко не идеал...

ГЛАВА 16

Статичные страницы Django

В предыдущей главе мы создавали на своем сайте средства комментирования. Мы разобрались со встроенной подсистемой комментариев Django, научились адаптировать ее под свои нужды и даже реализовывать автоматическое модерирование комментариев.

В этой главе мы познакомимся с другим инструментарием Django, позволяющим нам создавать обычные Web-сайты со статичными страницами, содержимое которых не генерируется на основе хранящихся в моделях данных, а задается в виде обычного HTML-кода. Подсистема статичных страниц Django (в документации по этой библиотеке их называют «плоскими» страницами — flatpages, а на страницах встроенного административного сайта — простыми страницами), хоть и несколько неуклюжа, но во многих случаях может оказаться полезной.

Введение в статичные страницы

Прежде чем начать работать с подсистемой статичных страниц Django, нам следует уяснить несколько важных моментов:

- как уже говорилось, содержимое статичных страниц не генерируется программно на основе данных, хранящихся в моделях, а задается разработчиком сайта и представляет собой обычный HTML-код;
- содержимое статичных страниц должно представлять собой «чистый» HTML-код. Теги шаблонов (см. *главу 7*) в нем недопустимы, поскольку содержимое таких страниц не проходит обработку шаблонизатором Django. В результате могут возникнуть проблемы при вставке в код изображений и ссылок на файлы — как статичные, так и выгруженные на сайт посетителем (как решить эти проблемы, мы узнаем позже);
- содержимое статичных страниц хранится в особой модели. Оно включает в себя интернет-адрес страницы, ее заголовок, собственно содержимое, путь к файлу шаблона (если для страницы должен быть использован особый шаблон), признак, разрешено ли комментирование этой страницы, и признак, является ли

страница частью административного раздела, и, соответственно, должен ли посетитель выполнить процедуру входа на сайт, чтобы попасть на нее;

- работа со статичными страницами выполняется через встроенный административный сайт Django;
- мы не можем поместить в контекст данных шаблонов, применяемых для вывода статичных страниц, произвольные данные. Поэтому, если брать наш случай, мы не сможем вывести на этих страницах в составе меню навигации список категорий товаров. Вероятно, это основной недостаток подсистемы статичных страниц Django.

В качестве статичных страниц обычно реализуются служебные страницы сайта: сведения о разработчиках, контакты, всевозможные пояснительные статьи и др. Но это только в том случае, если на статичных страницах нам не требуется выводить сторонние данные, — тогда лучше будет создать обычные страницы с контроллерами и шаблонами.

На основе подсистемы статичных страниц можно создавать и обычные сайты, которым не требуется выводить данные, хранящиеся в моделях, или требуется, но в ограниченных количествах. Так, например, можно создать домашний сайт с гостевой книгой.

Настройка проекта для реализации статичных страниц

Как и в случае подсистемы комментирования, чтобы реализовать на сайте поддержку статичных страниц, нам придется внести некоторые модификации в настройки проекта. Так что первым делом откроем модуль `settings` пакета проекта.

Сначала проверим, включено ли в состав активных приложение `django.contrib.sites`, реализующее поддержку работы нескольких Web-сайтов на одной копии Django, и задан ли для нашего сайта идентификатор. (Список активных приложений хранится в переменной `INSTALLED_APPS`, а идентификатор сайта задается в переменной `SITE_ID`. За подробностями — к *главе 15*.)

Далее добавим в список активных приложение `django.contrib.flatpages`. Оно как раз и обеспечивает работу подсистемы статичных страниц:

```
INSTALLED_APPS = (
  . . .
  'django.contrib.sites',
  . . .
  'django.contrib.flatpages',
)
```

После этого выполним синхронизацию моделей с базой данных, чтобы Django создала в базе нужные таблицы.

Работа со статичными страницами

Работа со статичными страницами выполняется через встроенный административный сайт Django — это мы уже знаем. Войдем на сайт, набрав интернет-адрес **http://localhost:8000/admin/** и, если нужно, выполнив процедуру входа.

В наборе таблиц, находящемся на главной странице административного сайта, присутствует таблица с заголовком **Flatpages**, соответствующая приложению `django.contrib.flatpages`. В ней находится единственная гиперссылка **Простые страницы**, ведущая на список статичных страниц. (На страницах встроенного административного сайта статичные страницы почему-то называются *простыми*.)

Список статичных страниц ничем не примечателен — на подобные списки мы уже насмотрелись. Поэтому сразу же нажмем кнопку добавления записи, чтобы создать новую статичную страницу.

Интерфейс для создания новой статичной страницы можно увидеть на рис. 16.1. Посмотрим, что там есть.

В поле ввода **URL** заносится интернет-адрес создаваемой страницы. Его обязательно следует вводить с начальным и конечным слешами — вот так: **/about/**.

В поле ввода **Заголовок** вводится текст названия страницы (содержимого ее тега `<title>`).

Область редактирования **Содержимое**, как уже понятно, служит для ввода HTML-кода содержимого статичной страницы. Это содержимое может иметь произвольный размер.

В списке **Sites** выбирается сайт, для которого создается статичная страница. (Можно указать сразу несколько сайтов.) Сайт у нас всего один — он был создан самой Django и носит имя `example.com`, даваемое по умолчанию, — и нам обязательно следует его выбрать, иначе страница не будет создана.

В самом низу, над кнопками сохранения введенных данных, мы увидим небольшую раскрывающуюся панель («спойлер») **Расширенные настройки**. Чтобы ее развернуть, достаточно щелкнуть на расположенной правее ее заголовка гиперссылке **Показать**. Эта панель содержит элементы управления для указания дополнительных параметров страницы (рис. 16.2). Их немного.

Установка флажка **Включить комментарии** дает посетителям возможность создавать к этой странице комментарии. А установка флажка **Требуется регистрация** указывает Django, что эта страница является частью административного раздела сайта, и, чтобы попасть на нее, посетитель должен выполнить вход на сайт.

По умолчанию для вывода статичных страниц Django использует шаблон flatpages/default.html. Если нам нужно, чтобы та или иная страница выводилась с применением другого шаблона, мы укажем имя его файла в поле ввода **Имя шаблона**.

Шаблоны статичных страниц

Шаблоны статичных страниц также желательно хранить в папке *flatpages*.

Рис. 16.1. Интерфейс создания новой статичной страницы

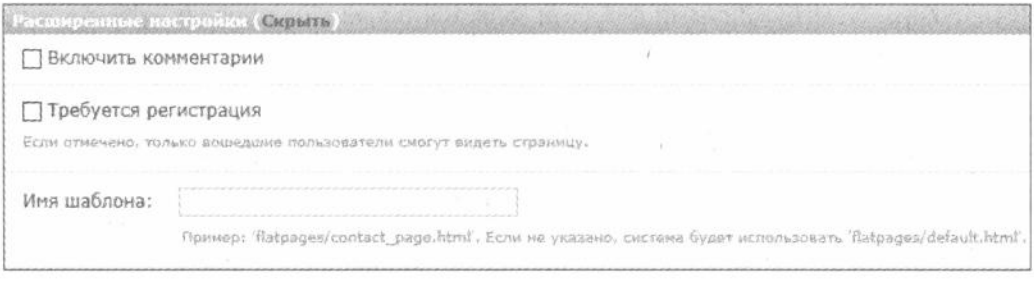


Рис. 16.2. Панель **Расширенные настройки**

Отметим, что статичные страницы в списке сортируются по интернет-адресам. И, к сожалению, изменить порядок их сортировки мы не можем. Так что, дабы задать нужный нам порядок сортировки статичных страниц, нам придется соответствующим образом задавать их интернет-адреса. Скажем, если мы хотим, чтобы страница сведений о сайте находилась в списке после страницы контактов, мы укажем для первой интернет-адрес **/contacts/**, а для второй — **/site_about/**.

Как указать интернет-адреса статичных файлов и файлов, выгруженных на сайт?

Когда мы начнем писать HTML-код статичных страниц, мы обязательно столкнемся с проблемой. Нам может потребоваться поместить на такую страницу графическое изображение, хранящееся в файле, который либо является статичным, либо был выгружен на сайт посетителем, или гиперссылку на какой-либо аналогичный файл.

Конечно, мы знаем, как записать нужный тег. Но каким образом получить самое главное — интернет-адрес такого файла? Очень просто.

Интернет-адреса статичных файлов (подробнее о них рассказывалось в *главе 8*) формируются по следующему шаблону:

/<префикс, указанный нами в переменной STATIC_URL в настройках проекта>/<путь к файлу, указанный относительно папки static>/

Так, если мы указали префикс `/static/`, а изображение, которое нам нужно вывести на страницу, хранится в файле images/logo.jpg в папке static, результирующий интернет-адрес этого файла будет **/static/images/logo.jpg**:

```
<img src="/static/images/logo.jpg">
```

Что касается файлов, выгруженных посетителем (о них разговор шел в *главе 13*), то их интернет-адреса формируются по похожему принципу:

/<префикс, указанный нами в переменной MEDIA_URL в настройках проекта>/<путь к файлу, указанный относительно папки выгруженных файлов>/

Путь к папке с выгруженными файлами задается также в настройках проекта, в переменной `MEDIA_ROOT`.

В случае если мы указали префикс `/media/`, а файл, гиперссылку на который нам нужно поместить на страницу, хранится в файле downloads/price_list.zip в папке выгруженных файлов, результирующий интернет-адрес этого файла будет **/media/downloads/price_list.zip**:

```
<a href="/media/downloads/price_list.zip">Прайс-лист</a>
```

Привязка статичных страниц

Хоть мы и указали интернет-адреса при создании статичных страниц, но все же должны выполнить их привязку. Привязка их выполняется на уровне проекта, т. е. в модуле `urls` пакета проекта.

Выполнить привязку статичных страниц можно тремя способами. Какой из них выбрать, зависит от структуры нашего сайта.

- Первый способ — привязать все статичные страницы к одной виртуальной папке. После этого все статичные страницы получат интернет-адреса вида:

  /<указанная нами виртуальная папка>/<интернет-адрес страницы>/.

  Сделать это очень просто. Достаточно вставить в список привязок вот такую строку:

```
urlpatterns = patterns('',
  . . .
  url(r'^flatpages/', include('django.contrib.flatpages.urls')),
)
```

  Тогда, если для страницы сведений о сайте мы указали интернет-адрес **/about/**, то попасть на эту страницу можно будет по интернет-адресу **/flatpages/about/**.

  Такой способ пригоден для сайтов, на которых статичные страницы объединены в один раздел. Именно в этом случае появляется смысл привязать их разом к одной виртуальной папке.

- Второй способ — сделать так, чтобы при обращении к любой виртуальной папке, кроме тех, для которых привязка была указана явно, активизировалась подсистема статичных страниц. Для этого в конец списка привязок следует вставить такую строку:

```
urlpatterns = patterns('',
  . . .
  url(r'^(?P<url>.*/)$', 'django.contrib.flatpages.views.flatpage'),
)
```

  После этого при обращении по интернет-адресу **/about/** будет выведена страница сведений о сайте, а при обращении к адресу, скажем, **/contacts/**, на экране появится статичная страница, в параметрах которой был указан этот интернет-адрес (т. е. страница контактов).

  Второй способ наилучшим образом подходит для сайтов, на которых бóльшая часть страниц являются статичными.

- Третий способ — привязать каждую статичную страницу к своему интернет-адресу.

```
urlpatterns = patterns('',
  . . .
  url(r'^about/$', 'django.contrib.flatpages.views.flatpage',↵
  {'url': '/about/'}, name = 'about'),
)
```

Привязываем страницу со сведениями о сайте к виртуальной папке about — после чего она будет у нас доступна по интернет-адресу **/about/**.

Здесь в первом параметре функции `url` мы, как обычно, задаем виртуальную папку, к которой привязывается статичная страница. А в третьем ее параметре, который представляет собой словарь, в элементе `url` этого словаря мы указываем интернет-адрес, который задали при создании привязываемой статичной страницы.

Третий способ хорош для сайтов, где бóльшую часть страниц составляют страницы, генерируемые на основе данных, хранящихся в моделях. К таким сайтам можно отнести и тот, что мы создадим в шестой части этой книги.

Мы можем комбинировать различные способы привязки:

```
urlpatterns = patterns('',
  url(r'^$', 'django.contrib.flatpages.views.flatpage',↳
  {'url': '/homepage/'}, name = 'index'),
  url(r'^(?P<url>.*/)$', 'django.contrib.flatpages.views.flatpage'),
)
```

Здесь мы создаем сайт, все страницы которого являются статичными. К корневой папке сайта мы привязываем страницу с интернет-адресом **/homepage/** — это главная страница нашего сайта, — применив третий способ. А привязку остальных страниц выполняем вторым способом.

Создание шаблонов для статичных страниц

Осталось создать шаблон, с применением которого статичные страницы будут выводиться на экран. Django не включает такой шаблон в свой состав, так что нам придется сделать это самим.

По умолчанию для вывода статичных страниц служит шаблон flatpages/default.html — это мы уже знаем. Его лучше создавать на уровне проекта (о шаблонах уровня приложения и уровня проекта рассказывалось в *главе 8*).

Контекст шаблона для статичных страниц содержит переменную `flatpage`. Ее значением является объект класса `FlatPage`, объявленный в модуле `django.contrib.flatpages.models`. (Да, и этот класс является моделью.) Он поддерживает набор свойств, хранящих различные сведения о статичной странице (табл. 16.1).

Таблица 16.1. *Свойства класса* `FlatPage`

Свойство	Описание
`url`	Интернет-адрес статичной страницы
`title`	Название
`content`	HTML-код содержимого
`enable_comments`	`True`, если посетитель может создавать комментарии к странице

Таблица 16.1 (окончание)

Свойство	Описание
`registration_required`	`True`, если страница входит в состав административного раздела и недоступна для гостей

Далее приведен полный код шаблона, который мы можем использовать для вывода статичных страниц:

```
{% extends "base.html" %}
{% load static %}
{% block title %}{{ flatpage.title }}{% endblock %}
{% block main %}
  <h2>{{ flatpage.title }}</h2>
  {{ flatpage.content }}
{% endblock %}
```

Если бы еще подсистема статичных страниц позволяла включать в контекст данных шаблона произвольные данные, цены бы ей не было! Но пока что остается надеяться, что в будущих версиях Django такая возможность появится...

Получение списка статичных страниц в шаблонах

Django предоставляет нам возможность получить список статичных страниц сайта прямо в шаблонах. Это позволит нам создать динамически формируемую панель навигации по всем статичным страницам сайта вместо того, чтобы создавать каждую гиперссылку вручную.

Прежде всего, нам необходимо загрузить модуль `flatpages` шаблонизатора, который обрабатывает нужные нам теги шаблона. Для этого следует поместить в код шаблона перед первым вызовом любого тега подсистемы статичных страниц следующий код:

```
{% load flatpages %}
```

Собственно получение списка статичных страниц выполняет тег `get_flatpages`, записываемый в таком формате:

```
{% get_flatpages [<префикс>] [for <пользователь>] as <переменная> %}
```

Список страниц будет помещен в параметр *переменная*, которую мы указали.

Каждая статическая страница в списке является объектом класса `FlatPage`. Мы можем получить сведения о ней, обратившись к свойствам этого класса, описанным в табл. 16.1:

```
{% get_flatpages as flatpages %}
<ul>
  {% for page in flatpages %}
```

```
    <li><a href="{{ page.url }}">{{ page.title }}</a></li>
  {% endfor %}
</ul>
```

Здесь мы формируем панель навигации для сайта, в котором привязка статичных страниц выполнена вторым способом (см. ранее в этой главе).

```
{% get_flatpages as flatpages %}
<ul>
  {% for page in flatpages %}
    <li><a href="/flatpages/{{ page.url }}">{{ page.title }}</a></li>
  {% endfor %}
</ul>
```

А здесь формируем панель навигации для случая, когда привязка статичных страниц выполнена первым способом.

```
{% get_flatpages as flatpages %}
<ul>
  <li><a href="{% url "index" %}">Главная</a></li>
  {% for page in flatpages %}
    {% if page.url != "/homepage/" %}
      <li><a href="{{ page.url }}">{{ page.title }}</a></li>
    {% endif %}
  {% endfor %}
</ul>
```

Здесь же мы формируем панель навигации для сайта с комбинированной привязкой, выводя гиперссылку, указывающую на главную страницу, отдельно от гиперссылок на остальные страницы.

Мы можем указать Django включить в список только те статичные страницы, чей интернет-адрес начинается с заданного нами префикса (см. приведенный ранее формат написания тега `get_flatpages`):

```
{% get_flatpages "/contacts/" as flatpages %}
```

В результате получаем список статичных страниц, чей интернет-адрес начинается с символов `"/contacts/"`.

По умолчанию Django включает в список только те статичные страницы, для посещения которых не нужно выполнять процедуру входа на сайт (см. раздел, посвященный работе со статичными страницами). Однако мы можем включить в список и страницы, недоступные для гостей, для чего укажем в теге `get_flatpages` переменную, хранящую текущего пользователя (см. приведенный ранее формат написания этого тега):

```
{% get_flatpages for user as flatpages %}
```

Если посетитель не выполнил вход на сайт, список статичных страниц включит только те из них, которые не требуют входа (как и по умолчанию). Но если пользователь выполнил вход на сайт, список будет содержать и страницы, закрытые для

гостей. (Переменная контекста данных `user`, как мы помним из *главы 14*, хранит текущего пользователя.)

Что дальше?

В этой главе мы занимались подсистемой статичных страниц Django и выясняли, насколько она будет нам полезна. Нам, с нашим сугубо «динамическим» сайтом, она не пригодится, но вот разработчикам обычных сайтов, страницы которых, в основном, не формируются на основе хранящихся в моделях данных, вполне может понравиться.

Следующая часть книги будет посвящена дополнительным модулям Python. Они привносят в Python и Django новую функциональность, которая окажется нам очень кстати. Так, поддержка вывода миниатюр, форматирования текста с использованием кодов BBCode и добавления к данным тегов явно не будет лишней.

ЧАСТЬ V

Дополнительные библиотеки

ГЛАВА 17

Создание и вывод миниатюр. Библиотека easy-thumbnails

В предыдущих главах мы занимались реализацией разграничения доступа и комментирования и разбирались с инфраструктурой статичных страниц Django. Наш сайт постепенно становится все более и более профессиональным.

Django — исключительно мощная библиотека. Ее функциональности могут позавидовать многие из менее именитых конкурентов. Но абсолютно все она объять не в состоянии, и обязательно наступит момент, когда нам потребуется какая-либо функция, не реализованная ни в ней, ни в языке Python.

В таком случае нам на помощь придут дополнительные библиотеки от независимых разработчиков. Их существует огромное множество — реализующих практически любую функциональность.

В этой главе мы будем говорить о библиотеке easy-thumbnails, позволяющей создавать миниатюры на основе графических изображений.

Введение в библиотеку easy-thumbnails

Как уже говорилось ранее, библиотека easy-thumbnails позволяет создавать на основе графических изображений миниатюры. Эти миниатюры сохраняются (кэшируются) в особой папке, чтобы исключить их повторное создание при следующем запросе.

Библиотека предлагает несколько тегов и фильтров шаблонов, которые используются для генерирования и вывода миниатюр на экран. Мы можем управлять созданием миниатюр, задавая их размеры, качество и многие другие параметры.

> ***Установка библиотеки Pillow***
>
> Для нормальной работы библиотеки easy-thumbnails требуется предварительно установить библиотеку Pillow. Ее дистрибутивный комплект можно найти по интернет-адресу **http://www.lfd.uci.edu/~gohlke/pythonlibs/#pillow**.

Сама библиотека easy-thumbnails доступна по интернет-адресу **https://github.com/SmileyChris/easy-thumbnails/**, а документация к ней — по интернет-адресу **http://easy-thumbnails.readthedocs.org/en/latest/index.html**.

Настройка проекта

После установки библиотеки нам следует соответственно настроить наш проект. Поэтому откроем модуль `settings` пакета проекта.

Базовые настройки

Первое, что нам необходимо сделать, — занести в список активных приложение `easy_thumbnails`, которое, собственно, и занимается генерированием и выводом миниатюр:

```
INSTALLED_APPS = (
  . . .
  'easy_thumbnails',
)
```

После чего обязательно выполним синхронизацию с базой данных, чтобы Django создала все необходимые для работы этого приложения таблицы.

Все остальные настройки являются необязательными и влияют лишь на процесс генерирования миниатюр. Имена переменных, в которых указываются наиболее полезные, на взгляд автора, настройки, перечислены в табл. 17.1.

***Таблица 17.1.** Переменные модуля `settings`, в которых задаются настройки приложения `easy_thumbnails`*

Переменная	Описание	Значение по умолчанию
`THUMBNAIL_BASEDIR`	Имя папки хранения миниатюр, находящейся в папке, где хранятся выгруженные файлы	`""`
`THUMBNAIL_DEFAULT_OPTIONS`	Параметры миниатюр по умолчанию	`None`
`THUMBNAIL_EXTENSION`	Расширение файлов миниатюр. Указывается без точки	`"jpg"`
`THUMBNAIL_HIGH_RESOLUTION`	Если `True`, будут отдельно генерироваться файлы миниатюр для экранов сверхвысокого разрешения (Retina и 4K)	`False`
`THUMBNAIL_PREFIX`	Префикс, который будет добавляться к именам файлов миниатюр	`""`
`THUMBNAIL_QUALITY`	Качество миниатюр формата JPEG. Указывается в виде числа от 1 (наихудшее качество) до 95 (наилучшее качество)	`85`
`THUMBNAIL_SUBDIR`	Имя папки хранения миниатюр, находящейся в папке, где хранится оригинальный файл	`""`

Таблица 17.1 (окончание)

Переменная	Описание	Значение по умолчанию
THUMBNAIL_TRANSPARENCY_EXTENSION	Расширение файлов миниатюр, имеющих канал полупрозрачности. Указывается без точки	"png"

Теперь дадим пояснения к этим переменным и указываемым в них настройкам.

По умолчанию файлы миниатюр сохраняются в тех же папках, где находятся файлы с оригинальными изображениями. При этом каждый сгенерированный на их основе файл миниатюры получит другое имя, включающее имя оригинального файла, его расширение, обозначение размера миниатюры и ее качества. Так, для файла brush.jpg, хранящегося в папке goods папки выгруженных файлов, для хранения миниатюры размером 200×100 пикселов и качеством 90 будет создан файл с именем brush_jpg_200x100_q90.jpg.

Если мы с помощью переменной THUMBNAIL_PREFIX укажем для имен файлов миниатюр префикс, он будет добавлен в начало имени файла. Так, если мы укажем префикс thumb, то наш файл получит имя thumb_brush_jpg_200x100_q90.jpg.

Мы можем указать приложению easy_thumbnails сохранять файлы миниатюр в отдельной папке, расположенной в папке выгруженных файлов, задав имя этой папки в переменной THUMBNAIL_BASEDIR. При этом файлы миниатюр будут сохраняться в этой папке с сохранением структуры папок, в которые последовательно вложен оригинальный файл. Так, если мы зададим папку thumbnails, то миниатюра для файла goods\brush.jpg будет сохранена в файле thumbnails\goods\brush_jpg_200x100_q90.jpg.

Также мы можем указать приложению easy_thumbnails сохранять файлы миниатюр в папке, находящейся непосредственно в папке, где хранится оригинальный файл, задав имя этой папки в переменной THUMBNAIL_SUBDIR. И если мы зададим, скажем, папку thumbnails, то миниатюра для файла goods\brush.jpg будет сохранена в файле goods\thumbnails\ brush_jpg_200x100_q90.jpg.

> ***ОТДЕЛЕНИЕ МИНИАТЮР ОТ ОРИГИНАЛЬНЫХ ИЗОБРАЖЕНИЙ***
>
> Правила хорошего тона Web-программирования рекомендую отделять миниатюры от оригинальных изображений. Поэтому следует задать папку хранения миниатюр в переменной THUMBNAIL_BASEDIR или THUMBNAIL_SUBDIR.
>
> ```
> THUMBNAIL_BASEDIR = "thumbnails"
> ```

Параметры миниатюр по умолчанию

Используемые по умолчанию параметры миниатюр указываются в переменной THUMBNAIL_DEFAULT_OPTIONS. Ее значением должен быть словарь, каждый элемент которого задает один из параметров.

Библиотека easy-thumbnails поддерживает довольно много параметров, которые мы можем задать для генерируемых ей миниатюр. Рассмотрим их.

- `size` — задает размеры миниатюры. Они должны быть указаны либо в виде кортежа из двух числовых значений — ширины и высоты, — либо в виде строки с этими двумя значениями, разделенными запятой. Значение, отличное от нуля, собственно задает размер в пикселах, нулевое значение указывает библиотеке подобрать такой размер, чтобы миниатюра сохранила свои пропорции:

  ```
  size = (200, 100)
  ```

  Здесь мы задаем размер миниатюры 200×100 пикселов.

  ```
  size = (200, 0)
  ```

  А здесь задаем ширину миниатюры в 200 пикселов. Ее высота при этом станет такой, чтобы миниатюра сохранила пропорции сторон.

- `upscale` — если `True`, слишком маленькие изображения в процессе формирования миниатюр будут увеличиваться в размерах. Значение по умолчанию — `False`.

- `crop` — задает параметры обрезки изображений. Они указываются в виде строки из двух числовых значений, задающих собственно величину обрезки в процентах, эти значения разделяются запятой. Первое число задает величину обрезки с левой стороны, если оно положительное, или с правой, если оно отрицательное. Второе число, будучи положительным, задает величину обрезки с верхней стороны, а будучи отрицательным — с нижней. Если задать значение 0, то величина обрезки с соответствующей стороны — левой или верхней — будет выбрана автоматически. Если же не указать какое-либо значение, то обрезка будет выполнена сразу с обеих сторон: с левой и правой или с верхней и нижней соответственно.

  Например:

  ```
  crop = "10, 0"
  ```

  Здесь мы задаем величину обрезки, равную 10 %, с левой стороны. Величина обрезки с верхней стороны будет выбрана автоматически.

  ```
  crop = ", -20"
  ```

  А здесь указываем выполнить обрезку на 20 % с нижней стороны. При этом также будет выполнена обрезка с левой и правой сторон, и величина ее будет выбрана автоматически.

  ```
  crop = ","
  ```

  Здесь указываем выполнить автоматическую обрезку со всех сторон.

  Также мы можем задать для параметра `crop` два предопределенных значения:

  - `"smart"` — «умная» обрезка, при которой библиотека сама будет выбирать величину обрезки с каждой стороны так, чтобы не затронуть сюжетно важную часть изображения;
  - `False` — вообще отключает обрезку. Это, кстати, значение параметра `crop` по умолчанию.

- `target` — задает координаты центра изображения, относительно которого будет выполняться его обрезка. Указывается в таком же формате, что и размер миниатюры (см. ранее). Координаты задаются в процентах. Если вместо какой-либо из координат задать значение `None` или вообще его не указать, будет выбрано значение 50. Значение этого параметра по умолчанию: `(50, 50)`.

  Например:

  ```
  target = (10, 90)
  ```

  Здесь мы задаем координаты центра изображения [10 %, 90 %].

  ```
  target = ", 0"
  ```

  А здесь задаем координаты центра изображения [50 %, 0 %].

- `background` — задает цвет, которым будут заполняться края изображения при его увеличении до нужного размера. Значение по умолчанию: `None`.

  Например:

  ```
  background = 0
  ```

  Здесь мы задаем черный цвет заполнения.

- `bw` — если `True`, создает черно-белую миниатюру. Значение по умолчанию: `False`.

- `replace_alpha` — задает цвет, которым будет заменены полупрозрачные участки изображения. Значение по умолчанию — `False`, т. е. замена полупрозрачных участков отключена.

  Например:

  ```
  replace_alpha = 16777215
  ```

  Здесь мы задаем замену полупрозрачных участков белым цветом.

- `sharpen` — если `True`, делает изображение более резким. Значение по умолчанию: `False`.

- `detail` — то же самое, что `sharpen`, но более мягкий вариант.

Таким образом, запись, задающая для миниатюр «умную» обрезку и «мягкое» повышение резкости, будет выглядеть так:

```
THUMBNAIL_DEFAULT_OPTIONS = {"crop": "smart", "detail": True}
```

Псевдонимы

Указанные нами в переменной `THUMBNAIL_DEFAULT_OPTIONS` параметры по умолчанию будут применены ко всем миниатюрам в проекте. Но часто бывает необходимо указать для разных приложений или даже для разных полей моделей, входящих в состав одного приложения, разные параметры миниатюр. Конечно, можно задавать их каждый раз явно (как, мы вскоре узнаем), но проще установить их в настройках проекта в виде так называемого *псевдонима* (alias — в терминологии библиотеки easy-thumbnails).

Псевдонимы указываются в переменной THUMBNAIL_ALIASES. Ее значением должен быть словарь, каждый элемент которого задает набор псевдонимов для определенной *цели*. Ключ такого элемента должен представлять собой наименование цели, а значение — словарь со списком относящихся к ней псевдонимов.

Цель псевдонимов задает приложение, модель и поле, для которого будут применимы данные псевдонимы. Ее наименование записывается в следующем формате:

<имя приложения>[.*<имя модели>*[.*<имя поля>*]]

Если не указывать *имя поля*, псевдонимы будут применимы ко всем полям модели, имеющим подходящий тип (FileField или ImageField). Если не указывать *имя модели*, псевдонимы будут применимы ко всем моделям приложения.

Если в качестве цели указать пустую строку, псевдонимы могут быть применены ко всем приложениям проекта.

Вернемся к словарю — списку относящихся к определенной цели псевдонимов. Ключ элемента этого словаря задаст имя псевдонима, а значение — собственно параметры миниатюр в том же формате, в котором мы указали параметры по умолчанию (см. ранее):

```
THUMBNAIL_ALIASES = {
  "page.Good.thumbnail": {
    "bw": {"size": (200, 100), "bw": True},
  },
  "news": {
    "small": {"size": (100, 60)},
    "large": {"size": (400, 300)},
  },
}
```

Здесь мы создаем три псевдонима: bw, применимый лишь для поля thumbnail модели Good приложения page, и small и large, применимые для всех полей всех моделей приложения news.

Вывод миниатюр

Теперь рассмотрим инструменты для вывода миниатюр. Они включают в себя один тег шаблона и один фильтр.

Перед тем как использовать их в коде шаблонов, нам следует загрузить модуль thumbnail шаблонизатора. Для этого мы вставим в начало кода шаблона, перед первым использованием инструментов для вывода миниатюр, вот такой тег:

```
{% load thumbnail %}
```

Вывод на основе псевдонима

Если мы уже создали псевдоним со всеми нужными параметрами, наша задача существенно упрощается. Мы используем фильтр шаблона thumbnail_url:

<изображение>|thumbnail_url:*<имя псевдонима>*

Параметр *имя псевдонима* указывается в виде строки:

```
<img src="{{ good.thumbnail|thumbnail_url:"bw" }}">
```

Здесь мы выводим миниатюру изображения товара, хранящуюся в поле `thumbnail` модели `Good`, на основе параметров, указанных в псевдониме `bw`.

Если псевдоним с указанным именем отсутствует, заданная переменная имеет неподходящий тип или вообще пуста, фильтр `thumbnail_url` возвращает пустую строку. С одной стороны, это обеспечивает нормальный вывод страницы, но, с другой, затрудняет выявление ошибок. Так что будем внимательны в таких случаях.

Вывод с указанием параметров

Если же мы не создали подходящий псевдоним, нам потребуется при выводе миниатюры указать все необходимые параметры. В первую очередь, это размеры миниатюры — если мы их не укажем, библиотека easy-thumbnails не сможет создать миниатюру.

В этом случае нам понадобится тег шаблона `thumbnail`. Вот его формат:

```
{% thumbnail <изображение> <размеры>|<имя псевдонима>↵
[<остальные параметры>] %}
```

Размеры миниатюры указываются в формате *<ширина>x<высота>*. Вместо размеров мы можем указать *имя псевдонима* — оно также задается в виде строки.

Мы можем также задать остальные параметры создаваемой миниатюры: режим обрезки, цвет фона, повышение резкости и пр. Эти параметры разделяются пробелами. Если какой-либо параметр принимает в качестве значения логическую величину, то, чтобы присвоить ему значение `True`, достаточно указать лишь имя этого параметра.

Тег `thumbnail` вставляет в код интернет-адрес файла миниатюры в том месте, в котором находится:

```
<img src="{% thumbnail good.thumbnail 200x100 %}">
```

Здесь мы выводим миниатюру товара размерами 200×100 пикселов.

```
<img src="{% thumbnail good.thumbnail 200x100 sharpen %}">
```

Здесь выводим миниатюру товара размерами 200×100 пикселов с повышением резкости.

```
<img src="{% thumbnail good.thumbnail "bw" %}">
```

А здесь выводим миниатюру товара с применением параметров, указанных в псевдониме `bw`. (Хотя в этом случае удобнее будет воспользоваться описанным ранее фильтром `thumbnail_url`.)

Если псевдоним с указанным именем отсутствует, заданная переменная имеет неподходящий тип или вообще пуста, тег `thumbnail` также вернет пустую строку.

Вывод изображения по умолчанию

Если поле, где хранится оригинальное изображение, пусто, тег `thumbnail` выводит пустую строку. В результате на странице вместо миниатюры будет присутствовать пустое пространство, что некрасиво и может нарушить дизайн сайта.

В таких случаях обычно выводят некое изображение по умолчанию. Мы, поклонники библиотек Django и easy-thumbnail, можем указать его прямо в теге `thumbnail`, применив фильтр `default`:

```
{% thumbnail <изображение>|<файл изображения по умолчанию> . . . %}
```

Например:

```
<img src="{% thumbnail good.thumbnail|default:"goods/default.jpg"↵
200x100 %}">
```

Теперь, если поле `thumbnail` пусто, на экран будет выведено изображение, хранящееся в файле **goods\default.jpg** папки выгруженных файлов.

Что дальше?

Хороша библиотека easy-thumbnail! Если нам нужно вывести миниатюры хранящихся на сайте изображений, мы сделаем это без особых хлопот, просто поместив в код шаблонов нужный тег или фильтр.

А библиотека django-taggit, с которой мы познакомимся в следующей главе, позволит привязать теги к любой позиции, опубликованной на сайте. Думается, в блоге эта возможность будет нелишней...

ГЛАВА 18

Привязка тегов к данным. Библиотека django-taggit

В предыдущей главе мы выводили на страницы нашего сайта миниатюры графических изображений, для чего использовали замечательную библиотеку easy-thumbnail.

В этой главе мы займемся другой дополнительной библиотекой. Она носит название django-taggit и предназначена для привязки к опубликованным на сайте позициям тегов и выполнения поиска по ним.

Введение в теги

Но сначала давайте выясним, что такое теги и чем они могут помочь посетителям нашего сайта.

Все сайты, публикующие множество более или менее однотипных позиций: статей, товаров, файлов, изображений, электронных писем и пр., — предоставляют возможность поиска нужной позиции по введенному посетителем ключевому слову. Посетитель вводит его в особой форме, нажимает кнопку поиска и получает список позиций, в названиях и описаниях которых встретилось это ключевое слово.

Создать такой поиск мы сможем без особого труда, поскольку все необходимые для этого инструменты Django уже изучили в *главе 5*. И обязательно займемся этим впоследствии, когда собственно начнем работу над сайтом.

Но в последнее время различные сайты предоставляют посетителям возможность более быстрого и удобного поиска позиций, имеющих какие-либо схожие признаки. Здесь идея очень проста:

- к каждой позиции привязываются несколько ключевых слов, описывающих ее наиболее полно и исчерпывающе;
- привязанные к позиции ключевые слова выводятся в ее составе в виде гиперссылок. Каждая гиперссылка указывает на контроллер, выполняющий поиск, и содержит в своем составе само ключевое слово: либо прямо в интернет-адресе, либо в GET-параметре;

- посетитель, желающий найти другие позиции, которые описывает какое-либо ключевое слово, щелкает на соответствующей ему гиперссылке;
- контроллер сайта, выполняющий поиск, получает это ключевое слово, выполняет поиск по нему и выводит список всех остальных аналогичных позиций, к которым оно было привязано.

Такой подход к реализации поиска заметно удобнее, т. к. не требует от посетителя вводить что-либо вручную. Недостатком можно считать лишь тот факт, что, как правило, набор привязанных к позициям ключевых слов ограничен и не охватывает всех описывающих их признаков. (Впрочем, ради таких случаев на сайтах предусматривается и обычный, традиционный поиск с полем ввода ключевого слова.)

Осталось сказать, что такие предопределенные ключевые слова, привязываемые к опубликованным на сайте позициям, называются *тегами*. (Не путать с тегами HTML!)

Если взять наш случай, то для описания веника, предназначенного для уборки жилых помещений, можно использовать теги «веник» и «дом», а для метлы, что будет применяться для уборки производственных цехов, — теги «метла» и «производство».

К тегам предъявляются следующие требования:

- набор тегов должен быть минимален, т. е. для описания различных позиций следует создавать как можно меньше тегов, чтобы не запутать посетителя;
- к каждой позиции следует привязывать минимально необходимое для ее описания количество тегов;
- теги должны быть, по возможности, короткими и односложными (тут надо сказать, что теги, состоящие из нескольких слов, не так удобны в использовании);
- теги должны по возможности наиболее полно описывать позиции, к которым привязаны;
- теги следует записывать без ошибок, в именительном падеже и единственном числе.

Теги (как и обычный, традиционный поиск) имеет смысл использовать только на сайтах, где опубликовано большое количество однотипных позиций. В противном случае они просто не нужны.

Введение в библиотеку django-taggit

Библиотека django-taggit предоставляет все необходимые инструменты для реализации на сайте поддержки тегов и поиска по ним. Она позволяет привязывать произвольный набор тегов к любой записи модели, имеющейся в проекте.

Библиотека django-taggit включает:

- особый класс поля для моделей, предназначенный для хранения всех привязанных к записи тегов в удобном для обработки формате;
- особое поле формы для ввода набора привязываемых к записи тегов и особый элемент управления для него;

- инструменты для просмотра и правки набора всех созданных тегов через встроенный административный сайт Django.

Установочный комплект библиотеки django-taggit можно найти по интернет-адресу **https://github.com/alex/django-taggit**, а документацию по ней — по интернет-адресу **http://django-taggit.readthedocs.org/**.

Настройка проекта

Перед тем как начать реализацию поддержки тегов с применением библиотеки django-taggit, нам необходимо внести исправления в настройки проекта. Откроем модуль `settings` пакета проекта.

Единственное, что нам нужно выполнить здесь, — сделать приложение `taggit`, которое и реализует поддержку тегов, активным. Для этого мы включим его в состав элементов списка, являющегося значением переменной `INSTALLED_APPS`:

```
INSTALLED_APPS = (
  . . .
  'taggit'
)
```

После чего обязательно будет нужно выполнить синхронизацию с базой данных. В результате Django создаст таблицу, в которой будут храниться все созданные нами теги.

Добавление тегов к позициям

Как уже говорилось ранее, для хранения тегов, привязанных к какой-либо записи модели, библиотека django-taggit предлагает особый класс поля модели. Этот класс носит имя `TaggableManager` и объявлен в модуле `taggit.managers`:

```
from taggit.managers import TaggableManager
class Good(models.Model):
  . . .
  tags = TaggableManager()
```

Здесь мы объявляем в модели `Good` поле `tags`, в котором будут храниться привязанные к товару теги.

Класс `TaggableManager` поддерживает знакомые нам по табл. 5.3 параметры `blank`, `help_text` и `verbose_name`. Все прочие теги, указанные в той таблице, не поддерживаются.

```
class Good(models.Model):
  . . .
  tags = TaggableManager(blank = True, verbose_name = "Теги")
```

Здесь мы делаем поле `tags` необязательным и задаем для него надпись `"Теги"`.

> **Класс TaggableManager**
>
> Отметим два интересных момента. Во-первых, класс `TaggableManager` — это фактически не поле, а своего рода «посредник» между моделью, в которой было создано поле этого класса, и таблицей приложения `taggit`, где хранятся все теги. Во-вторых, как вытекает из только что сказанного, никакого поля в таблице базы данных, соответствующей модели, при этом не создается. Но мы для простоты будем считать, что данный класс — это обычный класс поля.

Для ввода тегов в формах применяется класс поля `TagField`, объявленный в модуле `taggit.forms`. Он поддерживает те же параметры, что и класс поля `CharField`, знакомый нам по *главе 11* (поскольку класс `CharField` является его родителем):

```
from taggit.forms import TagField
class GoodForm(forms.ModelForm):
  class Meta:
    model = Good
  . . .
  tags = TagField(label = "Теги")
```

Здесь мы создаем в форме ввода товара новое поле `tags`, в котором будут указываться теги.

По умолчанию для вывода такого поля на экран применяется класс элемента управления `TagWidget`, также объявленный в модуле `taggit.forms`, являющийся потомком класса `TextInput` (обычное поле ввода, за подробностями — к *главе 12*) и внешне выглядящий так же. Менять этот элемент управления на другой не рекомендуется, иначе теги не будут обрабатываться должным образом.

Поле формы `TagField` обрабатывает введенное в него строковое значение по следующим правилам:

- если введенная строка не содержит ни запятых, ни двойных кавычек, каждое ее слово трактуется как отдельный тег;
- фрагмент строки, взятый в двойные кавычки, считается одним тегом. Такой тег может содержать пробелы и запятые. Незакрытые кавычки игнорируются;
- фрагменты строки, разделенные запятыми, считаются отдельными тегами. Пробелы между запятой и соседними словами игнорируются.

В табл. 18.1 приведено несколько примеров разбора строк, введенных в такое поле формы.

***Таблица 18.1.** Примеры формирования набора тегов на основе строк, введенных в поле формы `TagField`*

Введенное значение	Результирующий набор тегов
веник жилое помещение	"веник", "жилое", "помещение"
веник "жилое помещение"	"веник", "жилое помещение"
метла "уборка улиц	"метла", "уборка", "улиц"

Таблица 18.1 (окончание)

Введенное значение	Результирующий набор тегов
метла "уборка улиц, предприятий и учреждений"	"метла", "уборка улиц, предприятий и учреждений"
метла, уборка улиц	"метла", "уборка улиц"

Как видим, библиотека django-taggit достаточно «умна», чтобы сформировать корректные теги на основе введенной нами строки.

Обработка тегов

Итак, теги к позициям мы привязали. Теперь нужно каким-то образом выполнить поиск позиций, к которым был привязан указанный тег. И неплохо было бы узнать, как управлять набором привязанных к какой-либо позиции тегов непосредственно, без применения форм, привязанных к моделям.

Поиск по тегам

Выполнить поиск по тегам очень просто. Все, что нам нужно для этого знать, было описано еще в *главе 5*, посвященной моделям.

Все теги, что мы привязали ко всем записям всех моделей всех приложений нашего проекта, хранятся в особой таблице — это мы уже знаем. Поле `name` этой таблицы хранит имя тега. Следовательно, чтобы найти, скажем, все товары с привязанным тегом `"веник"`, мы напишем вот такое выражение:

```
goods = Good.objects.filter(tags__name = "веник")
```

Если нам нужно произвести поиск товаров, к которым были привязаны теги `"веник"` или `"щетка"`, мы напишем вот что:

```
goods = Good.objects.filter(tags__name__in = ["веник", "щетка"]).↵
distinct()
```

Поскольку при этом возможно дублирование записей, мы используем метод `distinct`, чтобы получить в результирующем наборе только уникальные записи. (Вот нам и пригодился этот метод!)

Этот код мы можем использовать в написанном в *главе 10* классе-контроллере списка товаров для реализации поиска по тегам:

```
class GoodListView(ListView, CategoryListMixin):
  . . .
  def get(self, request, *args, **kwargs):
    if self.kwargs["cat_id"] == None:
      self.cat = Category.objects.first()
    else:
      self.cat = Category.objects.get(pk = self.kwargs["cat_id"])
```

```
    return super(GoodListView, self).get(request, *args, **kwargs)
  . . .
  def get_queryset(self):
    goods = Good.objects.filter(category = self.cat).order_by("name")
    try:
      goods = goods.filter(tags _name = self.request.GET["tag"])
    except KeyError:
      pass
    return goods
```

Здесь мы получаем указанный посетителем тег через GET-параметр `tag`. На взгляд автора, так удобнее, нежели вставлять его в состав интернет-адреса.

Программное управление тегами

Иногда приходится заносить значения в поля моделей не через привязанные к моделям формы, а программно. Как это делается в случае обычных полей, мы узнали в *главе 11*. Осталось выяснить, как это сделать для поля класса `TaggableManager`.

Этот класс поддерживает ряд методов (табл. 18.2), позволяющих нам манипулировать набором тегов, что привязаны к записи.

Таблица 18.2. Методы класса `TaggableManager`, предназначенные для манипуляции привязанными к записи тегами

Метод	Описание
`add(<теги>)`	Добавляет к записи теги, указанные в виде строк. Каждая строка должна представлять собой отдельный параметр
`remove(<теги>)`	Удаляет из записи теги, указанные в виде строк. Если какой-либо из указанных тегов не был привязан к записи, ошибки не возникает, никаких исключений не генерируется. Каждая строка должна представлять собой отдельный параметр
`clear()`	Удаляет из записи все теги
`set(<теги>)`	Удаляет из записи все теги и добавляет к ней указанные. Каждый из привязываемых тегов должен представлять собой отдельный параметр, и каждый задается в виде строки

Вот примеры:

```
good = Good.objects.all().first()
good.tags.add("дом")
```

Здесь мы добавляем к первому товару тег `"дом"`.

```
good.tags.remove("предприятие", "учреждение")
```

Здесь удаляем из этого товара теги `"предприятие"` и `"учреждение"`.

```
good.tags.set("веник", "дом")
```

А здесь удаляем из этого товара все привязанные ранее теги и добавляем к нему теги "веник" и "дом".

И, разумеется, не забываем сохранить сделанные в записи изменения:

```
good.save()
```

Вывод тегов на экран

Последний этап — вывод тегов на экран и формирование на их основе гиперссылок, на которых посетитель будет щелкать, чтобы выполнить поиск по тегам.

Класс `TaggableManager` поддерживает не принимающий параметров метод `names`. Он возвращает список имен всех тегов, привязанных к данной записи. Поскольку этот метод не принимает параметров, мы можем использовать его в коде шаблонов:

```
{% with names=good.tags.names %}
  {% if names.count > 0 %}
    <p>Теги: {% for name in names %}{% if not forloop.first %},
    {% endif %}{{ name }}{% endfor %}</p>
  {% endif %}
{% endwith %}
```

Здесь мы выводим список имен привязанных к товару тегов, разделив их запятыми.

```
{% with names=good.tags.names %}
  {% if names.count > 0 %}
    <p>Теги: {% for name in names %}{% if not forloop.first %},
    {% endif %}<a href="{% url "index" cat_id=good.category.id %}↵
    ?tag={{ name|urlencode }}">{{ name }}</a>{% endfor %}</p>
  {% endif %}
{% endwith %}
```

А здесь формируем аналогичный список, каждый тег в котором представляет собой гиперссылку, указывающую на список товаров и передающую выводящему его контроллеру имя тега в GET-параметре `tag`. Отметим, что для вывода значения этого параметра мы используем фильтр шаблона `urlencode`, который преобразует выводимое значение к виду, пригодному для вставки в интернет-адрес (подробнее об этом фильтре говорилось в *главе 7*).

Конечно, список товаров — не лучший объект для испытания библиотеки django-taggit в действии, теги были бы намного уместнее в блоге. Но здесь мы лишь экспериментируем ради практики, а когда начнем разработку сайта, то обязательно учтем этот момент.

Администрирование списка тегов

И, наконец, библиотека django-taggit дает нам возможность просмотреть и, при необходимости, исправить список всех созданных нами тегов. Сделать это можно через хорошо знакомый нам встроенный административный сайт Django.

На главной странице этого сайта приложение `taggit` представляется таблицей **Taggit**, содержащей гиперссылку **Метки**. (Почему-то именно так разработчики Django решили назвать теги.) Щелчок на этой гиперссылке открывает список тегов.

Список тегов

Не забываем, что все теги, привязанные ко всем записям всех моделей всех приложений проекта, хранятся в одном списке.

Список тегов не несет для нас чего-либо нового. Поэтому сразу же щелкнем на каком-либо из тегов, чтобы перейти на страницу его правки (рис. 18.1) и, таким образом, просмотреть сведения о нем.

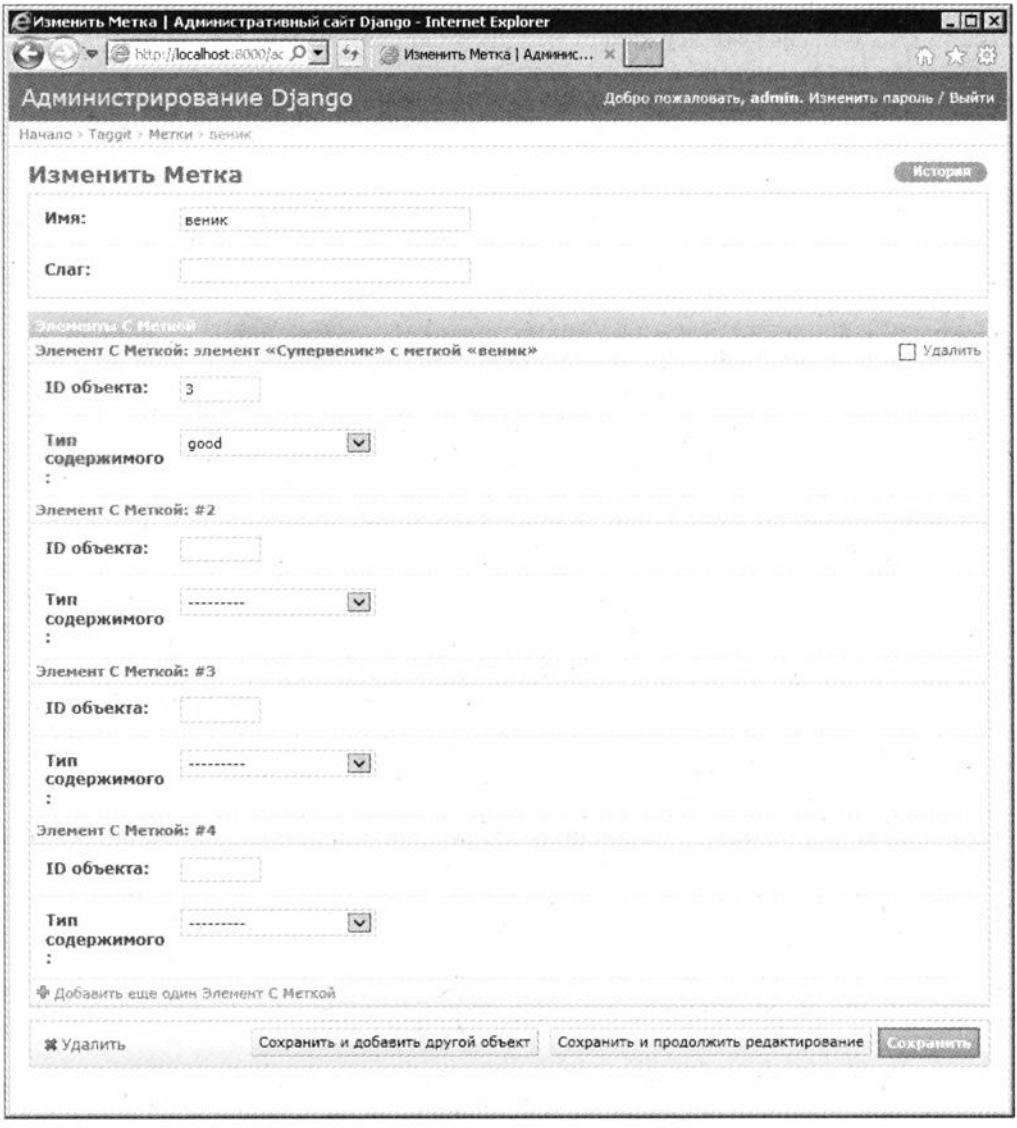


Рис. 18.1. Интерфейс для правки тега

Уже понятно, что поле ввода **Имя** служит для ввода имени тега.

В поле ввода **Слаг** заносится то же имя тега, но преобразованное к формату, подходящему для помещения в состав интернет-адреса. К огромному сожалению, для тегов, включающих символы кириллицы, это поле не всегда заполняется, поэтому для нас, пользующихся кириллической азбукой, совершенно бесполезно.

Группа элементов управления **Элементы с меткой** служит для просмотра и изменения списка записей, к которым привязан данный тег. Она включает ряд других групп, каждая из которых описывает одну привязанную запись и содержит:

- поле ввода **ID объекта** — задает идентификатор привязанной к тегу записи (значение ее свойства `pk` или `id`, за подробностями — к *главе 5*);
- раскрывающийся список **Тип содержимого** — устанавливает модель, к записи которой привязан данный тег.

В этом списке выводится минимум четыре подобных группы. Если к тегу привязано менее четырех записей, какие-то группы будут пустыми.

К сожалению, править список привязанных к тегу записей не очень удобно — нужно знать идентификатор записи, а получить его не так просто. Но мы, по крайней мере, сразу увидим, что к какому-то тегу не привязано ни одной записи, и поймем, что его следует удалить.

Для добавления, правки и удаления тегов пользователь должен обладать правами `taggit | Метка | Can add tag`, `taggit | Метка | Can change tag` и `taggit | Метка | Can delete tag` соответственно. Как назначить пользователю необходимые права, рассказывалось в *главе 14*.

Что дальше?

В этой главе мы делали возможной привязку тегов к опубликованным на нашем сайте позициям. Теги помогут посетителю быстро отыскать подобные позиции, не отрывая рук от мыши.

А в следующей главе мы познакомимся с дополнительной библиотекой django-precise-bbcode. Она позволит нам дать сайту поддержку кодов BBCode и, соответственно, возможность форматировать выводимый текст и даже вставлять в него изображения.

ГЛАВА 19

Форматирование текста с применением тегов BBCode. Библиотека django-precise-bbcode

В предыдущей главе мы занимались реализацией привязки тегов к записям моделей и поиском по тегам. В этом нам помогла дополнительная библиотека django-taggit.

В этой главе мы сделаем так, чтобы большие фрагменты текста, хранящиеся в моделях, выводились на Web-страницах отформатированными: с разбиением на абзацы, с выравниванием по центру или правому краю, с выделением полужирным шрифтом и курсивом, с графическими изображениями, вставленными прямо в текст. Для этого мы воспользуемся тегами BBCode и библиотекой django-precise-bbcode, которая их поддерживает.

Как Web-обозреватель форматирует текст при выводе

Описания товаров и содержимое новостей на нашем сайте представляют собой большие фрагменты текста, хранящиеся в полях типа memo базы данных, представленных в моделях классом `TextField`.

Но при выводе таких фрагментов текста мы обязательно столкнемся с проблемой. Заключается она в том, что эти фрагменты выводятся как сплошные блоки текста, фактически одной длинной строкой. (Разумеется, если строка не помещается в отведенное для нее место, Web-обозреватель автоматически выполнит переносы.) Мы можем при вводе разбить такой фрагмент на абзацы, а можем и не разбивать, — результат будет одним и тем же.

Почему так происходит? Дело в том, что Web-обозреватель при выводе текста, являющегося содержимым какого-либо тега, например, абзаца (`<p>`) или заголовка (`<h1>`…`<h6>`), следует следующим правилам:

- последовательно следующие друг за другом пробелы преобразуются в один пробел;

- символы возврата каретки и перевода строки (разрывы строк) преобразуются в пробел;
- все содержимое тега выводится в виде строки;
- если строка не помещается в отведенное для нее место на странице, автоматически выполняются переносы по пробелам и символам дефиса.

Так что, даже если мы разобьем текст на абзацы, поставив в нужные места символы возврата каретки и перевода строки, Web-обозреватель при выводе все равно преобразует их в пробелы. В результате текст, опять же, будет выведен в одну строку...

Но сплошной блок текста, выведенный таким образом, очень плохо читается. Есть ли возможность хотя бы разбить его на абзацы?

Можно, конечно, вводить в поля не сам текст, а формирующий его HTML-код. Тогда мы сможем с помощью различных HTML-тегов отформатировать текст как пожелаем: разбить его на абзацы, задать выравнивание по центру для заголовков, выделить важные фрагменты полужирным шрифтом и курсивом и даже вставить в текст изображения и создать таблицы. (Кстати, при создании статичных страниц, описанных в *главе 16*, используется именно такой подход.)

Однако здесь могут возникнуть две серьезные проблемы:

- пользователь, вводящий такой текст, может допустить ошибку в HTML-коде: забыть поставить закрывающий тег, нарушить последовательность открывающих и закрывающих тегов и др. В результате HTML-код может быть обработан некорректно (хотя современные Web-обозреватели весьма «снисходительны» к мелким ошибкам подобного рода), текст отобразится на экране не так, как задумывалось, а в особо сложных случаях нарушится дизайн страницы;
- в число зарегистрированных пользователей, имеющих права на ввод и правку внутренних данных сайта, может попасть злоумышленник, который вставит в HTML-код фрагмент, выполняющий недопустимые действия, — например, вызывающий Web-сценарий с вирусным кодом.

Конечно, в случае статичных страниц, которые обычно создает и правит один-единственный пользователь — разработчик сайта, такой вариант будет приемлем. Но не для работы с данными сайта — ведь эту задачу могут выполнять сразу несколько пользователей, которых труднее проверить на благонадежность.

Теги BBCode

Поэтому в большинстве случаев ныне используют другой подход. Текст форматируют с применением особых тегов, не являющихся тегами HTML. А контроллер, который выводит страницы, преобразует эти теги в соответствующий им HTML-код.

В настоящее время сложился своего рода стандарт де-факто, описывающий набор таких тегов. Он носит имя *BBCode* (Bulletin Board Code, код досок объявлений), а

теги, применяемые для форматирования текста согласно этому стандарту, — *тегами BBCode*.

> **Стандарт BBCode**
>
> В реальности стандарт BBCode описывает лишь принципы написания тегов и основной их набор, поддержка которого обязательна в любом случае. Однако разные сайты и разные библиотеки, предназначенные для создания сайтов, обычно поддерживают расширенный набор таких тегов. Так что по поводу поддержки BBCode следует в любом случае обращаться к документации по конкретному сайту и конкретной библиотеке.

Теги BBCode записываются так же, как и теги HTML, за тем исключением, что вместо символов < и > для их выделения применяются квадратные скобки ([и]). Теги BBCode могут быть одинарными и двойными — в последнем случае между ними помещается содержимое тега. Вероятно, единственное их серьезное отличие от HTML-тегов заключается в том, что они никогда не имеют атрибутов.

Основной набор тегов BBCode включает парные теги `[b]` и `[i]`. Первый выделяет текст полужирным шрифтом, а второй — курсивом:

> `[b]Этот текст[/b]` будет выделен **полужирным шрифтом**,
> а `[i]этот[/i]`— *курсивом*.

Парный тег `[quote]` служит для создания цитат, взятых из сообщений других пользователей, и обычно применяется в блогах и форумах:

```
[quote]Пользователь User34 писал: "Если не хотите иметь проблем с Python,
сразу устанавливайте библиотеку setuptools".[/quote]
Спасибо! Так и сделаю.
```

Парный тег `[img]` используется для вставки в текст изображений. Интернет-адрес файла изображения указывается в качестве содержимого этого тега:

```
[img]/images/logo.png[/img]
```

Теги BBCode могут принимать параметры. Одновременно тег может получить всего один параметр, который указывается в открывающем теге после его имени через символ равенства (=).

Так, тег `[url]`, используемый для вставки в текст гиперссылок, принимает в качестве параметра интернет-адрес:

```
Посмотрите на [url=http://www.python.org/]домашнем сайте языка
Python[/url].
```

Помимо этого, любой текст, даже если он не содержит тегов BBCode, автоматически разбивается на абзацы в тех местах, где находятся символы возврата каретки и перевода строки. Это выполняется самим контроллером, выводящим страницу, в обязательном порядке.

Библиотека django-precise-bbcode

Дополнительная библиотека django-precise-bbcode призвана дать поддержку BBCode сайтам, написанным на Python и Django.

Введение в библиотеку django-precise-bbcode

Эта библиотека включает в себя:

- особый класс поля модели, предназначенный для хранения текста, который отформатирован с применением тегов BBCode;
- набор тегов и фильтров шаблонов, которые можно применить для вывода текста, отформатированного по правилам BBCode;
- административные инструменты, с помощью которых можно управлять набором поддерживаемых тегов BBCode и даже создавать свои собственные теги;
- встроенную поддержку смайликов (правда, чтобы ее активизировать, придется немного потрудиться).

> ***Обработка смайликов***
>
> Для успешной обработки смайликов требуется установить также библиотеку Pillow. Установочный пакет этой библиотеки под Windows можно найти по интернет-адресу **http://www.lfd.uci.edu/~gohlke/pythonlibs/#pillow**.

Установочный комплект библиотеки django-precise-bbcode можно найти по интернет-адресу **https://pypi.python.org/pypi/django-precise-bbcode/**, а ее описание — по интернет-адресу **http://django-precise-bbcode.readthedocs.org/en/latest/**.

Теги BBCode, поддерживаемые django-precise-bbcode

Рассмотрим набор тегов BBCode, которые поддерживаются этой библиотекой. Все они перечислены в табл. 19.1.

Таблица 19.1. *Теги BBCode, поддерживаемые библиотекой django-precise-bbcode*

Тег	Описание
[b]*текст*[/b]	Выводит *текст* **полужирным шрифтом**
[i]*текст*[/i]	Выводит *текст* *курсивом*
[u]*текст*[/u]	Выводит *текст* подчеркнутым
[s]*текст*[/s]	Выводит *текст* ~~зачеркнутым~~
[list]*пункты*[/list]	Создает маркированный список
[list=*нумерация*]*пункты*[/list]	Создает нумерованный список
[*]*текст*	Создает пункт списка на основе указанного в параметре *текст*
[code]*текст*[/code]	Выводит *текст* с сохранением форматирования

Таблица 19.1 (окончание)

Тег	Описание
[quote]*<текст>*[/quote]	Выводит *текст* как цитату
[center]*<текст>*[/center]	Выравнивает *текст* по центру
[color=*<цвет>*]*<текст>*[/color]	Окрашивает *текст* в указанный *цвет*
[url]*<интернет-адрес>*[/url]	Выводит *интернет-адрес*, формируя на его основе гиперссылку
[url=*<интернет-адрес>*]*<текст>*[/url]	Выводит текст в виде гиперссылки с заданным интернет-адресом
[img]*<интернет-адрес>*[/img]	Выводит графическое изображение, файл которого хранится по заданному интернет-адресу

Многие из этих тегов нам уже знакомы. Но некоторые требуют дополнительного пояснения.

В парном теге [color] цвет можно указать либо в виде принятого в CSS наименования, либо в виде RGB-кода:

```
[color=red]Красный текст[/color]
[color=#00FF00]Зеленый текст[/color]
```

Парный тег [list] служит для создания маркированных списков. Набор пунктов помещается внутрь этого тега, а каждый пункт должен предваряться тегом [*]:

```
[list]
  [*]Python;
  [*]Django;
  [*]setuptools;
  [*]Pillow;
  [*]gjango-precise-bbcode.
[/list]
```

Здесь мы создаем маркированный список из пяти пунктов.

Аналогичный ему парный тег [list=*<нумерация>*] используется для создания нумерованных списков. Поддерживаются следующие виды нумерации и соответствующие им значения параметра этого тега:

- 1 — арабские цифры;
- 01 — арабские цифры с начальным нулем;
- i — римские цифры, набранные строчными буквами;
- I — римские цифры, набранные прописными буквами;
- a — строчные латинские буквы;
- A — прописные латинские буквы.

Вот пример:

```
[list=I]
  [*]модель;
  [*]контроллер;
  [*]шаблон.
 [/list]
```

Здесь мы создаем список из трех пунктов, нумерованных римскими цифрами, набранными прописными буквами.

Настройка проекта

Перед тем как начать «знакомить» наш сайт с тегами BBCode, нам следует внести изменения в настройки проекта. Они, как мы уже давно знаем, хранятся в модуле `settings` пакета проекта.

Базовые настройки

Ключевые настройки заключаются лишь во внесении приложения `precise_bbcode`, которое и обеспечивает поддержку форматирования BBCode, в список активных приложений:

```
INSTALLED_APPS = (
  . . .
  'precise_bbcode',
)
```

И обязательно выполним синхронизацию с базой данных, чтобы Django создала в ней все потребные для нормальной работы приложения `precise_bbcode` таблицы.

Настройки библиотеки django-precise-bbcode

Помимо того, мы можем указать настройки для самой библиотеки django-precise-bbcode. Они не являются обязательными и затрагивают некоторые нюансы в функционировании этой библиотеки.

Наиболее полезные для нас настройки и соответствующие им переменные приведены в табл. 19.2. Их немного.

По умолчанию библиотека django-precise-bbcode вместо обычных текстовых смайликов будет выводить графические изображения, которые указаны для этих смайликов в особом списке (что это за список, мы узнаем в конце главы). Чтобы отключить эту функцию, достаточно присвоить переменной `BBCODE_ALLOW_SMILIES` значение `False`.

Переменная `SMILIES_UPLOAD_TO` указывает имя папки, в которой хранятся файлы графических смайликов (по умолчанию — precise_bbcode\smilies). Эта папка будет автоматически создана в папке, где хранятся выгруженные файлы (подробнее — в *главе 13*), так что нам делать этого не придется.

Таблица 19.2. Настройки библиотеки django-precise-bbcode и переменные, в которых они указываются

Переменная	Описание	Значение по умолчанию
BBCODE_NEWLINE	Задает HTML-тег для разбиения текста на абзацы в виде строки	" "
BBCODE_ESCAPE_HTML	Задает список недопустимых символов, которые должны быть заменены соответствующими им литералами HTML. Указывается в виде кортежа, каждый элемент которого задает замену для одного недопустимого символа и также должен представлять собой кортеж из двух элементов-строк: собственно недопустимого символа и заменяющего его литерала	(('&', '&'), ('<', '<'), ('>', '>'), ('"', '"'), ('\'', '''),)
BBCODE_ALLOW_SMILIES	Если True, смайлики будут обрабатываться	True
SMILIES_UPLOAD_TO	Задает папку, в которой хранятся файлы графических смайликов	"precise_bbcode/smilies"

Реализация поддержки BBCode

Библиотека django-precise-bbcode предлагает нам два способа дать приложениям нашего сайта поддержку BBCode:

- хранение отформатированного с помощью тегов BBCode текста в поле модели, принадлежащем особому классу;
- использование для обработки тегов BBCode специальных тега и фильтра шаблона.

Мы можем также использовать встроенный в эту библиотеку класс форматировщика, чтобы обработать теги BBCode в коде контроллера.

Использование класса поля *BBCodeTextField*

Проще всего, вероятно, сразу создать в модели поле, в котором будет храниться текст, форматируемый с помощью BBCode-тегов. Для этого в библиотеке django-precise-bbcode предусмотрен класс поля модели `BBCodeTextField`, объявленный в модуле `precise_bbcode.fields`. Этот класс является потомком класса `TextField` и поддерживает те же параметры конструктора, что и родитель (см. *главу 5*).

```
from precise_bbcode.fields import BBCodeTextField
class New(models.Model):
  title = models.CharField(max_length = 100, db_index = True)
  description = models.TextField()
  content = BBCodeTextField()
  pub_date = models.DateField(db_index = True, auto_now_add = True)
```

Здесь мы пишем новую модель для хранения новостей. Поле `content`, где будет храниться содержимое новости, мы создаем на основе класса `BBCodeTextField`.

Этот класс поддерживает также и свойство `rendered`. Оно возвращает хранящийся в поле текст, уже обработанный и представляющий собой HTML-код. Мы можем использовать это свойство как в коде контроллера, так и в шаблонах.

> ***Дополнительное поле, хранящее обработанный текст***
>
> Класс `BBCodeTextField` создает в таблице дополнительное поле с именем вида `_<имя поля>_rendered`, помеченное как нередактируемое и, соответственно, не выводящееся на экран. В этом поле хранится собственно уже обработанный и преобразованный в HTML-код текст, взятый из основного поля. При обращении к свойству `rendered` упомянутого ранее класса будет возвращено как раз содержимое дополнительного поля.

Вот примеры:

```
first_new = New.objects.first()
html_content = first_new.content.rendered
```

Здесь мы получаем обработанное содержимое первой новости в списке.

```
<div>{{ new.content.rendered|safe }}</div>
```

А здесь выводим обработанное содержимое в шаблоне.

Отметим, что в шаблоне для вывода уже обработанного текста, представляющего собой HTML-код, мы задействуем фильтр шаблона `safe`, описанный в *главе 7*. Этот фильтр отключает замену недопустимых в HTML-коде символов литералами. Если мы его не укажем, то на страницу будет выведен сам HTML-код, полученный в результате обработки текста, что нам совсем не нужно.

> ***Ошибка библиотеки django-precise-bbcode версии 0.4.2***
>
> К сожалению, в версии 0.4.2 библиотеки django-precise-bbcode, которую использовал автор, присутствует ошибка, которая может привести к потере значения, хранящегося в поле класса `BBCodeTextField`, при его правке. Вследствие этого в настоящее время лучше не пользоваться этим классом, а форматируемый текст хранить в обычных полях класса `TextField`.

Использование тега шаблона *bbcode* и фильтра *bbcode*

Если же мы по каким-либо причинам не можем воспользоваться классом поля `BBCodeTextField`, не беда. Мы можем сохранить форматированный текст в поле другого класса, а потом выполнить его обработку прямо в шаблоне, применив тег `bbcode` или одноименный фильтр.

Сначала мы загрузим модуль `bbcode_tags` шаблонизатора с помощью тега:

```
{% load bbcode_tags %}
```

Этот тег должен присутствовать в коде шаблона перед первым использованием тега или фильтра `bbcode`.

Тег bbcode записывается в таком формате:

```
{% bbcode <выводимая строка> %}
```

и выводит строку, указанную параметром *выводимая строка*, в том месте, где находится сам:

```
<h2>{% bbcode new.content %}</h2>
```

Здесь мы обрабатываем и выводим содержимое новости.

Фильтр bbcode использовать еще проще:

```
<h2>{{ new.content|bbcode|safe }}</h2>
```

Здесь мы тоже не забываем указать фильтр safe, иначе получим на странице сам HTML-код, а не отформатированный текст.

Использование программного форматировщика

Для тех редких случаев, когда нам может понадобиться выполнить обработку строки с BBCode-тегами в коде контроллера, библиотека django-precise-bbcode предусматривает особый класс форматировщика. Объект этого класса можно получить, вызвав не принимающий параметров метод get_parser, объявленный в модуле precise_bbcode.parser:

```
from precise_bbcode.parser import get_parser
parser = get_parser()
```

Этот класс поддерживает метод render. Он принимает в качестве параметра строку с текстом, который следует обработать, и возвращает строку с готовым HTML-кодом:

```
first_new = New.objects.first()
html_content = parser(first_new.content)
```

Какими HTML-тегами заменяются теги BBCode?

В некоторых случаях требуется знать, какими HTML-тегами библиотека django-precise-bbcode заменяет теги BBCode. Зная эти теги, мы можем предпринять необходимые меры, чтобы предотвратить искажение дизайна страниц нашего сайта.

Список тегов BBCode и HTML-тегов, которые их заменят в результате обработки, приведен в табл. 19.3.

***Таблица 19.3.** Теги BBCode и заменяющие их HTML-теги*

Тег BBCode	Заменяющий его HTML-тег
[b]*текст*[/b]	*текст*
[i]*текст*[/i]	*текст*
[u]*текст*[/u]	<u>*текст*</u>

Таблица 19.3 (окончание)

Тег BBCode	Заменяющий его HTML-тег
[s]<текст>[/s]	<strike><текст></strike>
[list]<пункты>[/list]	<пункты>
[list=1]<пункты>[/list]	<пункты>
[list=01]<пункты>[/list]	<пункты>
[list=a]<пункты>[/list]	<пункты>
[list=A]<пункты>[/list]	<пункты>
[list=i]<пункты>[/list]	<пункты>
[list=I]<пункты>[/list]	<пункты>
[*]<текст>	<текст>
[quote]<текст>[/quote]	<blockquote><текст></blockquote>
[code]<текст>[/code]	<code><текст></code>
[center]<текст>[/center]	<текст>
[color=<цвет>]<текст>[/color]	<span style="color: <цвет>;"><текст>
[url]<интернет-адрес>[/url]	<a href="<интернет-адрес>"><интернет-адрес>
[url=<интернет-адрес>]<текст>[/url]	<a href="<интернет-адрес>"><текст>
[img]<интернет-адрес>[/img]	<img src="<интернет-адрес>" alt="">

Создание собственных тегов BBCode

Одна из замечательных возможностей библиотеки django-precise-bbcode — ее расширяемость. Если нам не хватает стандартного набора поддерживаемых ею тегов BBCode, мы с легкостью можем добавить в него свои теги. И самое интересное в том, что для этого нам не придется писать никакого программного кода.

Добавление новых тегов выполняется через встроенный административный сайт Django. На его главной странице найдем таблицу с заголовком **Precise_Bbcode** (так

там представляется приложение `presize_bbcode` — ключевая часть этой библиотеки) и щелкнем на имеющейся в ней гиперссылке **BBCode tags**.

Изначально список тегов BBCode, созданных самим разработчиком сайта, пуст, что и неудивительно. Так что сразу нажмем кнопку добавления записи.

Страница, предназначенная для ввода параметров создаваемого BBCode-тега, показана на рис. 19.1.

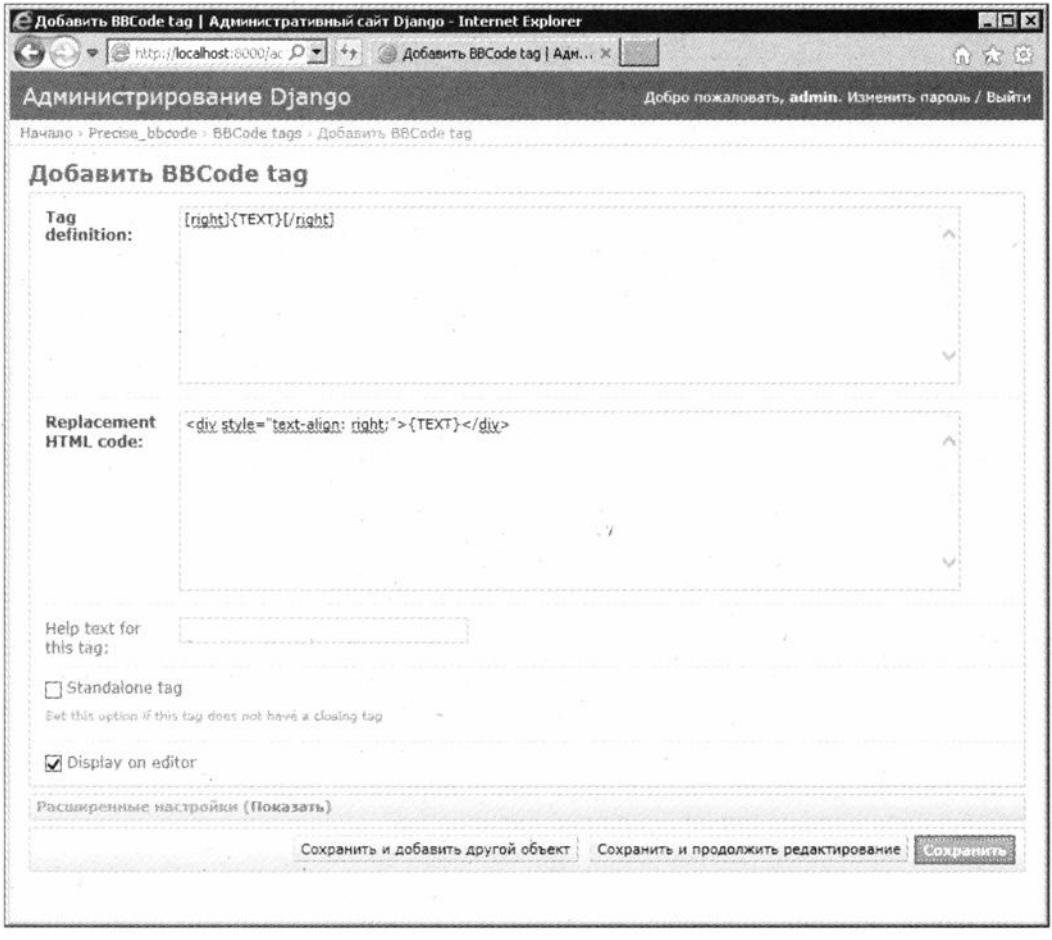


Рис. 19.1. Страница создания нового тега BBCode

В области редактирования **Tag definition** вводится код создаваемого BBCode-тега. Чем-то он схож регулярным выражением (см. *главу 6*): мы указываем сам тег и используем особые литералы, чтобы задать содержимое этого тега и значение параметра, если, конечно, наш тег его принимает. Вот перечень доступных для использования литералов:

- `{TEXT}` — совпадает с любым текстом, который может содержать любые символы. Обычно служит для указания содержимого парных тегов;

- {URL} — совпадает с интернет-адресом;
- {EMAIL} — совпадает с адресом электронной почты;
- {COLOR} — совпадает с обозначением цвета. Можно использовать как наименования цветов, принятые в CSS (red, blue, black и др.), так и коды цветов в формате RGB, которые также надлежит указывать в формате, принятом в CSS (например, #FF0000);
- {NUMBER} — совпадает с числом;
- {SIMPLETEXT} — совпадает с текстом, содержащим лишь латинские буквы, цифры, пробелы, точки, запятые, дефисы и символы подчеркивания, плюса и минуса.

Скажем, если мы хотим создать парный тег [right], который выведет содержимое в виде абзаца, выровненного по правому краю, мы запишем его код вот так:

```
[right]{TEXT}[/right]
```

А для парного тега [size], задающего размер шрифта, этот код будет таким:

```
[size=(NUMBER)]{TEXT}[/size]
```

Здесь параметр тега задает размер шрифта в пикселах.

В области редактирования **Replacement HTML code** указывается HTML-код, которым будет заменен создаваемый нами тег BBCode. Здесь для указания содержимого тега и значений его атрибутов также применяются рассмотренные нами ранее литералы.

Например, для тега [right] мы укажем такой HTML-код замены:

```
<div style="text-align: right;">{TEXT}</div>
```

А для тега [size] — такой код:

```
<span style="font-size: {NUMBER}px;">{TEXT}</span>
```

Если мы создаем одинарный тег, то обязательно установим флажок **Standalone tag**. Для парных тегов его следует держать сброшенным.

> ***Особенности текущей реализации библиотеки***
>
> Поле ввода **Help text for this tag** в текущей реализации библиотеки django-precise-bbcode не используется. А установленный по умолчанию флажок **Display on editor** лучше не сбрасывать.

В раскрывающейся панели **Расширенные настройки** находятся элементы управления, задающие дополнительные параметры создаваемого тега. Они приведены на рис. 19.2.

Здесь мы видим набор флажков. Все они перечислены в табл. 19.4.

После указания всех параметров нового тега их следует сохранить. После чего мы сразу же сможем использовать этот тег для форматирования текста.

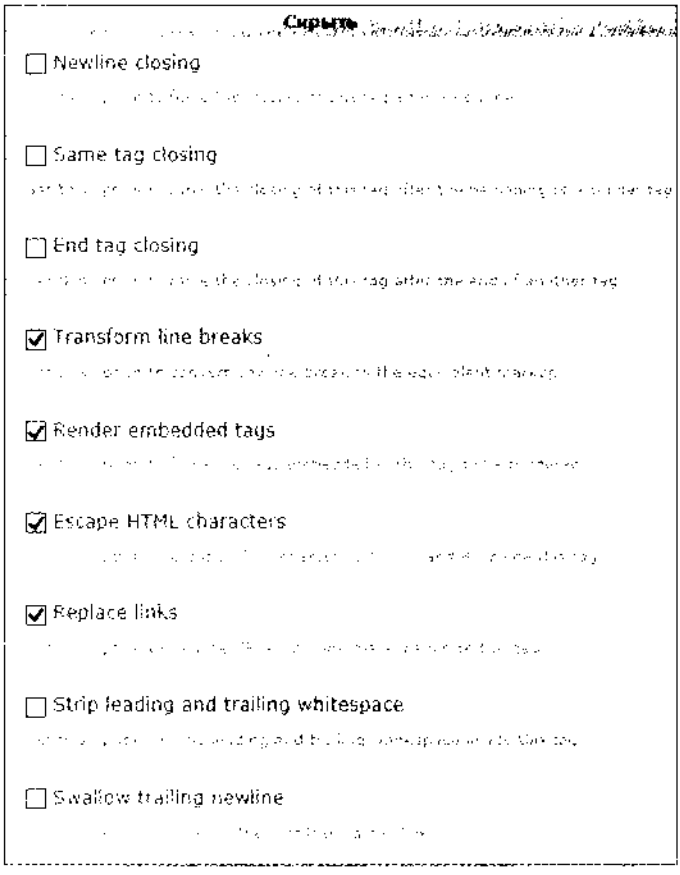


Рис. 19.2. Раскрывающаяся панель **Расширенные настройки**

Таблица 19.4. *Дополнительные параметры создаваемого тега и представляющие их флажки*

Флажок	Описание	Установлен по умолчанию
Newline closing	Закрывать тег перед ближайшим разрывом строк	
Same tag closing	Закрывать тег перед началом такого же тега	
End tag closing	Закрывать тег после окончания другого тега	
Transform line breaks	Обрабатывать присутствующие в содержимом тега разрывы строк, преобразуя их в соответствующий HTML-код	+
Render embedded tags	Обрабатывать BBCode-теги, присутствующие в содержимом этого тега	+
Escape HTML characters	Заменять все недопустимые символы в содержимом тега соответствующими литералами HTML	+
Replace links	Преобразовывать интернет-адреса, присутствующие в содержимом тега, в гиперссылки	+

Таблица 19.4 (окончание)

Флажок	Описание	Установлен по умолчанию
Strip leading and trailing whitespaces	Удалять начальные и конечные пробелы в содержимом тега	
Swallow trailing newline	Удалять конечный разрыв строк	

Добавление поддержки смайликов

Хоть библиотека django-precise-bbcode и поддерживает вывод графических смайликов вместо их текстового представления, но изначально она не включает в свой состав ни самих файлов изображений со смайликами, ни какого-либо списка соответствий текстовых представлений этим файлам. Нам самим придется найти первые и задать вторые.

Сначала нужно подобрать подходящие к стилистике нашего сайта изображения смайликов. Автор рекомендует наведаться на сайт **http://smayli.ru/**, где можно найти смайлики на все случаи жизни.

Далее мы снова откроем встроенный административный сайт, найдем на его главной странице в таблице **Precise_Bbcode** гиперссылку **Smilies**, ведущую на список уже созданных смайликов, и щелкнем на ней.

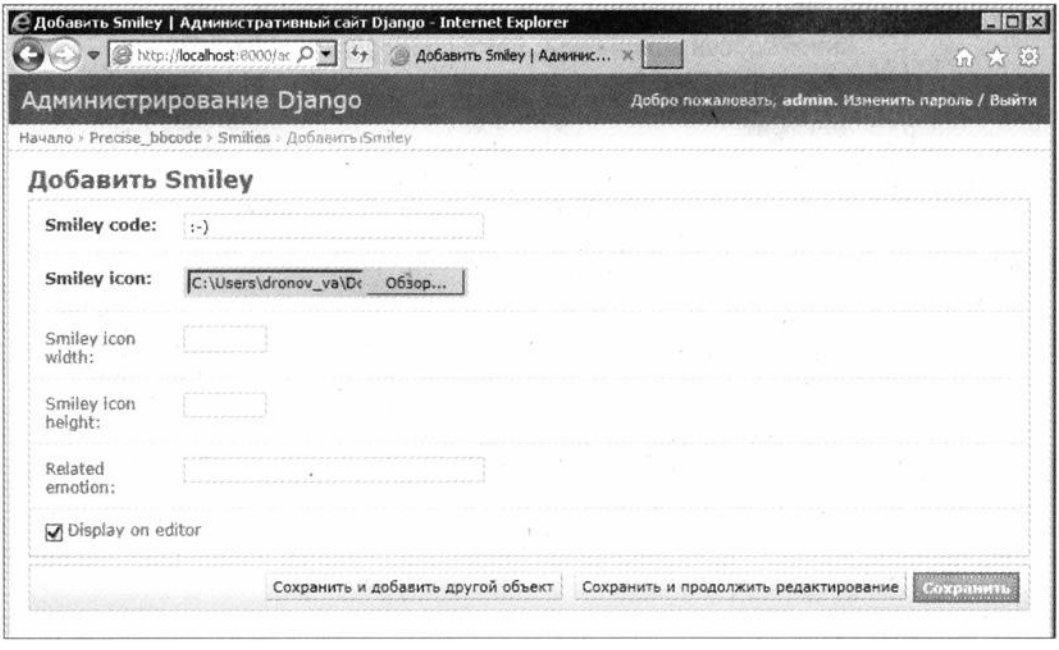


Рис. 19.3. Страница для добавления нового смайлика

Как и следовало ожидать, изначально список смайликов пуст. Поэтому сразу же нажмем гиперссылку добавления новой записи, после чего перейдем на соответствующую страницу (рис. 19.3).

В поле ввода **Smiley code** вводится текстовое представление смайлика: :-), ;-), :D и пр. А в поле ввода файла **Smiley icon** указывается файл с его графическим изображением.

При желании мы можем указать ширину и высоту изображения смайлика в полях ввода **Smiley icon width** и **Smiley icon height** соответственно. Но, если изображение смайлика уже имеет подходящие размеры, делать это не нужно.

> ***Особенности текущей реализации библиотеки***
>
> Поле ввода **Related emotion** в текущей реализации библиотеки django-precise-bbcode не используется. А установленный по умолчанию флажок **Display on editor** лучше не сбрасывать.

Останется лишь сохранить введенные нами данные — и смайлик готов!

Что дальше?

В этой главе мы занимались поддержкой форматирования текста по правилам BBCode. И вполне преуспели в этом — теперь пользователи нашего сайта смогут красиво форматировать содержимое новостей.

Уф! Наконец-то теоретическая часть и практические занятия, призванные закрепить полученные знания, закончились. Настала пора реальной практики, создания настоящего сайта. Чем мы и займемся в следующей части книги.

ЧАСТЬ VI

Создание Web-сайта

ГЛАВА 20

Планирование и предварительные действия

В предыдущих частях книги мы занимались теорией и выполняли своего рода лабораторные работы, чтобы закрепить полученные знания. Но теперь теория кончилась, и наступает настоящая работа.

Мы создадим полностью работоспособный Web-сайт гипотетической фирмы «Веник-Торг», торгующей вениками, метлами, щетками и всем подобным. Этот сайт включит в свой состав список товаров, разбитый на категории, блог, гостевую книгу, список новостей и вспомогательные страницы: список контактов, сведения о самой фирме и данные о способах оплаты и вывоза приобретенного товара. Разумеется, он будет содержать и главную страницу, где, в том числе, будут выводиться перечень рекомендуемых товаров и три последние новости.

На примере этого сайта мы рассмотрим более сложные приемы Django-программирования — в частности, ввод связанных записей одновременно с родительской, реализацию поиска по введенному ключевому слову и обмен данными по технологии AJAX. И, разумеется, подробно рассмотрим работу каждой модели, каждого контроллера и каждого шаблона, что мы напишем.

Планирование сайта

Но сначала мы займемся планированием нашего будущего сайта. В самом деле, прежде чем что-то делать, необходимо твердо уяснить, что же мы хотим получить в результате. И спланировать это нужно в самом начале, перед тем, как приниматься за дело. Ведь когда работа сделана, менять что-либо значительно труднее, чем сразу делать все как надо.

Основные этапы планирования сайта

Итак, о чем нужно подумать перед началом работы над сайтом?

1. Прежде всего, четко определить назначение сайта. Что он должен делать: рассказывать о ком-либо или о чем-либо, привлекать клиентов, помогать посетите-

лям решать какие-то проблемы, просвещать или развлекать их. В зависимости от предполагаемого назначения сайта его структура может сильно различаться.

2. Определить, какая конкретно информация должна присутствовать на сайте. Здесь главный принцип — ничего лишнего, только то, что действительно нужно потенциальным посетителям.
3. Собрать всю необходимую информацию. И сделать это прямо сейчас, чтобы потом не работать по принципу «хватай мешки — вокзал тронулся». Все тексты, изображения, файлы, которые мы намерены выложить в Сеть, уже должны быть на нашем компьютере.
4. Продумать *логическую структуру сайта* — список входящих в его состав страниц — и, желательно, нарисовать ее. Здесь лучше не изобретать самому велосипед, а посетить какой-нибудь уже существующий и популярный Web-сайт сходной тематики и посмотреть, как он организован. Например, для сайта фирмы идеальна такая структура:
   - главная страница с краткими сведениями о самой фирме, списком рекомендуемых товаров и самыми последними новостями;
   - полный перечень предлагаемых товаров с разбиением по категориям и возможностью просмотра сведений о выбранном товаре;
   - список новостей, относящихся к фирме и товарам, с возможностью просмотра более полного содержимого выбранной новости;
   - блог, где сотрудники фирмы будут писать статьи о продаваемых фирмой товарах, особенностях их эксплуатации, сведениях о гарантийной поддержке, проведенных корпоративных вечеринках и пр.;
   - гостевая книга, где покупатели смогут оставить отзывы о фирме и товарах;
   - прочие страницы (контакты, данные о фирме, сведения о сайте и его разработчиках и др.).
5. Придумать *физическую структуру сайта*, т. е. как будут называться отдельные приложения, входящие в состав сайта, где будут храниться статичные файлы и файлы, выгруженные пользователями, и как они будут организованы.
6. Решить вопрос с административным разделом сайта. Будет ли для работы с внутренними данными сайта использован встроенный административный сайт Django? Или мы создадим для этого свой административный раздел, включающий страницы для добавления, правки и удаления категорий, товаров, новостей и записей гостевой книги? (Это не касается блога — для работы с его статьями нам, так или иначе, придется создавать специальные страницы.) А может быть, мы применим смешанный подход: для работы с часто изменяемыми данными (например, товарами и новостями) мы создадим специальные страницы, а для манипулирования редко изменяемыми (пользователями и группами) задействуем возможности встроенного административного сайта?
7. Решить, в каком ключе будет выполнен дизайн сайта. Будет ли он консервативным, строгим или затейливым. Соответственно, домашняя страничка должна отражать эстетические наклонности автора, развлекательный сайт лучше сделать

повеселее, а новостной — поскромнее, чтобы пестрота дизайна не заслоняла главное — информацию. На этом этапе лучше всего будет набросать на бумаге, как должна выглядеть та или иная страница.

8. Проверить, ничего ли мы не забыли. Это последний этап планирования сайта, но не менее важный, чем остальные.

Это что касается собственно планирования сайта. Теперь поговорим о его структуре — логической и физической — и административном разделе, а также скажем пару слов о дизайне.

Логическая структура Web-сайта

Как мы только что выяснили, логическая структура сайта определяет состав входящих в него приложений и их функциональность, выполняемые ими задачи. Над ней нужно подумать как следует.

Django не оставляет нам особого выбора, настоятельно рекомендуя создавать отдельное приложение на каждый раздел сайта. Так, для обработки списка категорий, на которые будут разбиваться товары, — его вывода и правки — удобнее создать отдельное приложение. (Хотя никто не мешает нам перенести функциональность по выводу категорий на приложение, которое выводит список товаров, как мы сделали ранее, в предыдущих главах.) А для обработки списка новостей — их вывода, добавления, правки и удаления — отдельное приложение следует создать однозначно.

Теперь что касается функциональности приложений. Понятно, что одно приложение проекта станет выполнять сразу несколько разных задач. Приложение для работы с категориями будет и формировать их список, и реализовывать его правку. Приложение для работы с товарами будет как выводить список товаров, так и предоставлять посетителю сведения о выбранном товаре, ну, и, безусловно, реализует их добавление, правку и удаление. То же самое касается и приложения, обеспечивающего работу блога.

А еще каждое приложение будет включать в свой состав модель, где сохранятся обрабатываемые им данные. Так, приложение списка категорий задаст модель категории, приложение списка товаров — модель товара и т. д.

Далее нам следует определиться с тем, как будут формироваться вспомогательные страницы сайта:

- мы можем задействовать для их вывода инфраструктуру статичных страниц Django, избавив себя тем самым от необходимости в написании для них контроллеров и шаблонов. Но этот подход пригоден лишь для случаев, когда на таких страницах не нужно выводить данные, хранящиеся в моделях (а ведь у нас на каждой странице присутствует панель навигации, включающая, в том числе, и список категорий, который как раз берется из модели);
- можно выводить эти страницы и традиционным путем — с применением контроллера и шаблона. Тогда мы сможем включить в их состав и данные, хранящиеся в моделях. Пожалуй, для нашего случая это единственный вариант.

И, наконец, нам понадобится решить, что будет выводиться на каждой странице сайта. Понятно, что на странице списка товаров будет выводиться список товаров, а на странице блога — перечень входящих в его состав статей. Но что должно присутствовать на главной странице и на странице гостевой книги? Итак, примем, что:

- на главной странице сайта фирмы обычно выводят список рекомендуемых товаров, несколько самых «свежих» новостей и, возможно, последнюю запись гостевой книги («последний отзыв»);
- на странице гостевой книги, помимо списка уже оставленных в ней записей, должна также присутствовать форма для добавления новой записи;
- страницы блога также будут иметь форму для поиска статей;
- если фирма продает большое количество разнообразных товаров, стоит предусмотреть на странице списка товаров и форму для поиска по ним. Такой поиск должен быть сквозным, т. е. выполняться по товарам, относящимся к разным категориям;
- на страницах сведений о товаре, содержимого отдельной новости и содержимого отдельной статьи блога нужно предусмотреть гиперссылку для возврата на соответствующую страницу списка, при этом учитывая пагинацию.

Повторимся: все это желательно решить прямо сейчас, чтобы потом не переделывать уже готовый и работающий сайт по нескольку раз.

Физическая структура Web-сайта

Физическая структура сайта описывает:

- имена входящих в его состав приложений;
- уровень, на котором будут размещены статичные файлы и шаблоны, — проекта или приложений;
- место хранения базы данных;
- структуру папок, в которых будут храниться статичные и выгруженные файлы;
- способы передачи параметров в контроллеры (непосредственно в составе интернет-адреса или через GET-параметры).

С именами приложений все ясно. Они должны быть как можно более короткими и четко объяснять назначения приложений. Хороший пример имени приложения: `goods` — мы сразу поймем, что данное приложение «отвечает» за вывод и обработку списка товаров.

Статичные файлы, по мнению автора, лучше поместить на уровень проекта. Во-первых, все равно бóльшая часть статичных файлов применяются во всех страницах, входящих в сайт. А во-вторых, удобнее хранить все эти файлы в отдельном месте, чем разбрасывать их по отдельным папкам.

С файлами базовых шаблонов, описывающих общие для всех страниц элементы, также все ясно — их следует поместить на уровень проекта. Что касается шаблонов, относящихся к отдельным страницам, то их можно как оставить на уровне

приложений, так и вынести на уровень проекта. Последний вариант удобнее — так все файлы шаблонов, опять же, будут сведены в одно место.

Базу данных можно хранить как непосредственно в папке проекта (как это сделали мы ранее), так и в специально созданной папке. Эта папка может находиться как в папке проекта, так и вне ее.

Теперь о структуре папок, предназначенной для организации статичных файлов. Понятно, что хранить уйму файлов в одной папке не очень удобно — если нам понадобится найти какой-либо файл, чтобы его заменить или исправить, нам придется искать его среди десятков, а то и сотен его «соседей». Так что разнородные файлы часто помещают в разные папки: таблицы стилей — в одну, графические изображения — в другую, файлы Web-сценариев — в третью и т. д.

Здесь приведен пример структуры папок, которую мы можем использовать для организации статичных файлов. Вложенность папок и файлов друг в друга показана отступами, имена папок набраны прописными буквами, имена файлов — строчными:

```
STATIC
  STYLES
    main.css
    goods.css
    . . .
  IMAGES
    logo.png
    NAV
      main_up.png
      main_down.png
      main_up_hover.png
      . . .
    SCRIPTS
      jquery.js
      main.js
      . . .
```

Выгруженные файлы тоже нежелательно сваливать в одну папку. Как правило, в ней создаются несколько папок — для хранения файлов, относящихся к разным моделям. Так, отдельные папки предусматриваются для хранения основных изображений товаров, которые отображаются в списке, изображений, которые выводятся на странице сведений о товаре, изображений, используемых в статьях блога, и др.

Вот пример структуры папок, в которых хранятся выгруженные файлы различного назначения:

```
MEDIA
  GOODS
    LIST
      good1.jpg
```

```
      good2.jpg
      . . .
    OTHERS
      good1_1.jpg
      good1_2.jpg
      good1_3.jpg
      good2_1.jpg
      good2_2.jpg
      . . .
  BLOG
    image1.jpg
    image2.jpg
    . . .
```

Количество статей блога и, соответственно, использованных в них изображений имеет тенденцию к увеличению. И со временем в папке, где хранятся файлы этих изображений, может скопиться столько файлов, что ими будет сложно управлять. Поэтому в этой папке создаются вложенные папки, соответствующие годам и месяцам, когда был выгружен тот или иной файл. Вот пример подобной структуры папок:

```
BLOG
  2013
    11
      image1.jpg
      image2.jpg
      . . .
    12
      image3.jpg
      image4.jpg
  2014
    1
      image5.jpg
      image6.jpg
      image7.jpg
      . . .
```

Средства Django для выгрузки файлов, рассмотренные нами в *главе 13*, предоставляют встроенные средства для автоматического формирования таких папок согласно заданному нами правилу. Так что создавать эти папки нам самим не придется.

Что касается способа передачи параметров: идентификаторов записей, подстрок поиска, номеров страниц — в контроллеры, то он выбирается согласно следующим правилам:

- основные параметры — например, идентификаторы записей, рекомендуется передавать в составе интернет-адреса;

- служебные параметры: подстроки поиска, номера страниц и пр., — как правило, не являющиеся обязательными, удобнее передавать через GET-параметры.

Впрочем, это всего лишь рекомендации. В документации по Django можно найти примеры как передачи идентификаторов записей через GET-параметр, так и отправки номеров страниц в составе интернет-адреса.

Средства для администрирования сайта

Наконец, мы должны решить, с помощью каких средств будет выполняться администрирование сайта, т. е. работа с его внутренними данными. Применительно к нашему случаю нам следует ответить на вопрос: как будет выполняться добавление, правка и удаление категорий, товаров и новостей.

У нас есть два варианта: использовать встроенный административный сайт Django или создать для этого собственные инструменты.

В случае блога выбор будет однозначным — создание своих собственных инструментов. Согласитесь, что добавление, правка и удаление статей блога через встроенный административный сайт выглядит, по меньшей мере, странно.

Для добавления записей в гостевую книгу мы также создадим либо отдельную страницу, либо, что предпочтительнее, выведем форму на той же странице, где отображается список добавленных ранее записей.

Во всех остальных случаях мы будем руководствоваться следующими рекомендациями:

- если администрированием сайта будут заниматься один-два человека, проще использовать для этого встроенный административный сайт Django. Совсем не обязательно ради них создавать отдельный административный раздел;
- если администрирование будет выполняться бóльшим числом сотрудников, лучше создать свои средства администрирования. Встроенный сайт может оказаться для некоторых из них слишком сложным или непривычным. При этом:
  - для работы с часто изменяемыми данными (категориями, товарами, новостями) мы создадим специальные страницы;
  - для работы с редко изменяемыми, второстепенными и служебными данными (пользователями, группами, файлами) можно использовать встроенный административный сайт. Все равно, как правило, данными такого рода занимается всего один человек — главный администратор.

К тому же, создавая свой собственный административный раздел, мы можем реализовать в нем какие-либо инструменты для упрощения работы с данными: наборы форм, в том числе и вложенные (см. *главу 12*), особые элементы управления и т. п.

Немного о дизайне сайта

Конечно, дизайн сайта не имеет прямого отношения к программированию. Однако его следует учитывать при создании структуры шаблонов.

Страницы большинства современных сайтов имеют множество однотипных элементов. К таким элементам относятся, прежде всего, заголовок сайта с логотипом, панели навигации и поддон (нижняя часть).

Однотипные элементы такого рода следует вынести в шаблон-родитель. Это позволит нам не дублировать их во всех шаблонах проекта и тем самым значительно уменьшить размер файлов шаблонов и объем работ по их созданию. (Собственно, об этом уже говорилось в *главе 8*, но повторить лишний раз не помешает.)

> ***Готовые дизайны сайтов***
>
> Готовые дизайны сайтов различного назначения, выполненные в формате Adobe Photoshop и доступные для свободного использования, можно найти по интернет-адресу **http://ruseller.com/adds.php?rub=36**.

Проект сайта «Веник-Торг»

Что ж, с основами проектирования сайтов мы ознакомились. Настало время попрактиковаться, подготовив проект нашего будущего сайта фирмы «Веник-Торг».

Итак, этот сайт включит в свой состав следующие приложения:

- главной страницы — `main`;
- списка категорий — `categories`;
- списка товаров — `goods`;
- списка новостей — `news`;
- гостевой книги — `guestbook`;
- блога — `blog`;
- сведений о контактах — `contacts`;
- сведений о фирме — `about`;
- сведений об оплате и вывозе товаров — `howtobuy`;
- служебного приложения, которое мы используем для реализации функциональности хранилища изображений, — `imagepool`.

Мы не станем использовать инструментарий статичных страниц Django, но применим встроенную систему комментирования.

Мы дадим пользователям возможность при вводе описаний товаров и статей блога использовать теги BBCode и вставлять в текст изображения.

Мы создадим административный раздел для работы со списками категорий, товаров и новостей, а также для публикации статей в блоге. Для работы со списками пользователей, групп, записей гостевой книги и комментариев будет применяться встроенный административный сайт Django (все равно такого рода задачами будет заниматься, скорее всего, один человек).

На страницах входа и выхода мы не станем размещать заголовок сайта, панель навигации и поддон — так мы сделаем эти страницы более стильными.

Мы не будем создавать отдельные папки для хранения статичных файлов, относящихся к разным типам. Наш сайт будет довольно простым, и статичных файлов окажется не настолько много, чтобы разбрасывать их по отдельным подпапкам.

Для хранения графических изображений в папке выгруженных файлов мы создадим такие папки:

- goods/list — основные изображения, которые будут выводиться в списке товаров;
- goods/detail — дополнительные изображения, присутствующие на странице сведений о товаре;
- imagepool — изображения, присутствующие в тексте новостей и статей блога.

В последней папке будут создаваться подпапки для отдельных годов, а в них — папки для отдельных месяцев. (Также можно создать и папки для отдельных дней, но, думается, это лишнее, поскольку список новостей и блог нашего сайта не будут содержать слишком много позиций.) И тогда файл, выгруженный, скажем, в апреле 2014 года, попадет в подпапку 2014/4.

Базу данных сайта мы будем хранить в подпапке data папки проекта. Ее файл будет называться site.dat.

На этом пока все. Подробные характеристики каждого приложения проекта мы определим непосредственно перед его созданием.

Предварительные действия

Теперь начнем собственно разработку сайта. Точнее, выполним предварительные работы: создадим проект, укажем его настройки и реализуем для зарегистрированных пользователей возможность входа и выхода.

Здесь предполагается, что мы уже установили все необходимые для начала разработки дополнительные библиотеки. Их список можно найти в *приложении 1*.

Создание проекта сайта

Создадим проект сайта способом, описанным в *главе 4*. Назовем его, скажем, `broomtrade`.

Сразу же, чтобы не делать этого потом, создадим в папке проекта следующие папки:

- data — для хранения базы данных;
- templates, static и upload — для хранения шаблонов и статичных файлов уровня проекта и выгруженных файлов;
- generic — на основе этой папки мы создадим пакет, в котором объявим базовые классы, используемые во всех приложениях проекта.

В папке generic сразу же создадим пустой, не содержащий кода модуль с именем `__init__`. Этот модуль скажет программному ядру Python, что эта папка представляет собой пакет.

Настройки проекта

Чтобы не отвлекаться впоследствии, мы, по возможности, зададим для проекта сразу все необходимые настройки. Откроем модель `settings` пакета проекта и укажем параметры базы данных в переменной `DATABASES`:

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

Зададим для сайта русский язык в переменной `LANGUAGE_CODE`:

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

Активизируем все необходимые приложения, добавив в список, являющийся значением переменной `INSTALLED_APPS`, следующие элементы:

```
INSTALLED_APPS = (
  . . .
  'django.contrib.sites',
  'django.contrib.comments',
  'easy_thumbnails',
  'taggit',
  'precise_bbcode',
)
```

Пропишем папки для хранения шаблонов, статичных файлов уровня проекта:

```
TEMPLATE_DIRS = (os.path.join(BASE_DIR, 'templates'),)
STATICFILES_DIRS = (os.path.join(BASE_DIR, 'static'),)
```

Укажем папку для хранения выгруженных файлов и используемый для них префикс интернет-адреса:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'uploads')
MEDIA_URL = '/media/'
```

Зададим имена привязок для страниц входа и выхода:

```
LOGIN_URL = "login"
LOGOUT_URL = "logout"
```

Поскольку мы активизировали приложение `django.contrib.sites`, обеспечивающее поддержку нескольких сайтов одной копией Django, — иначе мы не сможем использовать подсистему комментирования, — обязательно зададим идентификатор сайта:

```
SITE_ID = 1
```

Укажем параметры для формирования миниатюр:

```
THUMBNAIL_BASEDIR = "thumbnails"
THUMBNAIL_ALIASES = {
```

```
  "goods.Good.image": {
    "base": {"size": (200, 100)},
  },
}
```

Здесь мы создали единственный псевдоним `base`, содержащий параметры для миниатюр товаров, которые будут выводиться в списке.

И зададим параметры рассылки уведомлений о вновь добавленных комментариях:

```
MANAGERS = (("admin", "admin@someserver.ru"),)
```

Во время отладки сайта в качестве получателя уведомлений укажем лишь одного пользователя — администратора.

```
EMAIL_HOST = "someserver.ru"
EMAIL_HOST_USER = "user"
EMAIL_HOST_PASSWORD = "123456789"
DEFAULT_FROM_EMAIL = "mailer@someserver.ru"
```

Разумеется, нам следует задать здесь реальные адрес электронной почты, интернет-адрес SMTP-сервера, имя пользователя и пароль.

> ***НЕ ЗАБЫВАЕМ СОХРАНЯТЬ МОДУЛИ!***
>
> Не забываем сохранять все вновь созданные и исправленные модули! В дальнейшем автор более не будет об этом напоминать.

После указания параметров выполним синхронизацию с базой данных. Django потребует от нас создать пользователя с правами главного администратора — создадим его.

Начальные привязки

Следующий этап — указание начальных привязок. Они включат в себя привязки страниц входа и выхода, а также привязку папки выгруженных файлов к соответствующему ей префиксу.

Откроем модуль `urls` пакета проекта и включим в состав списка привязок следующие элементы:

```
urlpatterns = patterns('',
  ...
  url(r'^login/', "django.contrib.auth.views.login", name = "login"),
  url(r'^logout/', "django.contrib.auth.views.logout", name = "logout"),
)
```

Хороший стиль Python-программирования рекомендует помещать код, выполняющий импорт, в начало модуля. Соответственно, вставим в начало модуля выражения:

```
from django.conf import settings
from django.conf.urls.static import static
```

И нам останется лишь поместить в конец модуля вот такое выражение:

```
urlpatterns += static(settings.MEDIA_URL,
document_root = settings.MEDIA_ROOT)
```

Создание страниц входа и выхода

Сразу же создадим страницы для входа на сайт и выхода из него. Поскольку Django уже включает в свой состав все необходимые контроллеры, нам останется сделать лишь шаблоны этих страниц. Заодно создадим базовые шаблоны и базовые стили.

Базовые шаблоны

Прежде всего, подготовим базовый шаблон, который станет основой для всех страниц сайта. Вот его код:

```
{% load staticfiles %}
<!doctype html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link type="text/css" href="{% static "base.css" %}"
    rel="stylesheet">
    {% block additional_css %}
    {% endblock %}
    {% block additional_js %}
    {% endblock %}
    <title>{% block title %}{% endblock %} :: Веник-Торг</title>
  </head>
  <body>
    {% block content %}
    {% endblock %}
  </body>
</html>
```

Он включает в свой состав лишь инфраструктурные теги HTML (`<html>`, `<head>` и `<body>`), указание на версию языка HTML (HTML5), используемую кодировку и привязку основной таблицы стилей base.css, что задаст стили, применяемые на всех страницах сайта.

В этом шаблоне мы предусмотрели следующие блоки:

- `additional_css` — для привязок дополнительных таблиц стилей;
- `additional_js` — для привязок файлов Web-сценариев;
- `title` — для части названия, уникальной для каждой страницы;
- `content` — для собственно содержимого страницы.

Сохраним этот шаблон в файле base.html в папке templates, где хранятся файлы шаблонов уровня проекта.

Следующий базовый шаблон станет потомком шаблона base.html и основой для страниц входа и выхода. Вот его код:

```
{% extends "base.html" %}
{% load staticfiles %}
{% block additional_css %}
  <link type="text/css" href="{% static "login_logout.css" %}"
  rel="stylesheet">
{% endblock %}
{% block content %}
  <div id="ll">
    {% block ll %}
    {% endblock %}
  </div>
{% endblock %}
```

В нем мы укажем привязку таблицы стилей login_logout.css, которая включит в свой состав стили, задающие оформление для страниц входа и выхода. Также мы создадим в нем блок, внутри которого будет помещаться форма входа и текст, сообщающий об успешном выходе. В нем мы сформируем блок шаблона `ll`, с помощью которого зададим соответствующее странице содержимое в шаблонах-потомках.

Сохраним этот шаблон также в той же папке и дадим его файлу имя login_logout.html.

Универсальный шаблон формы

Страницы нашего сайта будут содержать множество форм, предназначенных для добавления, правки и удаления товаров, новостей, статей блога, записей гостевой книги и, наконец, для входа на сайт. И все эти формы будут создаваться с помощью одного и того же HTML-кода. Поэтому целесообразнее вынести его в отдельный подгружаемый шаблон (о подгружаемых шаблонах рассказывалось в *главе 8*).

Сначала решим, какой код вынести в отдельный шаблон. С одной стороны, он должен выполнять как можно больше задач по выводу формы — так мы уменьшим трудоемкость разработки каждого шаблона. С другой стороны, он должен позволять нам, в случае чего, вставить в форму дополнительные элементы управления.

Поскольку мы хотим создать универсальный шаблон формы, который можно было бы использовать везде, в том числе и для ввода комментариев, нам следует предусмотреть вывод используемых в последнем случае скрытых и служебных элементов. (Более подробно о подсистеме комментирования Django рассказывалось в *главе 15*.)

Давайте посмотрим вот на такой код:

```
{% csrf_token %}
{% for field in form.hidden_fields %}
  {{ field }}
{% endfor %}
{% for field in form.visible_fields %}
```

```
  {% if field.name == "honeypot" %}
    <div class="honeypot">{{ field }}</div>
  {% else %}
    <div class="form-field">
      {% if field.errors|length > 0 %}
        <div class="error-list">
          {% for error in field.errors %}
            <div class="error-description">{{ error }}</div>
          {% endfor %}
        </div>
      {% endif %}
      <div class="label">{{ field.label }}</div>
      <div class="control">{{ field }}</div>
      {% if field.help_text %}
        <div class="help">{{ field.help_text }}</div>
      {% endif %}
    </div>
  {% endif %}
{% endfor %}
```

Он создает набор видимых элементов управления формы со всеми надписями, поясняющим текстом и списками ошибок ввода. Скрытые поля, используемые в формах добавления комментариев, выводятся отдельно. Кроме того, поле ввода, предусмотренное в качестве «ловушки» для систем автоматической рассылки спама, заключается в отдельный блок с привязанным к нему стилевым классом, который сделает этот блок невидимым.

Отметим, что ни кнопки отправки данных, ни тега `<form>` шаблон не включает, что позволит нам без проблем указать надпись для кнопки отправки и дополнительные параметры для самой формы.

Создадим в папке templates, где хранятся шаблоны уровня проекта, вложенную папку generic. И сохраним там приведенный ранее код, указав для файла имя form.html. Это и есть наш универсальный шаблон формы.

Собственно шаблоны страниц входа и выхода

Теперь можно создать шаблон для страницы входа, являющийся потомком шаблона login_logout.html. Вот его код:

```
{% extends "login_logout.html" %}
{% block title %}Вход{% endblock %}
{% block ll %}
  <h2>Вход</h2>
  <form action="" method="post">
    {% include "form.html" %}
    <div class="submit-button"><input type="submit" value="Войти"></div>
    <input type="hidden" name="next" value="{{ next }}">
  </form>
{% endblock %}
```

Здесь мы используем только что созданный универсальный шаблон формы form.html, подгружая его с помощью знакомого нам по *главе 8* тега шаблона include.

В папке templates, где хранятся шаблоны уровня проекта, создадим папку registration. В ней и сохраним только что созданный шаблон, дав ему имя login.html.

Шаблон страницы выхода также будет потомком шаблона login_logout.html. Вот его код:

```
{% extends "login_logout.html" %}
{% block title %}Выход{% endblock %}
{% block ll %}
  <h2>Выход</h2>
  <p>Вы успешно выполнили процедуру выхода с сайта.</p>
  <p><a href="{% url "login" %}">Войти снова</a></p>
{% endblock %}
```

Здесь также нечего комментировать. Так что сразу же сохраним его в созданной ранее папке registration под именем logged_out.html.

Оформление

Осталось лишь создать таблицы стилей base.css и login_logout.css, задающие оформление для всех страниц сайта и страниц входа и выхода соответственно. Обе эти таблицы стилей будут храниться в папке static, предназначенной для статичных файлов уровня проекта.

> ***Кодировка таблиц стилей***
>
> Все таблицы стилей должны быть сохранены в кодировке UTF-8.

Код таблицы стилей base.css таков:

```
@charset "utf-8";
body {
  color: #000000;
  background-color: #FFFFFF;
  margin: 0px;
  padding: 0px;
}
.form-field {
  width: 100%;
  margin: 20px 0px 20px 0px;
}
.form-field .help {
  font-size: smaller;
  font-style: italic;
}
.form-field .label {
  margin-bottom: 2px;
}
```

```
.form-field .control input[type=name],
.form-field .control input[type=password],
.form-field .control input[type=number],
.form-field .control input[type=file],
.form-field .control textarea, .form-field .control select {
  width: 100%;
}
.submit-button {
  width: 100%;
}
.error-list .error-description {
  color: #FF0000;
}
```

Здесь мы убираем отступы между краями страницы и ее содержимым и задаем параметры шрифта, размеры и величины отступов у блоков, в которые будут помещаться элементы управления, и самих элементов управления.

А код таблицы стилей login_logout.css будет следующим:

```
@charset "utf-8";
#ll {
  width: 160px;
  margin: 50px auto 0px auto;
  padding: 30px;
  border: 1px solid #AAAAAA;
}
#ll h2 {
  margin-top: 0px;
}
```

Здесь мы с помощью известного HTML-кодировщикам приема располагаем по центру блок, где будут выводиться форма входа и текст с сообщением об успешном выходе. Также мы задаем для этого блока ширину, внутренние отступы и рамку. И, наконец, мы убираем отступ сверху у заголовка, чтобы уменьшить просвет между ним и верхней границей блока.

И проверим только что созданные страницы. Запустим отладочный Web-сервер Django и наберем в Web-обозревателе интернет-адрес страницы входа — **http://localhost:8000/login/**. Эта страница должна выглядеть так, как показано на рис. 20.1.

После чего проверим страницу выхода, набрав интернет-адрес **http://localhost:8000/logout/**. Полученная страница выхода представлена на рис. 20.2.

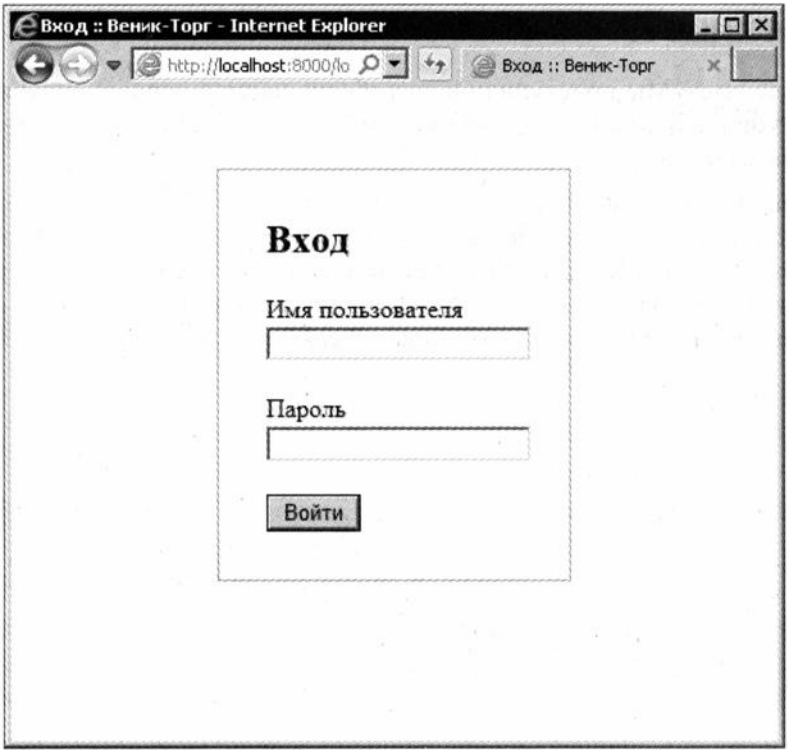


Рис. 20.1. Страница входа

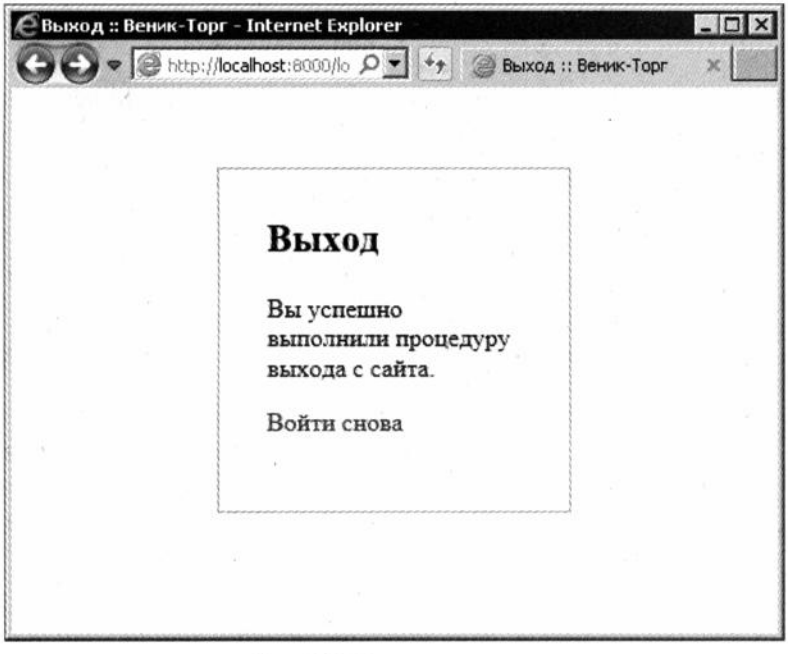


Рис. 20.2. Страница выхода

Что дальше?

Начало положено! Мы приступили к разработке нашего первого настоящего сайта на языке Python с применением библиотеки Django. Теперь мы с уверенностью можем называть себя программистами!

В следующей главе мы создадим главную страницу сайта, пока еще неполную, можно сказать, ее заготовку. (Дорабатывать мы ее будем по ходу дела, после создания приложений, обрабатывающих списки категорий, товаров и новостей.) А еще мы дополним таблицы стилей и создадим третий базовый шаблон, на котором будут основаны шаблоны остальных страниц сайта.

ГЛАВА 21

Главная страница

В предыдущей главе мы приступили к разработке нашего первого настоящего Web-сайта. Мы подготовили его план, создали и настроили проект, реализовали возможность входа и выхода и создали базовые шаблоны и таблицы стилей.

В этой главе мы создадим заготовку для главной страницы, напишем еще один базовый шаблон и дополним набор стилей. Также нам потребуется создать кое-какие базовые классы, которые мы будем широко использовать в дальнейшем.

Что должно выводиться на главной странице, мы уже решили. Так что можем сразу же приступать к работе.

Приложение и привязка

Для главной страницы мы создадим приложение `main`. Сделаем это в порядке, описанном в *главе 4*.

Откроем модуль `urls` пакета проекта и вставим в список привязок следующий элемент:

```
urlpatterns = patterns('',
  . . .
  url(r'^$', include('main.urls')),
)
```

Создадим в пакете приложения `main` модуль `urls` и вставим в него код:

```
from django.conf.urls import patterns, url
from main.views import MainPageView
urlpatterns = patterns('',
  url(r'^$', MainPageView.as_view(), name = "main"),
)
```

Для привязки мы указали «говорящее» имя `main`. Класс-контроллер `MainPageView`, выводящий главную страницу, мы создадим потом.

Осталось внести кое-какие исправления в настройки проекта, а именно — вставить вновь созданное приложение в список активных и задать интернет-адрес, куда будет перенаправляться пользователь после успешного входа. Пусть это будет интернет-адрес главной страницы — с нее пользователь сможет перейти куда угодно.

Откроем модуль `settings` пакета проекта и поместим в список активных приложение, созданное нами:

```
INSTALLED_APPS = (
  . . .
  'main',
)
```

И добавим в код модуля выражение:

```
LOGIN_REDIRECT_URL = "main"
```

Контроллер

Приступим к созданию контроллера, который станет формировать главную страницу. Поскольку всю логику по выводу двух списков — товаров и новостей — нам придется писать самим, сделаем этот контроллер потомком самого простого класса — `TemplateView`.

Но сначала нам следует создать один базовый класс. Мы вынесем в него функциональность, которая будет присутствовать во всех классах-контроллерах нашего проекта.

Базовый класс *CategoryListMixin*

Все страницы нашего сайта (за исключением страниц входа и выхода) будут выводить панель навигации. Она, в числе прочих пунктов, включит и перечень категорий, каждая позиция которого будет вести на страницу списка товаров, относящихся к этой категории. Для хранения списка категорий мы впоследствии (в *главе 24*), создадим особую модель.

Далее, каждая приличная панель навигации должна подсвечивать или выделять как-то иначе гиперссылку, соответствующую текущей, т. е. загруженной в данный момент странице. Чтобы сделать такое, нам потребуется знать интернет-адрес текущей страницы, — тогда мы без труда сравним его с интернет-адресом каждой гиперссылки и, в зависимости от результата сравнения, привяжем к ней задающий нужное оформление стиль.

Чтобы вывести на странице некое значение или использовать его в коде шаблона для каких-либо вычислений, нам следует создать в контексте данных шаблона переменную, которая будет хранить это значение. Вместо того чтобы включать код, который будет формировать такую переменную, в состав каждого-класса контроллера, мы можем вынести его в отдельный класс, который потом указать в числе родителей каждого создаваемого класса-контроллера.

Класс TemplateView, что мы выбрали в качестве родителя для контроллера главной страницы, получает функциональность по формированию контекста данных от одного из своих родителей — класса ContextMixin. Следовательно, базовый класс, добавляющий в контекст данных переменную со списком категорий и текущим интернет-адресом (назовем его CategoryListMixin), следует породить именно от него.

Создадим в сформированном нами самими пакете generic модуль mixins. В него введем код объявления класса CategoryListMixin:

```
from django.views.generic.base import ContextMixin
class CategoryListMixin(ContextMixin):
  def get_context_data(self, **kwargs):
    context = super(CategoryListMixin, self).get_context_data(**kwargs)
    context["current_url"] = self.request.path
    return context
```

Получить объект класса HttpRequest, хранящий сведения о запросе, можно через свойство request, унаследованное от родительского класса ContextMixin. Текущий интернет-адрес хранится в знакомом нам по *главе 6* свойстве path класса HttpRequest. Этот интернет-адрес сохраняется в переменной current_url контекста данных.

Код, помещающий в контекст данных список категорий, мы впишем сюда в *главе 24*, когда создадим соответствующее приложение. А пока что закончим с классом CategoryListMixin.

Собственно контроллер главной страницы

Вот теперь можно приступать к написанию класса-контроллера, «отвечающего» за главную страницу.

Откроем модуль views пакета приложения и вставим в него такой код:

```
from django.views.generic.base import TemplateView
from generic.mixins import CategoryListMixin
class MainPageView(TemplateView, CategoryListMixin):
  template_name = "mainpage.html"
```

В числе предков этого класса мы указали также созданный ранее базовый класс CategoryListView, добавляющий в контекст данных список категорий и текущий интернет-адрес. В целом же новый класс-контроллер не содержит практически никакого кода, кроме указания на используемый шаблон. Точнее, пока не содержит.

Шаблон

Осталось лишь написать шаблон главной страницы. Следует также создать базовый шаблон, на котором будут основаны шаблоны остальных страниц сайта, и дополнить состав стилей.

Базовый шаблон

Для создания страниц сайта мы используем привычный многим дизайн. Он включает заголовок сайта, находящийся в его «шапке», боковую панель навигации слева, расположенную справа область для основного содержимого и поддон.

Тогда код базового шаблона будет следующим:

```
{% extends "base.html" %}
{% load staticfiles %}
{% block additional_css %}
  <link type="text/css" href="{% static "main.css" %}" rel="stylesheet">
{% endblock %}
{% block content %}
  <div id="header">
    <h1>Веник-Торг</h1>
  </div>
  <div id="leftmenu">
    <ul>
      {% url "main" as page_url %}
      <li><a href="{{ page_url }}"{% if page_url == current_url %}
      class="current"{% endif %}>Главная</a></li>
      {% if user.is_authenticated %}
        <li class="indented"><a href="/admin/">Админка</a></li>
        <li class="indented"><a href="{% url "logout" %}">Выйти</a></li>
      {% endif %}
    </ul>
  </div>
  <div id="main">
    {% block main %}
    {% endblock %}
  </div>
  <div id="footer">
    <p>Все права принадлежат разработчикам сайта.</p>
  </div>
{% endblock %}
```

Новый шаблон является потомком созданного в *главе 20* шаблона base.html. В нем мы указываем привязку таблицы стилей main.css, которая задаст оформление для всех остальных страниц сайта. (Эту таблицу стилей мы создадим чуть позже.)

Блок `leftmenu` служит для размещения панели навигации. Пока что она будет содержать три гиперссылки, указывающие на главную страницу, встроенный административный сайт и страницу выхода.

Давайте посмотрим на код, создающий гиперссылку на главную страницу:

```
{% url "main" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}
class="current"{% endif %}>Главная</a></li>
```

Здесь мы сравниваем интернет-адрес главной страницы с интернет-адресом текущей страницы и, если они равны (т. е. если в данный момент загружена главная страница), привязываем к тегу `<a>`, создающему гиперссылку, стилевой класс `current`. Он задаст соответствующее оформление для гиперссылки, указывающей на текущую страницу.

Что касается текущего интернет-адреса, то он хранится в переменной `current_url` контекста данных шаблона, формируемой объявленным ранее базовым классом `CategoryListMixin`.

Гиперссылки, ведущие на встроенный административный сайт и страницу выхода, мы выводим лишь в том случае, если пользователь ранее выполнил процедуру входа.

Стилевой класс `indented` задаст для пункта панели навигации дополнительный отступ сверху. Так мы визуально отделим гиперссылки разного назначения друг от друга.

Содержимое, уникальное для каждой страницы, будет выводиться в блоке шаблона `main`.

Сохраним этот шаблон в папке templates шаблонов уровня проекта, дав ему имя main.html.

Собственно шаблон страницы

Шаблон главной страницы будет очень простым. Вот его код:

```
{% extends "main.html" %}
{% block title %}Главная страница{% endblock %}
{% block main %}
  <h2>Все для уборки</h2>
  <p>Фирма &amp;quot;Веник-Торг&amp;quot; продает все необходимое для уборки
  помещений различного назначения: веники, щетки, метлы, совки и пр.</p>
  <h3>Новости</h3>
  <h3>Рекомендуемые товары</h3>
{% endblock %}
```

Под заголовками «Новости» и «Рекомендуемые товары» будут выводиться списки последних новостей и рекомендуемых товаров. Мы создадим их позже.

Этот шаблон можно сохранить на уровне приложения. Создадим в папке пакета приложения папку templates и поместим его там, дав хранящему его файлу имя mainpage.html.

Оформление

И зададим оформление для главной и всех последующих страниц сайта, создав таблицу стилей main.css. Вот ее код:

```
@charset "utf-8";
#header {
```

```
  text-align: center;
  border-bottom: 1px #cccccc solid;
}
#header h1 {
  margin: 20px 0px 20px 0px;
  letter-spacing: 0.2em;
}
#leftmenu {
  float: left;
  width: 100px;
  padding: 10px;
}
#leftmenu ul {
  margin: 0px;
  padding: 0px;
}
#leftmenu ul li {
  font-size: larger;
  list-style-type: none;
  margin: 0px 10px 10px 0px;
}
#leftmenu ul li a {
  display: block;
}
#leftmenu ul li.indented {
  margin-top: 20px;
}
#leftmenu ul li a:link, #leftmenu ul li a:visited,
#leftmenu ul li a:active, #leftmenu ul li a:hover {
  text-decoration: none;
}
#leftmenu ul li a:link, #leftmenu ul li a:visited {
  color: #0000FF;
  background-color: #FFFFFF;
}
#leftmenu ul li a:active, #leftmenu ul li a:hover {
  color: #FFFFFF;
  background-color: #000000;
}
#leftmenu ul li a.current:link, #leftmenu ul li a.current:visited,
#leftmenu ul li a.current:active, #leftmenu ul li a.current:hover {
  color: #000000;
  background-color: #FFFFFF;
}
#main {
  margin-left: 110px;
```

```
  border-left: 1px #cccccc solid;
  padding: 10px;
}
#main h2:first-child {
  margin: 0px 0px 10px 0px;
}
#footer {
  text-align: right;
  border-top: 1px #cccccc solid;
  padding-right: 10px;
}
#footer p {
  font-size: smaller;
  font-style: italic;
}
```

Здесь мы, в том числе, указываем местоположение и размеры блоков и параметры гиперссылок. Обычные гиперссылки будут иметь белый фон и текст синего цвета. При наведении курсора мыши они получат черный фон и белый цвет текста. Гиперссылка, указывающая на текущую страницу, будет всегда выводиться черным цветом на белом фоне. (Не будем создавать слишком сложный дизайн, в конце концов, книга посвящена не этому.)

Завершающие действия

Закончив, посмотрим на готовую главную страницу, Для чего наберем в Web-обозревателе интернет-адрес **http://localhost:8000/**, разумеется, не забыв запустить отладочный Web-сервер.

Далее проверим, работает ли механизм входа на сайт и выхода с него. Наберем интернет-адрес **http://localhost:8000/login/**, чтобы попасть на страницу входа, и выполним вход. Нас перенаправят на главную страницу сайта, где мы увидим гиперссылки **Админка** и **Выход** (рис. 21.1), выводимые только выполнившим вход на сайт пользователям. И выполним выход с сайта.

На странице выхода не помешало бы поместить гиперссылку, ведущую на главную страницу. Так что откроем шаблон logged_out.html и вставим в его код такую строку (выделена полужирным шрифтом):

```
{% block ll %}
  . . .
  <p><a href="{% url "login" %}">Войти снова</a></p>
  <p><a href="{% url "main" %}">На главную страницу</a></p>
{% endblock %}
```

Рис. 21.1. Главная страница сайта

Что дальше?

Наш сайт обзавелся главной страницей. Пусть она еще не вполне функциональна, но со временем мы ее доделаем.

Полюбовавшись на творение рук своих, оставим хвалебный отзыв разработчикам. Что? Негде оставлять хвалебные отзывы? Нет гостевой книги? Ничего, в следующей главе мы это исправим...

ГЛАВА 22

Гостевая книга

В предыдущей главе мы начали разработку главной страницы сайта и почти ее сделали. Осталось лишь поместить на страницу списки рекомендованных товаров и самых последних новостей, но это потом.

Сейчас же мы приступим к работе над гостевой книгой. Мы сделаем вывод списка оставленных в ней записей и добавим на ту же страницу форму ввода новой записи. Работа с уже добавленными в гостевую книгу записями — их правка и удаление, — как мы решили в *главе 20*, будет выполняться через встроенный административный сайт Django.

Защита от спама

О том, насколько надоедливы спамеры, говорить не надо. Давайте лучше поговорим о том, как от них защититься.

Программы автоматической отправки «мусорных» сообщений в гостевые книги обычно вводят значения во все поля соответствующих форм. На этом можно построить нашу систему защиты. Мы создадим особое поле ввода — «ловушку», которое будет присутствовать в форме добавления новой записи, но не будет выводиться на экран. Если запись заносит посетитель, он не увидит этого поля и, соответственно, не внесет в него значения. В случае же, если запись заносится программой, поле-«ловушка» получит какое-либо значение.

Нам останется лишь проверить в контроллере, добавляющем запись в гостевую книгу, содержит ли это поле какое-либо значение. Если не содержит, следовательно, запись оставлена посетителем, и ее следует добавить в гостевую книгу. Иначе мы проигнорируем внесенную запись, оставив программу автоматической отправки сообщений в счастливом неведении о том, что ее усилия тщетны...

Кстати, такая же система используется и в стандартной подсистеме комментирования Django (см. *главу 15*).

Приложение

Приложение гостевой книги будет иметь имя `guestbook`. Создадим его. После чего откроем модуль `settings` пакета проекта и добавим вновь созданное приложение в список активных:

```
INSTALLED_APPS = (
  . . .
  'guestbook',
)
```

Модель

Модель гостевой книги — назовем ее `Guestbook` — должна включать поля для хранения имени оставившего запись посетителя, даты и времени добавления записи и собственно ее содержимого. Более подробные параметры полей новой модели вместе с их именами сведены в табл. 22.1.

***Таблица 22.1.** Поля модели `Guestbook`*

Имя	Тип	Параметры
`user`	Текстовый	Длина — 20 символов. Обязательное поле
`posted`	Дата и время	Значение заносится при добавлении записи
`content`	Memo	Обязательное поле

Сделаем так, чтобы записи модели выводились отсортированными по значению поля `posted` по убыванию. Это поле мы сделаем индексированным — в таком случае сортировка по его значению будет выполняться быстрее.

Определившись с характеристиками модели, откроем модуль `models` пакета приложения `guestbook` и введем в него такой код:

```
class Guestbook(models.Model):
  user = models.CharField(max_length = 20, verbose_name = "Пользователь")
  posted = models.DateTimeField(auto_now_add = True, db_index = True,↵
  verbose_name = "Опубликовано")
  content = models.TextField(verbose_name = "Содержание")
  class Meta:
    ordering = ["-posted"]
    verbose_name = "запись гостевой книги"
    verbose_name_plural = "записи гостевой книги"
```

Особых пояснений здесь не требуется — все давно знакомо нам по *главе 5*. Так что сразу выполним синхронизацию с базой данных.

Привязки

И приступим к созданию привязок контроллеров к интернет-адресам — как уровня проекта, так и уровня приложения.

Откроем модуль `urls` пакета проекта, где указываются привязки уровня проекта. Добавим в список привязок следующий элемент:

```
urlpatterns = patterns('',
  . . .
  url(r'^guestbook/', include('guestbook.urls')),
)
```

Далее создадим в пакете приложения `guestbook` модель `urls` и укажем в нем привязку уровня приложения. Вот код, который ее создает:

```
from guestbook.views import GuestbookView
urlpatterns = patterns('',
  url(r'^$', GuestbookView.as_view(), name = "guestbook"),
)
```

Привязка здесь всего одна, поскольку мы используем и для вывода списка записей, и для добавления новой записи один и тот же класс-контроллер `GuestbookView`. Поскольку форма для добавления записи будет выводиться на той же странице, где перечисляются уже присутствующие в гостевой книге заметки, а процедура создания новой записи не требует много кода, разносить функциональность по двум контроллерам и неудобно, и нерационально.

Форма

Следующий шаг — создание формы, куда посетитель будет заносить новую запись. Это будет форма, связанная с моделью.

Поскольку мы собираемся добавить в эту форму еще и поле — «ловушку» для спамеров, нам придется явно определить все ее поля со всеми необходимыми параметрами.

Исходя из всего этого, напишем код нашей формы:

```
from django import forms
from guestbook.models import Guestbook
class GuestbookForm(forms.ModelForm):
  class Meta:
    model = Guestbook
  user = forms.CharField(max_length = 20, label = "Пользователь")
  content = forms.CharField(widget = forms.Textarea,↵
  label = "Содержание")
  honeypot = forms.CharField(required = False,↵
  label = "Ловушка для спамеров")
```

Сохраним его в модуле `forms` в пакете приложения.

Сохранение кода форм

Правила хорошего тона Django-программирования рекомендуют сохранять код форм в отдельном модуле `forms`.

Контроллер

Класс-контроллер `GuestbookView` мы создадим на основе класса `ArchiveIndexView`, рассмотренного в *главе 10*. Так нам не придется беспокоиться о формировании набора записей и обеспечении пагинации.

Вот полный код класса-контроллера `GuestbookView`:

```
from django.shortcuts import redirect
from django.views.generic.dates import ArchiveIndexView
from django.contrib import messages
from guestbook.models import Guestbook
from guestbook.forms import GuestbookForm
from generic.mixins import CategoryListMixin

class GuestbookView(ArchiveIndexView, CategoryListMixin):
  model = Guestbook
  date_field = "posted"
  template_name = "guestbook.html"
  paginate_by = 20
  allow_empty = True
  form = None
  def get(self, request, *args, **kwargs):
    self.form = GuestbookForm()
    return super(GuestbookView, self).get(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(GuestbookView, self).get_context_data(**kwargs)
    context["form"] = self.form
    return context
  def post(self, request, *args, **kwargs):
    self.form = GuestbookForm(request.POST)
    if self.form.is_valid():
      if self.form.cleaned_data["honeypot"] == "":
        self.form.save()
        messages.add_message(request, messages.SUCCESS,↵
        "Запись успешно добавлена в гостевую книгу")
      return redirect("guestbook")
    else:
      return super(GuestbookView, self).get(request, *args, **kwargs)
```

Поскольку впоследствии нам придется выводить на странице гостевой книги и список категорий и подсвечивать текущий пункт панели навигации, мы указываем в числе родителей этого класса еще и объявленный в *главе 21* класс `CategoryListMixin`.

(Впрочем, практически все классы-контроллеры, что мы напишем далее, будут иметь его в списке родителей.)

Не забываем присвоить свойству `allow_empty` значение `True`. Если мы этого не сделаем, то при попытке зайти в пустую гостевую книгу получим сообщение об «ошибке 404».

В свойстве `form` мы будем хранить форму для добавления записи. Она будет создаваться в методе `get` и добавляться в контекст данных шаблона в методе `get_contect_data`.

Перед сохранением введенной в форму записи обязательно проверим, было ли занесено в поле `honeypot` какое-либо значение. Если не было занесено, мы сохраняем новую запись, в противном случае игнорируем ввод.

Все остальное нам давно знакомо. Поэтому вставим приведенный здесь код в модуль `views` пакета приложения.

Шаблоны

Помимо шаблона guestbook.html, на основе которого должна генерироваться страница гостевой книги, мы объявим два универсальных шаблона. Они будут содержать код двух типовых элементов, которые мы используем на различных Web-страницах. Также не забудем дополнить набор стилей.

Универсальный шаблон вывода сообщений

Наш класс-контроллер `GuestbookView`, чтобы дать посетителю знать об успешном добавлении записи в гостевую книгу, использует инфраструктуру сообщений Django (за подробностями — к *главе 12*). Поэтому нам понадобится вставить в шаблон код, который сможет выводить сообщения на экран.

Поскольку такие сообщения должны выводиться на нескольких страницах сайта, давайте вынесем выводящий их код в подгружаемый шаблон. Вот этот код:

```
{% if messages %}
  <div id="messages-list">
    {% for message in messages %}
      <p class="{{ message.tags }}">{{ message }}</p>
    {% endfor %}
  </div>
{% endif %}
```

Сохраним его в файле messages.html, поместив этот файл в подпапку generic, которую создали ранее в папке templates шаблонов уровня проекта.

Универсальный шаблон пагинации

Поскольку гостевую книгу мы выводим с применением пагинации, нам придется вставить в шаблон код, создающий набор гиперссылок для перехода между страни-

цами. Этот код также будет использован на разных страницах, и его также лучше оформить как отдельный подгружаемый шаблон:

```
{% if paginator.num_pages > 1 %}
  <div id="pagination">
    {% if page_obj.has_previous %}
      <div id="previous-page"><a href="?page=
      {{ page_obj.previous_page_number }}">&lt;</a></div>
    {% endif %}
    {% if page_obj.has_next %}
      <div id="next-page"><a href="?page=
      {{ page_obj.next_page_number }}">&gt;</a></div>
    {% endif %}
    <div id="num-pages">
      {% for pn in paginator.page_range %}
        {% if not forloop.first %}
            |  
        {% endif %}
        {% if page_obj.number != pn %}
          <a href="?page={{ pn }}">
        {% endif %}
        {{ pn }}
        {% if page_obj.number != pn %}
          </a>
        {% endif %}
      {% endfor %}
    </div>
  </div>
{% endif %}
```

В интернет-адресах гиперссылок мы указываем лишь GET-параметр `page` с номером страницы. Встретив такую гиперссылку, Web-обозреватель «поймет», что она указывает на интернет-адрес текущей страницы. Такой прием позволит нам сделать шаблон пагинации действительно универсальным.

Сохраним этот шаблон в файле pagination.html, поместив его в подпапку generic папки templates уровня проекта.

Шаблон гостевой книги

Код шаблона, что будет выводить гостевую книгу, ничем не примечателен. Давайте посмотрим сами:

```
{% extends "main.html" %}
{% block title %}Гостевая книга{% endblock %}
{% block main %}
  {% include "generic/messages.html" %}
  <h2>Гостевая книга</h2>
  <div class="form bottom-indent">
```

```
    <form action="" method="post">
      {% include "generic/form.html" %}
      <div class="submit-button"><input type="submit"
      value="Добавить"></div>
    </form>
  </div>
  {% if object_list.count > 0 %}
    {% for object in latest %}
      <table class="guestbook">
        <tr>
          <td class="user">{{ object.user }}</td>
          <td class="posted">{{ object.posted|date:"j.m.Y H:i:s" }}</td>
        </tr>
        <tr>
          <td colspan="2" class="content">{{ object.content }}</td>
        </tr>
      </table>
    {% endfor %}
  {% endif %}
  {% include "generic/pagination.html" %}
{% endblock %}
```

Обратить внимание здесь имеет смысл лишь на формат, которым выводится дата и время добавления записи (об указании формата вывода таких значений говорилось в *главе 7*), и на способ оформления отдельных записей гостевой книги — каждая запись выводится в таблице.

Сохраним этот шаблон в файле guestbook.html, поместив его в папку templates уровня приложения `guestbook`.

Оформление

В коде новых шаблонов мы использовали большое количество новых стилей. Настало время создать их.

Откроем таблицу стилей main.css, определяющую оформление всех страниц сайта, за исключением страниц входа и выхода, и вставим в нее такой код:

```
.form {
  width: 400px;
  margin: 20px 0px 20px 0px;
}
.bottom-indent {
  margin-bottom: 50px;
}
.guestbook {
  width: 100%;
  margin-bottom: 20px;
  border: 1px solid #cccccc;
}
```

```
.guestbook .user {
  font-weight: bold;
}
.guestbook .posted {
  font-style: italic;
  text-align: right;
  width: 200px;
}
.guestbook .content {
  padding-top: 20px;
}
.honeypot {
  display: none;
}
#messages-list {
  background-color: #EEEEEE;
  padding: 5px;
}
#messages-list .ERROR {
  color: #FF0000;
}
#pagination {
  margin-top: 20px;
}
#pagination #previous-page {
  float: left;
}
#pagination #num-pages {
  text-align: center;
}
#pagination #next-page {
  float: right;
}
```

Эти стили зададут оформление для гостевой книги, формы добавления новой записи, списка сообщений и элементов пагинации, а также скроют поле-«ловушку» для систем автоматической рассылки спама.

Завершающие действия

Осталось сделать немного. А именно — создать гиперссылку, указывающую на гостевую книгу, и сделать так, чтобы содержимое гостевой книги выводилось в составе встроенного административного сайта Django.

Начнем с гиперссылки на гостевую книгу. Код, который нам для этого потребуется вставить в базовый шаблон main.html, в приведенном далее листинге выделен полужирным шрифтом:

```
. . .
{%. url "main" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}
class="current"{% endif %}>Главная</a></li>
{% url "guestbook" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}
class="current"{% endif %}>Гостевая</a></li>
{% if user.is_authenticated %}
. . .
```

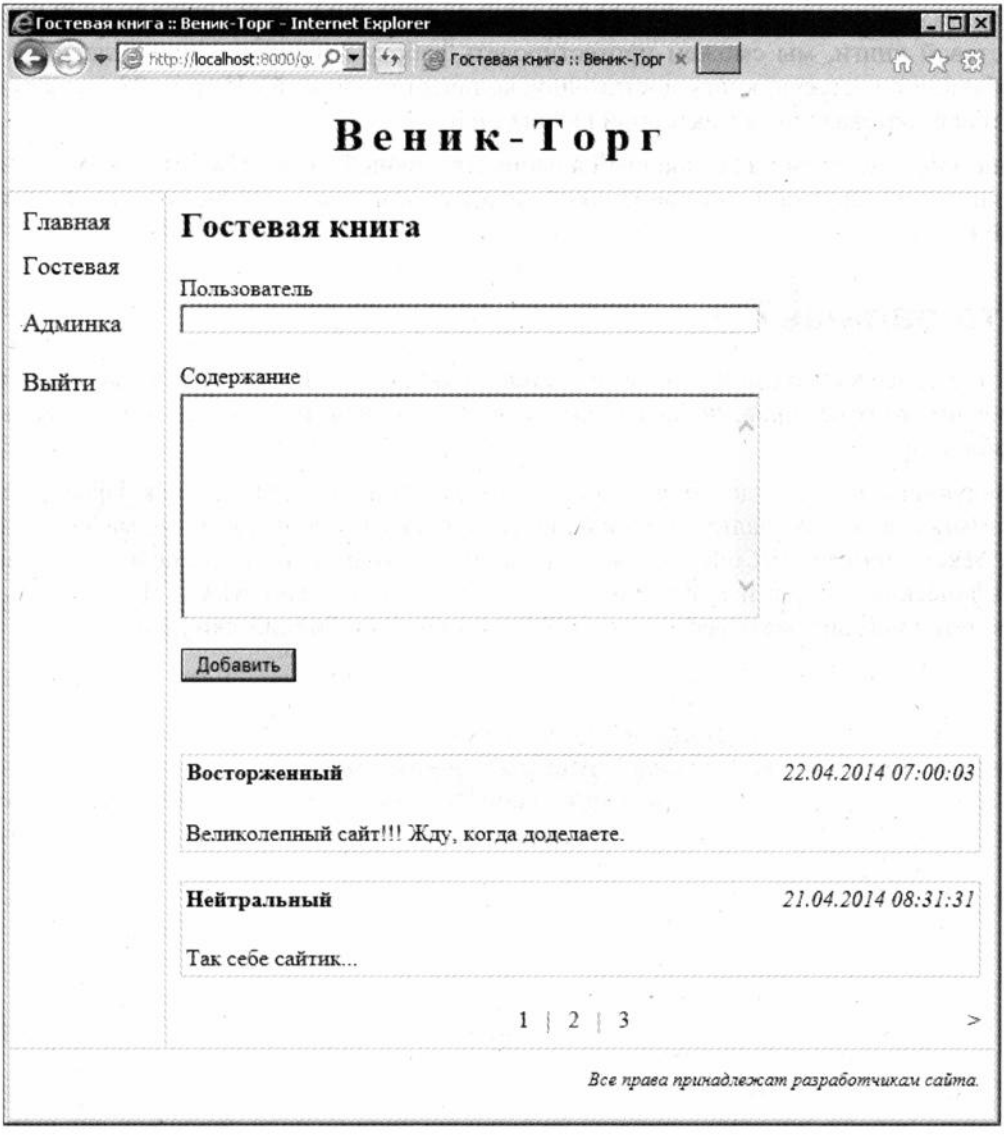


Рис. 22.1. Гостевая книга

Теперь «познакомим» встроенный административный сайт с созданной ранее моделью `Guestbook`. Откроем модуль `admin` пакета приложения и вставим в него такой код:

```
from guestbook.models import Guestbook
admin.site.register(Guestbook)
```

Наконец, проверим, как работает гостевая книга. Запустим отладочный Web-сервер, перейдем на страницу гостевой книги, добавим в нее несколько записей и посмотрим, как они выводятся.

Чтобы проверить, работает ли пагинация, откроем модуль `views` и зададим для свойства `paginate_by` класса `GuestbookView` значение 1 или 2. Обновив страницу гостевой книги, мы сможем протестировать работу пагинации (при условии, что добавили в гостевую книгу достаточное количество записей). На рис. 22.1 показана готовая гостевая книга с активной пагинацией.

И не забудем зайти на встроенный административный сайт и посмотреть, выводится ли на его главной странице гиперссылка, указывающая на список записей гостевой книги.

Что дальше?

В этой главе мы создали для сайта гостевую книгу. Заодно попрактиковались в написании контроллеров, добавляющих запись в модель, и создании подгружаемых шаблонов.

Следующая глава будет более серьезным испытанием для нас как Django-программистов. Мы создадим список новостей, реализуем поддержку форматирования их текста тегами BBCode, сделаем хранилище изображений, обеспечим выгрузку графических файлов и получение их списка по технологии AJAX. Для этого нам придется мобилизовать все знания HTML, JavaScript и библиотеки jQuery.

ГЛАВА 23

Список новостей. Хранилище изображений

В предыдущей главе мы занимались гостевой книгой. Мы написали модель, где будут храниться ее записи, сделали контроллер и все нужные шаблоны. И даже добавили в гостевую книгу несколько записей, положив начало обратной связи между фирмой «Веник-Торг» и ее клиентами.

В этой главе мы приступим к работе над списком новостей. Мы сделаем так, чтобы пользователи могли форматировать их содержимое с помощью тегов BBCode и вставлять в них изображения. А для хранения этих изображений мы создадим особое хранилище, работающее по технологии AJAX.

Собственно список новостей

Начнем мы с реализации основных функций списка новостей: вывода собственно списка, вывода содержимого выбранной новости, добавления, правки и удаления новостей. Хранилище изображений мы сделаем потом, когда отладим основные функции.

Приложение

Приложение — а оно, как мы решили ранее, будет иметь имя `news`, — создается точно так же, как и любое приложение, входящее в состав проекта Django. И точно так же оно добавляется в список активных приложений, хранящийся в модуле `settings` пакета проекта:

```
INSTALLED_APPS = (
  . . .
  'news',
)
```

Модель

Модель новостей, которой мы дадим имя `New`, должна включать следующие поля:

- заголовок новости;
- краткое описание, которое будет выводиться в списке новостей;

- содержимое, выводящееся на странице выбранной новости;
- дата и время создания новости.

Содержимое новости, отформатированное с применением тегов BBCode, мы будем хранить в обычном поле типа memo. Этим самым мы исключим досадную проблему с классом поля `BBCodeTextField`, приводящую в некоторых случаях к потере хранящегося в этом поле значения (подробнее об этом говорилось в *главе 19*).

Для последнего поля в качестве значения по умолчанию мы зададим текущее значение даты и времени.

Значение поля заголовка не помешало бы сделать уникальным в пределах даты, что хранится в последнем поле. Так мы исключим возможность многократного ввода одной и той же новости.

Определившись с составом полей новой модели, давайте распишем их параметры более подробно. Эти параметры сведены в табл. 23.1.

***Таблица 23.1.** Поля модели `New`*

Имя	Тип	Параметры
`title`	Текстовый	Максимальная длина — 100 символов. Значение должно быть уникальным в пределах даты, хранящейся в поле `posted`
`description`	Memo	
`content`	Memo	
`posted`	Дата и время	Значение по умолчанию — текущие дата и время

Все поля модели являются обязательными.

Сделаем так, чтобы записи модели выводились отсортированными по значению поля `posted` по убыванию. Чтобы сортировка выполнялась быстрее, создадим на основе данного поля индекс.

Теперь переведем все описанное в табл. 23.1 на язык, понятный Django. Откроем модуль `models` пакета приложения и вставим в него такой код:

```
from datetime import datetime
class New(models.Model):
  title = models.CharField(max_length = 100, unique_for_date = "posted",↳
  verbose_name = "Заголовок")
  description = models.TextField(verbose_name = "Краткое содержание")
  content = models.TextField(verbose_name = "Полное содержание")
  posted = models.DateTimeField(default = datetime.now(),↳
  db_index = True, verbose_name = "Опубликована")
  class Meta:
    ordering = ["-posted"]
    verbose_name = "новость"
    verbose_name_plural = "новости"
```

Метод now класса datetime, объявленного в одноименном стандартном модуле Python, возвращает текущее значение даты и времени.

И не забудем выполнить синхронизацию с базой данных, чтобы Django создала в ней все необходимые структуры.

Привязки

В модуле urls пакета проекта (т. е. на уровне проекта) добавим в список привязок элемент, соответствующий новому приложению:

```
urlpatterns = patterns('',
  . . .
  url(r'^news/', include('news.urls')),
)
```

Создадим в пакете приложения модуль urls и вставим в него код, задающий привязки уровня приложения:

```
from django.contrib.auth.decorators import permission_required
from news.views import NewsListView, NewDetailView, NewCreate,↵
NewUpdate, NewDelete
urlpatterns = patterns('',
  url(r'^$', NewsListView.as_view(), name = "news_index"),
  url(r'^(?P<pk>\d+)/$', NewDetailView.as_view(), name = "news_detail"),
  url(r'^add/$', permission_required("news.add_new")↵
  (NewCreate.as_view()), name = "news_add"),
  url(r'^(?P<pk>\d+)/edit/$', permission_required("news.change_new")↵
  (NewUpdate.as_view()), name = "news_edit"),
  url(r'^(?P<pk>\d+)/delete/$', permission_required("news.delete_new")↵
  (NewDelete.as_view()), name = "news_delete"),
)
```

Здесь мы сразу же задаем привязки для контроллеров, выполняющих вывод списка новостей, вывод содержимого выбранной новости, добавление, правку и удаление новостей. Также мы указываем, что для перехода на страницы добавления, правки и удаления новости пользователь должен иметь соответствующие права.

Как мы знаем из *главы 10*, по умолчанию классы-контроллеры, выводящие сведения о выбранной записи и реализующие ее правку и удаление, будут искать идентификатор нужной записи в параметре pk. Этот параметр может быть получен из одноименной группы регулярного выражения, созданной в интернет-адресе привязки, или одноименного параметра, переданного методом GET. Потому-то мы и создали в интернет-адресах группы регулярных выражений с именами pk.

Контроллеры

Указав привязки, займемся контроллерами, которых в этом случае будет целых пять. Но сначала напишем два базовых класса, реализующих функциональность, которая будет общей для всех контроллеров.

Базовые классы

Когда пользователь заканчивает правку новости или подтверждает ее удаление нажатием соответствующей кнопки в форме, он будет вновь перенаправлен на список новостей. При этом нам следует выполнить перенаправление на ту страницу списка, с которой он перешел на страницу правки или удаления новости.

Следовательно, в контроллерах этих страниц нам потребуется получить номер страницы, который передается особым GET-параметром, имеющим имя `page`. Номер страницы понадобится нам как в контроллере, чтобы сформировать интернет-адрес перенаправления, так и в шаблоне, чтобы сгенерировать гиперссылку возврата.

Первый из написанных нами базовых классов — `PageNumberMixin` — будет помещать номер страницы в особую переменную контекста данных:

```
class PageNumberMixin(CategoryListMixin):
  def get_context_data(self, **kwargs):
    context = super(PageNumberMixin, self).get_context_data(**kwargs)
    try:
      context["pn"] = self.request.GET["page"]
    except KeyError:
      context["pn"] = "1"
    return context
```

Мы сделали этот класс потомком написанного ранее базового класса `CategoryListMixin`. Так удобнее — вместо того чтобы указывать в списке родителей класса-контроллера оба класса: `CategoryListMixin` и `PageNumberMixin`, нам достаточно будет указать лишь один — последний.

Поместим код, объявляющий новый базовый класс, в модуль `mixins` пакета `generic`.

Второй базовый класс — `PageNumberView` — будет получать номер страницы и добавлять его к интернет-адресу переадресации после успешного сохранения или удаления записи:

```
from django.views.generic.base import View
class PageNumberView(View):
  def post(self, request, *args, **kwargs):
    try:
      pn = request.GET["page"]
    except KeyError:
      pn = "1"
    self.success_url = self.success_url + "?page=" + pn
    return super(PageNumberView, self).post(request, *args, **kwargs)
```

Он является потомком объявленного в модуле `django.views.generic.base` класса `View`, родителя всех классов-контроллеров, в том числе и класса `TemplateView` (см. *главу 11*).

Создадим в пакете `generic` отдельный модуль `controllers` и сохраним код нового класса в нем.

Контроллеры списка новостей и отдельной новости

Контроллеры, выводящие список новостей и выбранную новость, настолько просты, что мы напишем их за один присест.

Сначала вставим в начало модуля `views` код, выполняющий импорт необходимых классов и функций:

```
from django.views.generic.dates import ArchiveIndexView
from django.views.generic.detail import DetailView
from news.models import New
from generic.mixins import CategoryListMixin, PageNumberMixin
```

Напишем код класса-контроллера `NewsListView`, выводящего список новостей:

```
class NewsListView(ArchiveIndexView, CategoryListMixin):
  model = New
  date_field = "posted"
  template_name = "news_index.html"
  paginate_by = 10
  allow_empty = True
  allow_future = True
```

Мы породили его от класса `ArchiveIndexView`, который сам сформирует набор записей и отсортирует записи в нем по указанному нами полю. И при этом задали значение `True` для свойства `allow_future`, чтобы новости, созданные сегодня, присутствовали в списке записей (за подробностями — к *главе 10*).

Код класса-контроллера `NewDetailView`, выводящего сведения о выбранной новости, еще короче:

```
class NewDetailView(DetailView, PageNumberMixin):
  model = New
  template_name = "new.html"
```

Он является потомком класса `DetailView`, который предназначен для выборки из модели записи с полученным через интернет-адрес или GET-параметр идентификатором. По умолчанию он извлекается из группы регулярного выражения или GET-параметра `pk` — для этого мы и создали ранее в интернет-адресе соответствующей привязки группу с таким именем.

Также в списке родителей класса `NewDetailView` присутствует написанный ранее класс `PageNumberMixin`, который, в дополнение к списку категорий и текущему интернет-адресу, поместит в контекст данных полученный с GET-параметром `page` номер страницы.

Контроллеры для добавления, правки и удаления новости

Сразу же напишем классы-контроллеры для добавления, правки и удаления новостей. Они тоже не станут отличаться чрезмерной сложностью.

Вставим в начало модуля `views` код, выполняющий импорт классов и функций, которые будут использоваться во вновь написанном коде:

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.core.urlresolvers import reverse_lazy
from django.contrib.messages.views import SuccessMessageMixin
from django.contrib import messages
from generic.controllers import PageNumberView
```

После чего добавим в конец модуля код, собственно объявляющий эти классы:

```
class NewCreate(SuccessMessageMixin, CreateView, CategoryListMixin):
  model = New
  template_name = "new_add.html"
  success_url = reverse_lazy("news_index")
  success_message = "Новость успешно создана"

class NewUpdate(SuccessMessageMixin, PageNumberView, UpdateView,↪
PageNumberMixin):
  model = New
  template_name = "new_edit.html"
  success_url = reverse_lazy("news_index")
  success_message = "Новость успешно изменена"

class NewDelete(PageNumberView, DeleteView, PageNumberMixin):
  model = New
  template_name = "new_delete.html"
  success_url = reverse_lazy("news_index")
  def post(self, request, *args, **kwargs):
    messages.add_message(request, messages.SUCCESS,↪
    "Новость успешно удалена")
    return super(NewDelete, self).post(request, *args, **kwargs)
```

Отметим следующие моменты:

- классы `UpdateView` и `DeleteView`, от которых порождены классы `NewUpdate` и `NewDelete`, по умолчанию извлекают идентификатор записи из группы регулярных выражений `pk`, созданной в привязке, или одноименного GET-параметра. Ранее в привязках мы создали группы регулярных выражений с таким именем;
- для указания интернет-адреса, по которому будет выполнено перенаправление после успешного добавления, изменения или удаления записи, мы воспользуемся свойством `success_url`. Поскольку мы присваиваем ему значение непосредственно в объявлении класса, то должны применить функцию `reverse_lazy` вместо `reverse`;
- в числе родителей классов `NewUpdate` и `NewDelete`, которые будут изменять и удалять новости, мы укажем написанный ранее класс `PageNumberView`, который добавит к интернет-адресу переадресации номер страницы. Это позволит нам вернуть пользователя на ту страницу списка новостей, с которой он перешел на страницу правки или удаления.

  Класс `PageNumberView` мы указываем в списке родителей самым первым (за исключением класса `SuccessMessageMixin` — по описанной в *главе 12* причине). Это

нужно для того, чтобы его код был выполнен раньше кода классов-контроллеров NewUpdate и NewDelete и успел сформировать в свойстве success_url корректный интернет-адрес перенаправления до того момента, как перенаправление будет выполнено;

- в числе родителей тех же классов мы укажем еще и класс PageNumberMixin. Он поместит в контекст данных шаблона переменную с номером страницы, и мы сможем создать гиперссылку для возврата на страницу списка с этим номером;
- чтобы выполнить отправку сообщения об успешном добавлении или сохранении новости, мы укажем в списке родителей классов-контроллеров NewCreate и NewUpdate класс SuccessMessageMixin, причем обязательно вставим его в самое начало этого списка (в том числе и перед классом PageNumberView). Текст сообщения мы укажем в свойстве success_message;
- в классе NewDelete нам придется писать код отправки сообщения самим — в теле переопределенного метода post.

Вообще-то, все эти моменты были оговорены в предыдущих главах книги. Однако обратить на них внимание еще раз не помешает.

Шаблоны

Осталось написать шаблоны для всех пяти страниц, относящихся к разделу новостей нашего сайта. Все эти шаблоны мы сохраним в папке templates уровня приложения news.

Шаблон списка новостей

Шаблон списка новостей, вероятно, самый сложный из всех пяти. Вот его код:

```
{% extends "main.html" %}
{% block title %}Новости{% endblock %}
{% block main %}
  {% include "generic/messages.html" %}
  <h2>Новости</h2>
  {% if perms.news.add_new %}
    <p><a href="{% url "news_add" %}">Добавить новость</a></p>
  {% endif %}
  {% for object in latest %}
    <h4><a href="{% url "news_detail" pk=object.pk %}
    ?page={{ page_obj.number }}">{{ object.title }}</a></h4>
    <p>{{ object.description}}</p>
    <p class="posted">{{ object.posted|date:"j.m.Y H:i:s" }}</p>
    <p class="buttons bottom-indent">
      {% if perms.news.change_new %}
        <a href="{% url "news_edit" pk=object.pk %}
        ?page={{ page_obj.number }}">Изменить</a>
      {% endif %}
```

```
      {% if perms.news.delete_new %}
        <a href="{% url "news_delete" pk=object.pk %}↵
        ?page={{ page_obj.number }}">Удалить</a>
      {% endif %}
    </p>
  {% endfor %}
  {% include "generic/pagination.html" %}
{% endblock %}
```

Здесь мы, в зависимости от того, имеет ли текущий пользователь необходимые права, выводим гиперссылки для добавления, правки и удаления новости. В остальном здесь все уже нам знакомо.

Сохраним этот шаблон в файле news_index.html.

Шаблон сведений о выбранной новости

Шаблон страницы, выводящей сведения о выбранной новости, мы сохраним в файле new.html. Его код совсем прост:

```
{% extends "main.html" %}
{% load bbcode_tags %}
{% block title %}{{ object.title }}{% endblock %}
{% block main %}
  <h2>{{ object.title }}</h2>
  <div>{{ object.content|bbcode|safe }}</div>
  <p class="posted">{{ object.posted|date:"j.m.Y H:i:s" }}</p>
  <p><a href="{% url "news_index" %}?page={{ pn }}">Назад</a></p>
{% endblock %}
```

Поскольку мы использовали для хранения текста содержимого новости, отформатированного тегами BBCode, обычное поле класса `TextField`, для его вывода применим фильтр шаблона `bbcode`. При этом не забудем загрузить дополнительный модуль шаблонизатора `bbcode_tags`, который обрабатывает этот фильтр. (За подробностями — к *главе 19*.)

Шаблоны добавления, правки и удаления новости

Шаблоны добавления, правки и удаления новости будут храниться в файлах new_add.html, new_edit.html и new_delete.html соответственно.

Код шаблона new_add.html:

```
{% extends "main.html" %}
{% block title %}Добавление новости{% endblock %}
{% block main %}
  <h2>Добавление новости</h2>
  <div class="form">
    <form action="" method="post">
      {% include "generic/form.html" %}
```

```
      <div class="submit-button"><input type="submit"
      value="Добавить"></div>
    </form>
  </div>
  <p><a href="{% url "news_index" %}">Назад</a></p>
{% endblock %}
```

Код шаблона new_edit.html:

```
{% extends "main.html" %}
{% block title %}Правка новости :: {{ object.title }}{% endblock %}
{% block main %}
  <h2>Правка новости</h2>
  <div class="form">
    <form action="" method="post">
      {% include "generic/form.html" %}
      <div class="submit-button"><input type="submit"
      value="Сохранить"></div>
    </form>
  </div>
  <p><a href="{% url "news_index" %}?page={{ pn }}">Назад</a></p>
{% endblock %}
```

Код шаблона new_delete.html:

```
{% extends "main.html" %}
{% load bbcode_tags %}
{% block title %}Удаление новости :: {{ object.title }}{% endblock %}
{% block main %}
  <h2>Удаление новости</h2>
  <h4>{{ object.title }}</h4>
  <div>{{ object.content|bbcode|safe }}</div>
  <p class="posted">{{ object.posted|date:"j.m.Y H:i:s" }}</p>
  <form action="" method="post">
    {% csrf_token %}
    <input type="submit" value="Удалить">
  </form>
  <p><a href="{% url "news_index" %}?page={{ pn }}">Назад</a></p>
{% endblock %}
```

Здесь уж совсем нечего комментировать.

Оформление

Откроем таблицу стилей main.css, что хранится в папке статичных файлов уровня проекта, и добавим в нее пару стилей, использованных в только что написанных шаблонах:

```
.posted {
  font-style: italic;
```

```
  text-align: right;
}
.buttons {
  text-align: right;
}
```

Вывод списка новостей на главной странице

Теперь сделаем так, чтобы на главной странице выводился список из пяти последних новостей.

Сначала внесем небольшие правки в контроллер главной страницы. Откроем модуль `views` пакета приложения `main` и вставим в его начало выражение, импортирующее модель новостей:

```
from news.models import New
```

После чего дополним код класса `MainPageView`, чтобы он выглядел следующим образом (добавленный код выделен полужирным шрифтом):

```
class MainPageView(TemplateView, CategoryListMixin):
  template_name = "mainpage.html"
  news = New.objects.all()[0:5]
  def get_context_data(self, **kwargs):
    context = super(MainPageView, self).get_context_data(**kwargs)
    context["news"] = self.news
    return context
```

Останется лишь вставить в код шаблона главной страницы mainpage.html фрагмент, который выведет список новостей (выделен полужирным шрифтом):

```
...
<h3>Новости</h3>
{% for object in news %}
  <h4><a href="{% url "news_detail" pk=object.pk %}">↵
  {{ object.title }}</a></h4>
  <p>{{ object.description}}</p>
  <p class="posted">{{ object.posted|date:"j.m.Y H:i:s" }}</p>
{% endfor %}
<h3>Рекомендуемые товары</h3>
. . .
```

Заключительные действия

Остается только добавить в панель навигации, что находится в базовом шаблоне main.html, пункт, ведущий на страницу списка новостей. Вставленный в шаблон код выделен полужирным шрифтом:

```
. . .
{% url "main" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}
```

```
class="current"{% endif %}>Главная</a></li>
{% url "news_index" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}
class="current"{% endif %}>Новости</a></li>
{% url "guestbook" as page_url %}
. . .
```

Этим мы обеспечим вывод пункта **Новости** сразу за пунктом **Главная**.

Самым последним действием будет, как обычно, проверка раздела новостей в работе. Сначала выполним вход на сайт, перейдем на страницу списка новостей, а оттуда — на страницу добавления новости (рис. 23.1).

Рис. 23.1. Страница добавления новости

Добавим несколько новостей. (Заодно прямо сейчас можем проверить, как работают их правка и удаление.) Откроем код класса-контроллера `NewsListView` и уменьшим количество записей, выводящихся на страницу записей, до двух, чтобы проверить пагинацию. Вернемся на страницу списка новостей и удостоверимся, что она выглядит так, как показано на рис. 23.2.

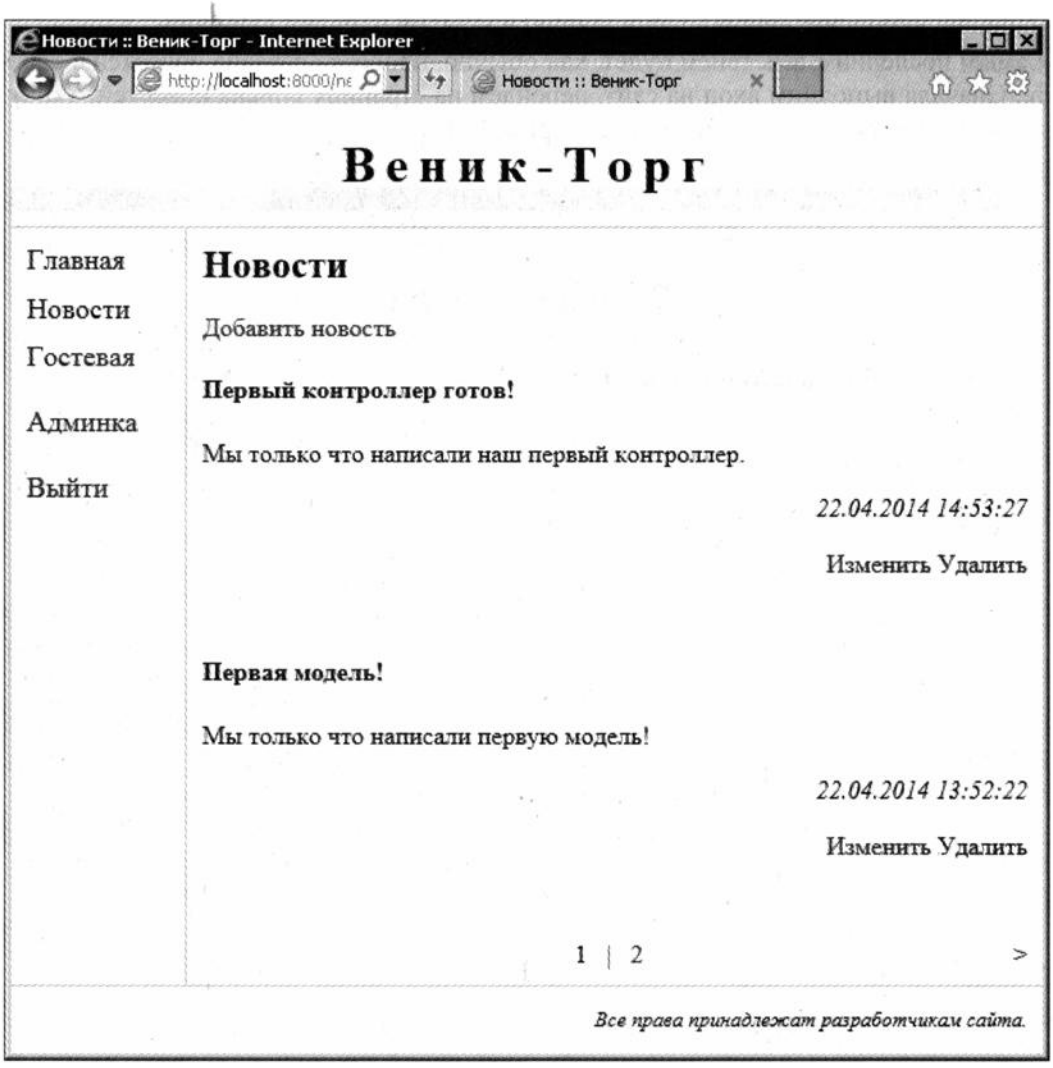


Рис. 23.2. Страница списка новостей

Наконец, перейдем на главную страницу и посмотрим, как выводится на ней список последних новостей (рис. 23.3).

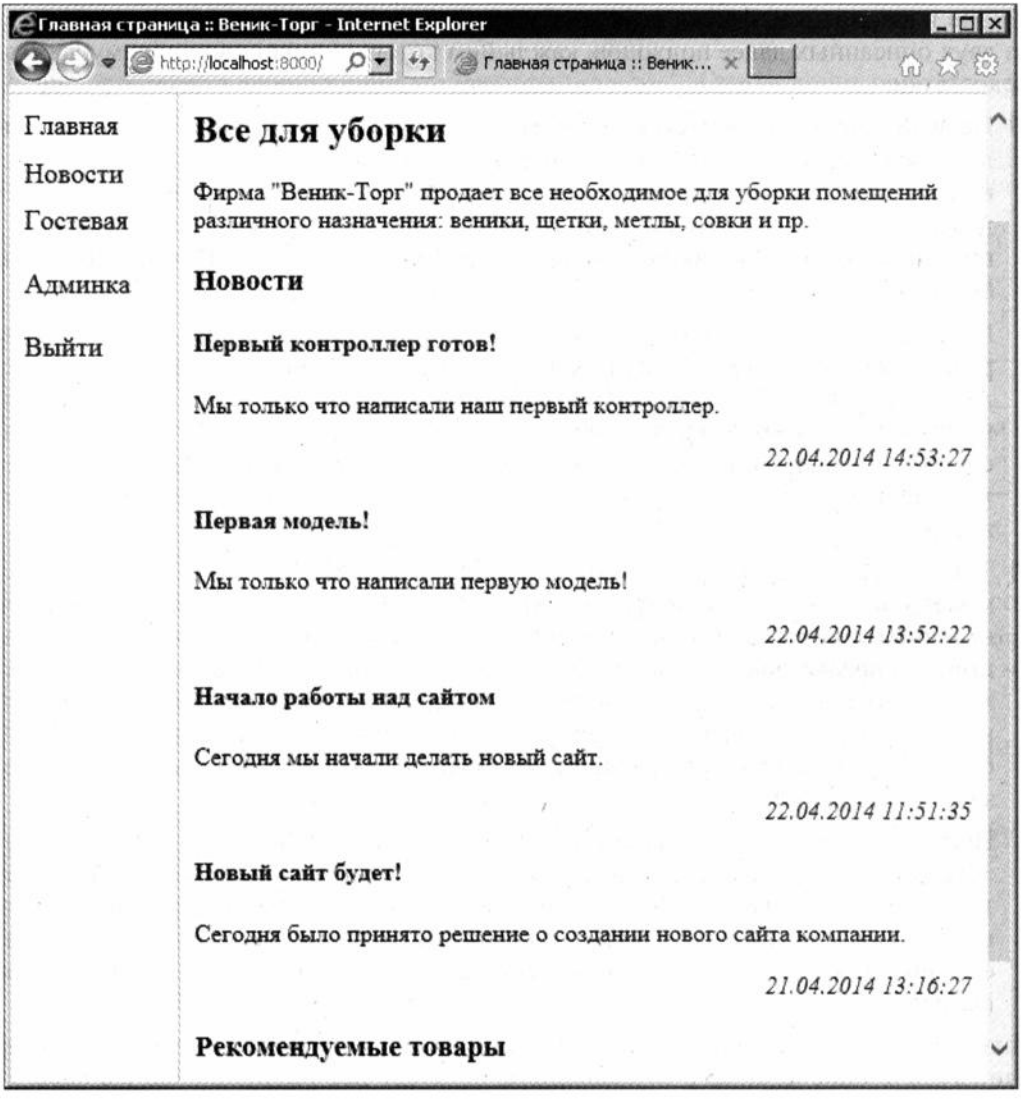


Рис. 23.3. Главная страница со списком последних новостей

Хранилище изображений

Основную функциональность списка новостей мы сделали. Теперь можно приступить к работе над хранилищем графических изображений.

Где и как хранить изображения?

Ранее мы дали пользователям возможность использовать для форматирования текста новостей теги BBCode. А эти теги позволяют, в числе прочего, выводить прямо в тексте графические изображения.

Но где и как хранить файлы этих изображений? Здесь можно прибегнуть к одному из двух описанных далее подходов, каждый из которых имеет свои преимущества и недостатки.

- Первый подход заключается в том, что изображения хранятся в отдельной модели, связанной с основной, в нашем случае, моделью — `New`. К его преимуществам можно отнести простоту как реализации, так и использования, — в самом деле, задавать графические файлы можно прямо на страницах добавления и правки новости — в связанном наборе форм (см. *главу 12*). Почему бы не выбрать его?

  К сожалению, такой подход имеет один огромный недостаток. Тег `[img]`, который помещает в текст изображение, требует указания его интернет-адреса. А этот интернет-адрес может быть сформирован только после того, как файл изображения будет выгружен на сайт и сохранен на диске. И, если брать наш случай, при создании новости мы вставить изображения в ее текст не сможем, т. к. хранящие их файлы будут выгружены на сайт только после сохранения новости.

- Тогда нам придется прибегнуть к другому подходу. Мы создадим отдельную модель, в которой станем хранить все файлы, выгруженные на сайт определенным пользователем, и отдельное приложение для манипуляции ими. При создании или правке новости мы будем выводить список этих файлов с гиперссылками, позволяющими удалить ненужный файл, и форму для выгрузки на сайт новых файлов. Причем и получение списка файлов, и выгрузка, и удаление будут выполняться с применением технологии AJAX — так мы исключим перезагрузки страницы.

  Непосредственно при создании или правке новости пользователь сможет загрузить файлы в хранилище, получить их список и удалить файл, который почему-то стал ему не нужен. После чего он просто щелкнет на требуемом файле и вставит в содержимое новости полностью сформированный тег `[img]`, включающий его интернет-адрес. Последний мы сможем получить без труда — файл-то уже выгружен.

Иначе говоря, мы создадим *хранилище изображений*. Оно будет универсальным, единым для определенного пользователя, включит все выгруженные им файлы и позволит использовать их в тексте новостей, статьях блога и, вообще, везде, где допускается применение BBCode-форматирования.

Хранилище изображений довольно сложно и трудоемко в реализации. Однако выгоды его применения перевешивают все недостатки.

Приложение

В *главе 20* мы решили создать для хранилища изображений отдельное приложение `imagepool`. Создадим его и включим в список активных:

```
INSTALLED_APPS = (
  . . .
  'imagepool',
)
```

Модель

Модель хранилища изображений мы назовем `ImagePool`. Она включит поля для хранения пользователя, выгрузившего изображение, дату и время его выгрузки (по этому полю мы можем потом отсортировать записи) и собственно выгруженное изображение. Более полно параметры полей вместе с их именами приведены в табл. 23.2.

***Таблица 23.2.** Поля модели `ImagePool`*

Имя	Тип	Параметры
`user`	Связь	Устанавливает связь с моделью `User`
`uploaded`	Дата и время	Значение заносится при создании новой записи
`image`	Графический файл	

Все поля этой модели сделаем обязательными.

Поле `uploaded` нужно сделать индексированным (индекс по поля `user` Django создаст сама). И укажем изначальную сортировку модели по полям `user` и `uploaded`. Сортировка по последнему полю будет выполняться по убыванию — так мы вынесем недавно выгруженные файлы в начало списка.

В качестве папок хранения выгруженных файлов зададим путь вида news/<год>/<месяц>, как решили еще в *главе 20*.

В соответствии с принятыми решениями, откроем модуль `models` пакета вновь созданного приложения и занесем в него код модели:

```
from django.contrib.auth.models import User
class ImagePool(models.Model):
  user = models.ForeignKey(User)
  uploaded = models.DateTimeField(db_index = True, auto_now_add = True,↳
  verbose_name = "Выгружен")
  image = models.ImageField(upload_to = "imagepool/%Y/%m",↳
  verbose_name = "Изображение")
  class Meta:
    ordering = ["user", "-uploaded"]
    verbose_name = "изображение"
    verbose_name_plural = "изображения"
  def delete(self, *args, **kwargs):
    self.image.delete(save = False)
    super(ImagePool, self).delete(*args, **kwargs)
```

Поскольку мы также собираемся дать пользователю возможность удаления выгруженных файлов, то должны позаботиться о том, чтобы удаленные файлы не стали «мусорными». Для этого мы переопределим метод `delete`, в котором реализуем удаление и самого хранящегося в записи файла.

Закончив ввод кода модели, выполним синхронизацию с базой данных.

Привязки

Выполним привязку приложения `imagepool` на уровне проекта — в модуле `urls` его пакета, — добавив в список привязок элемент:

```
urlpatterns = patterns('',
  . . .
  url(r'^imagepool/', include('imagepool.urls')),
)
```

Привязки уровня приложения мы укажем в модуле `urls` пакета приложения, который нам в какой уже раз приходится создавать самим, т. к. Django «забывает» это сделать:

```
from django.contrib.auth.decorators import login_required
from imagepool.views import get_list, upload_file, delete_file
urlpatterns = patterns('',
  url(r'^$', login_required(get_list), name = "imagepool_index"),
  url(r'^upload/$', login_required(upload_file),↵
  name = "imagepool_upload"),
  url(r'^(?P<pk>\d+)/delete/$', login_required(delete_file),↵
  name = "imagepool_delete"),
)
```

Здесь мы задали привязки для контроллеров `get_list`, `upload_file` и `delete_file`, которые будут выполнять, соответственно, формирование списка файлов, выгрузку файла на сайт и удаление выбранного пользователем файла. Поскольку выполняемые ими задачи достаточно специфичны, мы реализуем их в виде обычных функций.

Для всех контроллеров мы указали требование обязательного выполнения пользователем входа на сайт.

Контроллеры

Итак, нам нужно написать три контроллера. Но подождем открывать модуль `views` пакета приложения. Давайте определимся с тем, какие сведения о файлах будет содержать список, генерируемый контроллером `get_list`, и, вообще, определимся с принципами, по которым будет функционировать наше хранилище изображений.

Принципы работы хранилища изображений

Технология *AJAX* (Asynchronous JavaScript and XML, асинхронный JavaScript и XML), как мы знаем, позволяет вести обмен данными между серверной программой и Web-обозревателем асинхронно и без перезагрузки страницы. В частности, она позволяет оперативно отправлять и загружать данные, которые могут быть помещены на страницу или обработаны каким-либо иным способом. Ее-то мы и применим для загрузки списка графических файлов, находящихся в хранилище, и удаления файла.

Однако выгрузить файл таким же образом мы не сможем — технология AJAX просто не позволяет сделать это. Тем не менее, мы сможем выгрузить его традиционным способом — с помощью обычной формы. Мы создадим на странице невидимый встроенный фрейм (тег `<iframe>`) и перенаправим вывод контроллера `upload_file`, который получает и сохраняет выгружаемый файл, в него. Тогда, с одной стороны, файл будет благополучно выгружен, а с другой, страница не будет перезагружена.

Теперь подумаем, какие данные включит список графических файлов, генерируемый контроллером `get_list`. Чтобы сделать Web-сценарий, выводящий полученный список файлов на экран, как можно проще, мы переложим бóльшую часть работы по подготовке данных на этот контроллер.

Далее, список будет разбиваться на страницы, каждая из которых включит четыре файла. Здесь мы применим хорошо знакомый нам по *главе 9* пагинатор Django.

Список файлов мы оформим в виде словаря Django (в терминологии языка JavaScript такие структуры данных называются *объектами*). Этот словарь будет содержать следующие элементы (*свойства* в терминологии JavaScript):

- `images` — собственно перечень графических файлов, оформленный в виде списка Python (в языке JavaScript поддерживаются аналогичные структуры данных, называемые *массивами*). Каждый элемент этого списка будет представлять собой словарь с элементами:
  - `src` — интернет-адрес, используемый для вывода файла на экран;
  - `delete_src` — интернет-адрес, предназначенный для удаления файла. Мы создадим на его основе гиперссылку, щелкнув на которой, пользователь выполнит удаление файла;
- `prev_url` — интернет-адрес для получения предыдущей страницы списка файлов. Если в настоящий момент выводится первая страница, этот элемент будет хранить пустую строку;
- `next_url` — интернет-адрес для получения следующей страницы списка файлов или пустая строка, если в настоящий момент выводится последняя страница.

Для упрощения программирования Web-сценариев мы применим известную библиотеку jQuery, загрузить которую можно с ее «домашнего» сайта (**http://jquery.com/**). Исполняемый файл библиотеки поместим в папку static уровня проекта и дадим ему имя jquery.js.

Хоть расшифровка аббревиатуры «AJAX» и включает слово «XML», нам совсем не обязательно пересылать данные в этом формате. Мы можем применить более простой и компактный формат *JSON* (JavaScript Object Notation, объектная нотация JavaScript), который представляет собой исходный код JavaScript, создающий нужный нам объект. К тому же, и язык Python, и библиотека jQuery уже включают в свой состав поддержку этого формата.

Контроллер, формирующий список файлов

Вот теперь можно открыть модуль `views` пакета приложения и начать программирование контроллеров.

Код функции-контроллера `get_list`, которая будет заниматься созданием списка файлов, довольно велик:

```
from django.http import HttpResponse
from django.core.paginator import Paginator
from django.core.urlresolvers import reverse
import json
from imagepool.models import ImagePool
def get_list(request):
  try:
    page_num = request.GET["page"]
  except KeyError:
    page_num = 1
  paginator = Paginator(ImagePool.objects.filter(user = request.user), 4)
  try:
    page = paginator.page(page_num)
  except InvalidPage:
    page = paginator.page(1)
  output = {}
  output["images"] = []
  for image in page:
    output["images"] = output["images"] + [{"src": image.image.url,↵
    "delete_src": reverse("imagepool_delete",↵
    kwargs = {"pk": image.pk})}]
  if page.has_previous():
    output["prev_url"] = reverse("imagepool_index") + "?page=" + ↵
    str(page.previous_page_number())
  else:
    output["prev_url"] = ""
  if page.has_next():
    output["next_url"] = reverse("imagepool_index") + "?page=" + ↵
    str(page.next_page_number())
  else:
    output["next_url"] = ""
  return HttpResponse(json.dumps(output),↵
  content_type = "application/json")
```

Сначала мы получаем из GET-параметра `page` номер страницы, извлекаем из модели `ImagePool` все изображения, сохраненные текущим пользователем, формируем на основе результирующего списка записей сначала пагинатор, а потом — страницу с полученным ранее номером. Далее мы создаем словарь `output`, который и станет списком файлов, и наполняем его данными.

А теперь — внимание! В написанном нами коде встречаются два момента, не рассмотренные в предыдущих главах:

- для преобразования словаря Python в JSON-код мы используем метод `dumps` из стандартного модуля Python `json`. Строку с полученным кодом мы передадим первым параметром конструктору класса `HttpResponse` (см. *главу 6*);
- нам следует явно указать, что Web-обозревателю передается именно JSON-код. Для этого передадим в качестве значения необязательного параметра `content_type` конструктора класса `HttpResponse` строку `"application/json"`.

Контроллеры, сохраняющие и удаляющие файл

Функции-контроллеры `upload_file` и `delete_file`, которые будут выполнять выгрузку и удаление файла, не столь сложны. Посмотрим, например, на код контроллера `upload_file`:

```
def upload_file(request):
  if request.method == "POST":
    if request.FILES["file_to_upload"]:
      image = ImagePool(user = request.user,↵
      image = request.FILES["file_to_upload"])
      image.save()
      return HttpResponse("!!!")
    else:
      return HttpResponse("!!!")
  else:
    return HttpResponse("!!!")
```

Здесь мы проверяем, были ли получены методом POST какие-либо данные и включают ли они выгруженный файл (параметр `file_to_upload`, соответствующий одноименному полю формы, которую мы потом создадим в шаблоне). Если все это верно, мы добавляем в модель `ImagePool` новую запись, хранящую выгруженный файл.

Любой контроллер Django по окончании работы должен что-то отправить Web-обозревателю. Обычно это HTML-код страницы, но в нашем случае делать это излишне, т. к. отправленные контроллером данные будут выводиться в невидимом фрейме и, по сути, просто не нужны. Так что мы можем отправить любое строковое значение — автор выбрал три восклицательных знака, но с таким же успехом можно отправить и пустую строку.

Код контроллера `delete_file` еще проще:

```
def delete_file(request, pk):
  try:
    ImagePool.objects.get(pk = pk).delete()
  except ImagePool.DoesNotExist:
    pass
  return HttpResponse(json.dumps({"status": 1}),↵
  content_type = "application/json")
```

Он удаляет из модели ImagePool запись с полученным в интернет-адресе идентификатором и отправляет Web-обозревателю сообщение об успехе выполненной операции. В этом случае Web-сценарий, который запустит процесс удаления, будет ожидать получения данных в формате JSON, которые должен отправить наш контроллер. Опять же, это могут быть произвольные данные.

Шаблоны

HTML-код, создающий на экране интерфейс хранилища изображений: список файлов и форму для выгрузки — мы вынесем в отдельный подгружаемый шаблон. Нам также понадобится внести небольшие правки в код шаблонов страниц добавления и правки новости.

Универсальный шаблон хранилища изображений

Универсальному шаблону хранилища изображений мы дадим имя file_lister.html и сохраним его в подпапке generic папки шаблонов уровня проекта, как и остальные универсальные шаблоны:

```
<table id="imagepool_image_list" href="{% url "imagepool_index" %}">
  <tr>
    <td class="link"><a id="imagepool_prev" href="#">&lt;</a></td>
    <td class="image"> </td>
    <td class="image"> </td>
    <td class="image"> </td>
    <td class="image"> </td>
    <td class="link"><a id="imagepool_next" href="#">&gt;</a></td>
  </tr>
</table>
<div class="form">
  <form id="imagepool_form" action="{% url "imagepool_upload" %}"
  method="post" enctype="multipart/form-data" target="imagepool_output">
    {% csrf_token %}
    <div class="form-field">
      <div class="label">Файл для выгрузки</div>
      <div class="control"><input type="file" id="file_to_upload"
      name="file_to_upload"></div>
    </div>
    <div class="submit-button"><input type="submit"
    value="Выгрузить"></div>
  </form>
  <iframe id="imagepool_output" name="imagepool_output"></iframe>
</div>
```

Таблица imagepool_image_list послужит в качестве списка файлов. Ее первая и последняя ячейки (с привязанным к ним стилевым классом link) содержат гиперссылки для «листания» списка, а в остальных ячейках (к ним привязан стилевой класс image) будут выводиться сами файлы.

Отметим, что в теге <table>, формирующем эту таблицу, мы создали атрибут href. В качестве его значения мы указали интернет-адрес, к которому ранее привязали контроллер get_list, — поскольку этот интернет-адрес не содержит номера страницы, при обращении к нему будет получена первая страница списка. Так мы сможем впоследствии вывести первую страницу списка сразу после загрузки страницы или выгрузки очередного файла.

Форма imagepool_form предназначена для выгрузки файлов. Файл указывается в поле ввода файла file_to_upload. В качестве цели для вывода полученной от контроллера информации мы указали встроенный фрейм imagepool_output, который находится ниже формы. Впоследствии с помощью особого стиля мы сделаем его невидимым.

Исправленные шаблоны добавления и правки новости

Сразу же внесем правки в код шаблонов для страниц добавления и правки новости new_add.html и new_edit.html. Начнем с первого шаблона, во втором правки будут аналогичными.

Сначала добавим в шаблон вот такой код (выделен полужирным шрифтом):

```
{% extends "main.html" %}
{% load staticfiles %}
{% block additional_js %}
  <script src="{% static "jquery.js" %}" type="text/javascript"></script>
  <script src="{% static "imagepool.js" %}"
  type="text/javascript"></script>
{% endblock %}
{% block title %}Добавление новости{% endblock %}
. . .
```

Еще в базовом шаблоне base.html мы создали блок additional_js, предназначенный для размещения привязок файлов Web-сценариев. Настала пора применить его в деле. Здесь мы поместили в этот блок привязки файла jquery.js с кодом библиотеки jQuery и файла imagepool.js с кодом, реализующим клиентскую функциональность нашего хранилища изображений.

И добавим в шаблон код, который загрузит только что созданный универсальный шаблон хранилища изображений file_lister.html (добавленный код выделен полужирным шрифтом):

```
. . .
<div class="form">
  . . .
</div>
{% include "generic/file_lister.html" %}
<p><a href="{% url "news_index" %}">Назад</a></p>
. . .
```

Оформление

Дополним содержимое таблицы стилей main.css следующим кодом:

```
#imagepool_output {
  display: none;
}
#imagepool_image_list {
  margin-top: 60px;
}
#imagepool_image_list td {
  text-align: center;
  width: 160px;
  height: 120px;
}
#imagepool_image_list td.link {
  font-size: xx-large;
  width: 20px;
}
#imagepool_image_list td img {
  max-width: 160px;
  max-height: 100px;
}
#imagepool_image_list td a:link, #imagepool_image_list td a:visited,
#imagepool_image_list td a:active, #imagepool_image_list td a:hover{
  text-decoration: none;
}
#imagepool_image_list td a.disabled:link,
#imagepool_image_list td a.disabled:visited,
#imagepool_image_list td a.disabled:active,
#imagepool_image_list td a.disabled:hover{
  display: none;
}
```

Здесь мы задаем параметры различных элементов интерфейса хранилища изображений, в том числе размеры списка изображений и его элементов. Мы также предписываем, чтобы гиперссылки «листания» списка изображения при привязке к ним стилевого класса `disabled` становились невидимыми, равно как и встроенный фрейм `imagepool_output`.

Web-сценарий

Web-сценарий, реализующий клиентскую функциональность нашего хранилища изображений, довольно сложен. Поэтому мы рассмотрим его код по частям, по мере того, как будем его писать. Создадим в папке static уровня проекта файл imagepool.js, откроем его и приступим.

Сначала напишем код, объявляющий служебную функцию `getCaretPos`. В качестве единственного параметра она принимает объект jQuery, представляющий поле вво-

да или область редактирования, и возвращает текущую позицию текстового курсора в виде числа:

```
function getCaretPos(jQobject) {
  var obj = jQobject.get(0);
  obj.focus();
  if (document.selection) {
    var sel = document.selection.createRange();
    var clone = sel.duplicate();
    sel.collapse(true);
    clone.moveToElementText(obj);
    clone.setEndPoint("EndToEnd", sel);
    return clone.text.length;
  } else if (obj.selectionStart !== false) return obj.selectionStart;
  else return 0;
}
```

> ***Модифицированная версия кода***
>
> Этот код представляет собой несколько модифицированную автором версию кода, доступного по интернет-адресу **http://kurilka.co.ua/archives/getcaretpos/**.

Далее объявим функцию `getImageList`. Она получит в качестве параметра интернет-адрес очередной страницы списка файлов в виде строки, выполнит загрузку списка, его обработку и вывод на экран всех хранящихся в нем файлов:

```
function getImageList(url) {
  $.getJSON(url, function(data) {
    var images = $("#imagepool_image_list td.image");
    images.empty();
    for (var i = 0; i < data.images.length; i++) {
      $(images.get(i)).append("<a class='insert' href='" +
      data.images[i].src + "'><img src='" + data.images[i].src +
      "'></a><br><a class='delete' href='" + data.images[i].delete_src +
      "'>Удалить</a>");
    }
    var link = $("#imagepool_prev");
    if (data.prev_url == "") {
      link.addClass("disabled");
      link.attr("href", "#");
    } else {
      link.removeClass("disabled");
      link.attr("href", data.prev_url);
    }
    var link = $("#imagepool_next");
    if (data.next_url == "") {
      link.addClass("disabled");
      link.attr("href", "#");
```

```
    } else {
      link.removeClass("disabled");
      link.attr("href", data.next_url);
    }
  });
}
```

Как только JSON-код перечня файлов, сформированного контроллером `get_list`, будет загружен, функция выполнит следующие действия:

- очистит список — таблицу `imagepool_file_list` — от уже присутствующих в ней файлов;
- создаст в каждой ее ячейке с привязанным стилевым классом `image` две гиперссылки, первая из которых будет содержать само изображение, хранящееся в файле, а другая послужит для его удаления;
- если это первая страница списка, скроет гиперссылку для перехода на предыдущую страницу, привязав к ней стилевой класс `disabled`, в противном случае запишет в атрибут `href` ее тега `<a>` интернет-адрес предыдущей страницы;
- выполнит аналогичные действия с гиперссылкой для перехода на следующую страницу списка файлов.

Наберем теперь основной код, который будет запущен на выполнение сразу после загрузки страницы:

```
$(function() {
  var contentField = $("form textarea[name=content]");
  $("#imagepool_prev, #imagepool_next").click(function(evt) {
    evt.preventDefault();
    getImageList($(this).attr("href"));
  });
  $("#imagepool_output").on("load", function() {
    getImageList($("#imagepool_image_list").attr("href"));
  });
  $("#imagepool_image_list").on("click", "td.image a.insert",
  function(evt) {
    evt.preventDefault();
    var content = contentField.val();
    var position = getCaretPos(contentField);
    content = content.substring(0, position) + "[img]" +
    location.protocol + "//" + location.host + $(this).attr("href") +
    "[/img]" + content.substring(position);
    contentField.val(content);
  });
  $("#imagepool_image_list").on("click", "td.image a.delete",
  function(evt) {
    evt.preventDefault();
    if (window.confirm("Удалить изображение?")) {
      $.getJSON($(this).attr("href"), function(data) {
```

```
            getImageList($("#imagepool_image_list").attr("href"));
        });
    }
});
getImageList($("#imagepool_image_list").attr("href"));
});
```

Он сделает следующее:

- получит объект jQuery, представляющий область редактирования `content`, где вводится содержимое новости, чтобы не выполнять поиск каждый раз, когда нам понадобится получить к ней доступ. Соответствующее поле модели носит у нас имя `content`, следовательно, элемент управления получит то же самое имя — это обычное поведение шаблонизатора Django;
- привяжет к гиперссылкам «листания» списка файлов обработчик, который будет выполнять загрузку и вывод соответствующей страницы списка;
- привяжет к событию `load` вложенного фрейма `imagepool_output` обработчик. Он будет выполнен сразу же после выгрузки файла и загрузит первую страницу списка, взяв ее интернет-адрес из атрибута `href` тега `<table>`, который формирует таблицу `imagepool_file_list`;
- опосредованно привяжет ко всем гиперссылкам, в которых выводятся графические файлы в списке, обработчик, который при щелчке мышью вставит в содержимое области редактирования `content` на место текстового курсора тег `[img]`, выводящий это изображение.

> ***Требование полного интернет-адреса графического файла***
>
> К сожалению, библиотека django-precise-bbcode, обеспечивающая поддержку тегов BBCode (речь об этой библиотеке шла в *главе 19*), требует, чтобы в теге `[img]` находился полный интернет-адрес графического файла, иначе изображение не будет выведено на экран. Так что нам при реализации вставки этого тега придется формировать полный интернет-адрес графического файла.

- опосредованно привяжет ко всем гиперссылкам **Удалить**, находящимся в списке файлов, обработчик, который при щелчке мышью запустит процесс удаления выбранного изображения, предварительно спросив пользователя, хочет ли он этого;
- загрузит первую страницу списка файлов.

Проверим теперь хранилище изображений в работе. Войдем на наш сайт, создадим какую-либо новость, загрузим в хранилище несколько изображений (сразу штук 5–7 — чтобы проверить, как работает пагинация) и посмотрим, правильно ли выводится их список (рис. 23.4).

Вставим в текст содержимого новости одно из выгруженных изображений, щелкнув на нем мышью, сохраним новость и полюбуемся на результат (рис. 23.5).

Рис. 23.4. Интерфейс хранилища изображений

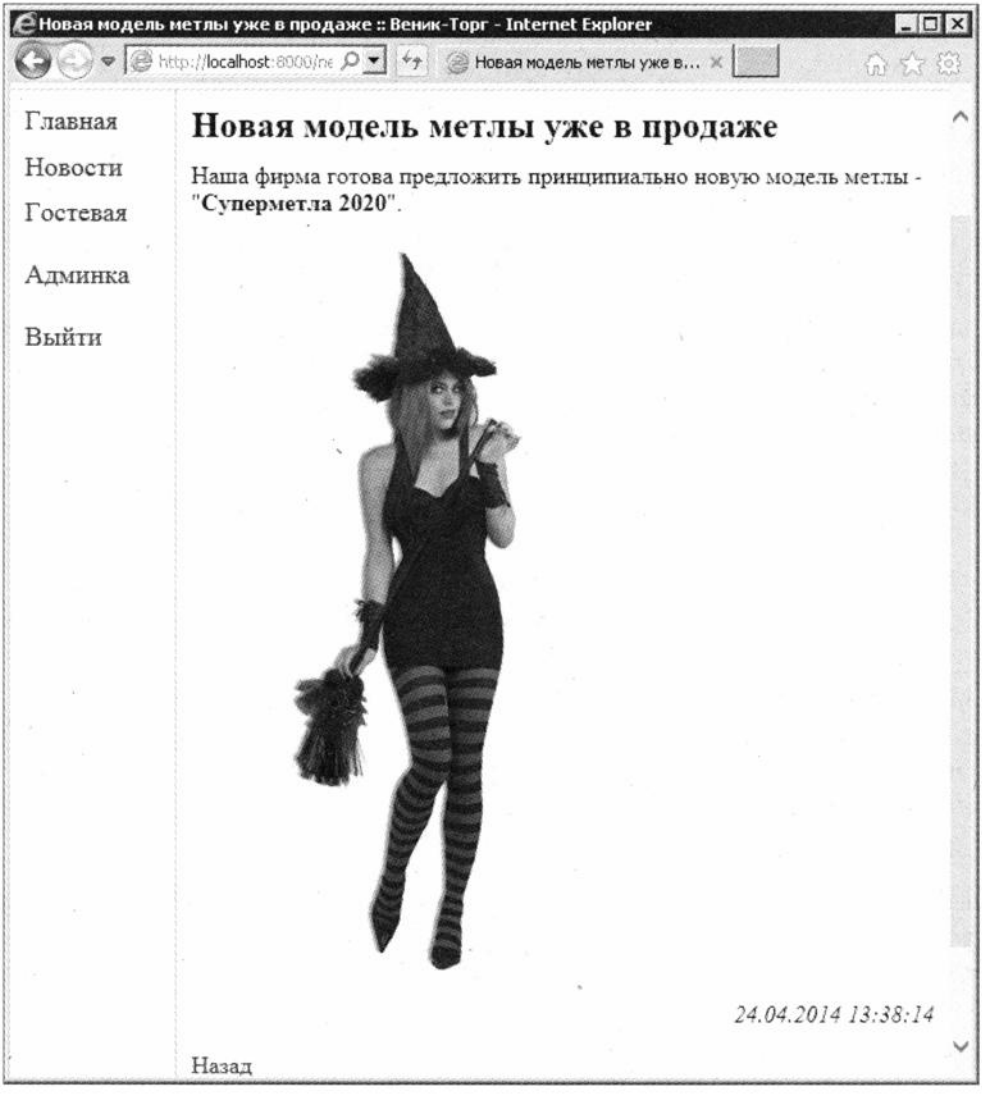


Рис. 23.5. Содержимое новости с графическим изображением

Что дальше?

В этой главе мы создали список новостей с возможностью форматирования их текста тегами BBCode и универсальное хранилище изображений, которые будут использоваться не только в тексте новостей, но и в статьях будущего блога. Уф! Теперь можно немного отдохнуть...

...Тем более, что вскоре нам предстоит работа над главной частью нашего сайта — каталогом товаров. В следующей главе мы сделаем список категорий.

ГЛАВА 24

Список категорий товаров

Предыдущая глава была очень насыщенной. Мы разработали приложение списка новостей и реализовали возможность их добавления, правки и удаления с помощью специальных страниц, входящих в состав административного раздела нашего сайта. Наконец, мы написали работающее по технологии AJAX приложение хранилища изображений, позволяющее свести воедино все изображения, что принадлежат текущему пользователю, представить их на экране в удобном списке, вставить в текст статьи выбранное изображение и при необходимости удалить его.

В этой главе мы приступим к разработке перечня товаров. И для начала — сделаем приложение, работающее со списком категорий.

Для простоты мы не станем делать отдельные страницы для добавления, правки и удаления категорий, а вместо этого используем набор форм, знакомый нам по *главе 12*.

Приложение

Приложение списка категорий будет называться `categories`. Создадим его и обязательно впишем в список активных приложений, иначе Django его не «увидит»:

```
INSTALLED_APPS = (
  . . .
  'categories',
)
```

Модель

Модель `Category`, хранящая список категорий, будет очень простой. Она включит поля для хранения названия и порядкового номера категории. Последнее поле позволит нам произвольно менять порядок следования категорий в списке, а первое поле следует сделать уникальным, чтобы исключить случайное дублирование какой-либо категории. Более подробно параметры полей перечислены в табл. 24.1.

Таблица 24.1. Поля модели `Categories`

Имя	Тип	Параметры
`name`	Текстовый	Максимальная длина — 30 символов. Должно хранить уникальные значения
`order`	Короткое положительное целое число	Значение по умолчанию — 0

Оба поля будут обязательными и индексированными. Также мы укажем для записей модели сортировку сначала по полю `order`, а потом по полю `name`.

Напишем код модели в модуле `models` пакета вновь созданного приложения:

```
class Category(models.Model):
  name = models.CharField(max_length = 30, db_index = True,↵
  unique = True, verbose_name = "Название")
  order = models.PositiveSmallIntegerField(default = 0,↵
  db_index = True, verbose_name = "Порядковый номер")
  def __str__(self):
    return self.name
  class Meta:
    ordering = ["order", "name"]
    verbose_name = "категория"
    verbose_name_plural = "категории"
```

Здесь мы переопределяем в модели метод `__str__` таким образом, чтобы он возвращал название категории. Если же мы этого не сделаем, в раскрывающемся списке категорий, находящемся на страницах добавления и правки товара, что мы создадим потом, все категории будут представлены в виде текста **Category object**. (Подробнее о методе `__str__` рассказывалось в *главе 5*.)

Не забываем после создания каждой модели выполнять синхронизацию с базой данных.

Привязки

Привязка уровня проекта выполняется давно знакомым нам способом. Достаточно добавить в список привязок модуль `urls` пакета проекта всего один элемент:

```
urlpatterns = patterns('',
  . . .
  url(r'^categories/', include('categories.urls')),
)
```

Привязки уровня приложения мы укажем в модуле `urls` пакета приложения, который нам, опять же, придется создать самим:

```
from django.contrib.auth.decorators import login_required
from categories.views import CategoriesEdit
urlpatterns = patterns('',
```

```
  url(r'^$', login_required(CategoriesEdit.as_view()),↵
  name = "categories_edit"),
)
```

Поскольку единственным действием, выполняемым этим приложением, будет реализация правки списка категорий, мы укажем здесь всего одну привязку — к классу-контроллеру `CategoriesEdit`, который и будет этим заниматься. Зададим для него требование обязательного входа на сайт.

Контроллер

Класс-контроллер `CategoriesEdit` мы сделаем потомком самого простого класса такого рода — `TemplateView`. Увы, ни один из классов-контроллеров более высокого уровня, изученных нами в *главах 10* и *12*, не поддерживает наборы форм. Поэтому нам придется писать весь код, создающий набор форм и сохраняющий введенные в него данные, самостоятельно, и в таком случае лучше взять за основу самый простой и быстрый в работе класс.

Получившийся класс-контроллер сравнительно несложен. Давайте посмотрим на его полный код:

```
from django.forms.models import modelformset_factory
from django.views.generic.base import TemplateView
from django.shortcuts import redirect
from django.contrib import messages
from categories.models import Category
from generic.mixins import CategoryListMixin

CategoriesFormset = modelformset_factory(Category, can_delete = True)

class CategoriesEdit(TemplateView, CategoryListMixin):
  template_name = "categories_edit.html"
  formset = None
  def get(self, request, *args, **kwargs):
    self.formset = CategoriesFormset()
    return super(CategoriesEdit, self).get(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(CategoriesEdit, self).get_context_data(**kwargs)
    context["formset"] = self.formset
    return context
  def post(self, request, *args, **kwargs):
    self.formset = CategoriesFormset(request.POST)
    if self.formset.is_valid():
      self.formset.save()
      messages.add_message(request, messages.SUCCESS,↵
      "Список категорий успешно изменен")
      return redirect("categories_edit")
    else:
      return super(CategoriesEdit, self).get(request, *args, **kwargs)
```

Мы создаем класс набора форм на основе модели `Category` с помощью функции `modelformset_factory`. При этом мы даем пользователю возможность только удалять записи, но никак не переупорядочивать их.

Дело в том, что при переупорядочивании записей набор форм меняет местами не сами записи в модели, а всего лишь содержимое полей этих записей. При этом значения поля `id`, где хранится уникальный идентификатор записи, не изменяются. Это значит, что, если мы, скажем, поменяем местами категории «Щетки» и «Метлы», то поменяются местами лишь значения полей `name` и `order` этих записей, а значения поля `id` не изменятся. И тогда все метлы окажутся привязанными к категории «Щетки», а все щетки — к категории «Метлы».

Вот именно поэтому, кстати, мы и ввели в модель `Category` поле `order`. Оно позволит нам переупорядочить категории в списке, задав для них подходящие порядковые номера.

После успешного сохранения введенных в набор форм данных мы выполняем перенаправление пользователя на ту же самую страницу. Так он сразу увидит результат своих действий.

Остальной код на приведенном ранее листинге уже знаком нам по *главе 12* и никаких пояснений не требует.

Шаблоны

Закончив с контроллером, приступим к написанию шаблонов, которых в этом случае будет два.

Универсальный шаблон набора форм

Поскольку мы собираемся применить набор форм еще на страницах добавления и правки товаров, удобнее вынести создающий его HTML-код в отдельный подгружаемый шаблон. Этот шаблон получит имя formset.html и будет храниться в папке templates/generic уровня проекта.

Оформим набор форм в виде таблицы, каждая строка которой соответствует одной из форм набора, а каждая ячейка — одному из полей формы и будет содержать относящийся к этому полю элемент управления. И не забудем создать строку «шапки», в которой укажем надписи для элементов управления.

Полный код нового шаблона довольно объемен:

```
{% csrf_token %}
{{ formset.management_form }}
<table class="form">
  <tr>
    <th></th>
    {% with form=formset|first %}
      {% for field in form.visible_fields %}
```

```
        <th>
          {{ field.label }}
          {% if field.help_text %}
            <br>{{ field.help_text }}
          {% endif %}
        </th>
      {% endfor %}
    {% endwith %}
  </tr>
  {% for form in formset %}
    <tr>
      <td>
        {% for field in form.hidden_fields %}
          {{ field }}
        {% endfor %}
      </td>
      {% for field in form.visible_fields %}
        <td>
          {% if field.errors.count > 0 %}
            <div class="error-list">
              {{ field.errors }}
            </div>
          {% endif %}
          <div class="control">{{ field }}</div>
        </td>
      {% endfor %}
    </tr>
  {% endfor %}
</table>
```

Прежде всего, мы привязываем к таблице, которая создаст набор форм, стилевой класс `form`. Так мы используем для оформления таблицы уже имеющиеся стили.

Чтобы создать «шапку» таблицы, мы извлекаем первую форму из набора (для чего применяем фильтр `first` — см. *главу 7*), перебираем в цикле все ее видимые поля и для каждого поля создаем ячейку заголовка, в которую помещаем текст надписи соответствующего поля и, если таковой имеется, поясняющий текст. Дополнительно в самом начале строки заголовка мы помещаем пустую ячейку, которую впоследствии с помощью особого стиля сделаем невидимой (зачем нужна пустая ячейка, мы узнаем совсем скоро).

Для каждой формы набора мы создаем обычную строку таблицы. В самом ее начале мы формируем ячейку, куда заносим все невидимые поля формы. (Вот поэтому нам и нужна пустая ячейка в начале строки заголовка — если ее не создать, таблица будет выведена на экран некорректно.) В последующих ячейках строки мы размещаем списки сообщений об ошибках ввода и элементы управления для всех видимых полей формы.

Шаблон страницы списка категорий

Шаблон списка категорий окажется совсем простым — ведь мы вынесли весь код, что генерирует набор форм, в отдельный универсальный шаблон. Давайте посмотрим сами:

```
{% extends "main.html" %}
{% block title %}Категории{% endblock %}
{% block main %}
  {% include "generic/messages.html" %}
  <h2>Категории</h2>
  <form action="" method="post">
    {% include "generic/formset.html" %}
    <div class="submit-button"><input type="submit"
    value="Сохранить"></div>
  </form>
{% endblock %}
```

Сохраним его в файле categories_edit.html в папке шаблонов уровня приложения categories.

Оформление

И добавим в таблицу стилей main.css очередную порцию стилей, которые зададут оформление для элементов новых шаблонов. На этот раз порция стилей будет небольшой:

```
.form th, .form td {
  padding: 10px;
  border: 1px solid #cccccc;
}
.form th:first-child, .form td:first-child {
  display: none;
}
```

Здесь мы указываем величину отступов для ячеек таблицы, в которой будет выводиться набор форм, и параметры рамок для них и скрываем первые ячейки каждой строки этой таблицы, где находятся скрытые поля форм.

Остается лишь немного дополнить код стилевого класса form (добавленный код выделен полужирным шрифтом):

```
.form {
  width: 400px;
  margin: 20px 0px 20px 0px;
  border-collapse: collapse;
}
```

Так мы зададим «схлопывание» границ ячеек в таблице — наборе форм. После этого рамки будут проводиться между ячейками, а не по их границам.

Завершающие действия

Пользователь сможет попасть на только что созданную нами страницу списка категорий, либо набрав ее интернет-адрес (что неудобно), либо щелкнув ведущую на нее гиперссылку. Давайте же создадим ее.

Откроем базовый шаблон main.html и исправим код, создающий гиперссылки на страницы административного раздела, чтобы он выглядел так:

```
. . .
{% if user.is_authenticated %}
  {% url "categories_edit" as page_url %}
  <li class="indented"><a href="{{ page_url }}"
  {% if page_url == current_url %} class="current"{% endif %}>
  Категории</a></li>
  <li><a href="/admin/">Админка</a></li>
  <li class="indented"><a href="{% url "logout" %}">Выйти</a></li>
{% endif %}
. . .
```

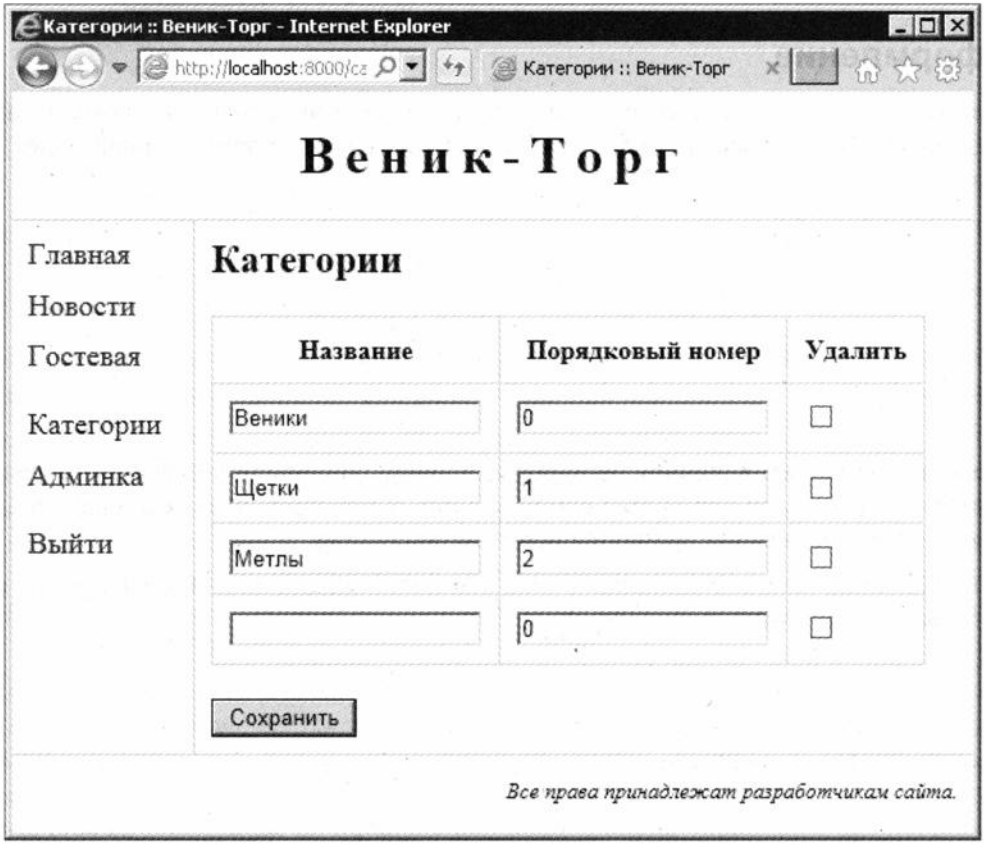


Рис. 24.1. Страница списка категорий

Мы выводим гиперссылку **Категории** в самом начале списка этих страниц и привязываем к ней стилевой класс `indented`, создающий дополнительный отступ сверху. И убираем тот же стилевой класс, привязанный нами ранее к гиперссылке **Админка**.

Посмотрим, что у нас получилось. Войдем на сайт, отправимся на страницу списка категорий, щелкнув на только что созданной гиперссылке, и создадим несколько категорий (рис. 24.1). Если мы не допустили ошибок в наборе кода, у нас все получится.

Что дальше?

В этой главе мы создали список категорий товаров и заодно попрактиковались в работе с наборами форм. Нет, положительно, наборы форм Django — замечательная вещь!

В следующей главе мы закончим создавать перечень товаров. Мы сделаем приложение для работы собственно со списком товаров: для вывода товаров, относящихся к выбранной пользователем категории, для вывода сведений об отдельном товаре, для добавления, правки и удаления товаров. Вот там нам придется повозиться...

ГЛАВА 25

Список товаров

В предыдущей главе мы начали разработку функциональности перечня товаров для нашего сайта и сделали список категорий и страницу для его правки, в которой использовали набор форм.

В этой главе мы тоже применим набор форм, на этот раз связанный. Он пригодится нам, когда мы будем создавать страницы для добавления и правки товара, — для указания набора дополнительных изображений, представляющих тот или иной товар.

Да, сейчас мы займемся приложением списка товаров, которое будет выводить товары, относящиеся к определенной категории, и сведения о выбранном товаре. Разумеется, мы также предусмотрим возможность добавлять, править и удалять товары с помощью особых страниц административного раздела.

В списке товаров мы дадим посетителю возможность сортировать позиции по названию, цене и факту наличия на складе щелчками на заголовках соответствующих столбцов. Такую возможность сейчас предоставляют многие сайты. А полные описания товаров, которые будут выводиться на странице сведений, можно будет форматировать тегами BBCode.

На главной странице сайтов мы выведем список рекомендованных товаров. Также не забудем вывести набор категорий в виде пунктов панели навигации, иначе посетитель не узнает, чем торгует наша гипотетическая фирма.

И, наконец, мы дадим посетителям возможность комментировать товары, для чего используем встроенную в Django подсистему комментирования.

Приложение

Приложение списка товаров получит название `goods`. Создадим его и добавим в список активных приложений:

```
INSTALLED_APPS = (
  . . .
  'goods',
)
```

Модели

Для хранения сведений о товарах нам понадобятся две модели, связанные друг с другом.

Первая модель сохранит собственно сведения о товарах: название, категорию, цену, цену с учетом скидки, факт наличия на складе, признак, является ли товар рекомендуемым, краткое описание и изображение, которые будут выводиться в списке товаров, и полное описание, выводящееся на странице сведений о товаре. Поле названия сделаем уникальным, чтобы исключить многократный ввод одного и того же товара.

В табл. 25.1 приведен список полей новой модели, которой мы дадим имя `Good`, и их параметры.

***Таблица 25.1.** Поля модели `Good`*

Имя	Тип	Параметры
`name`	Текстовый	Максимальная длина — 50 символов. Должно хранить уникальные значения
`category`	Связь	Устанавливает связь с моделью `Category` (см. *главу 24*)
`description`	Memo	
`content`		
`price`	Число с плавающей точкой	
`price_acc`		
`in_stock`	Логический	Значение по умолчанию — `True`
`featured`		Значение по умолчанию — `False`
`image`	Графическое изображение	

Все поля будут обязательными, за исключением `price_acc`. Поля `name`, `price`, `in_stock` и `featured` сделаем индексированными. В качестве папки хранения выгруженных изображений укажем путь goods/list. Поскольку порядок следования товаров в списке будет выбираться самим посетителем, изначальную сортировку записей в модели устанавливать не станем.

Вторая модель будет хранить графические изображения товара, выводимые на странице сведений. Она получит поле для хранения собственно графического изображения и, разумеется, поле связи с первой моделью. Более подробное описание полей второй модели, которую мы назовем `GoodImage`, приведено в табл. 25.2.

И здесь все поля будут обязательными. В качестве папки хранения выгруженных изображений укажем путь goods/detail. Сортировку задавать тоже не станем.

Поскольку мы будем отправлять уведомления о комментариях, нам понадобится указать в их тексте интернет-адрес страницы сведений о товаре, где был оставлен комментарий. Формировать его мы будем в методе `get_absolute_url` модели `Good`.

Таблица 25.2. Поля модели `GoodImage`

Имя	Тип	Параметры
good	Связь	Устанавливает связь с моделью Good
image	Графическое изображение	

Откроем модуль `models` пакета вновь созданного приложения `goods` и введем в него код обеих моделей:

```
from categories.models import Category
from django.core.urlresolvers import reverse

class Good(models.Model):
  name = models.CharField(max_length = 50, unique = True,↳
  db_index = True, verbose_name = "Название")
  category = models.ForeignKey(Category, verbose_name = "Категория")
  description = models.TextField(verbose_name = "Краткое описание")
  content = models.TextField(verbose_name = "Полное описание")
  price = models.FloatField(db_index = True, verbose_name = "Цена, руб.")
  price_acc = models.FloatField(null = True, blank = True,↳
  verbose_name = "Цена с учетом скидки, руб.")
  in_stock = models.BooleanField(default = True, db_index = True,↳
  verbose_name = "Есть в наличии")
  featured = models.BooleanField(default = False, db_index = True,↳
  verbose_name = "Рекомендуемый")
  image = models.ImageField(upload_to = "goods/list",↳
  verbose_name = "Основное изображение")
  def save(self, *args, **kwargs):
    try:
      this_record = Good.objects.get(pk = self.pk)
      if this_record.image != self.image:
        this_record.image.delete(save = False)
    except:
      pass
    super(Good, self).save(*args, **kwargs)
  def delete(self, *args, **kwargs):
    self.image.delete(save = False)
    super(Good, self).delete(*args, **kwargs)
  def get_absolute_url(self):
    return reverse("goods_detail", kwargs = {"pk": self.pk})
  class Meta:
    verbose_name = "товар"
    verbose_name_plural = "товары"

class GoodImage(models.Model):
  good = models.ForeignKey(Good)
```

```
  image = models.ImageField(upload_to = "goods/detail",↵
  verbose_name = "Дополнительное изображение")
  def save(self, *args, **kwargs):
    try:
      this_record = GoodImage.objects.get(pk = self.pk)
      if this_record.image != self.image:
        this_record.image.delete(save = False)
    except:
      pass
    super(GoodImage, self).save(*args, **kwargs)
  def delete(self, *args, **kwargs):
    self.image.delete(save = False)
    super(GoodImage, self).delete(*args, **kwargs)
  class Meta:
    verbose_name = "изображение к товару"
    verbose_name_plural = "изображения к товару"
```

Здесь нам обязательно следует принять меры по недопущению появления «мусорных» файлов изображений. Для этого мы переопределим методы `save` и `delete` каждой модели, в которые поместим код, удаляющий не нужные более файлы. Подробнее о проблеме «мусорных» файлов и ее решении рассказывалось в *главе 13*.

И добавим код, объявляющий автомодератор:

```
from django.contrib.comments.moderation import CommentModerator,↵
moderator
class GoodModerator(CommentModerator):
  email_notification = True
moderator.register(Good, GoodModerator)
```

Единственное, что он сделает, — активизирует отправку по электронной почте уведомлений о добавленных комментариях.

Закончив, выполним синхронизацию только что созданных моделей с базой данных проекта.

Привязки

Добавим в список привязок уровня проекта (модуль `urls` пакета проекта) пункт, соответствующий приложению `goods`. И, поскольку мы собираемся дать посетителю возможность комментировать товары, добавим также привязку к подсистеме комментирования Django:

```
urlpatterns = patterns('',
  . . .
  url(r'^goods/', include('goods.urls')),
  url(r'^comments/', include("django.contrib.comments.urls")),
)
```

Напишем код привязок уровня приложения, который поместим во вновь созданный модуль `urls` теперь уже пакета приложения:

```
from goods.views import GoodsListView, GoodDetailView, GoodCreate,↵
GoodUpdate, GoodDelete
urlpatterns = patterns('',
  url(r'^(?P<pk>\d+)/$', GoodsListView.as_view(), name = "goods_index"),
  url(r'^(?P<pk>\d+)/detail/$', GoodDetailView.as_view(),↵
  name = "goods_detail"),
  url(r'^(?P<pk>\d+)/add/$', permission_required("goods.add_good")↵
  (GoodCreate.as_view()), name = "goods_add"),
  url(r'^(?P<pk>\d+)/edit/$', permission_required("goods.change_good")↵
  (GoodUpdate.as_view()), name = "goods_edit"),
  url(r'^(?P<pk>\d+)/delete/$', permission_required("goods.delete_good")↵
  (GoodDelete.as_view()), name = "goods_delete"),
)
```

Первая привязка относится к контроллеру `GoodsListView`, который будет выводить список товаров, относящихся к какой-либо категории. В ее интернет-адресе мы указываем группу регулярного выражения, которая извлечет идентификатор категории.

Третья привязка относится к контроллеру `GoodCreate`, выполняющему добавление нового товара. В ее интернет-адресе мы создадим группу регулярного выражения, которая также извлечет идентификатор категории. Эта категория будет изначально выбрана в списке на странице добавления товара.

Группы регулярных выражений из интернет-адресов остальных привязок будут извлекать идентификаторы товаров.

Форма

Поскольку для указания дополнительных изображений товара мы собираемся использовать вложенный набор форм, нам придется использовать в качестве родителя для контроллеров добавления и правки товара низкоуровневый класс `TemplateView`. (Как мы уже знаем из *главы 12*, высокоуровневые классы не поддерживают работу с наборами форм.) И нам придется создать форму для ввода товара.

Создадим в пакете приложения модуль `forms` и внесем в него такой код:

```
from goods.models import Good
class GoodForm(forms.ModelForm):
  class Meta:
    model = Good
```

Мы не собираемся добавлять в форму дополнительные поля (в отличие от формы гостевой книги из *главы 22*), поэтому выберем для ее создания простой способ. (О способах создания форм, связанных с моделями, рассказывалось в *главе 11*.)

Контроллеры

Для приложения goods нам потребуется создать пять классов-контроллеров. Помимо того, мы напишем два базовых класса, которые реализуют базовую функциональность, общую для большей части контроллеров. С этих-то классов и начнем.

Базовые классы

Поскольку оба базовых класса будут использоваться лишь в приложении списка товаров, нет нужды помещать объявляющий их код в пакет generic. Напишем его прямо в модуле views пакета приложения goods.

Сначала условимся, что значение "0" GET-параметра sort укажет сортировку по названию товара (это сортировка по умолчанию), значение "1" — по цене и названию, а "2" — по признаку наличия товара и, опять же, по его названию. (Все GET-параметры возвращают строковые значения.) Значение "A" GET-параметра order задаст сортировку по возрастанию значения поля, а значение "D" — по его убыванию.

Класс SortMixin объявит свойства sort и order, в которых будут храниться обозначение поля, по которому в настоящий момент выполняется сортировка, и ее направление — по возрастанию или по убыванию. Также он поместит значения этих свойств в контекст данных шаблона, чтобы мы смогли сформировать корректные интернет-адреса гиперссылок возврата на страницу списка товаров:

```
from django.views.generic.base import ContextMixin
class SortMixin(ContextMixin):
  sort = "0"
  order = "A"
  def get_context_data(self, **kwargs):
    context = super(SortMixin, self).get_context_data(**kwargs)
    context["sort"] = self.sort
    context["order"] = self.order
    return context
```

Мы можем объявить отдельный класс, который будет извлекать из GET-параметров sort и order, соответственно, обозначение поля, по которому выполняется сортировка, и ее направление и заносить эти значения в одноименные поля. Но давайте лучше поместим выполняющий эту задачу код в объявленный в *главе 23* базовый класс PageNumberView, оформив его как метод get. Если уж создавать базовые классы, то делать их максимально универсальными.

Откроем модуль controllers из пакета generic и дополним код класса так, чтобы он выглядел следующим образом (добавленный код выделен полужирным шрифтом):

```
class PageNumberView(View):
  def get(self, request, *args, **kwargs):
    try:
      self.sort = self.request.GET["sort"]
```

```
    except KeyError:
      self.sort = "0"
    try:
      self.order = self.request.GET["order"]
    except KeyError:
      self.order = "A"
    return super(PageNumberView, self).get(request, *args, **kwargs)
  def post(self, request, *args, **kwargs):
    . . .
```

Контроллер списка товаров

Код класса-контроллера `GoodsListView`, выводящего список товаров, довольно сложен. Что и неудивительно: ведь мы собирались дать посетителю возможность сортировать товары в списке по названию, цене и признаку наличия на складе щелчками мышью на заголовках соответствующих им столбцов списка.

Вставим в начало модуля код, импортирующий необходимые классы, как встроенные в Django, так и написанные нами самими:

```
from django.views.generic.list import ListView
from generic.mixins import CategoryListMixin
from goods.models import Good
from categories.models import Category
from generic.controllers import PageNumberView
```

А вот и код, объявляющий сам класс-контроллер:

```
class GoodsListView(PageNumberView, ListView, SortMixin,↳
CategoryListMixin):
  model = Good
  template_name = "goods/index.html"
  paginate_by = 10
  cat = None
  def get(self, request, *args, **kwargs):
    if self.kwargs["pk"] == None:
      self.cat = Category.objects.first()
    else:
      self.cat = Category.objects.get(pk = self.kwargs["pk"])
    return super(GoodsListView, self).get(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(GoodsListView, self).get_context_data(**kwargs)
    context["category"] = self.cat
    return context
  def get_queryset(self):
    goods = Good.objects.filter(category = self.cat)
    if self.sort == "2":
      if self.order == "D":
        goods = goods.order_by("-in_stock", "name")
```

```
      else:
        goods = goods.order_by("in_stock", "name")
    elif self.sort == "1":
      if self.order == "D":
        goods = goods.order_by("-price", "name")
      else:
        goods = goods.order_by("price", "name")
    else:
      if self.order == "D":
        goods = goods.order_by("-name")
      else:
        goods = goods.order_by("name")
```

Прежде всего, мы создали этот класс-контроллер на основе высокоуровневого класса-контроллера `ListView`. Он возьмет на себя всю работу по формированию списка записей и обеспечению пагинации.

Мы также указали в составе родителей нашего класса объявленные ранее классы `PageNumberView` и `SortMixin`. Причем первый класс должен стоять в списке родителей первым, чтобы код, получающий параметры сортировки, был выполнен как можно раньше.

Метод `get` получает идентификатор категории, извлекает из модели `Category` категорию с этим идентификатором и сохраняет ее в специально объявленном свойстве `cat`. Метод `get_context_data` помещает эту категорию в контекст данных, чтобы мы смогли вывести на странице ее название и вставить в интернет-адрес гиперссылки, ведущей на страницу добавления товара, ее идентификатор.

Поскольку нам придется сортировать записи в модели `Good` по-разному, в зависимости от выбора посетителя, набор записей мы станем формировать программно, в методе `get_queryset` (он был описан в *главе 8*). Обратим внимание, по каким полям упорядочиваются записи набора при различных значениях параметров сортировки. А перед сортировкой записей не забудем отфильтровать их по полученному идентификатору категории.

Контроллер сведений о товаре

После монструозного `GoodsListView` класс-контроллер сведений о товаре `GoodDetailView` выглядит карликом:

```
from django.views.generic.detail import DetailView
from generic.mixins import PageNumberMixin

class GoodDetailView(PageNumberView, DetailView, SortMixin,↵
PageNumberMixin):
  model = Good
  template_name = "good.html"
```

А все потому, что мы, во-первых, использовали в качестве родителя высокоуровневый класс-контроллер подробных сведений `DetailView`, а во-вторых, в свое время

вынесли весь общий для всех контроллеров код в базовые классы, которые тоже указали в списке родителей. Так что в новом контроллере нам осталось лишь только задать модель и шаблон.

Контроллер добавления товара

Класс-контроллер добавления нового товара `GoodCreate` также довольно сложен. Рассмотрим его:

```
from django.views.generic.base import TemplateView
from django.forms.models import inlineformset_factory
from goods.models import GoodImage
from goods.forms import GoodForm
from django.contrib import messages
from django.shortcuts import redirect
from django.core.urlresolvers import reverse

GoodImagesFormset = inlineformset_factory(Good, GoodImage,↵
can_order = True)

class GoodCreate(PageNumberView, TemplateView, SortMixin,↵
PageNumberMixin):
  template_name = "good_add.html"
  cat = None
  form = None
  formset = None
  def get(self, request, *args, **kwargs):
    if self.kwargs["pk"] == None:
      self.cat = Category.objects.first()
    else:
      self.cat = Category.objects.get(pk = self.kwargs["pk"])
    self.form = GoodForm(initial = {"category": self.cat})
    self.formset = GoodImagesFormset()
    return super(GoodCreate, self).get(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(GoodCreate, self).get_context_data(**kwargs)
    context["category"] = self.cat
    context["form"] = self.form
    context["formset"] = self.formset
    return context
  def post(self, request, *args, **kwargs):
    self.form = GoodForm(request.POST, request.FILES)
    if self.form.is_valid():
      new_good = self.form.save()
      self.formset = GoodImagesFormset(request.POST, request.FILES,↵
      instance = new_good)
```

```
    if self.formset.is_valid():
      self.formset.save()
      messages.add_message(request, messages.SUCCESS,↵
      "Товар успешно добавлен")
      return redirect(reverse("goods_index",↵
      kwargs = {"pk": new_good.category.pk}) + "?page=" + ↵
      self.request.GET["page"] + "&sort=" + ↵
      self.request.GET["sort"] + "&order=" + self.request.GET["order"])
  if self.kwargs["pk"] == None:
    self.cat = Category.objects.first()
  else:
    self.cat = Category.objects.get(pk = self.kwargs["pk"])
  self.formset = GoodImagesFormset(request.POST, request.FILES)
  return super(GoodCreate, self).get(request, *args, **kwargs)
```

Формируя класс вложенного набора форм, мы указываем для него возможность переупорядочивания записей. Специально указывать для него возможность удаления записей не нужно — она уже установлена по умолчанию.

Мы объявляем свойства `cat`, `form` и `formset`, в которых будут храниться категория товара, форма, в которой вводятся основные сведения о товаре (мы объявили ее ранее, в модуле `forms`), и набор форм, где пользователь укажет дополнительные изображения товара.

В методе `get` мы получаем категорию товара и создаем форму и набор форм. Для формы мы указываем полученную ранее категорию в составе изначальных данных. В методе `get_context_data` мы помещаем все эти значения в контекст данных шаблона.

Код метода `post` нам, в принципе, знаком по примерам из *глав 11* и *12*. Отметим лишь наиболее примечательные моменты:

- метод `save` модели возвращает в качестве результата сохраненную запись. Эту запись мы укажем при повторном создании набора форм, чтобы последний смог привязать к этой записи все вновь созданные записи связанной модели `GoodImage` — списка дополнительных изображений;
- при формировании интернет-адреса переадресации не забываем указать в нем GET-параметры, задающие номер страницы, и параметры сортировки — это позволит пользователю вернуться на ту же страницу списка, отсортированную по тому же полю и в том же порядке. Все нужные значения мы можем извлечь непосредственно из GET-параметров, поскольку сохранение данных выполняет тот же класс-контроллер, что и вывод формы, и его интернет-адрес не изменяется. А идентификатор категории мы можем получить из только что созданного товара;
- если в форму были введены некорректные данные, мы повторно получаем категорию, создаем набор форм (форма у нас уже есть) и выводим страницу добавления товара, чтобы пользователь смог внести нужные исправления.

Контроллер правки товара

Класс-контроллер правки товара `GoodUpdate` во многом похож на только что рассмотренный нами контроллер `GoodCreate` и использует приемы, также знакомые нам по *главам 11* и *12*:

```
class GoodUpdate(PageNumberView, TemplateView, SortMixin,↵
PageNumberMixin):
  good = None
  template_name = "good_edit.html"
  form = None
  formset = None
  def get(self, request, *args, **kwargs):
    self.good = Good.objects.get(pk = self.kwargs["pk"])
    self.form = GoodForm(instance = self.good)
    self.formset = GoodImagesFormset(instance = self.good)
    return super(GoodUpdate, self).get(request, *args, **kwargs)
  def get_context_data(self, **kwargs):
    context = super(GoodUpdate, self).get_context_data(**kwargs)
    context["good"] = self.good
    context["form"] = self.form
    context["formset"] = self.formset
    return context
  def post(self, request, *args, **kwargs):
    self.good = Good.objects.get(pk = self.kwargs["pk"])
    self.form = GoodForm(request.POST, request.FILES,↵
    instance = self.good)
    self.formset = GoodImagesFormset(request.POST, request.FILES,↵
    instance = self.good)
    if self.form.is_valid():
      self.form.save()
      if self.formset.is_valid():
        self.formset.save()
        messages.add_message(request, messages.SUCCESS,↵
        "Товар успешно изменен")
        return redirect(reverse("goods_index",↵
        kwargs = {"pk": self.good.category.pk}) + "?page=" + ↵
        self.request.GET["page"] + "&sort=" + ↵
        self.request.GET["sort"] + "&order=" + self.request.GET["order"])
    return super(GoodUpdate, self).get(request, *args, **kwargs)
```

Контроллер удаления товара

Класс-контроллер удаления товара `GoodDelete` мы создадим на основе высокоуровневого класса `DeleteView`, поскольку здесь нам не нужно выводить на экран набора форм. В таком случае код класса станет очень компактным:

```
from django.views.generic.edit import DeleteView
class GoodDelete(PageNumberView, DeleteView, SortMixin, PageNumberMixin):
  model = Good
  template_name = "good_delete.html"
  def post(self, request, *args, **kwargs):
    self.success_url = reverse("goods_index",↳
    kwargs = {"pk": Good.objects.get(pk = kwargs["pk"]).category.pk}) + ↳
    "?page=" + self.request.GET["page"] + "&sort=" + ↳
    self.request.GET["sort"] + "&order=" + self.request.GET["order"]
    messages.add_message(request, messages.SUCCESS,↳
    "Товар успешно удален")
    return super(GoodDelete, self).post(request, *args, **kwargs)
```

Шаблоны

Помимо пяти шаблонов для страниц приложения списка товаров и шаблона почтового уведомления о добавлении комментария, мы напишем еще один универсальный шаблон и исправим написанный ранее.

Универсальный шаблон списка комментариев

Возможность комментирования мы хотим реализовать не только для товаров, но и для статей блога. Поэтому имеет смысл вынести код, выводящий список комментариев, в универсальный подгружаемый шаблон:

```
{% if comment_list|length > 0 %}
  <p> </p>
  {% for comment in comment_list %}
    {% if comment.is_removed %}
      <p class="comment-removed">Комментарий удален.</p>
    {% else %}
      <div class="comment">
        <div class="user">{{ comment.user_name }}</div>
        <div class="content">{{ comment.comment }}</div>
        <div class="date">{{ comment.submit_date|date:"j.m.Y H:i:s"↳
        }}</div>
      </div>
    {% endif %}
  {% endfor %}
{% endif %}
```

Создадим в папке templates шаблонов уровня проекта папку comments. И сохраним код нового универсального шаблона в этой папке, дав файлу имя list.html. (Как мы помним из *главы 15*, шаблон списка комментариев Django будет искать в файле comments\list.html.)

Исправленный универсальный шаблон пагинации

Гиперссылки, ведущие на другие страницы списка, содержат GET-параметр `page` с номером страницы. Но они не несут в себе ни указателя на поле, по которому ведется сортировка, ни обозначения направления сортировки. Поэтому после перехода на другую страницу списка товаров мы обнаружим, что список вновь отсортирован по умолчанию, т. е. по названию товара.

Следовательно, мы должны внести изменения в код универсального шаблона пагинации pagination.html, что хранится в подпапке generic папки templates уровня проекта. Изменения эти следующие (исправленный код выделен полужирным шрифтом):

```
{% if paginator.num_pages > 1 %}
  <div id="pagination">
    {% if page_obj.has_previous %}
      <div id="previous-page">↵
      <a href="?page={{ page_obj.previous_page_number }}↵
      {% if sort and order %}&sort={{ sort }}&order={{ order }}↵
      {% endif %}">&lt;</a></div>
    {% endif %}
    {% if page_obj.has_next %}
      <div id="next-page">↵
      <a href="?page={{ page_obj.next_page_number }}↵
      {% if sort and order %}&sort={{ sort }}&order={{ order }}↵
      {% endif %}">&gt;</a></div>
    {% endif %}
    <div id="num-pages">
      . . .
        {% if page_obj.number != pn %}
          <a href="?page={{ pn }}{% if sort and order %}↵
          &sort={{ sort }}&order={{ order }}{% endif %}">
        {% endif %}
      . . .
    </div>
  </div>
{% endif %}
```

Здесь мы добавляем в интернет-адреса гиперссылок, помимо GET-параметра `page`, еще и параметры `sort` и `order`, разумеется, если в контексте данных присутствуют одноименные переменные, хранящие их значения.

Шаблон списка товаров

Создадим в папке пакета приложения `goods` папку templates, где будут храниться шаблоны уровня этого приложения. Первым из них станет шаблон, выводящий список товаров, который мы сохраним в файле goods_index.html:

```
{% extends "main.html" %}
{% load thumbnail %}
```

```
{% block title %}{{ category.name }}{% endblock %}
{% block main %}
  {% include "generic/messages.html" %}
  <h2>{{ category.name }}</h2>
  {% if perms.goods.add_good %}
    <p><a href="{% url "goods_add" pk=category.pk %}
    ?page={{ page_obj.number }}&sort={{ sort }}
    &order={{ order }}">Добавить товар</a></p>
  {% endif %}
  <table class="list-table">
    <tr>
      <th>Миниатюра</th>
      <th><a href="?sort=0&order={% if sort == "0" and order == "A" %}
      D{% else %}A{% endif %}">{% if sort == "0" %}{% if order == "D" %}
      &#9660;{% else %}&#9650;{% endif %}{% endif %} Название
      </a></th>
      <th>Описание</th>
      <th><a href="?sort=1&order={% if sort == "1" and order == "A" %}
      D{% else %}A{% endif %}">{% if sort == "1" %}{% if order == "D" %}
      &#9660;{% else %}&#9650;{% endif %}{% endif %} Цена, руб.
      </a></th>
      <th><a href="?sort=2&order={% if sort == "2" and order == "A" %}
      D{% else %}A{% endif %}">{% if sort == "2" %}{% if order == "D" %}
      &#9660;{% else %}&#9650;{% endif %}{% endif %} Есть в наличии
      </a></th>
      {% if perms.goods.change_good %}
        <th> </th>
      {% endif %}
      {% if perms.goods.delete_good %}
        <th> </th>
      {% endif %}
    </tr>
    {% for object in object_list %}
      <tr>
        <td class="centered"><a href="{% url "goods_detail"
        pk=object.pk %}?page={{ page_obj.number }}
        &sort={{ sort }}&order={{ order }}">
        <img src="{% thumbnail object.image "base" %}"><a></td>
        <td><a href="{% url "goods_detail" pk=object.pk %}
        ?page={{ page_obj.number }}&sort={{ sort }}&order={{ order }}">
        {{ object.name }}</a></td>
        <td>{{ object.description }}</td>
        <td class="centered">{% if object.price_acc > 0 %}
        <span class="old-price">{{ object.price }}</span><br>
        {{ object.price_acc }}{% else %}{{ object.price }}
        {% endif %}</td>
```

```
        <td class="centered">{% if object.in_stock %}+{% endif %}</td>
        {% if perms.page.change_good %}
          <td class="centered"><a href="{% url "goods_edit"↵
          pk=object.pk %}?page={{ page_obj.number }}↵
          &sort={{ sort }}&order={{ order }}">Изменить</a></td>↵
        {% endif %}
        {% if perms.page.delete_good %}
          <td class="centered"><a href="{% url "goods_delete"↵
          pk=object.pk %}?page={{ page_obj.number }}↵
          &sort={{ sort }}&order={{ order }}">Удалить</a></td>
        {% endif %}
      </tr>
    {% endfor %}
  </table>
  {% include "generic/pagination.html" %}
{% endblock %}
```

Этот код объемен и сложен, но, по большей части, знаком нам по примерам из *главы 7*. Нужно прояснить лишь пару моментов:

- в интернет-адресах гиперссылок, ведущих на страницы сведений о товаре, добавления, правки и удаления товара, мы указываем, помимо номера страницы, еще и параметры сортировки. Таким образом, мы сможем реализовать возврат на ту же страницу списка товаров и с теми же параметрами сортировки;
- для формирования заголовков столбцов списка, по которым можно выполнять сортировку, мы используем следующий код (приведен фрагмент для столбца названий товаров):

  ```
  <th><a href="?sort=0&order={% if sort == "0" and order == "A" %}D↵
  {% else %}A{% endif %}">{% if sort == "0" %}{% if order == "D"↵
  %}&#9660;{% else %}&#9650;{% endif %}{% endif %} Название</a></th>
  ```

  Мы видим, что GET-параметр `sort`, задающий обозначение поля для сортировки, присутствует в интернет-адресе в любом случае;
- теперь что касается GET-параметра `order`, указывающего направление сортировки. Если в данный момент выполняется сортировка по тому или иному полю и по возрастанию, мы указываем для этого параметра значение "D" (сортировка по убыванию), в противном случае — значение "A" (сортировка по возрастанию). Тогда пользователь, щелкая на заголовке, будет тем самым менять направление сортировки на противоположное;
- если по какому-либо полю выполнена сортировка по возрастанию, мы выводим левее текста заголовка символ ▲, обозначаемый литералом `&#9650`. Если же была выполнена сортировка по убыванию, мы выводим там же символ ▼, который в HTML-коде обозначается литералом `&#9660`. Если же сортировка по полю вообще не выполнялась, мы не выводим в заголовке никакого символа-стрелки. Так мы дадим знать пользователю, что по этому полю выполняется сортировка, и обозначим ее направление.

Напоследок рассмотрим код, выводящий цену:

```
<td class="centered">{% if object.price_acc > 0 %}↵
<span class="old-price">{{ object.price }}</span><br>↵
{{ object.price_acc }}{% else %}{{ object.price }}{% endif %}</td>
```

Если для товара не указана цена с учетом скидки (поле `price_acc` пусто), мы выводим там обычную цену (содержимое поля `price`). В противном случае мы выводим обычную цену, привязываем к ней стилевой класс `old-price`, который задаст для нее серый цвет и зачеркивание, ставим разрыв строки (тег `<br>`) и выводим цену с учетом скидки. Это обычная практика перечней товаров подобного рода.

Шаблон сведений о товаре

Шаблон, выводящий сведения о выбранном товаре, будет сохранен в файле good.html в той же папке, что и предыдущий:

```
{% extends "main.html" %}
{% load bbcode_tags %}
{% load comments %}
{% block title %}{{ object.name }} :: {{ object.category.name }}↵
{% endblock %}
{% block main %}
  <h2>{{ object.name }}</h2>
  <p>Категория: {{object.category.name }}</p>
  <div class="good-images">
    <div><img src="{{ object.image.url }}"></div>
    {% for goodimage in object.goodimage_set.all %}
      <div><img src="{{ goodimage.image.url }}"></div>
    {% endfor %}
  </div>
  <div>{{ object.content|bbcode|safe }}</div>
  <p>Цена, руб.: {% if object.price_acc > 0 %}<span class="old-price">↵
  {{ object.price }}</span>  {{ object.price_acc }}{% else %}↵
  {{ object.price }}{% endif %}</p>
  {% if object.in_stock %}
    <p>Нет в наличии.</p>
  {% endif %}
  <p><a href="{% url "goods_index" pk=object.category.pk %}↵
  ?page={{ pn }}&sort={{ sort }}&order={{ order }}">Назад</a></p>
  <p> </p>
  {% render_comment_list for object %}
  <p> </p>
  {% get_comment_form for object as form %}
  <div class="form">
    <form action="{% comment_form_target %}" method="post">
      {% include "generic/form.html" %}
```

```
      <input type="hidden" name="next" value="{% url "goods_detail"%
      pk=object.pk %}?page={{ pn }}&sort={{ sort }}&order={{ order }}">
      <div class="submit-button"><input type="submit" name="submit"
      value="Отправить"></div>
    </form>
  </div>
{% endblock %}
```

Основное изображение товара, демонстрирующее его на странице списка, и дополнительные изображения, если таковые указаны, мы выводим в специальном блоке (теге `<div>`) с привязанным стилевым классом `good-images`. С помощью особых стилей мы указываем, чтобы эти изображения выводились в строку. (В более сложных реализациях подобных сайтов для этого можно использовать слайдер или фотогалерею, готовые решения для реализации которых можно без труда найти в Интернете.)

Цену товара мы выводим так же, как и в созданном чуть раньше шаблоне списка товаров. Единственное исключение — мы разделяем обычную цену и цену с учетом скидки не разрывом строк, а пробелом.

Признак того, является ли товар рекомендуемым, мы здесь не выводим. Посетителям сайта он ни к чему.

Интернет-адрес гиперссылки возврата включит в свой состав, помимо номера страницы списка товаров, также обозначение поля, по которому ведется сортировка, и ее направления.

Под гиперссылкой возврата мы выводим список комментариев к товару. Здесь будет задействован универсальный шаблон списка комментариев, написанный нами ранее.

Код формы для ввода комментария мы помещаем непосредственно в шаблон, а для вывода входящих в ее состав элементов управления применим написанный еще в *главе 20* универсальный шаблон. В саму эту форму не забудем поместить скрытое поле `next`, задав для него в качестве значения интернет-адрес комментируемой страницы, чтобы после добавления комментария посетитель вернулся на нее и сразу увидел отправленный им комментарий.

Шаблоны добавления, правки и удаления товара

Шаблоны страниц для добавления, правки и удаления товара достаточно просты и напоминают аналогичные шаблоны из приложения списка новостей `news`.

Код шаблона добавления товара good_add.html:

```
{% extends "main.html" %}
{% block title %}Добавление товара :: {{ category.name }}{% endblock %}
{% block main %}
  <h2>Добавление товара</h2>
  <div class="form">
    <form action="" method="post" enctype="multipart/form-data">
```

```
      {% include "generic/form.html" %}
      {% include "generic/formset.html" %}
      <div class="submit-button"><input type="submit"
      value="Добавить"></div>
    </form>
  </div>
  <p><a href="{% url "goods_index" pk=category.pk %}↵
  ?page={{ pn }}&sort={{ sort }}&order={{ order }}"">Назад</a></p>
{% endblock %}
```

Код шаблона правки товара good_edit.html:

```
{% extends "main.html" %}
{% block title %}Правка товара :: {{ good.name }}{% endblock %}
{% block main %}
  <h2>Правка товара</h2>
  <div class="form">
    <form action="" method="post" enctype="multipart/form-data">
      {% include "generic/form.html" %}
      {% include "generic/formset.html" %}
      <div class="submit-button"><input type="submit"
      value="Изменить"></div>
    </form>
  </div>
  <p><a href="{% url "goods_index" pk=good.category.pk %}↵
  ?page={{ pn }}&sort={{ sort }}&order={{ order }}"">Назад</a></p>
{% endblock %}
```

Код шаблона удаления товара good_delete.html:

```
{% extends "main.html" %}
{% load bbcode_tags %}
{% block title %}Удаление товара :: {{ object.name }}{% endblock %}
{% block main %}
  <h2>Удаление товара</h2>
  <h4>{{ object.name }}</h4>
  <p>Категория: {{object.category.name }}</p>
  <div>{{ object.content|bbcode|safe }}</div>
  <p>Цена, руб.: {% if object.price_acc > 0 %}<span class="old-price">↵
  {{ object.price }}</span>  {{ object.price_acc }}{% else %}↵
  {{ object.price }}{% endif %}</p>
  {% if object.in_stock %}
    <p>Нет в наличии.</p>
  {% endif %}
  {% if object.featured %}
    <p>Рекомендуемый.</p>
  {% endif %}
  <form action="" method="post">
    {% csrf_token %}
```

```
    <input type="submit" value="Удалить">
  </form>
  <p><a href="{% url "goods_index" pk=object.category.pk%}↵
  ?page={{ pn }}&sort={{ sort }}&order={{ order }}"">Назад</a></p>
{% endblock %}
```

Ради упрощения этого шаблона автор не стал реализовывать в нем вывод относящихся к товару изображений — на странице удаления товара они не очень-то и нужны. Если же кому-то из читателей понадобится вывести эти изображения на странице удаления, он может позаимствовать соответствующий фрагмент кода из шаблона good.html.

Шаблон почтового уведомления

Шаблон почтового уведомления об отправке комментария мы сохраним в файле comment_notification_email.txt в подпапке comments папки templates уровня проекта. Его код схож с кодом аналогичного шаблона, что мы создали в *главе 15*:

```
На сайте был оставлен новый комментарий:

  - товар: {{ content_object.name }} (http://localhost:8000↵
{{ content_object.get_absolute_url }});
  - пользователь: {{ comment.user_name }};
  - e-mail: {{ comment.user_email }};
  - содержимое: {{ comment.comment }};
  - дата: {{ comment.submit_date }};
  - IP-адрес: {{ comment.ip_address }}.

Гиперссылка на страницу комментария: ↵
http://localhost:8000/admin/comments/comment/{{ comment.pk }}/.
```

Отметим, что в тексте уведомления, помимо сведений о товаре и комментарии, мы выводим еще и интернет-адрес для перехода на страницу встроенного административного сайта, позволяющую исправить комментарий, в частности, при необходимости, пометить его как удаленный. Так мы заметно упростим жизнь модераторам комментариев.

> ***Интернет-адреса гиперссылок***
>
> В интернет-адресах обеих гиперссылок мы указали в качестве имени хоста **http://localhost:8000** — хост отладочного Web-сервера Django. Перед публикацией сайта мы изменим его на имя хоста, на котором будет опубликован наш сайт.

Оформление

Дополним таблицу стилей main.css следующим кодом:

```
.list-table {
  width: 100%;
  border-collapse: collapse;
}
```

```
.list-table td, .list-table th {
  border: 1px #666666 solid;
  padding: 5px;
}
.list-table td.centered {
  text-align: center;
}
.old-price {
  color: #999999;
  text-decoration: line-through;
}
.good-images {
  width: 100%;
  height: 200px;
  margin-bottom: 50px;
}
.good-images div {
  max-width: 300px;
  height: 200px;
  float: left;
}
.good-images div img {
  max-width: 300px;
  height: 200px;
}
.comment {
  border: 1px solid #cccccc;
  margin: 5px 0px 5px 0px;
}
.comment div {
  padding: 4px;
}
.comment .user {
  background-color: #eeeeee;
  font-weight: bold;
}
.comment .date {
  background-color: #eeeeee;
  font-style: italic;
  text-align: right;
}
.comment-removed {
  font-weight: bold;
  font-style: italic;
}
```

Добавленный код определяет стили, задающие оформление списка товаров, набора изображений, относящихся к товару и выводимых на странице сведений о товаре, и списка комментариев.

Вывод списка рекомендуемых товаров на главной странице

Наряду с последними пятью новостями, мы решили вывести на главной странице список рекомендуемых товаров. Сейчас самое время этим заняться.

Откроем модуль `views` пакета приложения `main` и отыщем в нем объявление класса-контроллера `MainPageView`. Исправим его, чтобы оно выглядело следующим образом (добавленный код выделен полужирным шрифтом):

```
from goods.models import Good
class MainPageView(TemplateView, CategoryListMixin):
  template_name = "mainpage.html"
  news = New.objects.all()[0:5]
  goods = Good.objects.filter(featured = True)
  def get_context_data(self, **kwargs):
    context = super(MainPageView, self).get_context_data(**kwargs)
    context["news"] = self.news
    context["goods"] = self.goods
    return context
```

Мы создали здесь свойство `goods`, в котором будет храниться список рекомендуемых товаров, и присвоили ему список товаров, у которых поле `featured` хранит значение `True` (т. е. товары, помеченные как рекомендуемые). После чего поместили этот список в контекст данных, создав для этого переменную `goods`.

Далее откроем шаблон mainpage.html, что хранится в папке templates уровня приложения `main`, и добавим в его код фрагменты, которые, собственно, сформируют список товаров. Добавленный код выделен полужирным шрифтом:

```
{% extends "main.html" %}
{% load thumbnail %}
{% block title %}Главная страница{% endblock %}
{% block main %}
  . . .
  <h3>Рекомендуемые товары</h3>
  <table class="list-table">
    {% for object in goods %}
      <tr>
        <td class="centered"><a href="{% url "goods_detail" ↵
        pk=object.pk %}"><img src="{% thumbnail object.image "base" ↵
        %}"><a></td>
        <td><a href="{% url "goods_detail" pk=object.pk %}">↵
        {{ object.name }}</a></td>
        <td>{{ object.description }}</td>
        <td class="centered">{% if object.price_acc > 0 %}<span
        class="old-price">{{ object.price }}</span><br>↵
        {{ object.price_acc }}{% else %}{{ object.price }}{% endif %}
        руб.</td>
```

```
        <td class="centered">{% if object.in_stock %}В наличии↵
        {% endif %}</td>
      </tr>
    {% endfor %}
  </table>
{% endblock %}
```

Список рекомендованных товаров также выводится в виде таблицы. Но здесь мы не создаем гиперссылки на страницы правки и удаления товара — они тут совершенно не нужны. Ради компактности мы также не выводим «шапку» таблицы.

Вывод списка категорий в составе панели навигации

Нам осталось (помимо обязательного этапа тестирования) вывести список категорий товаров в составе панели навигации. Тогда посетитель сможет просмотреть список товаров, относящихся к выбранной категории, щелкнув на ней мышью.

Сначала внесем изменения в код базового класса `CategoryListMixin`, объявленного в созданном нами самими модуле `generic` (добавленный код выделен полужирным шрифтом):

```
from categories.models import Category
class CategoryListMixin(ContextMixin):
  def get_context_data(self, **kwargs):
    context = super(CategoryListMixin, self).get_context_data(**kwargs)
    context["current_url"] = self.request.path
    context["categories"] = Category.objects.all()
    return context
```

Здесь мы создаем в контексте данных шаблона переменную `categories` и помещаем в нее список категорий.

Теперь дополним код базового шаблона main.html фрагментом, который выведет на экран этот список (добавленный код выделен полужирным шрифтом):

```
. . .
{% url "main" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}
class="current"{% endif %}>Главная</a></li>
{% for object in categories %}
  {% url "goods_index" pk=object.pk as page_url %}
  <li{% if forloop.first %} class="indented"{% endif %}>↵
  <a href="{{ page_url }}"{% if page_url == current_url %}
  class="current"{% endif %}>{{ object.name }}</a></li>
{% endfor %}
{% url "news_index" as page_url %}
<li class="indented"><a href="{{ page_url }}"
{% if page_url == current_url %} class="current"{% endif %}>↵
Новости</a></li>
{% url "guestbook" as page_url %}
. . .
```

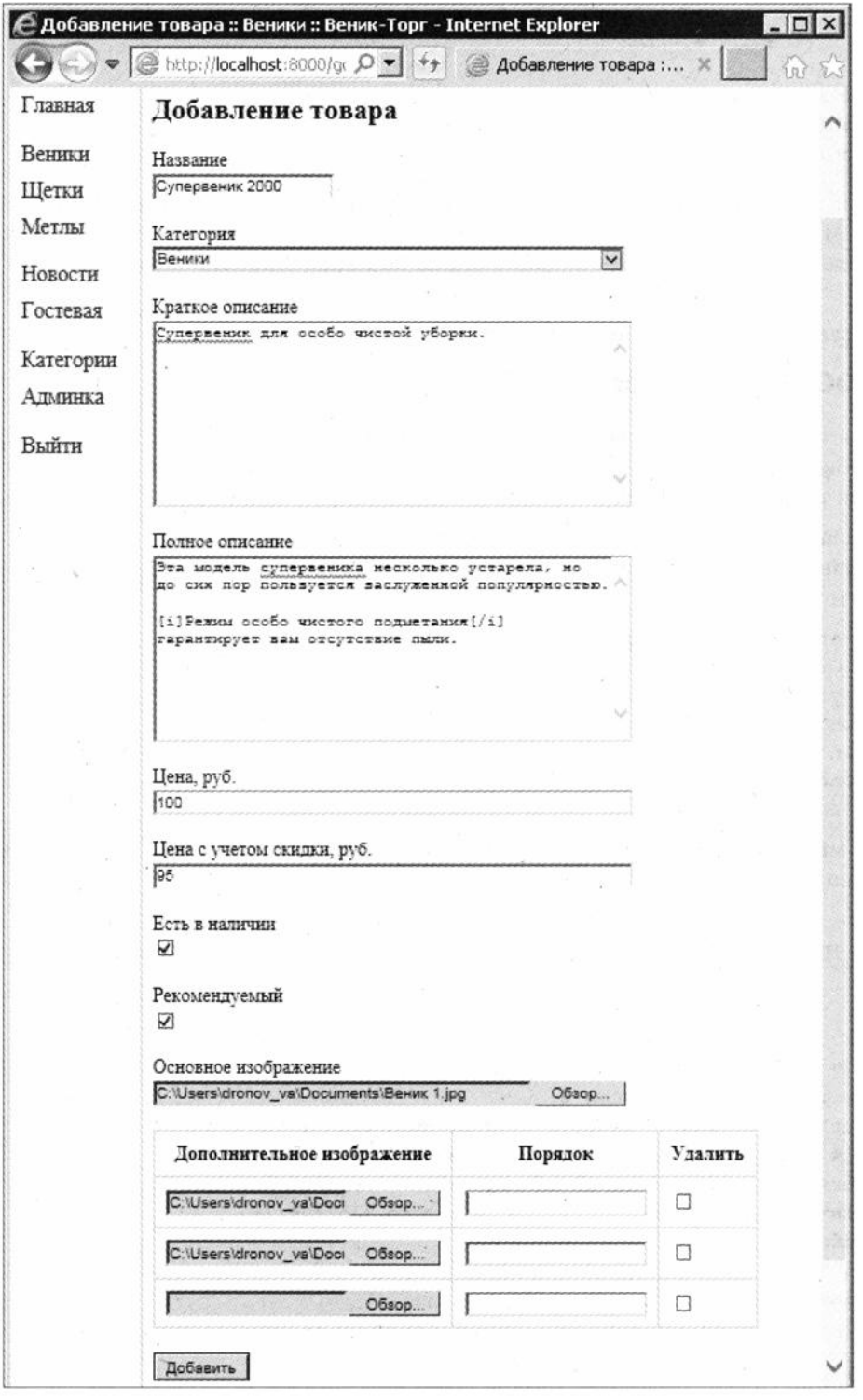


Рис. 25.1. Страница ввода товара

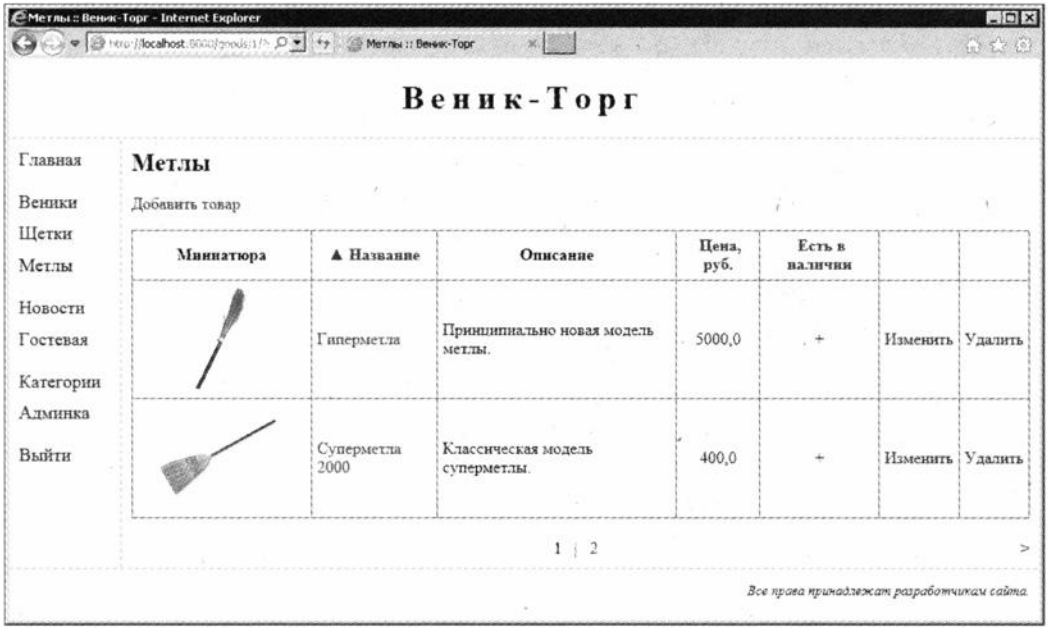


Рис. 25.2. Страница списка товаров

К первому пункту списка категорий мы привязываем стилевой класс `indented`, чтобы увеличить просвет между этим и предыдущим пунктами. Тот же стилевой класс мы привязываем к следующему за списком категорий пункту. Так список категорий будет визуально отделен от остальных пунктов панели навигации.

Запустим встроенный Web-сервер Django, выполним вход на сайт, перейдем на страницу списка товаров, относящихся к какой-либо категории, и откроем страницу ввода товара (рис. 25.1).

Введем несколько товаров, относящихся к разным категориям. Уменьшим количество позиций на странице списка до двух, чтобы проверить, как работает пагинация. И посмотрим, правильно ли отображается список товаров (рис. 25.2).

Откроем страницу сведений о каком-либо товаре, желательно того, для которого мы при добавлении указали дополнительные изображения (рис. 25.3).

Добавим к товару пару комментариев. Проверим электронную почту — должны прийти уведомления о поступлении новых комментариев (если, конечно, мы правильно настроили параметры отправки почты).

Напоследок откроем главную страницу и посмотрим на список рекомендуемых товаров (рис. 25.4).

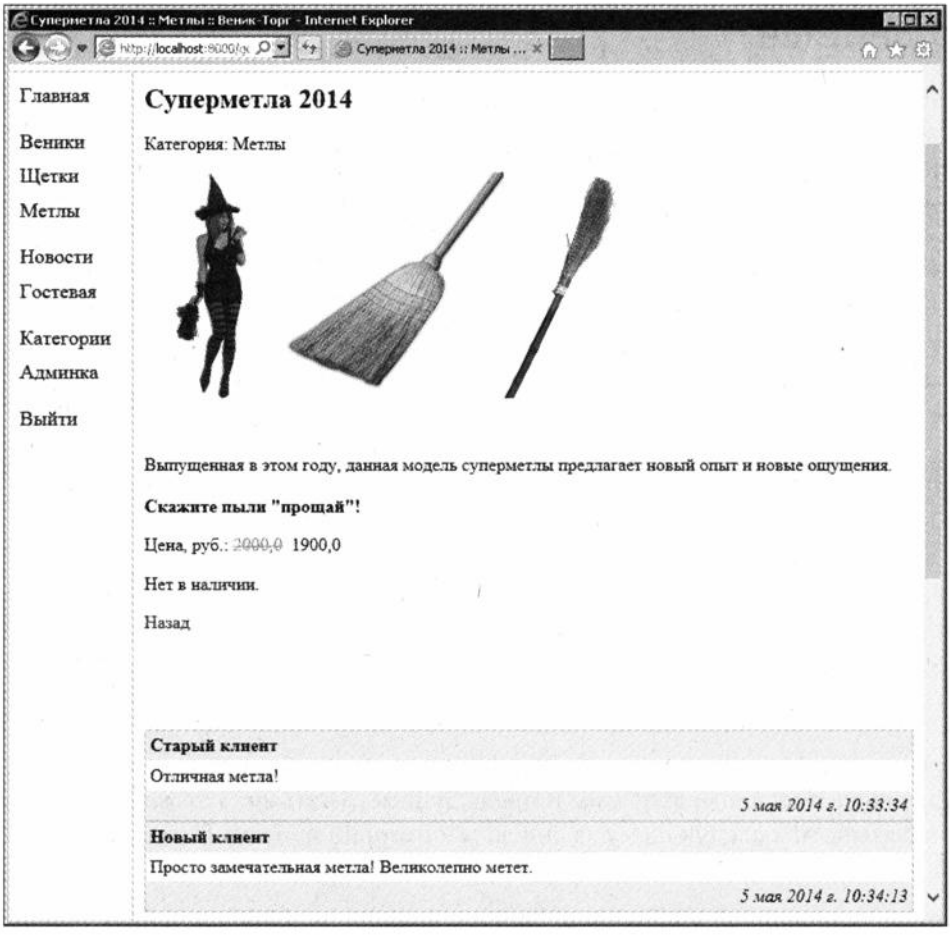


Рис. 25.3. Страница сведений о выбранном товаре

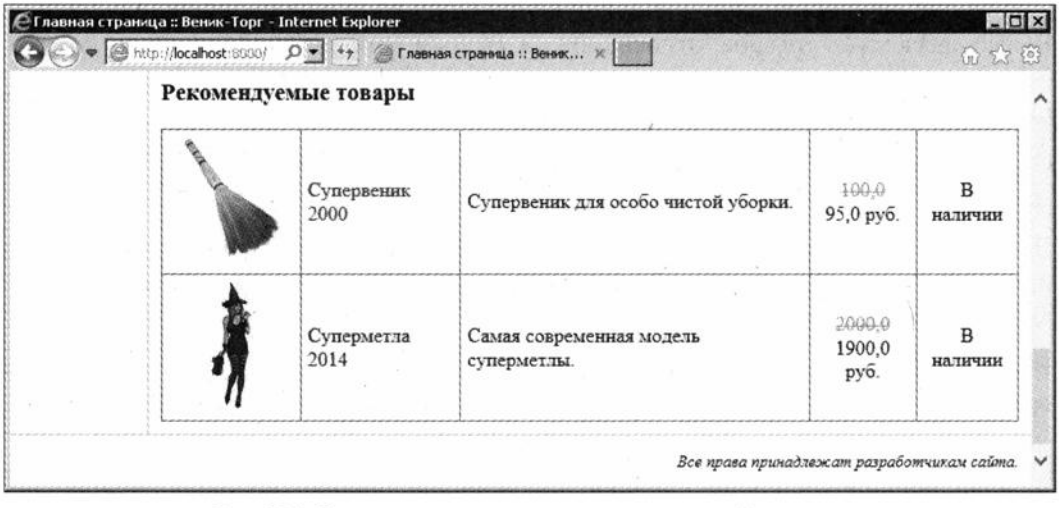


Рис. 25.4. Список рекомендуемых товаров на главной странице

Что дальше?

В этой главе мы сделали перечень товаров — основную часть нашего сайта. И приобрели неоценимый опыт в разработке сложных моделей, контроллеров и шаблонов...

...Опыт, который пригодится нам в следующей главе, в которой мы приступим к работе над блогом.

ГЛАВА 26

Блог

В предыдущей главе мы закончили работу над перечнем товаров, создав приложение, которое реализует функциональность собственно списка товаров. Над этим приложением нам пришлось повозиться, сделав вывод только тех товаров, что относятся к указанной посетителем категории, обеспечив посетителю возможность произвольно сортировать товары по названию, цене и признаку наличия на складе и оставлять комментарии к товарам.

Эта глава посвящена блогу. Здесь мы дадим возможность оставлять комментарии к отдельным статьям блога, искать их по введенному ключевому слову и тегам, а также сделаем так, чтобы править и удалять статьи мог лишь тот пользователь, что их написал, или суперпользователь, имеющие максимальные права (см. *главу 14*). Так что работы и здесь предстоит немало.

Приложение

Приложение блога получит «говорящее» имя `blog`. Создадим его и добавим в список активных приложений:

```
INSTALLED_APPS = (
  . . .
  'blog',
)
```

Модель

Данные блога будут храниться в модели `Blog`. Она получит следующие поля:

- заголовок статьи;
- краткое описание статьи, которое будет выводиться на странице списка статей;
- содержимое статьи;
- дата и время публикации статьи;

- признак, разрешены ли для статьи комментарии;
- набор привязанных к статье тегов;
- пользователь, написавший статью.

Поле заголовка сделаем уникальным в пределах текущей даты, чтобы какой-нибудь забывчивый пользователь по ошибке не опубликовал одну и ту же статью дважды. Для поля даты и времени публикации в качестве изначального значения зададим текущие дату и время, а для поля признака, указывающего, можно ли оставлять комментарии к данной статье, — `True`. Поле, хранящее написавшего статью пользователя, объявим нередактируемым, чтобы соответствующий ему элемент управления не выводился на страницы добавления и правки записей, — так мы исключим возможность указания в качестве автора другого пользователя.

Распишем все поля новой модели и их параметры более подробно, в результате чего получим табл. 26.1.

***Таблица 26.1.** Поля модели `Blog`*

Поле	Тип	Параметры
`title`	Текстовый	Максимальная длина — 100 символов Значение должно быть уникальным в пределах даты, хранящейся в поле `posted`
`description`	Memo	
`content`		
`posted`	Дата и время	Значение по умолчанию — текущие дата и время
`is_commentable`	Логический	Значение по умолчанию — `True`
`tags`	Теги	
`user`	Связь	Устанавливает связь с моделью `User`. Нередактируемое поле

Все поля будут обязательными, кроме поля `tags`. Для поля `posted` создадим индекс, чтобы сортировка по нему выполнялась быстрее. И укажем в модели изначальную сортировку по тому же полю по убыванию значений.

Для реализации привязки к статьям блога тегов мы используем описанную в *главе 18* библиотеку django-taggit.

Откроем модуль `models` пакета только что созданного приложения и введем в него код собственно модели `Blog`:

```
from datetime import datetime
from taggit.managers import TaggableManager
from django.contrib.auth.models import User
from django.core.urlresolvers import reverse
```

```
class Blog(models.Model):
  title = models.CharField(max_length = 100, unique_for_date = "posted",↵
  verbose_name = "Заголовок")
  description = models.TextField(verbose_name = "Краткое содержание")
  content = models.TextField(verbose_name = "Полное содержание")
  posted = models.DateTimeField(default = datetime.now(),↵
  db_index = True, verbose_name = "Опубликована")
  is_commentable = models.BooleanField(default = True,↵
  verbose_name = "Разрешены комментарии")
  tags = TaggableManager(blank = True, verbose_name = "Теги")
  user = models.ForeignKey(User, editable = False)
  def get_absolute_url(self):
    return reverse("blog_detail", kwargs = {"pk": self.pk})
  class Meta:
    ordering = ["-posted"]
    verbose_name = "статья блога"
    verbose_name_plural = "статьи блога"
```

Не забываем создать в модели метод `get_absolute_url` — он понадобится нам потом, когда мы займемся шаблоном почтового уведомления.

И напишем код автомодератора:

```
from django.contrib.comments.moderation import CommentModerator,↵
moderator
class BlogModerator(CommentModerator):
  email_notification = True
  enable_field = "is_commentable"
moderator.register(Blog, BlogModerator)
```

Здесь мы указываем в качестве поля, хранящего признак возможности комментирования статьи, `is_commentable`, и активизируем функцию отправки почтовых уведомлений о новых комментариях.

Привязки

Выполним привязку на уровне проекта, вставив в список привязок в модуле `urls` пакета проекта следующий элемент:

```
urlpatterns = patterns('',
  . . .
  url(r'^blog/', include('blog.urls')),
)
```

Создадим в папке пакета приложения модуль `urls` и напишем в нем код, задающий привязки уровня приложения:

```
from django.contrib.auth.decorators import permission_required
from blog.views import BlogListView, BlogDetailView, BlogCreate,↵
BlogUpdate, BlogDelete
```

```
urlpatterns = patterns('',
  url(r'^$', BlogListView.as_view(), name = "blog_index"),
  url(r'^(?P<pk>\d+)/detail/$', BlogDetailView.as_view(),
  name = "blog_detail"),
  url(r'^add/$', permission_required("blog.add_blog")
  (BlogCreate.as_view()), name = "blog_add"),
  url(r'^(?P<pk>\d+)/edit/$', permission_required("blog.change_blog")
  (BlogUpdate.as_view()), name = "blog_edit"),
  url(r'^(?P<pk>\d+)/delete/$', permission_required("blog.delete_blog")
  (BlogDelete.as_view()), name = "blog_delete"),
)
```

Здесь нам все знакомо по приложению списка новостей.

Форма

Ранее мы условились, что править и удалять статьи блога сможет лишь тот пользователь, что их написал, и суперпользователь. Следовательно, нам придется перед выводом страниц правки и удаления статьи и перед собственно сохранением исправленной статьи и ее удалением выполнять в контроллере соответствующую проверку.

К сожалению, высокоуровневые классы-контроллеры `CreateView` и `UpdateView` не предоставляют возможности выполнить такую проверку. Так что мы в очередной раз реализуем всю функциональность по сохранению и удалению записи вручную, породив нужные классы-контроллеры от низкоуровневого класса `TemplateView` (см. *главу 10*).

А раз так, нам нужно создать форму для правки статьи блога. Ее код будет очень простым:

```
from django import forms
from blog.models import Blog
class BlogForm(forms.ModelForm):
  class Meta:
    model = Blog
```

Сохраним его во вновь созданном модуле `forms` пакета приложения `blog`.

Контроллеры

Контроллеры, которые мы напишем для этого приложения, чем-то напомнят таковые из приложения `goods` (см. *главу 25*). Так, контроллер вывода списка статей будет выполнять формирование набора записей программно, в зависимости от введенного посетителем ключевого слова или выбранного им тега. А контроллеры для правки и удаления статьи будут порождены от класса `TemplateView`, о чем, собственно, уже говорилось ранее.

Базовые классы

Но сначала дополним код базового класса `PageNumberView`, объявленного нами в модуле `generic.controllers`.

И прежде условимся вот о чем:

- ключевое слово для поиска статей блога, введенное посетителем в особой форме, будет передаваться в GET-параметре `search`;
- имя тега, выбранного пользователем, будет передаваться в GET-параметре `tag`.

Нам понадобится извлечь значения этих параметров, занести их в особые свойства для дальнейшего использования и потом добавить в интернет-адрес перенаправления. Выполняющий все это код мы поместим в методы базового класса `PageNumberView` (добавленный код выделен полужирным шрифтом):

```
class PageNumberView(View):
  def get(self, request, *args, **kwargs):
    . . .
    try:
      self.search = self.request.GET["search"]
    except KeyError:
      self.search = ""
    try:
      self.tag = self.request.GET["tag"]
    except KeyError:
      self.tag = ""
    return super(PageNumberView, self).get(request, *args, **kwargs)
  def post(self, request, *args, **kwargs):
    . . .
    try:
      self.success_url = self.success_url + "&search=" + ↳
      request.GET["search"]
    except KeyError:
      pass
    try:
      self.success_url = self.success_url + "&tag=" + request.GET["tag"]
    except KeyError:
      pass
    return super(PageNumberView, self).post(request, *args, **kwargs)
```

Нам также потребуется класс, который поместит значения свойств `search` и `tag` в контекст данных шаблона и который мы назовем `SearchMixin`. Поскольку он будет использоваться только в приложении `blog`, мы объявим его в модуле `views` пакета этого приложения. Код нового класса очень прост:

```
from django.views.generic.base import ContextMixin
class SearchMixin(ContextMixin):
  search = ""
  tag = ""
```

```
  def get_context_data(self, **kwargs):
    context = super(SearchMixin, self).get_context_data(**kwargs)
    context["search"] = self.search
    context["tag"] = self.tag
    return context
```

В нем как раз и объявляются упомянутые ранее свойства, в которых будут храниться введенное посетителем ключевое слово и имя выбранного им тега.

Контроллер списка статей

Контроллер списка статей `BlogListView` будет выводить статьи в обратном хронологическом порядке, а также выполнять их фильтрацию по ключевому слову или имени тега. Поэтому мы создадим его на основе класса `ArchiveIndexView` и станем формировать набор записей сами — в переопределенном методе `get_queryset`:

```
from django.views.generic.dates import ArchiveIndexView
from generic.mixins import CategoryListMixin
from django.db.models import Q
from generic.controllers import PageNumberView
from blog.models import Blog

class BlogListView(PageNumberView, ArchiveIndexView, SearchMixin,↳
CategoryListMixin):
  model = Blog
  date_field = "posted"
  template_name = "blog_index.html"
  paginate_by = 2
  allow_empty = True
  allow_future = True
  def get_queryset(self):
    blog = super(BlogListView, self).get_queryset()
    if self.search:
      blog = blog.filter(Q(title__contains = self.search) | ↳
      Q(description__contains = self.search) | ↳
      Q(content__contains = self.search))
    if self.tag:
      blog = blog.filter(tags__name = self.tag)
    return blog
```

В коде метода `get_queryset` мы сначала вызываем этот же метод, унаследованный от класса-родителя, в результате чего получаем набор записей, уже отсортированный по полю `posted` по убыванию хранящихся в нем значений. А потом, в зависимости от того, присутствуют ли в свойствах `search` и `tag` какие-либо значения, выполняем фильтрацию записей.

Ключевое слово (значение свойства `search`) мы ищем в заголовках, кратких описаниях и содержимом записей (в полях `title`, `description` и `content` соответственно).

Здесь мы впервые используем описанный в *главе 5* класс Q, позволяющий выполнять сложный поиск, критерии которого объединяются по правилам *логического ИЛИ*.

Контроллер содержимого отдельной статьи

Контроллер BlogDetailView, выводящий содержимое отдельной статьи, совсем прост. Мы создадим его на основе класса DetailView:

```
from django.views.generic.detail import DetailView
from generic.mixins import PageNumberMixin

class BlogDetailView(PageNumberView, DetailView, SearchMixin,↳
PageNumberMixin):
  model = Blog
  template_name = "blog.html"
```

Контроллер добавления статьи

Контроллер добавления статьи BlogCreate также довольно прост. Посмотрим на его код:

```
from django.contrib.messages.views import SuccessMessageMixin
from django.views.generic.edit import CreateView
from django.core.urlresolvers import reverse_lazy

class BlogCreate(SuccessMessageMixin, CreateView, CategoryListMixin):
  model = Blog
  template_name = "blog_add.html"
  success_url = reverse_lazy("blog_index")
  success_message = "Статья успешно создана"
  def form_valid(self, form):
    form.instance.user = self.request.user
    return super(BlogCreate, self).form_valid(form)
```

Мы здесь переопределяем метод get_valid, в котором записываем в поле user создаваемой записи текущего пользователя — он и станет автором статьи. Этот прием мы рассмотрели в *главе 11*, и теперь настало время применить его в деле.

Контроллер правки статьи

А код контроллера BlogUpdate, выполняющего правку статьи слога, уже заметно сложнее:

```
from django.views.generic.base import TemplateView
from django.shortcuts import redirect
from django.core.urlresolvers import reverse
```

```
from django.contrib import messages
from blog.forms import BlogForm

class BlogUpdate(PageNumberView, TemplateView, SearchMixin,↳
PageNumberMixin):
  blog = None
  template_name = "blog_edit.html"
  form = None
  def get(self, request, *args, **kwargs):
    self.blog = Blog.objects.get(pk = self.kwargs["pk"])
    if self.blog.user == request.user or request.user.is_superuser:
      self.form = BlogForm(instance = self.blog)
      return super(BlogUpdate, self).get(request, *args, **kwargs)
    else:
      return redirect(reverse("login"))
  def get_context_data(self, **kwargs):
    context = super(BlogUpdate, self).get_context_data(**kwargs)
    context["blog"] = self.blog
    context["form"] = self.form
    return context
  def post(self, request, *args, **kwargs):
    self.blog = Blog.objects.get(pk = self.kwargs["pk"])
    if self.blog.user == request.user or request.user.is_superuser:
      self.form = BlogForm(request.POST, instance = self.blog)
      if self.form.is_valid():
        self.form.save()
        messages.add_message(request, messages.SUCCESS,↳
        "Статья успешно изменена")
        redirect_url = reverse("blog_index") + "?page=" + ↳
        self.request.GET["page"]
        try:
          redirect_url = redirect_url + "&search=" + ↳
          self.request.GET["search"]
        except KeyError:
          pass
        try:
          redirect_url = redirect_url + "&tag=" + self.request.GET["tag"]
        except KeyError:
          pass
        return redirect(redirect_url)
      else:
        return super(BlogUpdate, self).get(request, *args, **kwargs)
    else:
      return redirect(reverse("login"))
```

Обратим внимание, что проверка, является ли текущий пользователь автором исправляемой статьи или суперпользователем, выполняется дважды:

1. В методе `get` — перед выводом страницы правки статьи.
2. В методе `post` — перед сохранением записи.

Повторная проверка в методе `post` позволит нам обезопаситься от хакеров, которые могут попытаться внести несанкционированные изменения в статью, отправив контроллеру данные методом POST непосредственно, а не через страницу правки.

В интернет-адрес перенаправления после успешного сохранения статьи мы вставляем значения GET-параметров, обозначающие номер страницы, ключевое слово для поиска и имя тега. Таким образом, пользователь попадет на ту же страницу списка статей, отфильтрованных по тому же ключевому слову или тегу.

В случае если текущий пользователь не является автором исправляемой статьи или суперпользователем, мы перенаправляем его на страницу входа на сайт.

Контроллер удаления статьи

Контроллер удаления статьи `BlogDelete`, в общих чертах, похож на контроллер `BlogUpdate`. За одним исключением — он не сохраняет, а удаляет статью:

```
class BlogDelete(PageNumberView, TemplateView, SearchMixin,↵
PageNumberMixin):
  blog = None
  template_name = "blog_delete.html"
  def get(self, request, *args, **kwargs):
    self.blog = Blog.objects.get(pk = self.kwargs["pk"])
    if self.blog.user == request.user or request.user.is_superuser:
      return super(BlogDelete, self).get(request, *args, **kwargs)
    else:
      return redirect(reverse("login"))
  def get_context_data(self, **kwargs):
    context = super(BlogDelete, self).get_context_data(**kwargs)
    context["blog"] = self.blog
    return context
  def post(self, request, *args, **kwargs):
    self.blog = Blog.objects.get(pk = self.kwargs["pk"])
    if self.blog.user == request.user or request.user.is_superuser:
      self.blog.delete()
      messages.add_message(request, messages.SUCCESS,↵
      "Статья успешно удалена")
      redirect_url = reverse("blog_index") + "?page=" + ↵
      self.request.GET["page"]
      try:
        redirect_url = redirect_url + "&search=" + ↵
        self.request.GET["search"]
      except KeyError:
        pass
      try:
        redirect_url = redirect_url + "&tag=" + self.request.GET["tag"]
```

```
        except KeyError:
          pass
        return redirect(redirect_url)
      else:
        return redirect(reverse("login"))
```

Здесь мы также выполняем двойную проверку, является ли текущий пользователь автором этой статьи или суперпользователем: и в методе `get`, и в методе `post`.

Шаблоны

Для приложения `blog` мы создадим пять шаблонов страниц. Помимо этого, мы исправим шаблон почтового уведомления, чтобы в нем указывалось, к какой сущности был добавлен комментарий — к товару или статье блога, и универсальный шаблон пагинации.

Исправленный универсальный шаблон пагинации

Универсальный шаблон пагинации pagination.html мы поместили в папку templates\generic уровня проекта. В интернет-адреса присутствующих в нем гиперссылок нам понадобится вставить GET-параметры, задающие ключевое слово для поиска и имя тега. Дополнения в коде этого шаблона помечены полужирным шрифтом:

```
{% if paginator.num_pages > 1 %}
  <div id="pagination">
    {% if page_obj.has_previous %}
      <div id="previous-page"><a href="?page={{
      page_obj.previous_page_number }}{% if sort and order %}
      &sort={{ sort }}&order={{ order }}{% endif %}{% if search
      %}&search={{ search }}{% endif %}{% if tag %}&tag={{ tag }}
      {% endif %}">&lt;</a></div>
    {% endif %}
    {% if page_obj.has_next %}
      <div id="next-page"><a href="?page={{
      page_obj.next_page_number }}{% if sort and order %}&sort={{ sort
      }}&order={{ order }}{% endif %}{% if search %}&search={{ search
      }}{% endif %}{% if tag %}&tag={{ tag }}{% endif %}">&gt;</a></div>
    {% endif %}
    <div id="num-pages">
      . . .
        {% if page_obj.number != pn %}
          <a href="?page={{ pn }}{% if sort and order %}
          &sort={{ sort }}&order={{ order }}{% endif %}{% if search
          %}&search={{ search }}{% endif %}{% if tag %}&tag={{ tag }}
          {% endif %}">
```

```
        {% endif %}
      . . .
    </div>
  </div>
{% endif %}
```

Теперь пагинация будет нормально работать и в блоге с активизированным поиском.

Шаблон списка статей

Создадим в папке пакета приложения `blog` вложенную папку templates и сохраним в ней код шаблона списка статей блога, дав его файлу имя blog_index.html. В той же папке — на уровне приложения — мы сохраним и остальные шаблоны, что создадим позже:

```
{% extends "main.html" %}
{% block title %}{% if search %}{{ search }} :: {% elif tag %}{{ tag }}
:: {% endif %}Блог{% endblock %}
{% block main %}
  {% include "generic/messages.html" %}
  <div class="search-form">
    <form action="" method="get">
      <input type="text" id="search" name="search" value="{{ search }}">
      <input type="submit" value="Найти">
    </form>
  </div>
  <h2>Блог</h2>
  {% if perms.blog.add_blog %}
    <p><a href="{% url "blog_add" %}">Добавить статью</a></p>
  {% endif %}
  {% for object in latest %}
    <div class="blog-article">
      <h4><a href="{% url "blog_detail" pk=object.pk %}
      ?page={{ page_obj.number }}{% if search %}&search={{ search }}
      {% endif %}{% if tag %}&tag={{ tag }}{% endif %}">
      {{ object.title }}</a></h4>
      <p class="username">
        {% if object.user.get_full_name %}
          {{ object.user.get_full_name }}
        {% else %}
          {{ object.user.get_username }}
        {% endif %}
      </p>
      <p>{{ object.description}}</p>
      <p class="posted">{{ object.posted|date:"j.m.Y H:i:s" }}</p>
      {% with names=object.tags.names %}
```

```
        {% if names.count > 0 %}
          <p class="tags">{% for name in names %}↵
          {% if not forloop.first %},
          {% endif %}<a href="{% url "blog_index" %}↵
          ?tag={{ name|urlencode }}">{{ name }}</a>{% endfor %}</p>
        {% endif %}
      {% endwith %}
      {% if user == object.user or user.is_superuser %}
        <p class="buttons">
          {% if perms.blog.change_blog %}
            <a href="{% url "blog_edit" pk=object.pk %}↵
            ?page={{ page_obj.number }}{% if search %}↵
            &search={{ search }}{% endif %}{% if tag %}&tag={{ tag }}↵
            {% endif %}">Изменить</a>
          {% endif %}
          {% if perms.blog.delete_blog %}
            <a href="{% url "blog_delete" pk=object.pk %}↵
            ?page={{ page_obj.number }}{% if search %}↵
            &search={{ search }}{% endif %}{% if tag %}&tag={{ tag }}↵
            {% endif %}">Удалить</a>
          {% endif %}
        </p>
      {% endif %}
    </div>
  {% endfor %}
  {% include "generic/pagination.html" %}
{% endblock %}
```

В названии страницы (в теге `<title>`) мы выводим, помимо всего прочего, указанное посетителем ключевое слово или имя выбранного им тега. Так мы сообщим посетителю, что в настоящий момент в блоге выполняется фильтрация статей.

Форму поиска мы поместили в блок с привязанным стилевым классом `search-form` — для этого стилевого класса мы потом зададим сдвиг элемента страницы к правому краю его родителя. В результате форма поиска будет выводиться в верхнем правом углу блока содержимого страницы `main` (см. код шаблона родителя main.html, написанного в *главе 21*). Это обычное местоположение подобного элемента страницы.

В форме поиска мы создали поле ввода, в котором будет указываться искомое ключевое слово, и дали этому полю имя `search`. А для самой формы мы указали метод отправки данных GET. Тогда при нажатии кнопки **Найти** введенное ключевое слово будет передано контроллеру `BlogListView` с GET-параметром `search`.

В составе сведений о каждой статье мы выводим имя ее автора. Если для пользователя были указаны реальные имя и фамилия, мы выводим их, разделенные пробелом, вызвав метод `get_full_name` модели `User` (см. *главу 14*). В противном случае мы выводим имя пользователя, под которым он был зарегистрирован, воспользовавшись методом `get_username`.

Привязанные к статье теги мы выводим в отдельном абзаце, разделив их имена запятыми.

Кнопки для правки и удаления статьи мы выводим только в том случае, если в поле user записи, соответствующей статье, хранится текущий пользователь, или если текущий пользователь является суперпользователем.

Шаблон отдельной статьи

Шаблон, выводящий содержимое отдельной статьи, получит имя blog.html. Его код ничем не примечателен — все это мы уже видели в *главе 25*, когда создавали шаблон сведений о товаре:

```
{% extends "main.html" %}
{% load bbcode_tags %}
{% load comments %}
{% block title %}{{ object.title }}{% endblock %}
{% block main %}
  <h2>{{ object.title }}</h2>
  <p class="username">
    {% if object.user.get_full_name %}
      {{ object.user.get_full_name }}
    {% else %}
      {{ object.user.get_username }}
    {% endif %}
  </p>
  <div>{{ object.content|bbcode|safe }}</div>
  <p class="posted">{{ object.posted|date:"j.m.Y H:i:s" }}</p>
  {% with names=object.tags.names %}
    {% if names.count > 0 %}
      <p class="tags">{% for name in names %}{% if not forloop.first %},
      {% endif %}<a href="{% url "blog_index" %}
      ?tag={{ name|urlencode }}">{{ name }}</a>{% endfor %}</p>
    {% endif %}
  {% endwith %}
  <p><a href="{% url "blog_index" %}?page={{ pn }}
  {% if search %}&search={{ search }}{% endif %}{% if tag %}
  &tag={{ tag }}{% endif %}">Назад</a></p>
  <p> </p>
  {% render_comment_list for object %}
  <p> </p>
  {% get_comment_form for object as form %}
  <div class="form">
    <form action="{% comment_form_target %}" method="post">
      {% include "generic/form.html" %}
      <input type="hidden" name="next" value="{% url "blog_detail"
      pk=object.pk %}?page={{ pn }}{% if search %}&search={{ search }}
      {% endif %}{% if tag %}&tag={{ tag }}{% endif %}">
```

```
      <div class="submit-button"><input type="submit" name="submit"
      value="Отправить"></div>
    </form>
{% endblock %}
```

Шаблон добавления статьи

Шаблон для страницы добавления статьи мы сохраним в файле blog_add.html. Он похож на шаблон для добавления новости (см. *главу 23*):

```
{% extends "main.html" %}
{% load staticfiles %}
{% block title %}Добавление статьи{% endblock %}
{% block additional_js %}
  <script src="{% static "jquery.js" %}" type="text/javascript"></script>
  <script src="{% static "imagepool.js" %}"
  type="text/javascript"></script>
{% endblock %}
{% block main %}
  <h2>Добавление статьи</h2>
  <div class="form">
    <form action="" method="post">
      {% include "generic/form.html" %}
      <div class="submit-button"><input type="submit"
      value="Добавить"></div>
    </form>
  </div>
  {% include "generic/file_lister.html" %}
  <p><a href="{% url "blog_index" %}">Назад</a></p>
{% endblock %}
```

На странице добавления статьи мы выведем интерфейс разработанного в той же *главе 23* хранилища изображений. Так мы дадим пользователю возможность вставлять изображения в содержимое статьи. Не забудем привязать к странице необходимые файлы Web-сценариев.

Шаблон правки статьи

Шаблон правки статьи blog_edit.html также не принесет нам особых сюрпризов. Отметим, что на этой странице также присутствует хранилище изображений — как и на странице правки новости:

```
{% extends "main.html" %}
{% load staticfiles %}
{% block title %}Правка статьи :: {{ blog.title }}{% endblock %}
{% block additional_js %}
  <script src="{% static "jquery.js" %}" type="text/javascript"></script>
  <script src="{% static "imagepool.js" %}"
```

```
  type="text/javascript"></script>
{% endblock %}
{% block main %}
  <h2>Правка статьи</h2>
  <div class="form">
    <form action="" method="post">
      {% include "generic/form.html" %}
      <div class="submit-button"><input type="submit"
      value="Изменить"></div>
    </form>
  </div>
  {% include "generic/file_lister.html" %}
  <p><a href="{% url "blog_index" %}?page={{ pn }}
  {% if search %}&search={{ search }}{% endif %}{% if tag %}
  &tag={{ tag }}{% endif %}">Назад</a></p>
{% endblock %}
```

Шаблон удаления статьи

Шаблон удаления статьи мы сохраним в файле blog_delete.html. Он очень похож на шаблон содержимого статьи:

```
{% extends "main.html" %}
{% load bbcode_tags %}
{% block title %}Удаление статьи :: {{ blog.title }}{% endblock %}
{% block main %}
  <h2>Удаление статьи</h2>
  <h4>{{ blog.title }}</h4>
  <p class="username">
    {% if blog.user.get_full_name %}
      {{ blog.user.get_full_name }}
    {% else %}
      {{ blog.user.get_username }}
    {% endif %}
  </p>
  <div>{{ blog.content|bbcode|safe }}</div>
  <p class="posted">{{ blog.posted|date:"j.m.Y H:i:s" }}</p>
  <form action="" method="post">
    {% csrf_token %}
    <input type="submit" value="Удалить">
  </form>
  <p><a href="{% url "blog_index" %}?page={{ pn }}
  {% if search %}&search={{ search }}{% endif %}{% if tag %}
  &tag={{ tag }}{% endif %}">Назад</a></p>
{% endblock %}
```

Мы не выводим на экран набор привязанных к статье тегов — здесь он совершенно ни к чему.

Исправленный шаблон почтового уведомления

Осталось внести исправления в созданный в *главе 25* шаблон почтового уведомления comment_notification_email.txt. Мы сделаем так, чтобы в уведомлении указывалось, к какой сущности был добавлен комментарий, — к товару или статье блога. Исправленный код на приведенном далее листинге кода шаблона выделен полужирным шрифтом:

```
На сайте был оставлен новый комментарий:

{% if comment.content_type.model == "good" %}
  - товар: {{ content_object.name }};
{% else %}
  - статья блога: {{ content_object.title }};
{% endif %}
  - страница: http://localhost:8000{{ content_object.get_absolute_url }};
  - пользователь: {{ comment.user_name }};
. . .
```

Теперь модератор, получив по почте сообщение о новом комментарии, сразу увидит, к чему он относится.

Оформление

И добавим в код таблицы стилей main.css следующий фрагмент кода:

```
.blog-article {
  padding: 10px;
  margin: 10px 0px 10px 0px;
  border: 1px solid #cccccc;
}
.blog-article h4 {
  margin-top: 0px;
}
.username {
  font-size: smaller;
  font-weight: bold;
}
.tags {
  text-align: right;
}
.search-form {
  float: right;
}
```

Эти стили зададут оформление для различных элементов вновь созданных страниц, в частности, для списка статей блога и формы поиска.

Заключительные действия

Нашими заключительными действиями будут добавление в панель навигации пункта, ссылающегося на блог, и тестирование последнего.

Откроем шаблон main.html, задающий базовый дизайн для страниц сайта, и вставим в него такой код (выделен полужирным шрифтом):

```
. . .
<li class="indented"><a href="{{ page_url }}"↵
{% if page_url == current_url %} class="current"↵
{% endif %}>Новости</a></li>
{% url "blog_index" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}↵
class="current"{% endif %}>Блог</a></li>
{% url "guestbook" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}↵
class="current"{% endif %}>Гостевая</a></li>
. . .
```

Этим мы обеспечим следование пункта **Блог** сразу за пунктом **Новости**.

Запустим отладочный Web-сервер Django, выполним вход на сайт, перейдем на блог и откроем страницу добавления статьи (рис. 26.1).

Создадим несколько записей с произвольным содержимым (автор для тестовых целей написал несколько статей, посвященных Django-программированию). Уменьшим количество элементов на отдельной странице списка до двух, чтобы заодно проверить пагинацию, и посмотрим, правильно ли выводится список статей (рис. 26.2).

Щелкнем на заголовке какой-либо статьи, чтобы перейти на страницу ее содержимого, добавим несколько комментариев и посмотрим, что получится (рис. 26.3).

Попытаемся выполнить поиск статей по введенному в форму ключевому слову и по привязанным к статьям тегам. Напоследок создадим какую-либо пробную статью и проверим на ней, как работают правка и удаление.

Что дальше?

В этой главе мы создали для нашего сайта блог. Помимо вывода статей в виде списка и содержимого выбранной статьи, он «умеет» выполнять поиск по произвольному ключевому слову и тегам.

Наш сайт почти готов. Осталось сделать страницы контактов, сведений о самой фирме и способах оплаты и вывоза товара. Этим мы займемся в следующей главе.

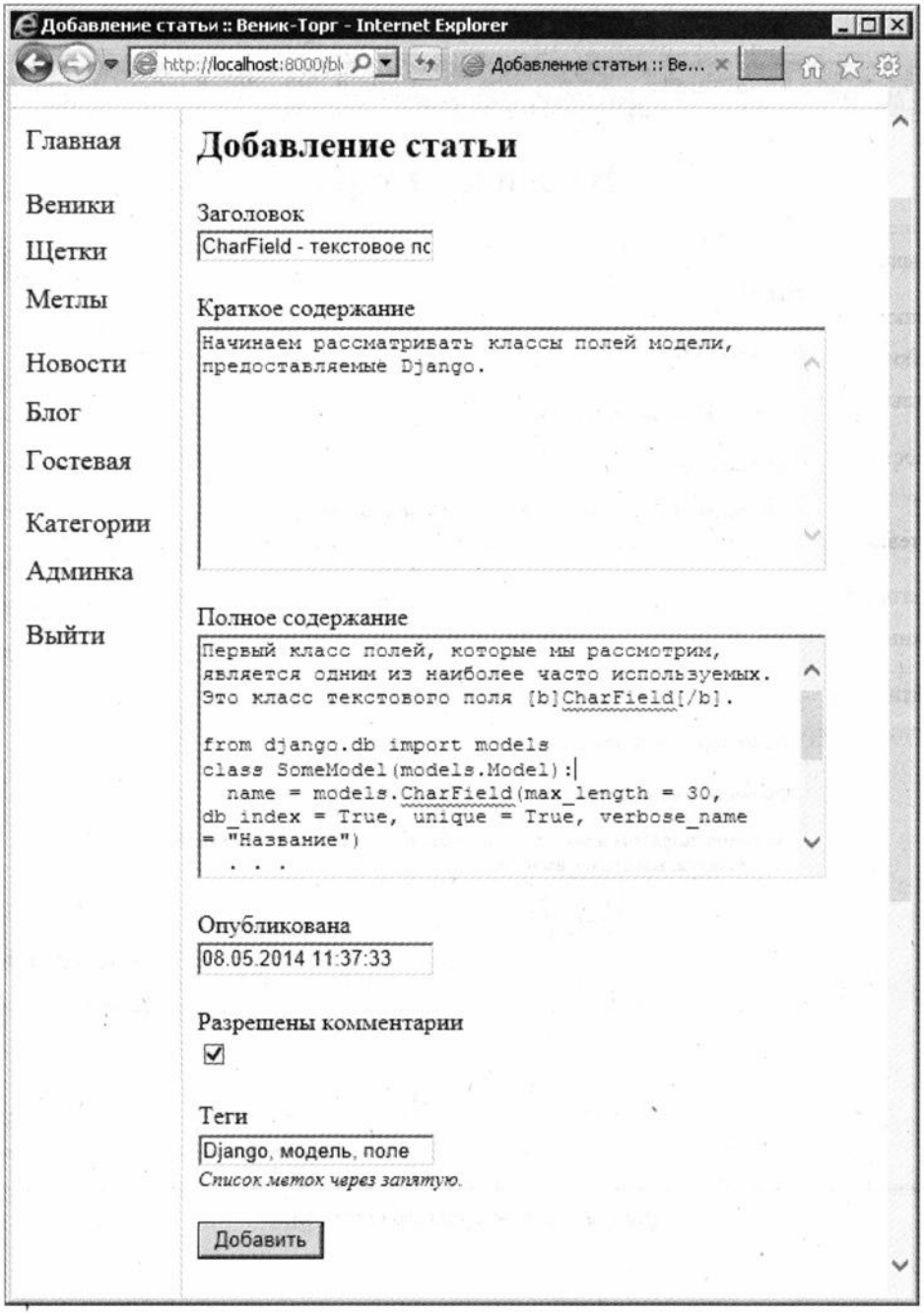


Рис. 26.1. Страница добавления статьи блога

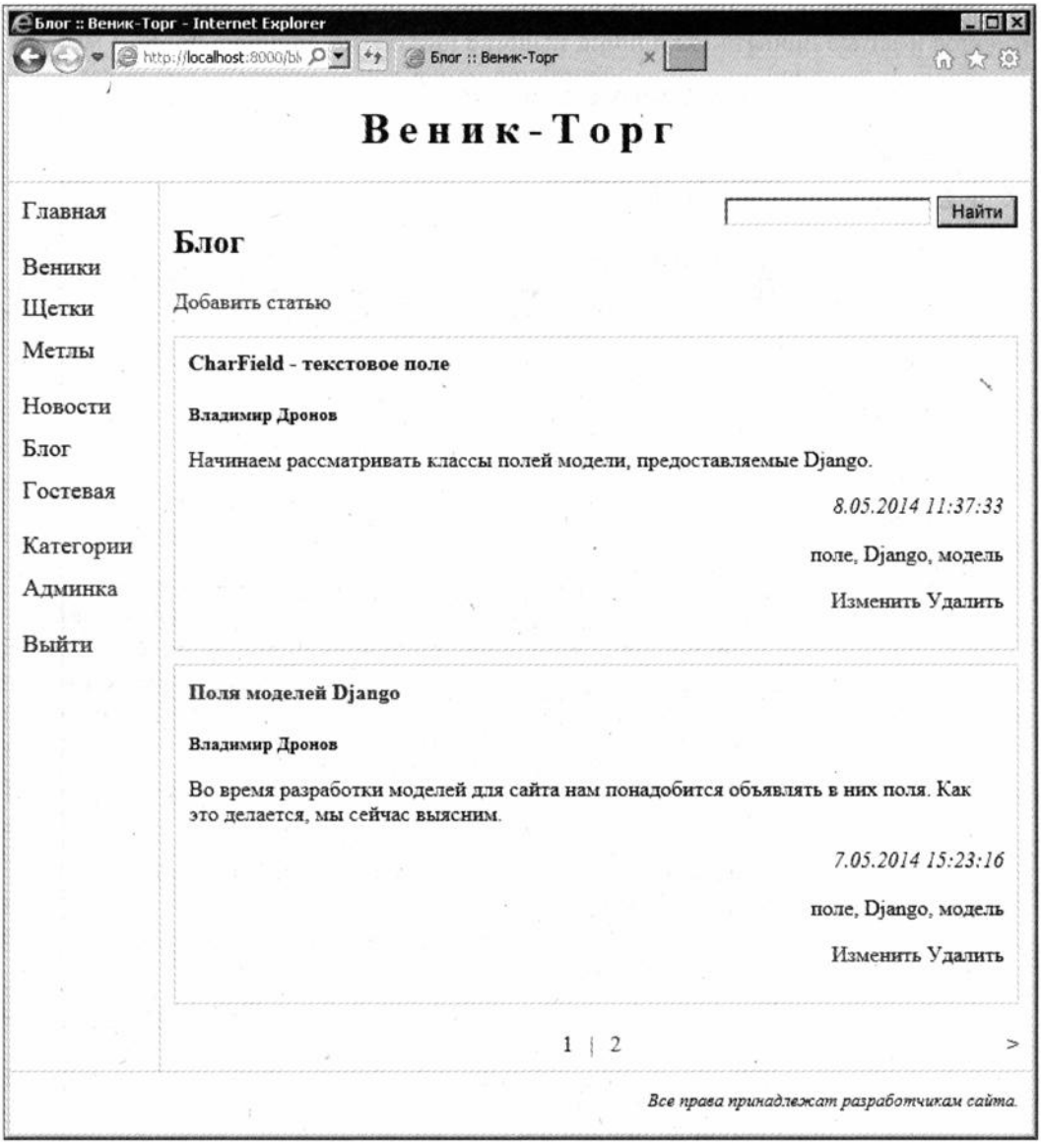


Рис. 26.2. Страница списка статей блога

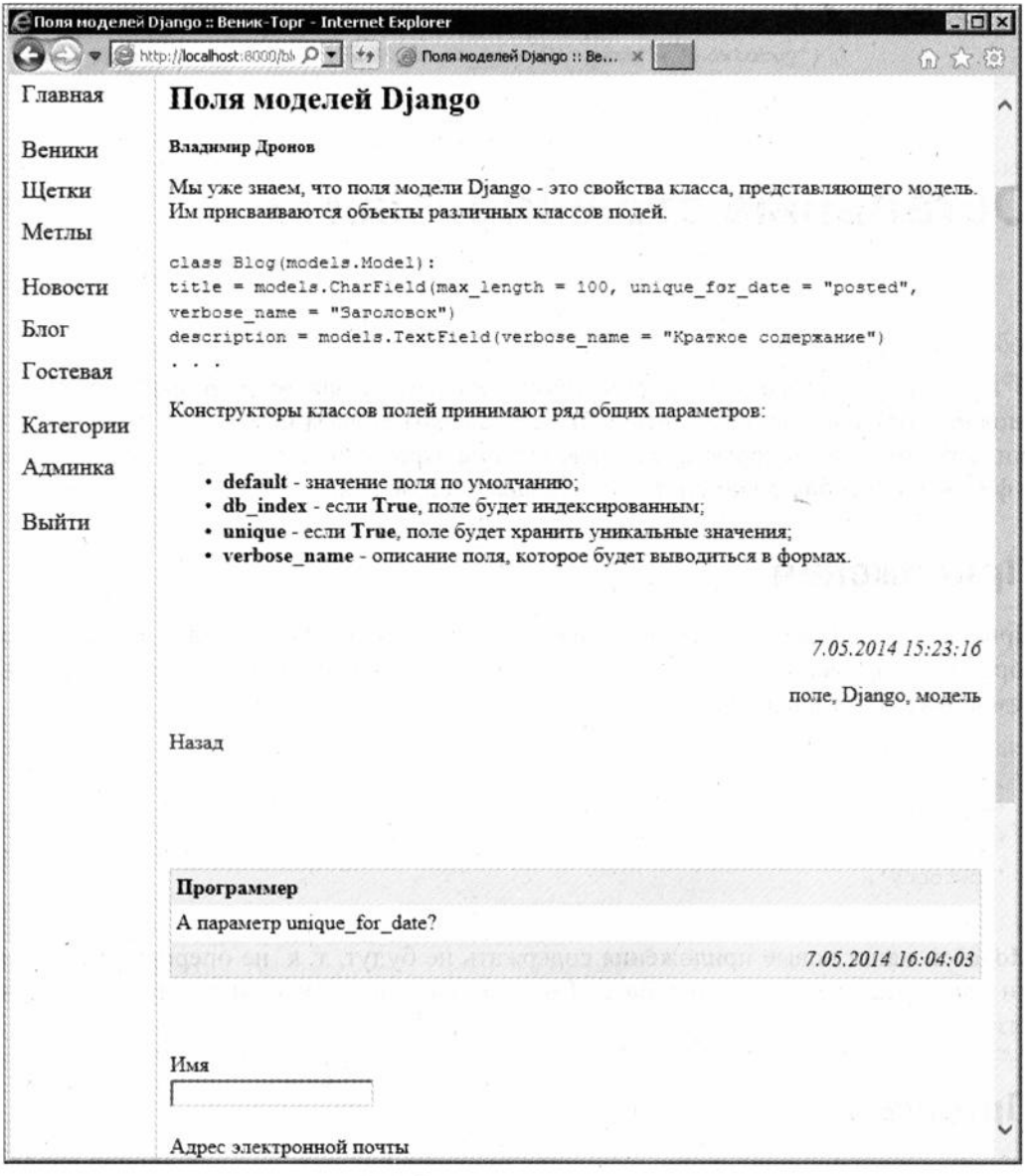


Рис. 26.3. Страница содержимого статьи

ГЛАВА 27

Остальные страницы сайта

В предыдущей главе мы создали блог, тем самым закончив разработку самых сложных разделов нашего сайта. Осталось сделать совсем немного — три простейших приложения, которые будут выводить на экран страницы контактов, сведений о фирме и способах оплаты и вывоза товара. Они не отнимут у нас много времени.

Приложения

Приложения списка контактов, сведений о фирме и способах оплаты и вывоза товара получат имена `contacts`, `about` и `howtobuy` соответственно. Создадим их и внесем в список активных приложений:

```
INSTALLED_APPS = (
  . . .
  'contacts',
  'about',
  'howtobuy',
)
```

Моделей наши новые приложения содержать не будут, т. к. не оперируют никакими данными, хранящимися в базе. Так что сразу же перейдем к написанию привязок.

Привязки

Давайте привяжем на уровне проекта не сами приложения, как делали это ранее, а непосредственно контроллеры, принадлежащие этим приложениям. Для чего добавим в модуль `urls` пакета проекта следующие фрагменты кода (выделены полужирным шрифтом):

```
. . .
from about.views import AboutView
from contacts.views import ContactsView
```

```
from howtobuy.views import HowToBuyView
urlpatterns = patterns('',
  . . .
  url(r'^about/', AboutView.as_view(), name = "about"),
  url(r'^contacts/', ContactsView.as_view(), name = "contacts"),
  url(r'^howtobuy/', HowToBuyView.as_view(), name = "howtobuy"),
)
. . .
```

Так мы несколько упростим себе работу, поскольку нам не придется выполнять привязки на уровне приложений.

Контроллеры

Контроллеры, которые мы напишем, совсем простые. Они представляют собой лишь объявления классов — потомков низкоуровневого класса-контроллера `TemplateView`. В этих классах мы лишь укажем имена соответствующих шаблонов.

Контроллер `AboutView` будет выводить страницу сведений о фирме:

```
from django.views.generic.base import TemplateView
from generic.mixins import CategoryListMixin
class AboutView(TemplateView, CategoryListMixin):
  template_name = "about.html"
```

Контроллер `ContactsView` выведет страницу со списком контактов:

```
from django.views.generic.base import TemplateView
from generic.mixins import CategoryListMixin
class ContactsView(TemplateView, CategoryListMixin):
  template_name = "contacts.html"
```

А контроллер `HowToBuyView` будет «отвечать» за вывод страницы сведений о способах оплаты и вывоза товаров:

```
from django.views.generic.base import TemplateView
from generic.mixins import CategoryListMixin
class HowToBuyView(TemplateView, CategoryListMixin):
  template_name = "howtobuy.html"
```

Поместим код всех этих классов в модули `views` пакетов приложений `about`, `contacts` и `howtobuy` соответственно.

Шаблоны

Шаблоны мы поместим на уровне соответствующих приложений — в папки templates, вложенные в папки пакетов этих приложений.

Шаблон about.html выведет сведения о фирме:

```
{% extends "main.html" %}
{% block title %}О нас{% endblock %}
```

```
{% block main %}
  <h2>О нас</h2>
  <p>Фирма &quot;Веник-Торг&quot; продает все необходимое для уборки
  помещений различного назначения: веники, щетки, метлы, совки и пр.</p>
{% endblock %}
```

Шаблон contacts.html выведет список контактов:

```
{% extends "main.html" %}
{% block title %}Контакты{% endblock %}
{% block main %}
  <h2>Контакты</h2>
  <h3>Телефоны:</h3>
  <ul>
    <li>директор: XXXXXXXXXX;</li>
    <li>отдел продаж: XXXXXXXXXX;</li>
    <li>бухгалтерия: XXXXXXXXXX.</li>
  </ul>
  <h3>Электронная почта:</h3>
  <ul>
    <li>директор: <a
    href="mailto:chief@someserver.ru">chief@someserver.ru</a>;</li>
    <li>отдел продаж: <a
    href="mailto:sales@someserver.ru">sales@someserver.ru</a>;</li>
    <li>бухгалтерия: <a
    href="mailto:finances@someserver.ru">finances@someserver.ru</a>.</li>
  </ul>
  <p>По поводу неполадок в работе сайта обращайтесь по адресу
  <a href="mailto:admin@someserver.ru">admin@someserver.ru</a>.</p>
{% endblock %}
```

Здесь мы обязательно указываем адрес электронной почты, по которому посетитель сможет отправить сообщение о возникшей в работе сайта неполадке.

Осталось написать шаблон howtobuy.html, что представит посетителю сведения о способах оплаты и вывоза приобретенного товара:

```
{% extends "main.html" %}
{% block title %}Оплата и вывоз{% endblock %}
{% block main %}
  <h2>Оплата и вывоз</h2>
  <h3>Способы оплаты:</h3>
  <ul>
    <li>наличный расчет (в случае приобретения в розничных
    магазинах);</li>
    <li>безналичный расчет.</li>
  </ul>
  <h3>Способы вывоза:</h3>
  <ul>
```

```
    <li>самовывоз;</li>
    <li>вывоз транспортом фирмы (10% наценки).</li>
  </ul>
  <p>По всем вопросам касательно оплаты и вывоза обращайтесь в отдел
  продаж и бухгалтерию (см. <a href="{% url "contacts" %}">страницу
  контактов</a>).</p>
{% endblock %}
```

Разумеется, поскольку фирма гипотетическая, все эти сведения взяты, что называется, с потолка. Для реальных фирм потребуется подставить в эти шаблоны реальные данные.

Заключительные действия

Наконец, вставим в панель навигации пункты, которые укажут на все созданные в этой главе страницы. Откроем шаблон-родитель main.html, что хранится в папке templates уровня проекта, и добавим в него следующий код (выделен полужирным шрифтом):

```
. . .
{% url "guestbook" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}
class="current"{% endif %}>Гостевая</a></li>
{% url "contacts" as page_url %}
<li class="indented"><a href="{{ page_url }}"↵
{% if page_url == current_url %} class="current"{% endif %}>↵
Контакты</a></li>
{% url "howtobuy" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}
class="current"{% endif %}>Оплата</a></li>
{% url "about" as page_url %}
<li><a href="{{ page_url }}"{% if page_url == current_url %}
class="current"{% endif %}>О нас</a></li>
{% if user.is_authenticated %}
. . .
```

Так мы поместим эти пункты после пункта **Гостевая**, отделив их от последнего увеличенным просветом, для чего привяжем к первому из вновь добавленных пунктов стилевой класс indented.

Запустим отладочный Web-сервер, откроем главную страницу сайта и перейдем, скажем, на страницу контактов (рис. 27.1). Мы сразу увидим, что содержимое левого блока, где выводится панель навигации, не помещается в него по высоте. Нам понадобится указать для обоих блоков — и левого и правого — такую минимальную высоту, чтобы содержимое левого блока не выходило за его границы.

Откроем таблицу стилей main.css, найдем код, определяющий именованные стили leftmenu и main, задающие оформление для левого и правого блоков соответственно. Добавим в них следующий код (выделен полужирным шрифтом):

```
#leftmenu {
  float: left;
  width: 100px;
  padding: 10px;
  min-height: 440px;
}
. . .
#main {
  margin-left: 110px;
  border-left: 1px #cccccc solid;
  padding: 10px;
  min-height: 440px;
}
```

Значение минимальной высоты 440 пикселов автор получил экспериментальным путем.

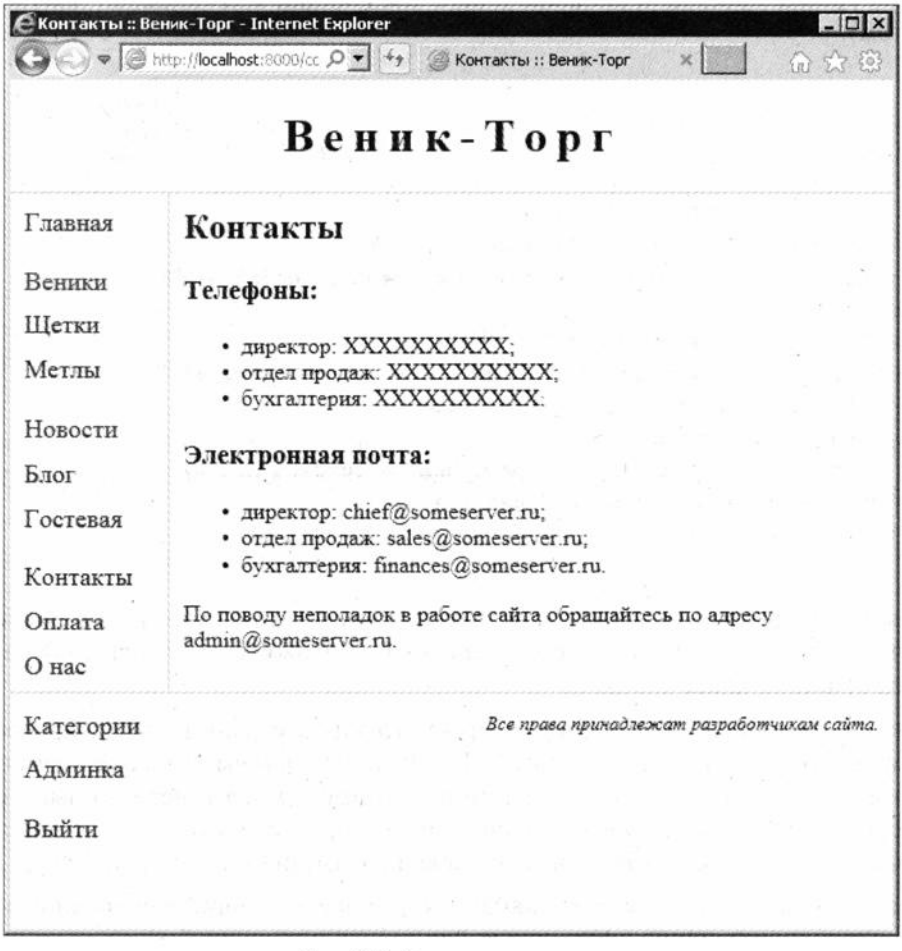


Рис. 27.1. Страница контактов

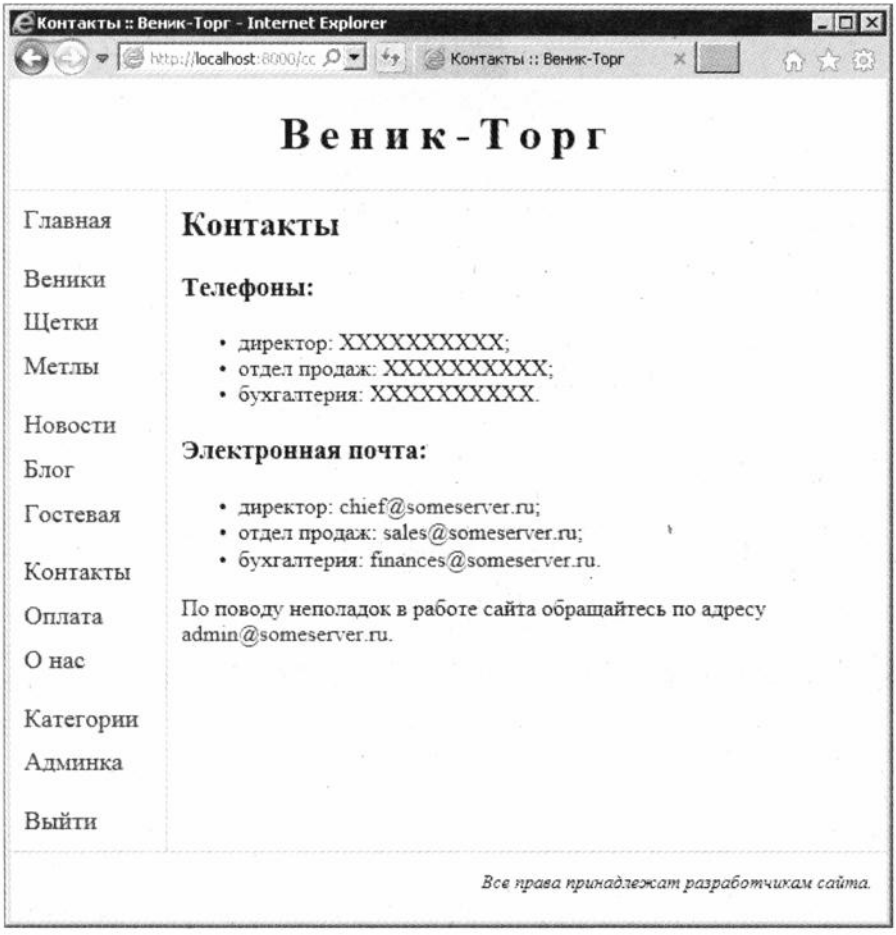


Рис. 27.2. Страница контактов с исправленным оформлением

После этого страницы нашего сайта будут отображаться нормально (рис. 27.2).

На этом разработку сайта, по крайней мере, его основной функциональности можно считать завершенной. В дальнейшем (в последней части книги) мы будем добавлять в него всяческие дополнительные возможности, наподобие генерирования каналов RSS и почтовой рассылки.

Что дальше?

Как уже было сказано, работа по созданию основной функциональности сайта закончена. В таком виде сайт, в принципе, можно публиковать в Сети. (Хотя, и тут никто не спорит, лучше сделать для него более презентабельный дизайн вместо сделанного автором и предназначенного исключительно для учебных целей.)

Далее мы рассмотрим дополнительную функциональность, предлагаемую Django, и подготовку сайта к публикации. И в следующей же главе познакомимся с инструментами для генерирования каналов RSS и Atom.

ЧАСТЬ VII

Прочие возможности Python и Django. Публикация готового Web-сайта

ГЛАВА 28

Генерирование каналов новостей RSS и Atom

В предыдущей главе мы закончили разработку сайта, сделав страницы контактов, сведений о фирме и способах оплаты и вывоза товаров. Теперь наш сайт полностью готов, и его можно публиковать в Интернете или использовать как основу для разработки более сложных решений.

Помимо основной функциональности: моделей, контроллеров, шаблонизатора, форм, подсистем разграничения доступа, комментирования, статичных страниц и пр., — Django предоставляет нам богатый набор дополнительных функций. В их число входят средства для рассылки электронной почты, генерирования каналов новостей RSS и Atom, кэширования страниц, журналирования и настройки встроенного административного Web-сайта под свои нужды. Этим средствам посвящена текущая, последняя часть книги. А в самом конце мы рассмотрим процесс подготовки Web-сайта к публикации.

И начнем мы с разговора о возможностях по генерированию каналов новостей в форматах RSS и Atom. Эти возможности столь же богаты, сколь просты в использовании.

Простейший генератор каналов новостей

Написать простейший генератор каналов новостей совсем несложно. Благо Django предоставляет для этого исключительно удобный базовый класс `Feed`.

Введение в генераторы каналов новостей

Прежде всего следует отметить, что генерирование каналов новостей в Django-сайтах выполняет особый класс-контроллер. Он является потомком упомянутого ранее класса `Feed`, объявленного в модуле `django.contrib.syndication.views`.

Контроллер-генератор новостей обычно помещается в состав приложения, обрабатывающего данные, на основе которых будут генерироваться входящие в состав канала позиции. Если брать наш случай, то для генерирования канала на основе

списка новостей сайта этот контроллер следует поместить в состав приложения news, а для вывода в виде канала списка товаров — в состав приложения goods.

Разумеется, следует также выполнить привязку контроллера-генератора новостей. Однако здесь следует быть внимательным — его привязка выполняется не совсем так, как в случае классов-контроллеров, знакомых нам по *главам 10* и *11*. Во втором параметре функции url указывается не вызов метода as_view нужного класса-контроллера, а его объект, обычно — вызов конструктора без параметров:

```
. . .
from news.views import NewsListFeed
urlpatterns = patterns('',
  . . .
  url(r'^feed/$', NewsListFeed(), name = "news_feed"),
)
```

Здесь мы выполнили привязку класса-контроллера NewsListFeed, генерирующего канал на основе списка новостей, указав во втором параметре функции url вызов конструктора этого класса.

Создание контроллера-генератора новостей

Выполнив привязку контроллера-генератора новостей, приступим к рассмотрению процесса его создания. Оно заключается в объявлении класса — потомка класса Feed, который и будет формировать канал:

```
from django.contrib.syndication.views import Feed
class NewsListFeed(Feed):
  . . .
```

Процесс написания класса-контроллера, генерирующего канал новостей, можно разделить на два этапа. Рассмотрим их по очереди.

Формирование сведений о самом канале новостей

Первое, что нам следует сделать, — указать сведения о самом канале новостей: его заголовок, описание, интернет-адрес относящейся к нему страницы, имя его автора и др.

Задать эти сведения можно двумя путями:

- если они представляют собой раз и навсегда заданные значения, мы можем указать их в соответствующих свойствах объявляемого класса;
- если же их значения вычисляются в процессе работы контроллера, мы напишем формирующий их код в переопределенных методах объявляемого класса.

Свойство, задающее какое-либо значение, что характеризует канал новостей, имеет то же имя, что и выполняющий аналогичную задачу метод. Например, свойство, указывающее заголовок канала, и задающий его метод имеют одно имя — title.

Свойства и аналогичные им методы класса Feed, что служат для задания сведений о канале новостей, приведены в табл. 28.1.

Таблица 28.1. Свойства и методы класса `Feed`, задающие параметры канала новостей

Свойство (метод)	Описание
`title`	Заголовок канала
`description`	Описание канала. Используется только в каналах формата RSS
`subtitle`	Описание канала. Используется только в каналах формата Atom
`link`	Интернет-адрес страницы, где выводятся данные, на основе которых генерируется канал
`author_name`	Имя автора канала
`author_email`	Адрес электронной почты автора канала
`author_link`	Интернет-адрес страницы со сведениями об авторе канала
`feed_copyright`	Сведения о правах автора канала
`ttl`	Время, в течение которого канал новостей является актуальным, заданное в секундах
`categories`	Список категорий, к которым относятся новости, включенные в состав канала

Для любого канала новостей обязательными к указанию являются заголовок, описание и интернет-адрес страницы с данными, на основе которых сгенерирован этот канал (свойства или методы `title`, `description` и `link`). Остальные данные указывать не обязательно (а многие из них указываются лишь в особых случаях).

Все приведенные в табл. 28.1 свойства и методы должны хранить или возвращать в качестве результата строки. Исключение составляет свойство (метод) `categories`, которое должно хранить (возвращать) список Python, содержащий строки. Что касается методов, то они не должны принимать параметров.

Следует задать и формат канала. Для этого служит свойство `feed_type`. В качестве его значения указывается значение одной из следующих переменных, объявленных в модуле `django.utils.feedgenerator`:

- `Rss201rev2Feed` — формат RSS версии 2.01 (значение по умолчанию);
- `RssUserland091Feed` — формат RSS версии 0.91;
- `Atom1Feed` — формат Atom версии 1.0.

Вот примеры:

```
from django.contrib.syndication.views import Feed
from django.core.urlresolvers import reverse_lazy
class NewsListFeed(Feed):
  title = "Новости сайта"
  description = "Новости сайта фирмы 'Веник-Торг'"
  link = reverse_lazy("news_index")
```

Здесь мы создаем контроллер для канала, хранящего список новостей нашего сайта. Поскольку мы не указали явно формат канала, последний будет сгенерирован в формате RSS 2.01.

```
from django.contrib.syndication.views import Feed
from django.core.urlresolvers import reverse
from django.utils.feedgenerator import Atom1Feed
from categories.models import Category
class GoodsListFeed(Feed):
  feed_type = Atom1Feed;
  def title(self):
    return "Товары"
  def subtitle(self):
    return "Новости сайта фирмы 'Веник-Торг'"
  def link(self):
    return reverse("news_index")
  def ttl(self):
    return "600"
  def categories(self):
    s = []
    for category in Categories:
      s = s + [category.name]
    return s
```

А здесь создаем контроллер для канала-списка товаров и указываем для него формат Atom 1.0.

Формирование отдельных позиций канала

Указав сведения о самом канале, мы можем начать формирование отдельных его позиций.

Прежде всего, нам необходимо подготовить список записей модели, на основе которых будут генерироваться позиции канала. Это выполняется в переопределенном методе `items` класса-контроллера, генерирующего канал. Метод `items` не принимает параметров и в качестве результата возвращает список записей:

```
. . .
from news.models import New
class NewsListFeed(Feed):
  . . .
  def items(self):
    return New.objects.all()[0:5]
```

Теперь контроллер `NewsListFeed` сгенерирует канал на основе пяти самых последних новостей.

```
. . .
from goods.models import Good
class GoodsListFeed(Feed):
  . . .
  def items(self):
    return Good.objects.order_by("name")
```

А контроллер `GoodsListFeed` сгенерирует канал на основе всего списка товаров, отсортированного по их названиям.

Далее мы укажем сведения об отдельной позиции канала. Для этого также используются особые свойства и методы класса `Feed` (табл. 28.2).

Таблица 28.2. *Свойства и методы класса* `Feed`*, задающие параметры отдельной позиции канала*

Свойство (метод)	Описание
`item_title`	Заголовок позиции
`item_description`	Описание позиции
`item_link`	Интернет-адрес страницы, где выводятся подробные сведения о позиции. Отметим, что для этого метода не существует одноименного свойства
`item_author_name`	Имя автора позиции
`item_author_email`	Адрес электронной почты автора позиции
`item_author_link`	Интернет-адрес страницы со сведениями об авторе позиции
`item_pubdate`	Дата публикации сведений, относящихся к позиции
`item_copyright`	Сведения о правах автора позиции
`item_categories`	Список категорий, к которым относится позиция

Здесь обязательными к указанию являются заголовок, описание и интернет-адрес страницы с подробными сведениями о позиции (свойства или методы `title`, `description` и `link`). Если свойство (метод) `link` не указано, для получения интернет-адреса будет автоматически вызван объявленный в модели метод `get_absolite_url` (см. *главы 5* и *15*).

Все приведенные в табл. 28.2 свойства и методы должны хранить или возвращать строки. Исключение составляют свойство (метод) `categories`, которое нам уже знакомо, и `pubdate`, которое должно хранить (возвращать) значение даты.

Что касается методов, предназначенных для получения сведений о позиции канала, то они имеют в классе `Feed` две разновидности:

- не принимающие параметров — служат для указания сведений, которые не берутся из модели;
- принимающие в качестве единственного параметра обрабатываемую в данный момент запись модели — служат для указания сведений, взятых из модели.

Вот примеры:

```
. . .
from django.core.urlresolvers import reverse
class NewsListFeed(Feed):
  . . .
```

```
  def item_title(self, item):
    return item.title
  def item_description(self, item):
    return item.description
  def item_pubdate(self, item):
    return item.posted
  def item_link(self, item):
    return reverse("news_detail", kwargs = {"pk": item.pk})
```

Этот код будет формировать сведения для отдельных позиций канала с новостями сайта. Отметим, что здесь мы используем методы, относящиеся только ко второй разновидности — получающие в качестве параметра запись модели (поскольку позиции канала формируются на основе записей модели).

```
. . .
from django.core.urlresolvers import reverse
class GoodsListFeed(Feed):
  . . .
  def item_title(self, item):
    return item.name
  def item_description(self, item):
    return item.description
  def item_categories(self, item):
    return [item.category.name]
  def item_link(self, item):
    return reverse("goods_detail", kwargs = {"pk": item.pk})
```

А этот код сформирует позиции канала с товарами.

Вывод гиперссылки на канал новостей

Закончив с контроллером-генератором канала, займемся шаблоном. Нам следует вставить в него гиперссылку, которая укажет на канал новостей.

Обычно для указания на наличие канала служат особые графические значки — отдельные для форматов RSS и Atom. Такой значок помещается в гиперссылку, ведущую на канал, выводится над списком позиций, на основе которых формируется канал, и выравнивается по правому краю. Вот HTML-код, который мы можем для этого использовать:

```
<div style="text-align: right;">
  <a href="{% url "news_feed" %}" target="_blank"><img
  src="{% static "images/rss.png" %}"></a>
</div>
```

Здесь предполагается, что графический значок канала хранится в файле rss.png в подпапке images папки статичных файлов. А в теге `<a>`, создающем гиперссылку на канал, следует указать атрибут `target` со значением `"_blank"` — тогда содержимое канала будет открыто в новом окне или вкладке Web-обозревателя.

Более сложный генератор каналов новостей

Если нам достаточно генерировать канал на основе новостей сайта или списка всех товаров, мы используем простейший класс-контроллер, написанный ранее. Но если мы хотим генерировать канал на основе списка товаров, относящихся к какой-то определенной категории, то столкнемся с проблемой.

Проблема эта заключается в том, что нам следует передать контроллеру-генератору канала идентификатор категории. И если подходящую случаю привязку мы создадим без труда:

```
urlpatterns = patterns('',
  . . .
  url(r'^(?P<pk>\d+)/feed/$', GoodsListFeed(), name = "goods_feed"),
)
```

то как нам получить этот идентификатор в самом контроллере?

В *главах 10* и *11*, разрабатывая классы-контроллеры, мы, чтобы получить переданные им в интернет-адресе параметры, переопределяли в классах-потомках метод `get`. В классе-потомке класса `Feed` мы в таком случае переопределим метод `get_object`. Этот метод принимает тот же набор параметров, что его «коллега» `get`, и должен возвращать запись модели, выбранную согласно значениям полученных параметров. Если же подходящей записи найдено не будет, должно генерироваться исключение `ObjectDoesNotExist`, объявленное в модуле `django.core.exceptions`:

```
from django.core.exceptions import ObjectDoesNotExist
class GoodsListFeed(Feed):
  . . .
  def get_object (self, request, *args, **kwargs):
    try:
      return Category.objects.get(pk = kwargs["pk"])
    except Category.DoesNotExist:
      raise ObjectDoesNotExist("Нет такой категории!")
```

В табл. 28.1 были приведены методы, которые возвращают сведения о самом канале. Тогда мы не рассматривали случай, когда эти сведения берутся из записи модели, и использовали обычную разновидность таких методов, не принимающую никаких параметров. Но в данном случае нам придется использовать их расширенную разновидность, которая принимает в качестве единственного параметра результат, возвращенный методом `get_object`, — извлеченную из модели запись:

```
class GoodsListFeed(Feed):
  . . .
  def title(self, obj):
    return obj.name
  def description(self, obj):
    return "Товары, относящиеся к категории '" + obj.name + "'"
```

```
  def link(self, obj):
    return reverse("goods_index", kwargs = {"pk": obj.pk})
  def categories(self, obj):
    return [obj.name]
```

Что касается метода `items`, создающего набор записей для генерирования на их основе канала, то мы также используем его расширенную разновидность:

```
class GoodsListFeed(Feed):
  . . .
  def items(self, obj):
    return Good.objects.filter(category = obj).order_by("name")
```

Формирование отдельных позиций канала в таком контроллере выполняется уже знакомым нам образом.

Одновременное формирование каналов в форматах RSS и Atom

Все сайты, генерирующие каналы новостей, предлагают две их разновидности: в форматах RSS и Atom. Чтобы не писать для каждого канала полноценный контроллер, мы можем создать один контроллер на основе другого.

Сначала создаем класс-контроллер для генерирования канала, скажем, формата RSS:

```
class RssNewsListFeed(Feed):
  . . .
```

После чего пишем класс-контроллер для канала другого формата (в нашем случае — Atom), сделав его потомком написанного ранее класса:

```
from django.utils.feedgenerator import Atom1Feed
class AtomNewsListFeed(RssNewsListFeed):
  feed_type = Atom1Feed
  subtitle = RssNewsListFeed.description
```

Не забываем, что описание канала RSS задает свойство или метод `description`, а описание канала Atom — свойство или метод `subtitle`.

И не забудем выполнить привязку обоих контроллеров:

```
from news.views import RssNewsListFeed, AtomNewsListFeed
urlpatterns = patterns('',
  . . .
  url(r'^feed/rss/$', RssNewsListFeed(), name = "news_feed_rss"),
  url(r'^feed/atom/$', AtomNewsListFeed(), name = "news_feed_atom"),
)
```

Генераторы каналов для сайта «Веник-Торг»

Для закрепления полученных знаний давайте добавим нашему сайту возможность генерирования каналов новостей. Один канал мы будем генерировать на основе новостей сайта, а другой — на основе списка товаров, относящихся к определенной категории. Оба канала мы предоставим в форматах RSS и Atom.

Генератор канала новостей сайта

Начнем с генератора канала новостей сайта как более простого в реализации. Пусть канал генерируется на основе пяти последних новостей. Как мы помним, списком новостей у нас «заведует» приложение `news`.

Привязки

Откроем модуль `urls` пакета приложения `news` и дополним список привязок следующими элементами, не забыв также указать выражение, импортирующее нужные классы-контроллеры:

```
. . .
from news.views import RssNewsListFeed, AtomNewsListFeed
urlpatterns = patterns('',
  . . .
  url(r'^feed/rss/$', RssNewsListFeed(), name = "news_feed_rss"),
  url(r'^feed/atom/$', AtomNewsListFeed(), name = "news_feed_atom"),
)
```

Контроллеры

Напишем классы-контроллеры, которые будут генерировать нам каналы в форматах RSS и Atom. Поместим их код в модуль `views` пакета приложения:

```
from django.contrib.syndication.views import Feed
from django.core.urlresolvers import reverse
from django.utils.feedgenerator import Atom1Feed

class RssNewsListFeed(Feed):
  title = "Новости сайта фирмы 'Веник-Торг'"
  description = title
  link = reverse_lazy("news_index")
  def items(self):
    return New.objects.all()[0:5]
  def item_title(self, item):
    return item.title
  def item_description(self, item):
    return item.description
  def item_pubdate(self, item):
    return item.posted
```

```
  def item_link(self, item):
    return reverse("news_detail", kwargs = {"pk": item.pk})

class AtomNewsListFeed(RssNewsListFeed):
  feed_type = Atom1Feed
  subtitle = RssNewsListFeed.description
```

Комментировать здесь совершенно нечего, т. к. все нам давно знакомо.

Шаблон

Внесем изменения в код шаблона news_index.html, вставив в него теги, которые выведут на экран графические значки каналов и сформируют на их основе соответствующие гиперссылки. Добавленный код выделен полужирным шрифтом:

```
{% extends "main.html" %}
{% load staticfiles %}
{% block title %}Новости{% endblock %}
{% block main %}
  . . .
  {% if perms.news.add_new %}
    <p><a href="{% url "news_add" %}">Добавить новость</a></p>
  {% endif %}
  <div class="feeds">
    <a href="{% url "news_feed_rss" %}" target="_blank"><img
    src="{% static "rss.png" %}"></a>
    <a href="{% url "news_feed_atom" %}" target="_blank"><img
    src="{% static "atom.png" %}"></a>
  </div>
  . . .
{% endblock %}
```

Найдем в Интернете значки, обозначающие каналы новостей форматов RSS и Atom, и поместим их в папку статичных файлов уровня проекта. Автор сохранил эти значки в файлах rss.png и atom.png — если вы дали файлам значков другие имена, соответственно исправьте приведенный здесь код.

Добавим также в таблицу стилей main.css стилевой класс `feeds`, который задаст для блока с гиперссылками на каналы новостей выравнивание по правому краю.

```
.feeds {
  text-align: right;
}
```

Заключительные действия

Теперь следует проверить контроллеры-генераторы каналов в работе. Запустим отладочный Web-сервер, зайдем на страницу новостей и щелкнем на любом из значков-гиперссылок, указывающем на канал. Содержимое канала откроется в новом окне или вкладке Web-обозревателя (рис. 28.1).

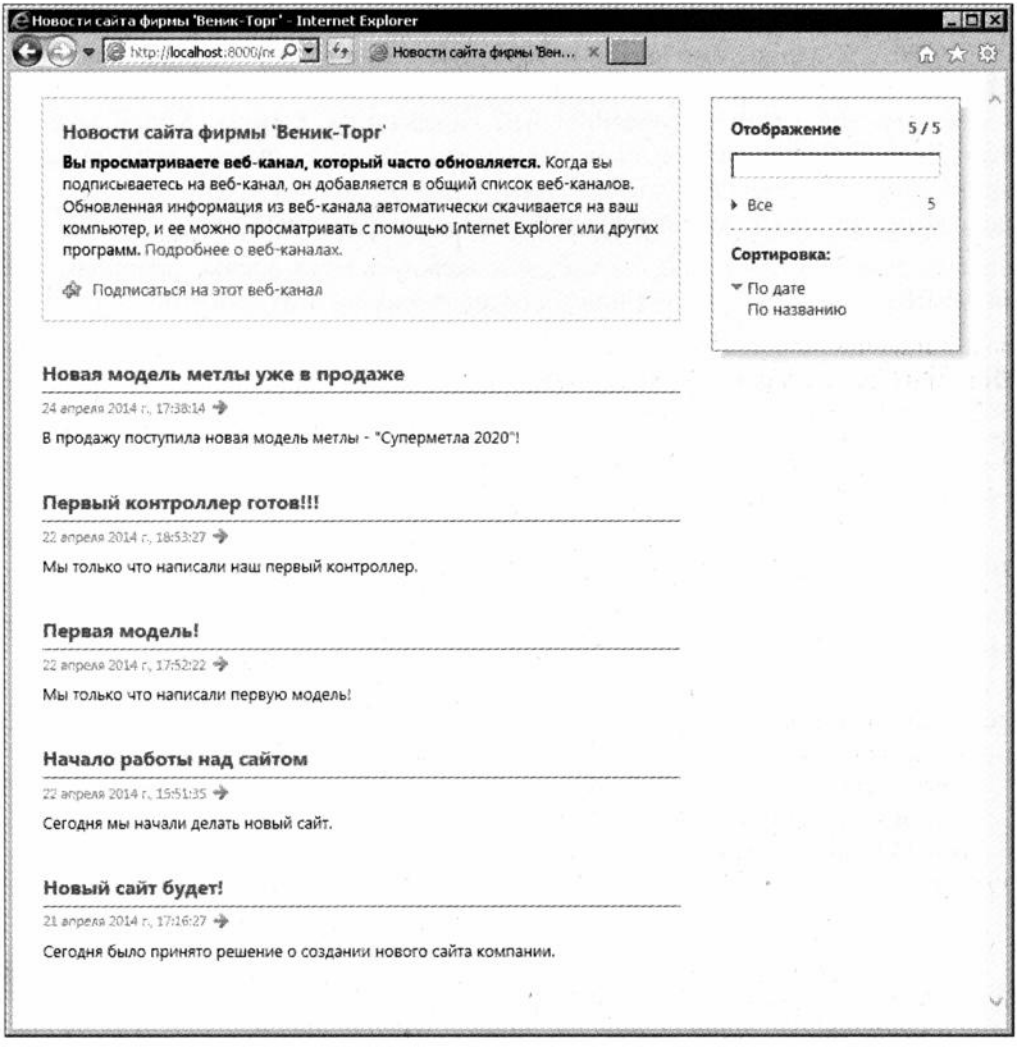


Рис. 28.1. Содержимое канала новостей сайта

Но если мы щелкнем на заголовке какой-либо из позиций канала новостей, то с удивлением обнаружим, что выполнили переход на непонятный сайт с доменным именем **example.com**. Что это еще такое?

Дело в том, что в процессе разработки сайта мы задействовали встроенное приложение `django.contrib.sites` — подсистему сайтов Django, которая обеспечивает работу нескольких сайтов на одной копии библиотеки. Эта подсистема требуется для успешной работы подсистемы комментирования.

Приложение `django.contrib.sites` ведет список сайтов, которые работают на данной копии Django. По умолчанию оно создает в списке одну позицию, соответствующую первому (и, в нашем случае, единственному) сайту. А по умолчанию этот сайт имеет доменное имя **example.com**, которое и подставляется в интернет-адреса гиперссылок в канале новостей.

Чтобы проверить, туда ли ведут гиперссылки в канале новостей, мы укажем для нашего сайта доменное имя **localhost:8000**. Зайдем на встроенный административный сайт, отыщем на его главной странице таблицу с заголовком **Sites** и щелкнем на единственной присутствующей в ней гиперссылке **Сайты**. Когда откроется страница со списком сайтов, щелкнем на единственном пункте этого списка — **example.com**. На странице правки сайта зададим доменное имя `localhost:8000` в поле ввода **Доменное имя** и сохраним сделанные изменения.

Осталось лишь обновить канал новостей и щелкнуть на заголовке любой его позиции. Мы должны попасть на страницу с содержимым данной новости.

Генератор канала товаров

Настала очередь класса-контроллера, который будет генерировать канал новостей на основе перечня товаров, относящихся к определенной категории. Работу со списком товаров у нас выполняет приложение `goods`.

Привязки

Дополним список привязок в модуле `urls` пакета приложения следующими элементами:

```
. . .
from goods.views import RssGoodsListFeed, AtomGoodsListFeed
urlpatterns = patterns('',
  . . .
  url(r'^(?P<pk>\d+)/feed/rss/$', RssGoodsListFeed(),
  name = "goods_feed_rss"),
  url(r'^(?P<pk>\d+)/feed/atom/$', AtomGoodsListFeed(),
  name = "goods_feed_atom"),
)
```

Здесь мы передаем через именованный параметр `pk` идентификатор категории.

Контроллеры

Код классов-контроллеров, генерирующих каналы на основе списка товаров, мы также поместим в модуль `views` пакета приложения. Где же еще находиться контроллерам?..

```
from django.contrib.syndication.views import Feed
from django.utils.feedgenerator import Atom1Feed
from django.core.exceptions import ObjectDoesNotExist

class RssGoodsListFeed(Feed):
  def get_object (self, request, *args, **kwargs):
    try:
      return Category.objects.get(pk = kwargs["pk"])
    except Category.DoesNotExist:
      raise ObjectDoesNotExist("Нет такой категории!")
```

```
  def title(self, obj):
    return obj.name
  def description(self, obj):
    return "Товары, относящиеся к категории '" + obj.name + "'"
  def link(self, obj):
    return reverse("goods_index", kwargs = {"pk": obj.pk})
  def categories(self, obj):
    return [obj.name]
  def items(self, obj):
    return Good.objects.filter(category = obj).order_by("name")
  def title(self, obj):
    return "Товары, относящиеся к категории '" + obj.name +↵
    "' :: Веник-Торг"
  def description(self, obj):
    return self.title(obj)
  def item_categories(self, item):
    return [item.category.name]
  def item_link(self, item):
    return reverse("goods_detail", kwargs = {"pk": item.pk})

class AtomGoodsListFeed(RssGoodsListFeed):
  feed_type = Atom1Feed
  subtitle = RssGoodsListFeed.description
```

Здесь нам также все знакомо.

Шаблоны

Исправим шаблон goods_index.html аналогично тому, как ранее исправили шаблон news_index.html. Добавленный код выделен полужирным шрифтом:

```
{% extends "main.html" %}
{% load thumbnail %}
{% load staticfiles %}
{% block title %}{{ category.name }}{% endblock %}
{% block main %}
  . . .
  {% if perms.goods.add_good %}
    <p><a href="{% url "goods_add" pk=category.pk %}↵
    ?page={{ page_obj.number }}&sort={{ sort }}&order={{ order }}">↵
    Добавить товар</a></p>
  {% endif %}
  <div class="feeds">
    <a href="{% url "goods_feed_rss" pk=category.pk %}"
    target="_blank"><img src="{% static "rss.png" %}"></a>
    <a href="{% url "goods_feed_atom" pk=category.pk %}"
```

```
    target="_blank"><img src="{% static "atom.png" %}"></a>
  </div>
  . . .
{% endblock %}
```

И проверим контроллер канала товаров в действии, как ранее проверили контроллер канала новостей сайта.

Что дальше?

В этой главе мы изучили инструменты Django, предназначенные для генерирования каналов новостей в формате RSS и Atom. А также дополнили такой функцией наш сайт.

В следующей главе мы будем разбираться со средствами для рассылки электронной почты. А в качестве практики встроим в сайт фирмы «Веник-Торг» почтовую рассылку уведомлений о недавно опубликованных новостях.

ГЛАВА 29

Рассылка электронной почты

В предыдущей главе мы познакомились с инструментами Django, предназначенными для генерирования каналов новостей в форматах RSS и Atom. Теперь наш сайт «умеет» генерировать подобные каналы на основе новостей сайта и списка товаров, относящихся к выбранной посетителем категории.

Многие современные сайты предоставляют посетителю возможность получать по электронной почте уведомления о каких-либо событиях — например, о появлении на сайте очередной новости. Есть ли в составе Django средства для чего-то подобного? Разумеется, есть.

> ***Параметры почтового SMTP-сервера***
>
> Для успешной отправки электронных писем в настройках проекта следует указать параметры почтового SMTP-сервера, которому будут отправляться письма. Как это сделать, было описано в *главе 15*.

Все функции, выполняющие отправку электронных писем, рассмотренные в этой главе, объявлены в модуле `django.core.mail`.

Разовая отправка электронного письма

Проще всего выполнить разовую отправку электронного письма. Для этого используется функция `send_mail`.

Функция `send_mail` принимает довольно много параметров. Их значения задаются в виде строк, если не указано иное:

- тема письма;
- содержимое письма;
- адрес отправителя;
- список адресов получателей;
- `fail_silently` — если `False` (значение по умолчанию), то в случае возникновения неполадок при отправке письма будет сгенерировано соответствующее ис-

ключение. Если `True`, функция завершит работу без генерирования исключения. Исключения, возникающие при работе функций отправки почты, будут описаны чуть позже;

- `auth_user` — имя пользователя для подключения к SMTP-серверу. Если не указано, будет использовано имя пользователя, заданное в параметрах почтового сервера (см. *главу 15*);
- `auth_password` — пароль для подключения к SMTP-серверу. Если не указан, будет использован пароль, записанный в параметрах почтового сервера.

Обязательными являются только первые четыре параметра этой функции. Остальные можно не указывать.

```
from django.core.mail import send_mail
send_mail("Привет!", "Привет от Python и Django!",↳
"sender@someserver.ru", ["receiver@otherserver.ru"],↳
fail_silently = True)
```

Здесь мы отправляем письмо с темой `Привет!` и содержимым `Привет от Python и Django!` от имени пользователя с адресом **sender@someserver.ru** по адресу **receiver@otherserver.ru** и указываем, чтобы при возникновении неполадок не генерировалось никаких исключений.

```
send_mail("Привет!", "Привет от Python и Django!",↳
"sender@someserver.ru",↳
["receiver1@otherserver.ru", "receiver2@otherserver.ru"],↳
auth_user = "admin", auth_password = "123", fail_silently = True)
```

А здесь отправляем письмо сразу по двум адресам и указываем, что для подключения к SMTP-серверу будут использоваться имя `admin` и пароль `123`.

Если для параметра `fail_silently` указано значение `False` (а это, как мы уже знаем, его значение по умолчанию), в случае возникновения неполадок при отправке почты функция `send_mail` будет генерировать различные исключения. Все они объявлены в модуле `smtplib` (кстати, входящего в состав не Django, а самого Python) и приведены в табл. 29.1.

***Таблица 29.1.** Исключения, генерируемые в случае возникновения неполадок при отправке почты*

Исключение	Условия возникновения
`SMTPException`	Класс-родитель для всех последующих классов исключений
`SMTPServerDisconnected`	Соединение с SMTP-сервером было внезапно разорвано
`SMTPResponseException`	Класс-родитель для всех последующих классов исключений
`SMTPConnectError`	Не удалось установить соединение с сервером
`SMTPAuthenticationError`	Неверные имя пользователя и (или) пароль
`SMTPHeloError`	Сервер отверг команду «приветствия» `HELO`, с которой начинается сеанс отправки письма

Таблица 29.1 (окончание)

Исключение	Условия возникновения
SMTPSenderRefused	Указанный почтовый адрес отправителя не существует
SMTPRecipientsRefused	Указанный почтовый адрес получателя не существует
SMTPDataError	Не удалось отправить письмо

Вот пример:

```
from smtplib import SMTPRecipientsRefused, SMTPException
try:
  send_mail("Привет!", "Привет от Python и Django!",↪
  "sender@someserver.ru", ["receiver@otherserver.ru"])
except SMTPRecipientsRefused:
  # Почтовый адрес получателя не существует
except SMTPException:
  # Возникла какая-то иная неполадка
```

Массовая рассылка электронных писем

При отправке письма функция `send_mail` устанавливает соединение с SMTP-сервером, а, закончив отправку, разрывает его. Процесс установления соединения требует определенного времени, которое в случае недостаточной скорости канала подключения к Интернету и сильной загруженности сервера может быть весьма значительным.

Поэтому, если нам нужно отправить множество разных писем различным адресатам, или, иными словами, выполнить массовую рассылку писем, использование данной функции может оказаться невыгодным именно по этой причине. Миллисекунды, в течение которых будет устанавливаться соединение с сервером, при многократном вызове функции `send_mail` вполне могут сложиться в минуты, а то и в часы.

В таком случае нам пригодится функция `send_mass_mail`, выполняющая массовую рассылку множества разных писем, которые предназначаются различным адресатам, в течение одного сеанса соединения с сервером. Что, безусловно, будет выполнено заметно быстрее.

Единственный обязательный параметр этой функции задает собственно набор писем, адресов их отправителя и получателей. Он должен представлять собой список Python, каждый элемент которого задает одно письмо и является кортежем из четырех следующих элементов:

- тема письма;
- собственно письмо;
- адрес отправителя;
- список адресов получателей.

Все эти элементы должны представлять собой строки.

Поддерживаются также необязательные параметры `fail_silently`, `auth_user` и `auth_password`, знакомые нам по функции `send_mail`:

```
from django.core.mail import send_mass_mail
letter1 = ("Привет!", "Привет от Python!", "sender@someserver.ru",↳
["receiver1@otherserver.ru"])
letter2 = ("Привет!", "Привет от Django!", "sender@someserver.ru",↳
["receiver2@otherserver.ru", "receiver3@otherserver.ru"])
send_mass_mail([letter1, letter2], fail_silently = True)
```

Здесь мы отправляем два письма за один сеанс связи с SMTP-сервером.

Отправка письма модераторам и администраторам сайта

Наконец, Django позволяет выполнить отправку электронного письма пользователям, указанным в списке модераторов и администраторов сайта.

Список модераторов указывается в переменной `MANAGERS` в модуле `settings` пакета проекта, т. е. в настройках проекта. Этот список организуется в виде кортежа, каждый элемент которого задает одного пользователя и также представляет собой кортеж из двух строковых элементов: имени пользователя и его адреса электронной почты. Все перечисленные в списке модераторов пользователи уже должны быть зарегистрированы на сайте:

```
MANAGERS = (("admin", "admin@someserver.ru"),↳
("moderator", "moderator@someserver.ru"))
```

> ***Рассылки модераторам***
>
> Кстати, именно модераторам сайта рассылаются почтовые уведомления о вновь добавленных комментариях. Об этом говорилось в *главе 15*. Модераторам также рассылаются уведомления о наличии на сайте «битых» гиперссылок.

Для собственно отправки письма модераторам служит функция `mail_managers`. Она принимает следующие параметры, указываемые в виде строк:

- тема письма;
- само письмо;
- `html_message` — письмо в формате HTML. Если указан, письмо будет отправлено именно в этом формате;
- уже знакомый нам `fail_silently`.

Первые два параметра являются обязательными.

Вот примеры:

```
from django.core.mail import mail_managers
mail_managers("Привет!", "Привет модераторам сайта 'Веник-Торг'!")
```

Здесь мы отправляем письмо в текстовом формате.

```
mail_managers("Привет!", "Привет модераторам сайта 'Веник-Торг'!",↵
html_message = "<p>Привет модераторам сайта &quot;Веник-Торг&quot;!</p>")
```

А здесь — в формате HTML.

Список администраторов сайта указывается в переменной `ADMINS` в модуле `settings` пакета проекта в том же формате, что и список модераторов:

```
ADMINS = (("admin", "admin@someserver.ru"),↵
("other_admin", "other_admin@someserver.ru"))
```

> ***РАССЫЛКИ АДМИНИСТРАТОРАМ***
>
> Администраторам рассылаются уведомления о возникновении ошибок при выполнении модулей, хранящих код сайта.

Отправку письма администраторам выполняет функция `mail_admins`. Она принимает те же параметры, что и функция `mail_managers`:

```
from django.core.mail import mail_admins
mail_admins("Привет!", "Привет администраторам сайта 'Веник-Торг'!")
```

Система рассылки уведомлений для сайта «Веник-Торг»

Нашим практическим заданием в этой главе станет разработка системы рассылки зарегистрированным в особом списке (*списке рассылки*) посетителям уведомлений о появлении новостей на сайте «Веник-Торг».

Мы не станем создавать для системы почтовой рассылки нового приложения, а используем созданное ранее приложение `contacts`, в текущей реализации лишь выводящее страницу контактов. Соответственно, форму для регистрации нового посетителя в списке рассылки мы поместим на этой странице.

Модель

Список рассылки, содержащий имена пользователей и их адреса электронной почты, нам нужно где-то хранить. Поэтому мы создадим для него новую модель.

Эта модель, названная нами `MailList`, будет содержать два текстовых поля: имя посетителя, подставляемое в текст письма в качестве приветствия, и адрес электронной почты. Последнее поле мы сделаем уникальным, чтобы какой-либо пользователь по ошибке не зарегистрировался в списке дважды.

Поля новой модели с их типами и параметрами приведены в табл. 29.2.

***Таблица 29.2.** Поля модели `MailList`*

Имя	Тип	Параметры
`username`	Текстовый	Максимальная длина — 50 символов
`email`	Адрес электронной почты	Уникальное поле

Все поля модели будут обязательными. Изначальную сортировку мы задавать не станем, т. к. список рассылки все равно нигде не выводится.

Откроем модуль `models` пакета приложения `contacts` и введем в него код модели `MailList`:

```
class MailList(models.Model):
  username = models.CharField(max_length = 50, verbose_name = "Имя")
  email = models.EmailField(unique = True, verbose_name = "E-mail")
  class Meta:
    verbose_name = "посетитель"
    verbose_name_plural = "список рассылки"
```

Не забудем выполнить синхронизацию с базой данных.

Контроллеры

Новых контроллеров мы создавать не станем, а лишь исправим два, написанных ранее:

- класс-контроллер `ContactsView` из приложения `contacts` — дополнив его функцией добавления в список рассылки нового посетителя;
- класс-контроллер `NewCreate` из приложения `news` — добавив в него код рассылки уведомлений сразу после создания новости.

> ***Другой способ рассылки уведомлений***
>
> На нашем сайте мы выполняем рассылку уведомлений после создания новости. Такой подход уместен в случае сайтов с не очень высокой посещаемостью и не очень большим списком рассылки. В противном случае следует придумать какой-либо другой способ рассылки уведомлений — скажем, добавить в модель новостей логическое поле, которое будет хранить признак, была ли выполнена рассылка уведомлений о поступлении той или иной новости, и рассылать уведомления, основываясь на значении этого поля, и по особой команде, отдаваемой администратором или модератором сайта. Также можно использовать планировщики заданий, разработанные специально для Django и выполняющие различные действия по расписанию.

Контроллер *ContactsView*

Откроем модель `views` пакета приложения `contacts` и найдем код, объявляющий класс `ContactsView`. И подумаем, что нам следует сделать.

Можно, конечно, вручную написать код, который будет создавать форму, выводить ее на экран, а потом сохранять введенные в форму данные в новой записи модели `MailList`. Но зачем это делать, если в составе Django имеется замечательный класс-контроллер `CreateView`, который прекрасно «умеет» все это делать! Нам останется лишь указать модель, где должна создаваться новая запись, интернет-адрес перенаправления после создания записи и текст сообщения.

Сказано — сделано. Перепишем хранящийся в модуле код, чтобы он выглядел следующим образом:

```
from django.views.generic.edit import CreateView
from django.core.urlresolvers import reverse_lazy
from django.contrib.messages.views import SuccessMessageMixin
from generic.mixins import CategoryListMixin
from contacts.models import MailList
class ContactsView(SuccessMessageMixin, CreateView, CategoryListMixin):
  model = MailList
  template_name = "contacts.html"
  success_url = reverse_lazy("contacts")
  success_message = "Вы успешно добавлены в список рассылки"
```

Контроллер *NewCreate*

Далее исправим код класса-контроллера `NewCreate` из модуля `views` пакета приложения `news`, дав ему функцию рассылки уведомлений сразу после сохранения добавленной новости.

Класс `CreateView`, являющийся родителем этого контроллера, поддерживает метод `form_valid`, выполняющийся при сохранении новой записи. Мы рассмотрели его еще в *главе 11*, а в *главе 26* использовали в коде контроллера `BlogCreate`. Код, рассылающий уведомления, мы также поместим в этот метод.

Если бы мы рассылали всем зарегистрированным в списке рассылки посетителям одно и то же письмо, то использовали бы функцию `send_mail`. Но, поскольку мы хотим вставлять в текст письма приветствие, адресованное конкретному посетителю-получателю письма, то применим для рассылки почты функцию `send_mass_mail`.

Исходя из сказанного, дополним код класса `NewCreate` объявлением метода `form_valid` (добавленный фрагмент выделен полужирным шрифтом):

```
from django.core.mail import send_mass_mail
from contacts.models import MailList
from broomtrade import settings
class NewCreate(SuccessMessageMixin, CreateView, CategoryListMixin):
  . . .
  def form_valid(self, form):
    output = super(NewCreate, self).form_valid(form)
    if MailList.objects.exists():
      s = "На сайте 'Веник-Торг' появилась новость:\n\n" + ↳
      form.instance.title + "\n\n" + form.instance.description + ↳
      "\n\nhttp://localhost:8000" + ↳
      reverse("news_detail", kwargs = {"pk": form.instance.pk})
      letters = []
      for maillist_item in MailList.objects.all():
        letters = letters + [("Уведомление с сайта 'Веник-Торг'",↳
        "Здравствуйте, " + maillist_item.username + "!\n\n" + s,↳
        settings.DEFAULT_FROM_EMAIL, [maillist_item.email])]
      send_mass_mail(letters, fail_silently = True)
    return output
```

В коде переопределенного метода `form_valid` мы сначала вызываем этот же метод, унаследованный от родителя. Тем самым мы дадим контроллеру возможность сохранить добавляемую новость перед тем, как будет выполнена рассылка уведомлений.

Далее мы формируем неизменяемую часть текста электронного письма, включающую заголовок и краткое описание новости и интернет-адрес страницы с ее полным содержимым. Поскольку контроллер уже сохранил новость, мы без проблем сможем получить идентификатор записи, в которой она хранится, чтобы сформировать этот интернет-адрес.

Потом мы перебираем список рассылки, извлекаем из него имена посетителей и их адреса электронной почты и создаем на их основе список, который потом передадим функции `send_mass_mail`. При этом:

- в начало подготовленного ранее содержимого письма мы добавляем текст приветствия с именем посетителя, которому адресуется данное письмо;
- в качестве адреса отправителя мы указываем значение переменной `DEFAULT_FROM_EMAIL` из модуля `settings` пакета проекта — в нем, как мы помним из *главы 15*, хранится адрес отправителя по умолчанию. Естественно, эта переменная должна существовать;
- в самом конце мы возвращаем результат, полученный от метода `form_valid` родителя. Если мы этого не сделаем, перенаправление не будет выполнено, и работа контроллера завершится с ошибкой.

Шаблон

Осталось лишь добавить в код шаблона contacts.html фрагмент, который выведет форму регистрации в списке рассылки, и не забыть обеспечить вывод на странице сообщения об успешной регистрации. Вот исправленный код (добавления выделены полужирным шрифтом):

```
{% block main %}
  {% include "generic/messages.html" %}
  <h2>Контакты</h2>
  . . .
  <p>По поводу неполадок в работе сайта обращайтесь по адресу
  <a href="mailto:admin@someserver.ru">admin@someserver.ru</a>.</p>
  <h3>Подпишитесь на наш список рассылки</h3>
  <div class="form">
    <form action="" method="post">
      {% include "generic/form.html" %}
      <div class="submit-button"><input type="submit"
      value="Подписаться"></div>
    </form>
  </div>
{% endblock %}
```

Запустим отладочный Web-сервер, зайдем на страницу контактов и зарегистрируемся в списке рассылки под разными именами, указав разные адреса электронной почты. Затем выполним вход на сайт, откроем список новостей, добавим какую-либо новость и проверим электронную почту.

Что дальше?

В этой главе мы занимались средствами отправки электронной почты, предоставляемыми Django, и выяснили, что эти средства исключительно развиты и просты в использовании. Настолько просты, что создание системы рассылки новостей для нашего сайта отняло у нас совсем немного времени.

А следующая глава будет посвящена подсистеме журналирования Django — замечательному средству просмотреть данные, обрабатываемые приложением, и, таким образом, выяснить, правильно ли оно работает.

ГЛАВА 30

Журналирование

В предыдущей главе мы рассылали средствами Django электронные письма. Выполнить это оказалось настолько просто, что нам понравилось, и мы решили дополнить свой сайт списком рассылки, высылающий зарегистрированным в нем посетителям уведомления о вновь добавленных новостях.

Эта глава будет посвящена инструменту, который очень пригодится при разработке сайтов, — подсистеме журналирования Django. Она позволит нам выяснить, какие данные обрабатывает в тот или иной момент какое-либо приложение, и понять, правильно ли оно работает.

Отладка Django-сайтов

Но сначала давайте поговорим об инструментах, которые Django предоставляет для отладки сайтов, и узнаем, насколько они могут быть нам полезны.

Если мы допустим ошибку в самом синтаксисе (*синтаксическую ошибку*), забудем импортировать класс или объявить функцию, то при попытке обратиться к странице, в коде контроллера которой присутствует ошибка, мы получим стандартную страницу сообщения, представленную на рис. 30.1.

На этой странице присутствует описание возникшей ошибки, путь к файлу модуля, где она присутствует, и даже номер строки кода. Так что локализовать и исправить ошибку не составляет никакого труда.

Но что делать, если никаких явных ошибок в коде нет, приложения функционируют, но работают не так, как нам нужно? Значит, где-то присутствует ошибка в логике приложения, *логическая ошибка*. И наша задача — поскорее найти ее и устранить.

Но как это сделать? Просматривать код, пытаться представить, как он работает, анализировать результаты работы приложения, возможно, даже временно блокировать некоторые фрагменты, превращая их в комментарии, чтобы узнать, работает ли остальной код. А это очень трудоемкий и длительный процесс, требующий внимания и выдержки.

Рис. 30.1. Стандартная страница сообщения об ошибке Django

В такой ситуации может оказаться полезным выяснить, какие данные в текущий момент обрабатывает приложение: какие значения хранятся в переменных и свойствах объектов, какие результаты возвращают функции и методы и, наконец, работают ли те или иные фрагменты кода. Но можно ли их получить? И если можно, то как?

Подсистема журналирования Django

К счастью, в составе Django поставляются инструменты, позволяющие сформировать на основе обрабатываемых приложением значений сообщение и вывести его в окне командной строки, где запущен отладочный Web-сервер, или записать эти значения в текстовый файл. Просмотрев впоследствии сохраненные таким образом данные, мы сможем понять, как работает приложение.

Помимо этого, Django по умолчанию выводит в окне командной строки сведения о полученных интернет-запросах и сообщения о критических ошибках, мешающих выполнению ключевого кода и не дающих даже вывести стандартную страницу с сообщением о возникшей ошибке.

Вывод сведений такого рода называется *журналированием*, а сам набор этих сведений, выведенный в командную строку или записанный в файл, — *журналом*.

Подсистема журналирования Django включает в свой состав набор элементов, которые можно разделить на две разновидности:

- *журнализаторы* — выполняют сбор полученных интернет-запросов, сообщений о критических ошибках и сформированных нами самими сообщений;

- *обработчики вывода* — выводят полученную от журнализаторов информацию в файл, окно командной строки или создают на его основе электронное письмо, которое отправляется администраторам сайта (пользователям, перечисленным в списке переменной ADMINS, — за подробностями обращайтесь к *главе 29*).

Кроме того, Django предоставляет программные инструменты для создания произвольных сообщений, которые будут выведены в журнал.

Использовать подсистему журналирования Django на практике очень просто. Главное — корректно ее сконфигурировать.

Настройки журналирования

Настройки подсистемы журналирования — журнализаторов и обработчиков вывода — указываются в модуле `settings` пакета проекта. Для этого используется переменная `LOGGING`.

Значение переменной `LOGGING` должно представлять собой словарь, каждый элемент которого задает определенный набор параметров. Далее приведен список ключей этих элементов:

- `version` — задает версию схемы журналирования в виде целого числа. В настоящее время поддерживается лишь версия `1`;
- `disable_existing_loggers` — если `True`, все уже существующие журнализаторы, в том числе журнализаторы, используемые по умолчанию, будут отключены. В противном случае они будут действовать параллельно с заданными нами;
- `handlers` — словарь, задающий обработчики вывода для журналирования;
- `loggers` — словарь, задающий собственно журнализаторы.

Как уже говорилось, значение параметра `handlers`, задающего список обработчиков вывода, должно представлять собой словарь, каждый элемент которого указывает параметры одного обработчика. Ключ элемента задает уникальное имя обработчика, по которому на него можно будет сослаться из описания параметров журнализатора. А значение элемента также должно представлять собой словарь, указывающий параметры этого обработчика.

Параметров обработчика не очень много, и здесь они приведены все:

- `level` — уровень сообщений в виде строки. Доступные в Django уровни сообщений указаны в порядке повышения их уровня:
  - `DEBUG` — сообщение отладочного характера;
  - `INFO` — информационное сообщение;
  - `WARNING` — предупреждение о нештатной ситуации, которая, тем не менее, не способна нарушить работу приложения;
  - `ERROR` — предупреждение о серьезной ошибке, могущей нарушить работу приложения;

- `CRITICAL` — предупреждение о критической ошибке, способной нарушить работу всего сайта.

Если указать для обработчика вывода какой-либо уровень сообщений, он будет выводить и сообщения, относящиеся к более высоким уровням. Например, если мы укажем уровень `WARNING`, обработчик выведет сообщения, относящиеся также к уровням `ERROR` и `CRITICAL`.

Отметим, что успешные интернет-запросы имеют уровень `INFO`, а успешные запросы к базе данных — уровень `DEBUG`. Неуспешные запросы имеют уровень `ERROR`;

- `class` — класс используемого обработчика вывода, заданный в виде строки:
  - `logging.StreamHandler` — выполняет вывод в командную строку;
  - `logging.FileHandler` — выполняет вывод в файл;
  - `django.utils.log.AdminEmailHandler` — отправляет полученную от журнализатора информацию по электронной почте администраторам сайта;
  - `logging.NullHandler` — вообще не выполняет вывод сообщений;
- `filename` — задает полный путь к файлу журнала в виде строки;
- `include_html` — если `True`, в электронное письмо будет вложена стандартная страница сообщения об ошибке Django. Значение по умолчанию — `False`.

Значение параметра `loggers`, определяющего набор журнализаторов, также должно представлять собой список, и указывается оно в том же формате, что и значение параметра `handlers`. Единственное исключение — ключи «внешнего» словаря здесь задают типы журнализаторов, которые будут задействованы в текущей конфигурации проекта. Нам доступны следующие типы:

- `django.request` — формирует сообщения обо всех интернет-запросах, полученных Django. Эти сообщения включают в свой состав интернет-адрес запроса и код ответа HTTP;
- `django.db.backends` — формирует сообщения на основе запросов к базе данных. Сообщения включают в себя код SQL-запроса с параметрами и продолжительность выполнения этого запроса;
- `django` — журнализатор по умолчанию, формирующий сообщения на основе любых событий, которые возникают при обработке кода;
- «пустой» тип, задаваемый пустой строкой, — обрабатывает сообщения, сформированные самим разработчиком.

А вот перечень параметров журнализаторов:

- `handlers` — список обработчиков вывода, через которые будет выполняться вывод сгенерированных журнализатором сообщений. Каждый элемент этого списка должен представлять собой строку с именем обработчика;
- `level` — уровень событий, на основе которых журнализатор будет генерировать сообщения. Список поддерживаемых Django уровней был приведен ранее;

- `propagate` — если `True`, сообщение на основе возникшего события также будет сгенерировано журнализатором `django`, в противном случае — не будет. Значение по умолчанию — `True`.

Рассмотрим примеры:

```
LOGGING = {
  'version': 1,
  'disable_existing_loggers': False,
  'handlers': {
    'file': {
      'level': 'DEBUG',
      'class': 'logging.FileHandler',
      'filename': os.path.join(BASE_DIR, 'output.log'),
    },
    'mail_admins': {
      'level': 'ERROR',
      'class': 'django.utils.log.AdminEmailHandler',
    },
  },
  'loggers': {
    'django.request': {
      'level': 'DEBUG',
      'handlers': ['file', 'mail_admins'],
    },
  },
}
```

Здесь мы определяем журнализатор, который будет перехватывать интернет-запросы уровня `DEBUG` и выше (фактически любого уровня) и передавать их двум обработчикам вывода. Первый обработчик — `file` — будет записывать все запросы в файл output.log, расположенный в папке проекта. Второй обработчик, с именем `mail_admins`, станет отсылать сообщения о запросах уровня `ERROR` и `CRITICAL` по электронной почте администраторам сайта.

```
LOGGING = {
  'version': 1,
  'disable_existing_loggers': False,
  'handlers': {
    'console': {
      'level': 'WARNING'
      'class': 'logging.StreamHandler',
    },
  },
  'loggers': {
    'django.db.backends': {
      'handlers': ['console'],
```

```
        'level': 'WARNING'
    },
  },
}
```

А здесь определяется журнализатор, который будет перехватывать все проблемные и ошибочные запросы к базе данных и передавать их обработчику вывода, который выведет их в окно командной строки.

```
LOGGING = {
  'version': 1,
  'disable_existing_loggers': False,
  'handlers': {
    'file': {
      'level': 'DEBUG',
      'class': 'logging.FileHandler',
      'filename': os.path.join(BASE_DIR, 'output.log'),
    },
    'null': {
      'level': 'DEBUG',
      'class': 'logging.NullHandler',
    },
  },
  'loggers': {
    'django.db.backends': {
      'level': 'DEBUG',
      'handlers': ['null'],
      'propagate': False,
    },
    '': {
      'level': 'DEBUG',
      'handlers': ['file'],
    },
  },
}
```

Здесь определяются два журнализатора, первый из которых будет подавлять вывод любых сообщений о запросах к базе данных, при этом блокируя их передачу журнализатору по умолчанию `django`, а второй, принадлежащий к «пустому» типу, станет выводить все сформированные разработчиком сообщения в файл output.log, который находится в папке проекта.

Эту конфигурацию журналирования мы можем использовать в своем проекте или взять ее за основу для разработки более сложной схемы журналирования.

> ***Определение журнализатора «пустого» типа***
>
> По умолчанию, любые заданные разработчиком произвольные сообщения не будут отправляться в журнал. Поэтому нам следует определить в настройках журнализатор «пустого» типа (об этом чуть далее).

Осталось лишь рассмотреть конфигурацию журналирования, установленную изначально. Она такова:

- журнализатор `django` перехватывает все информационные сообщения (класса `INFO` и выше, в том числе и сообщения об успешных интернет-запросах) и выводит их в окно командной строки;
- журнализатор `django.request` перехватывает все неуспешные интернет-запросы и отправляет соответствующие сообщения администраторам сайта по электронной почте.

Вывод в журнал произвольной информации

Ранее говорилось, что мы можем выводить в журнал произвольные данные: значения переменных и свойств, результаты, возвращенные функциями и методами, и какие-либо сообщения, например, оповещения о выполнении определенных участков кода. Такие данные перехватываются журнализатором, относящимся к «пустому» типу, который нам обязательно следует определить в настройках журналирования (см. ранее).

Настала пора выяснить, как поместить в журнал сообщение с произвольными данными.

Прежде всего, нам понадобится модуль `logging`, который мы в обязательном порядке импортируем:

```
import logging
```

Далее мы получим объект, собственно выполняющий вывод в журнал. Для этого мы вызовем функцию `getLogger`, объявленную в модуле `logging`, передав ей в качестве единственного параметра строку с полным именем модуля, к которому будет относиться весь вывод в журнал. *Системная*, т. е. создаваемая самой исполняющей средой Python переменная `__name__`, хранит строку с полным именем модуля, выполняющегося в текущий момент, — обычно ее и используют в таких случаях.

Функция `getLogger` возвращает в качестве результата то, что нам нужно, — объект, выполняющий вывод в журнал:

```
logger = logging.getLogger(__name__)
```

Для собственно выполнения вывода в журнал применяются следующие методы объекта, полученного от функции `getLogger`:

- `debug` — вывод сообщения отладочного характера;
- `info` — вывод информационного сообщения;
- `warning` — вывод предупреждения о нештатной ситуации;
- `error` — вывод предупреждения о серьезной ошибке;
- `critical` — вывод предупреждения о критической ошибке.

В качестве единственного параметра все эти методы принимают строку с содержимым выводимого в журнал сообщения:

```
logger.info("Код контроллера выполнен!")
```

Здесь мы выводим в журнал сообщение о выполнении кода контроллера.

```
a = 1
logger.debug(str(a))
```

А здесь выводим в журнал значение переменной `a`, предварительно преобразовав его в строку вызовом функции `str` (см. *главу 2*).

А теперь давайте рассмотрим более реальный пример. Откроем модуль `views` приложения `goods` и вставим в него код, который при выводе списка товаров, относящихся к какой-либо категории, будет записывать в журнал ее название. Понятно, что выражение, помещающее в журнал название категории, следует вставить в код метода `get` класса `GoodsListView` (добавленный код выделен полужирным шрифтом):

```
. . .
import logging
logger = logging.getLogger(__name__)
. . .
class GoodsListView(PageNumberView, ListView, SortMixin,↵
CategoryListMixin):
  . . .
  def get(self, request, *args, **kwargs):
    if self.kwargs["pk"] == None:
      self.cat = Category.objects.first()
    else:
      self.cat = Category.objects.get(pk = self.kwargs["pk"])
    logger.debug(self.cat.name)
    return super(GoodsListView, self).get(request, *args, **kwargs)
. . .
```

Запустим отладочный Web-сервер, зайдем на наш сайт и просмотрим списки товаров, относящихся к различным категориям. После этого завершим работу отладочного Web-сервера и откроем файл output.log, в который записывалось содержимое журнала. Там мы увидим перечень названий категорий.

> ***О СОХРАНЕНИИ ДАННЫХ В ФАЙЛЕ ЖУРНАЛА***
>
> Данные, записываемые в файл журнала, во время работы отладочного Web-сервера лишь накапливаются в оперативной памяти и реально сохраняются в файле только после того, как отладочный Web-сервер завершит работу. Поэтому перед просмотром содержимого файла журнала всегда следует завершать программу отладочного Web-сервера.

Что дальше?

В этой главе мы рассмотрели подсистему журналирования Django, которая может оказаться большим подспорьем при отладке сайта. (По крайней мере, автора книги она выручала не раз.)

В следующей главе мы займемся знакомым нам еще по *главе 4* встроенным административным сайтом Django. И узнаем, как раскрыть весь его потенциал.

ГЛАВА 31

Настройка встроенного административного сайта Django

В предыдущей главе мы рассмотрели мощный инструмент отладки Django-сайтов — подсистему журналирования. Она позволяет нам посмотреть, какими данными манипулируют приложения нашего сайта, и таким образом выяснить, правильно ли они работают.

Сайт, что мы создавали в процессе изучения этой книги, практически готов. Помимо списка товаров, гостевой книги, блога, главной и прочих страниц, он включает в свой состав средства для генерирования каналов RSS и Atom, почтовую рассылку и даже универсальное хранилище изображений, работающее по технологии AJAX. И, разумеется, в него входит административный раздел, позволяющий работать с новостями, категориями, товарами и статьями блога.

Но записи гостевой книги и списка рассылки недоступны через административный раздел нашего сайта. Для работы с ними мы планируем задействовать встроенный административный сайт, с которым познакомились еще в *главе 4*. Поскольку модерированием гостевой книги и списка рассылки, скорее всего, будет заниматься один человек — главный администратор сайта, — и этот человек достаточно технически подкован, нет большой нужды создавать для работы с ними специальные административные страницы.

Встроенный административный сайт Django — исключительно мощный инструмент, позволяющий получить доступ к любым данным сайта. Он дает нам возможность просматривать содержимое моделей, выполнять сортировку записей, добавлять, править и удалять их. Он предоставляет нам все необходимые средства для работы с данными, взамен не требуя от нас практически никакого программирования.

Но если мы владеем программированием (а мы им владеем!), то сможем без проблем настроить встроенный административный сайт под свои нужды. Мы можем изменить параметры сортировки записей на страницах списков, задать поля, по значениям которых будет выполняться фильтрация записей, пометить определенные поля как доступные только для чтения, объединить элементы управления на страницах добавления и правки записей в группы и пр. Как это сделать, показано в данной главе.

Администратор модели

Как сделать модель доступной через встроенный административный сайт, рассказывалось в *главе 5*. Нужно открыть модуль `admin`, который автоматически формируется при создании любого приложения, входящего в состав проекта, и содержит лишь следующее выражение:

```
from django.contrib import admin
```

К этому коду мы добавим вызов метода `register`, объявленного в модуле `django.contrib.admin.site`, передав ему единственным параметром класс нужной модели.

Так, чтобы сделать доступной через административный сайт модель `Guestbook` (гостевую книгу), мы откроем модуль `admin`, что входит в состав приложения `guestbook`, и добавим в него такой код:

```
from guestbook.models import Guestbook
admin.site.register(Guestbook)
```

После чего запустим отладочный Web-сервер, зайдем на административный сайт и удостоверимся в том, что гостевая книга стала через него доступна.

Но если мы хотим изменить внешний вид и поведение административного сайта при выводе содержимого какой-либо модели, нам придется добавить в модуль `admin` еще немного кода.

В первую очередь мы объявим *администратор модели* — особый класс, задающий ее представление на страницах административного сайта. Этот класс является потомком базового класса администратора `ModelAdmin`, объявленного в модуле `django.contrib.admin`. Все параметры представления записываются в качестве значений свойств этого класса:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
```

Далее мы зарегистрируем этот администратор, вызвав все тот же метод `register`. Но теперь мы передадим ему в качестве параметров, помимо модели, еще и класс администратора:

```
admin.site.register(Guestbook, GuestbookAdmin)
```

А теперь рассмотрим все возможности, предоставляемые нам Django в плане управления параметрами административного сайта и соответствующие им свойства класса администратора. Для удобства разобьем их на отдельные группы.

Настройка страниц списков записей

Начнем мы с рассмотрения возможностей по настройке внешнего вида и поведения страниц списков записей.

Настройки вывода записей

Зайдем на встроенный административный сайт и откроем страницу списка записей гостевой книги (рис. 31.1).

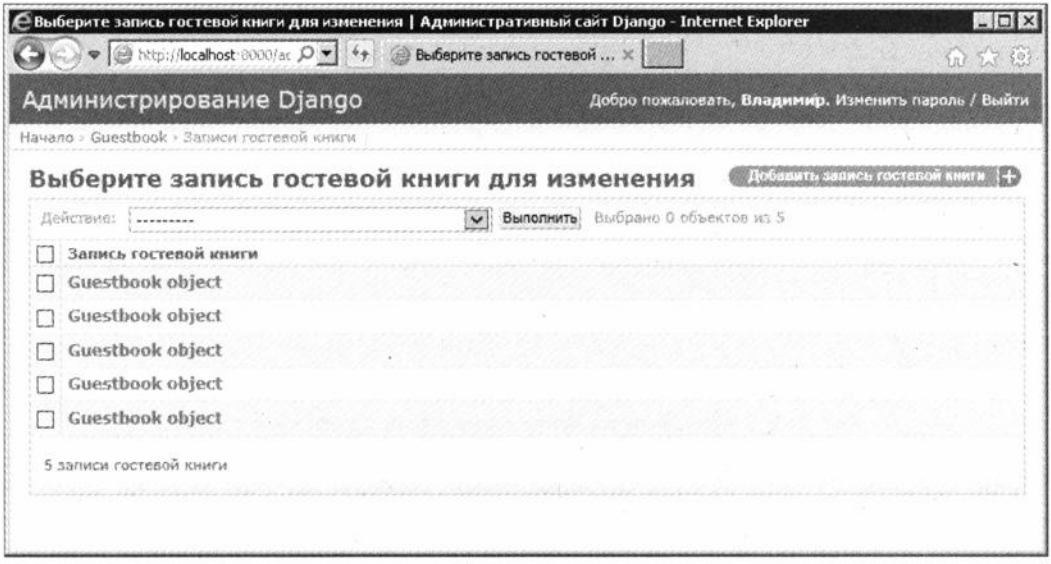


Рис. 31.1. Страница списка записей в изначальном виде

Мы видим, что все записи этой модели представлены одинаковыми строчками **Guestbook object**. И совершенно непонятно, какие данные хранит та или иная запись...

Дело в том, что по умолчанию на страницах списка выводятся строковые представления записей, генерируемые методом `__str__` модели (см. *главу 5*). Этот метод в изначальном виде возвращает для каждой записи строку вида `<имя класса модели> object`. Каковые строки мы и видим на рис. 31.1.

Конечно, мы можем переопределить в классе модели метод `__str__`, чтобы он возвращал строку, содержащую какие-либо реальные данные, что хранятся в записи. Но есть способ лучше.

Свойство `list_display` класса `ModelAdmin` задает список (обычно используют кортеж) полей, которые должны выводиться на странице списка записей. Каждый элемент этого списка должен представлять собой имя поля, заданное в виде строки. Поля выводятся в том порядке, в каком они перечислены в этом списке:

```
class GuestbookAdmin(admin.ModelAdmin):
  list_display = ("posted", "user", "content")
admin.site.register(Guestbook, GuestbookAdmin)
```

Здесь мы выводим в списке записей все поля модели `Guestbook`. Результат показан на рис. 31.2. Мы видим таблицу, каждый столбец которой содержит значения одного из указанных нами полей.

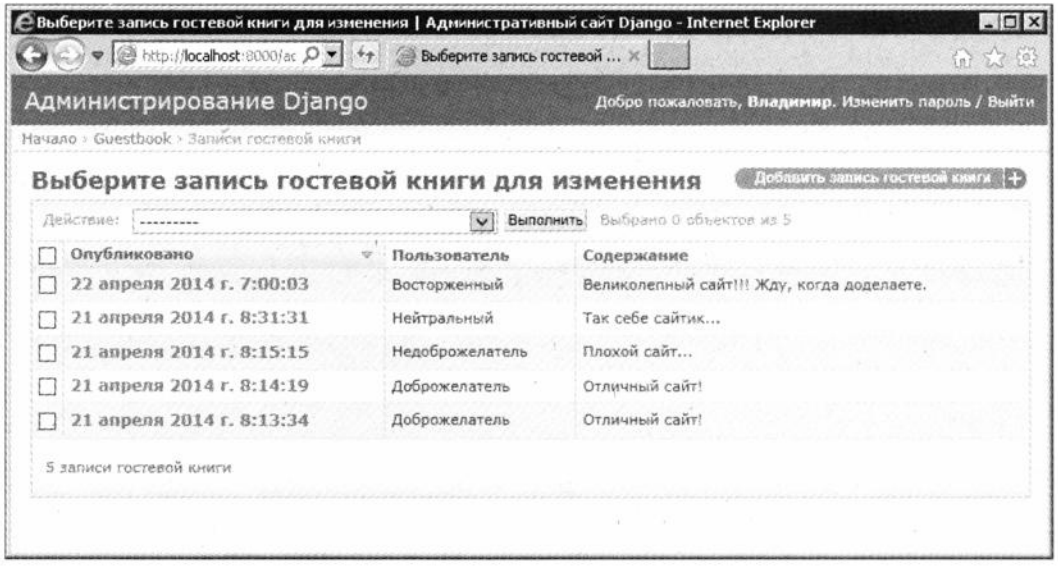


Рис. 31.2. Страница списка записей, выводящая содержимое полей

В списке свойства `list_display` мы можем использовать методы модели, задав их так же, как и имена полей:

```
class Guestbook(models.Model):
  . . .
  def get_formatted_datetime(self):
    return str(self.posted.day) + "." + str(self.posted.month) + "." +
    str(self.posted.year) + " " + str(self.posted.hour) + ":" +
    str(self.posted.minute) + ":" + str(self.posted.second)
  get_formatted_datetime.short_description = "Опубликовано"
```

Здесь мы объявили в модели `Guestbook` метод `get_formatted_datetime`, который возвращает строку с датой, взятой из поля `posted` и оформленной согласно формату *<число>.<месяц>.<год> <часы>:<минуты>:<секунды>*. Для извлечения числа, номера месяца, года, часов, минут и секунд мы использовали свойства `day`, `month`, `year`, `hour`, `minute` и `second` соответственно — все эти свойства поддерживаются классом `datetime`, хранящим значения даты и времени.

Все типы данных, поддерживаемые Python, можно рассматривать как объекты — об этом мы говорили еще в *главе 2*. Это касается и функций. Свойство `short_description` класса, представляющего функцию, задает для нее надпись, которая, в том числе, будет использована в качестве заголовка столбца на странице списка записей:

```
class GuestbookAdmin(admin.ModelAdmin):
  list_display = ("get_formatted_datetime", "user", "content")
```

Здесь мы указываем только что объявленный метод `get_formatted_datetime` в составе списка полей.

> ***О невозможности выполнения сортировки...***
>
> По столбцам, выводящим значения, которые возвращаются методами модели, нельзя выполнить сортировку.

Чтобы попасть на страницу правки записи, мы должны щелкнуть на гиперссылке, формируемой на основе содержимого одного из столбцов списка. По умолчанию это первый столбец, в нашем случае — **Опубликовано**. Однако мы можем указать для формирования такой гиперссылки любой другой столбец или даже несколько столбцов.

Список полей, на основе которых будут созданы гиперссылки, ведущие на страницы правки записей, задается в свойстве `list_display_links`. Этот список указывается в том же формате, что и список полей для вывода в свойстве `list_display`. При этом поля, перечисленные в списке свойства `list_display_links`, обязательно должны присутствовать и в списке свойства `list_display`:

```
class GuestbookAdmin(admin.ModelAdmin):
  list_display = ("posted", "user", "content")
  list_display_links = ("posted", "user")
```

Теперь, чтобы исправить запись, мы можем щелкнуть либо на гиперссылке в столбце **Опубликовано**, либо на гиперссылке, что находится в столбце **Пользователь**.

На страницах списков записей используется пагинация, причем по умолчанию на одной странице выводятся целых 100 записей. Мы можем указать другое количество записей, которое может присутствовать на одной странице, занеся его в свойство `list_per_page`:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  list_per_page = 10
```

Теперь на одной странице выводятся 10 записей гостевой книги.

Если общее количество записей в модели меньше или равно 200, рядом с гиперссылками пагинации будет выведена гиперссылка **Показать все**. Щелкнув на ней, мы выведем на страницу все записи, что хранятся в модели.

Мы можем указать другое значение предельного количества записей, при котором на экран будет выводиться гиперссылка **Показать все**. Это значение присваивается свойству `list_max_show_all`:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  list_per_page = 10
  list_max_show_all = 50
```

Теперь возможность вывода всего содержимого модели будет доступна только в том случае, если в модели `Guestbook` присутствует не более 50 записей.

Настройки фильтрации и сортировки записей

Встроенный административный сайт Django позволяет выполнять фильтрацию записей по введенному ключевому слову и сортировать записи щелчками на заголовках столбцов. И если сортировка доступна нам изначально, то чтобы получить возможность фильтровать записи, нам придется немного потрудиться.

Фильтрация записей активизируется указанием в свойстве `search_fields` списка полей, по значениям которых нужно выполнить фильтрацию. Список этот задается уже знакомым нам образом:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  search_fields = ("user", "content")
```

Так мы активизируем фильтрацию записей по полям `user` и `content`.

Если мы теперь откроем страницу списка записей, то увидим, что над списком появится поле ввода, где указывается искомое ключевое слово, и кнопка **Найти**. После нажатия этой кнопки в списке будут выведены лишь записи, в содержимом которой встретилось введенное ключевое слово (рис. 31.3).

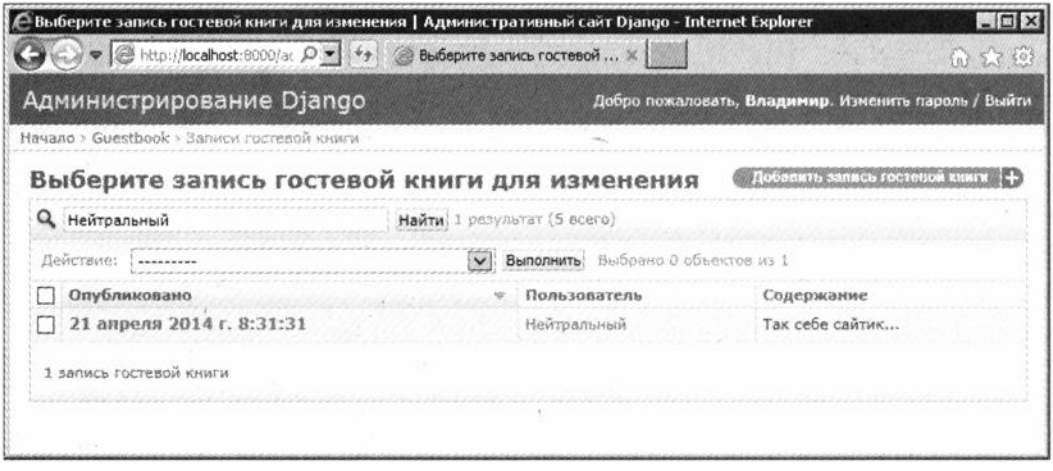


Рис. 31.3. Страница списка записей с активизированной возможностью фильтрации

Если модель включает поля типа даты или даты и времени, мы можем вывести над списком записей гиперссылки, соответствующие значениям даты, которые хранятся в этой модели. Щелкнув на такой гиперссылке, мы отфильтруем лишь те записи, что хранят это значение даты.

Чтобы задействовать такую возможность, нужно лишь указать строку с именем поля, по которому будет выполняться фильтрация подобного рода, в свойстве `date_hierarchy`:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  date_hierarchy = "posted"
```

Теперь мы можем быстро фильтровать записи гостевой книги по значениям даты, хранящимся в поле `posted`. Страница списка записей с активизированной возможностью быстрой фильтрации по дате показана на рис. 31.4.

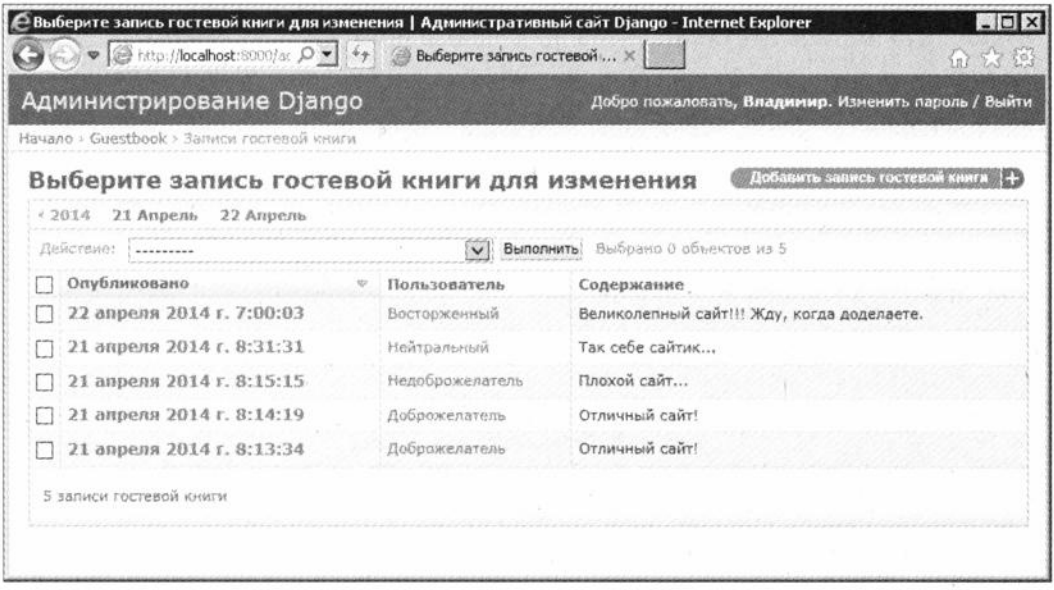


Рис. 31.4. Страница списка записей с активизированной возможностью быстрой фильтрации по дате

Мы также можем настроить быструю фильтрацию записей по значениям, хранящимся в полях других типов. При этом в правой части страницы появится перечень всех значений, хранящихся в указанных нами полях, — щелкнув на каком-либо значении, мы выведем на страницу хранящие его записи.

Включить такую полезную возможность можно, присвоив список полей, по которым должна выполняться быстрая фильтрация, свойству `list_filter`:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  list_filter = ("user",)
```

Здесь мы добавляем возможность быстрой фильтрации по значениям поля `user`. Страница, предоставляющая эту возможность, показана на рис. 31.5.

После добавления, правки или удаления записи мы обнаружим, что на странице списка, куда мы были перенаправлены, все записи отфильтрованы указанным нами ранее образом. Однако мы можем сделать так, чтобы Django в таких случаях выводила на страницы все записи, отменяя их фильтрацию. Для этого достаточно присвоить свойству `preserve_filters` значение `False`:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  list_filter = ("user",)
  preserve_filters = False
```

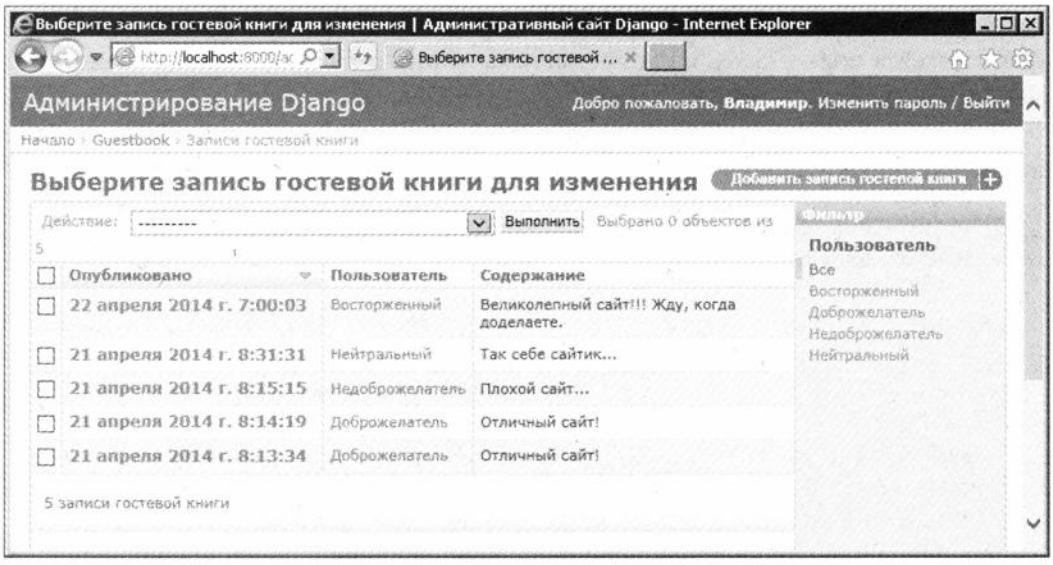


Рис. 31.5. Страница списка записей с возможностью быстрой фильтрации по значениям указываемых полей

Теперь после добавления, правки или удаления записи гостевой книги заданная ранее фильтрация будет сбрасываться.

Изначально записи на странице списка упорядочиваются согласно указанным в модели параметрам сортировки (свойство `ordering` вложенного класса `Meta`. За подробностями — к *главе 5*). Но мы можем указать для административного сайта свой порядок их сортировки. Для этого нужно составить список полей, по которым должна выполняться сортировка, в том же формате, что и для свойства `ordering` класса `Meta`, и присвоить его одноименному свойству класса администратора модели:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  ordering = ("user", "posted")
```

Теперь записи на страницах встроенного административного сайта сортируются сначала по имени пользователя, а потом — по дате добавления.

Настройки правки записей

Да-да, страница списка записей предоставляет возможность их правки! Хоть реализована она и не очень удобно, но может быть полезна, если потребуется быстро внести изменения в значения определенных полей.

Список полей, которые можно будет править прямо на странице списка записей, присваивается свойству `list_editable`:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
```

```
list_per_page = 2
list_editable = ("content",)
```

Здесь мы даем пользователю возможность править содержимое поля `content`. (Заодно уменьшаем количество записей, выводящихся на одной странице, до двух — для удобства.)

> ***О ЗАПИСЯХ, ПЕРЕЧИСЛЕННЫХ В СПИСКЕ СВОЙСТВА `LIST_EDITABLE`***
>
> Записи, перечисленные в списке свойства `list_editable`, не должны присутствовать в списке свойства `list_display_links`.

Если мы посмотрим после этого на страницу списка записей (рис. 31.6), то увидим, что в соответствующем столбце выводится элемент управления, с помощью которого и выполняется правка значения поля (в нашем случае это область редактирования). А в самом низу, под собственно списком, мы увидим кнопку **Сохранить**, нажав на которую сможем сохранить все сделанные изменения.

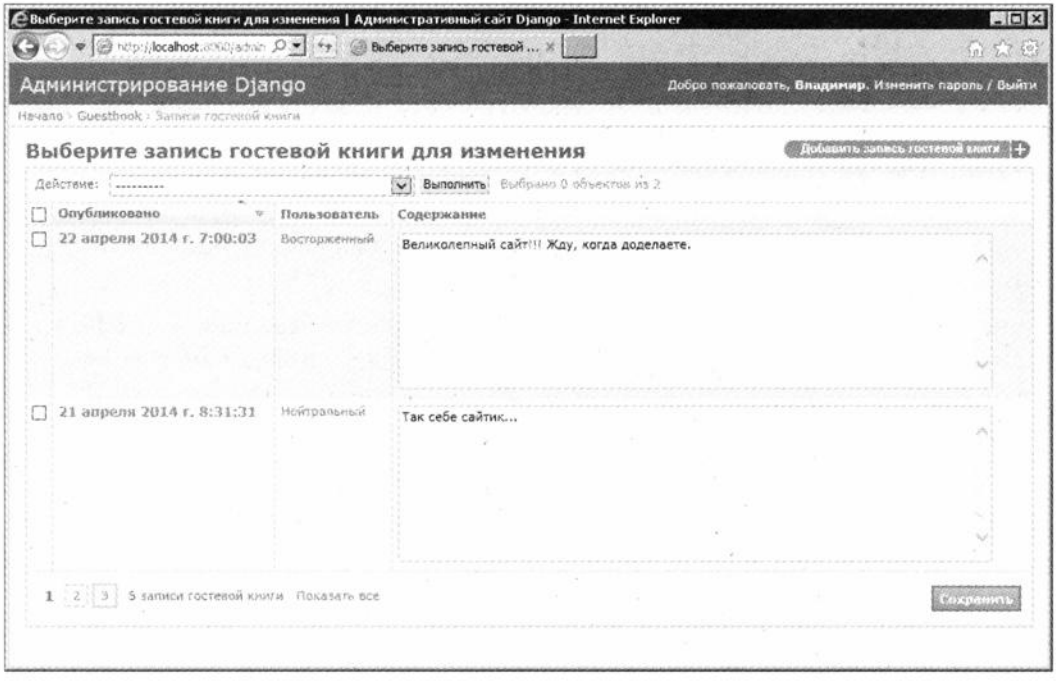


Рис. 31.6. Страница списка записей с возможностью правки

На каждой странице списка записей имеется раскрывающийся список **Действие** и кнопка **Выполнить**. Они нам уже знакомы по *главе 4* — мы пользовались ими для удаления записей.

По умолчанию эти элементы управления выводятся в верхней части страницы, над списком записей. Однако если мы присвоим свойству `actions_on_bottom` значение `True`, а свойству `actions_on_top` — значение `False`, они будут размещены внизу страницы:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  actions_on_top = False
  actions_on_bottom = True
```

Правда, непонятна практическая польза от такой перемены расположения, ведь для того, чтобы добраться до этих элементов, придется прокручивать страницу в самый низ.

Настройка страниц добавления и правки записей

Теперь поговорим о том, какие возможности Django дает нам в плане настройки страниц добавления и правки записей.

Настройка выводимых полей

По умолчанию Django выводит на страницы добавления и правки записи элементы управления для всех полей, присутствующих в модели, за исключением тех, что мы явно пометили как нередактируемые (параметр `editable` — подробнее см. *главу 5*) и автоматически заполняемые при добавлении и сохранении записи (параметры `auto_now` и `auto_now_add`). Однако допускается вывести на эти страницы лишь избранные поля — что может пригодиться, если значения каких-либо полей не вводятся пользователем, а формируются программно.

Во-первых, мы можем указать список выводимых полей в свойстве `fields`. Каждый элемент этого списка должен представлять собой строку с именем поля:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  fields = ("content",)
```

Здесь мы выводим на страницы добавления и правки только поле `content`.

Однако элементом списка свойства `fields` может быть и вложенный список, также хранящий строки с именами полей. В таком случае поля, перечисленные во вложенном списке, на странице будут выведены не друг под другом, как обычно, а по горизонтали, в одну строку:

```
from goods.models import Good
class GoodAdmin(admin.ModelAdmin):
  fields = (("name", "category"), "description", "content",↵
  ("price", "price_acc"), ("in_stock", "featured"), "image")
```

Здесь мы объявляем в модуле `admin` приложения `goods` класс-администратор для модели `Good` и выводим на экран все поля этой модели, указав, что поля `name` и `category`, `price` и `price_acc`, `in_stock` и `featured` должны быть выведены попарно в одну строку (рис. 31.7).

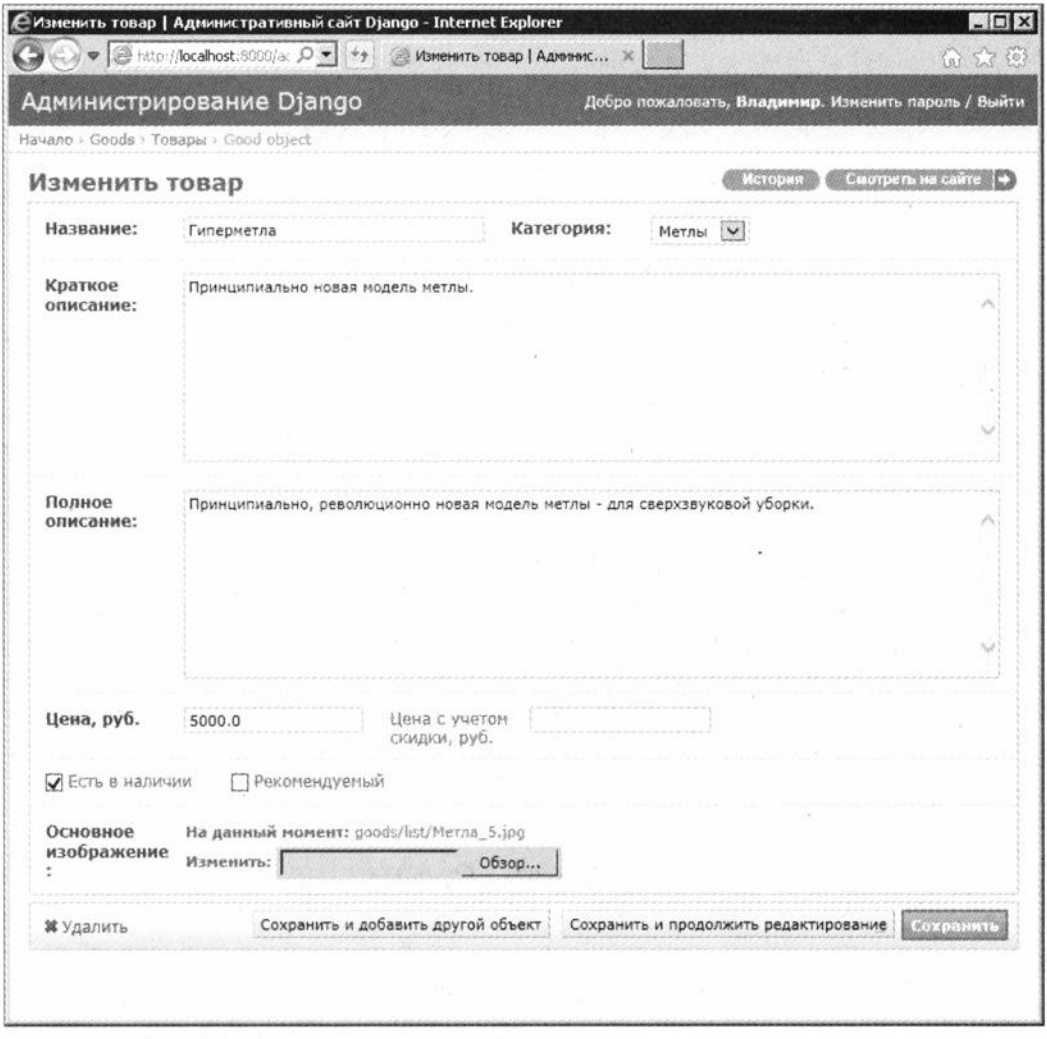


Рис. 31.7. Страница правки записей с полями, выведенными в одну строку

Во-вторых, мы можем указать поля, которые, наоборот, не должны выводиться на экран, присвоив их список свойству `exclude`:

```
class GuestbookAdmin(admin.ModelAdmin):
  . . .
  exclude = ("user",)
```

Так мы убираем с экрана поле `user` модели `Guestbook`.

По умолчанию все выводимые на экран поля доступны для ввода. Однако мы можем пометить некоторые из них как нередактируемые, задав их список в свойстве `readonly_fields`. При этом нередактируемые поля будут выводиться не в виде элемента управления, а как обычный текст:

```
class GoodAdmin(admin.ModelAdmin):
  . . .
  readonly_fields = ("category",)
```

Так мы делаем поле `category` модели `Good` нередактируемым.

Группировка полей

Мы также можем объединить определенные поля в логические группы. Такая группа будет включать заголовок, кратко описывающий ее, текст дополнительного пояснения, выводящегося под заголовком, и может быть свернута и развернута щелчком на заголовке.

Список групп, в которые объединяются поля, задается в поле `fieldsets`. Каждый его элемент указывает параметры одной группы и также представляет собой список из двух элементов.

Первый элемент списка второго уровня вложенности указывает заголовок группы, заданный в виде строки. Если группа не должна иметь заголовка, первым элементом списка следует указать значение `None`.

Второй элемент списка задает параметры создаваемой группы элементов управления и должен представлять собой словарь. Ключи его элементов должны совпадать с именами параметров создаваемых групп, а значения собственно и задают эти параметры.

Вот доступные нам параметры групп элементов управления:

- `fields` — список полей, входящих в группу. Он задается в том же формате, что применяется для указания списка полей в свойстве `fields` (см. ранее). Это единственный обязательный параметр;
- `classes` — список имен стилевых классов из собственной таблицы стилей административного сайта, которые будут привязаны к блоку, формирующему группу. Каждое имя стилевого класса должно представлять собой строку. Доступны два стилевых класса: `collapse` (группа будет изначально свернута) и `wide` (группа увеличенной ширины);
- `description` — строка с текстом дополнительного пояснения.

> ***Пояснение к свойству `fields`***
>
> Если мы указали в классе администратора модели список групп, то перечислять выводимые на экран поля в свойстве `fields` не нужно.

Вот пример:

```
class GoodAdmin(admin.ModelAdmin):
  fieldsets = (
    (None, {
      "fields": (("name", "category"),)
    }),
```

```
    ("Описание", {
      "fields": ("description", "content")
    }),
    ("Цена", {
      "fields": (("price", "price_acc"),)
    }),
    ("Дополнительные параметры", {
      "classes": ("collapse",),
      "fields": (("in_stock", "featured"),)
    }),
    ("Изображение", {
      "fields": ("image",)
    }),
)
```

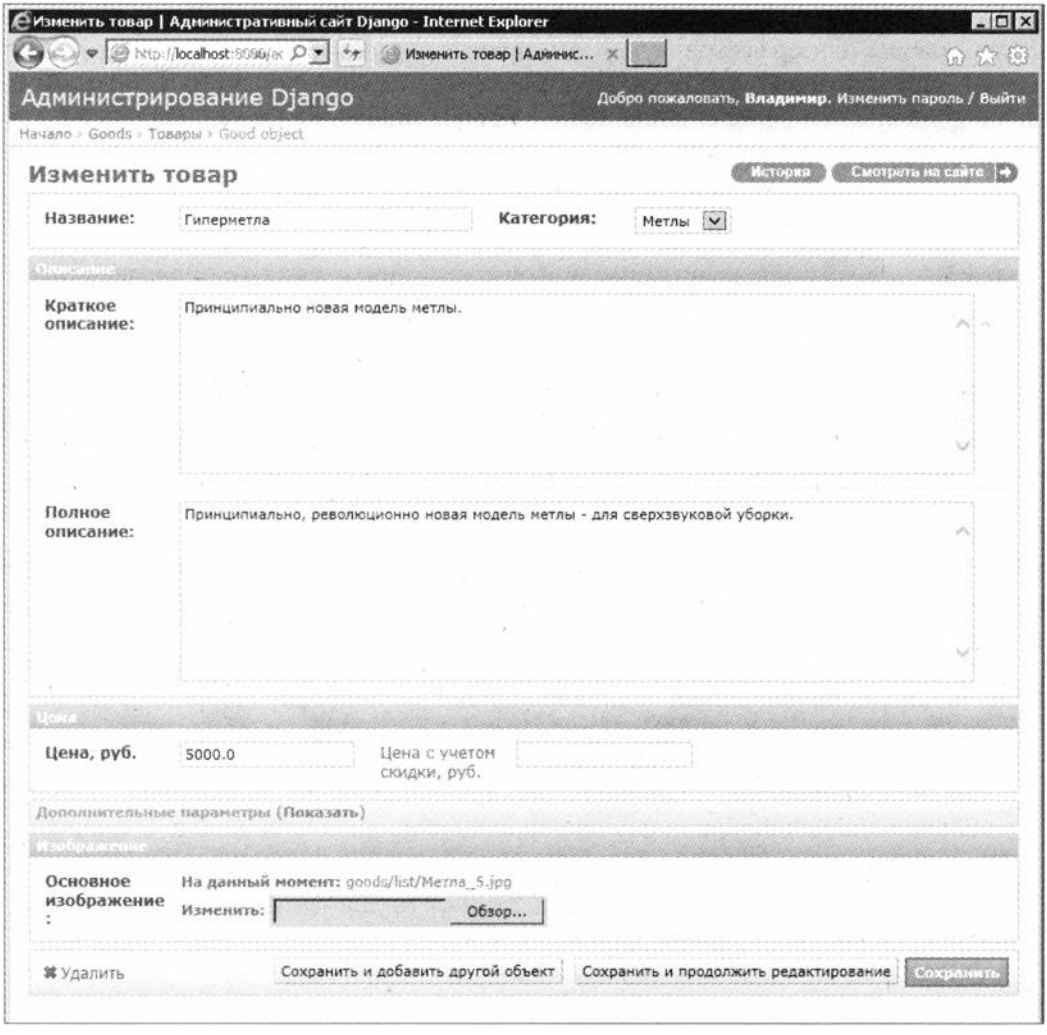


Рис. 31.8. Страница правки записи с группами элементов управления

Здесь мы формируем на страницах добавления и правки товаров пять групп:

- с полями `name` и `category`, выведенными в одну строку, без заголовка;
- с полями `description` и `content` и заголовком **Описание**;
- с полями `price` и `price_acc`, выведенными в одну строку, и заголовком **Цены**;
- с полями `in_stock` и `featured`, выведенными в одну строку, и заголовком **Дополнительные параметры**, изначально свернутую;
- с полем `image` и заголовком **Изображение**.

Страница, которую мы увидим на экране, показана на рис. 31.8.

Вывод связанных записей

Еще одна примечательная возможность, предлагаемая Django, — вывод на странице добавления или правки записи форм, предназначенных для работы со связанными записями, которые хранятся во вторичной модели. Для этого используются вложенные наборы форм, с которыми мы познакомились в *главе 12*.

Чтобы вывести на страницу такой набор форм, нам следует сначала объявить класс *вложенного администратора модели*, который, собственно, и будет его создавать. Этот класс должен быть потомком одного из двух классов:

- `StackedInline` — вложенный набор форм, в которых элементы управления располагаются один за другим по вертикали. Он подойдет в большинстве случаев;
- `TabularInline` — вложенный набор форм, организованный в виде таблицы, каждая строка которой соответствует одной записи вторичной модели. Такой вложенный администратор будет уместнее, если вторичная модель содержит небольшое количество полей. В противном случае пользователю, чтобы просмотреть все поля, придется прокручивать страницу по горизонтали, что неудобно.

Классы вложенных администраторов поддерживают свойства, позволяющие задавать различные параметры вложенного набора форм (табл. 31.1).

Таблица 31.1. Свойства классов — вложенных администраторов моделей

Свойство	Описание
`model`	Задает вторичную модель. Единственное обязательное поле
`max_num`	Задает максимальное количество форм в наборе. Значение по умолчанию — `None`
`extra`	Задает максимальное количество выводимых пустых форм для создания новых записей. Значение по умолчанию — 1
`can_delete`	Если `True`, пользователь получит возможность удалять записи. Значение по умолчанию — `True`
`verbose_name`	Задает надпись для формы. Если не указано, будет использовано значение одноименного свойства вложенного класса `Meta` модели

Таблица 31.1 *(окончание)*

Свойство	Описание
`verbose_name_plural`	Задает надпись для всего набора форм. Если не указано, будет использовано значение одноименного свойства вложенного класса `Meta` модели

Помимо этого, классы `StackedInline` и `TabularInline` поддерживают уже знакомые нам свойства `fields`, `exclude`, `fieldsets`, `readonly_fields`, `ordering`, а также свойство `radio_fields`, о котором мы скоро поговорим.

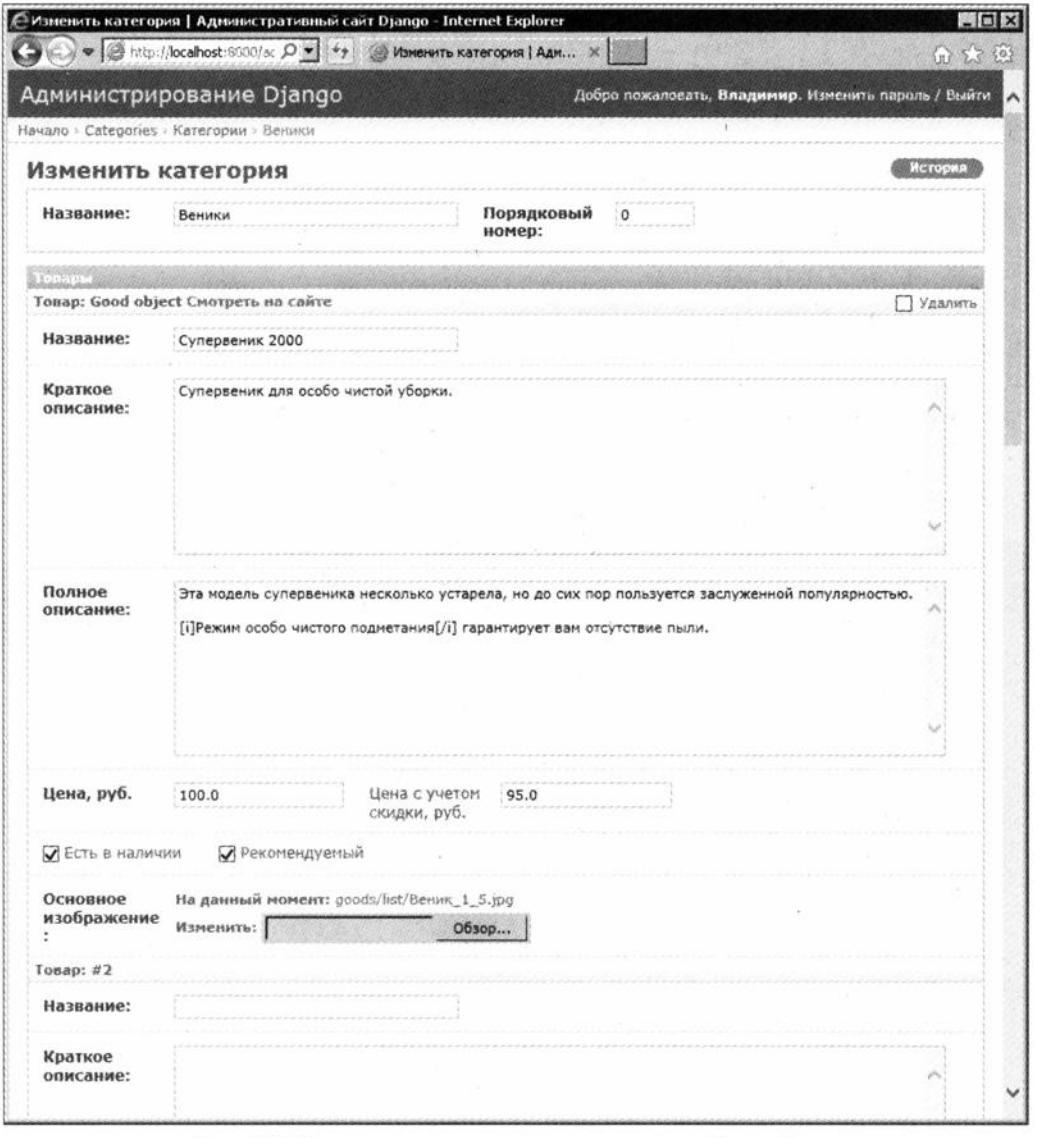


Рис. 31.9. Страница правки записи с вложенным набором форм (использовался класс-потомок `StackedInline`)

Вот пример:

```
from goods.models import Good
class GoodInline(admin.StackedInline):
  model = Good
  fields = ("name", "description", "content", ("price", "price_acc"),↵
  ("in_stock", "featured"), "image")
```

Здесь мы объявляем в модуле `admin` приложения `categories` вложенный администратор модели `Good` и указываем в нем, помимо модели, список выводимых полей.

Осталось создать класс-администратор для первичной модели и указать в нем все объявленные ранее вложенные администраторы, перечислив их в списке, присвоенном свойству `inlines`:

```
class CategoryAdmin(admin.ModelAdmin):
  fields = (("name", "order"),)
  inlines = (GoodInline,)
```

При этом не забудем его зарегистрировать:

```
from categories.models import Category
admin.site.register(Category, CategoryAdmin)
```

Если теперь мы перейдем на страницу правки любой категории, то увидим под формой, предназначенной для ввода сведений о собственно категории, набор форм, где указываются относящиеся к данной категории товары (рис. 31.9).

А вот так выглядит набор форм, созданный вложенным администратором-потомком класса `TabularInline` (рис. 31.10). Как видим, для нашего случая он не

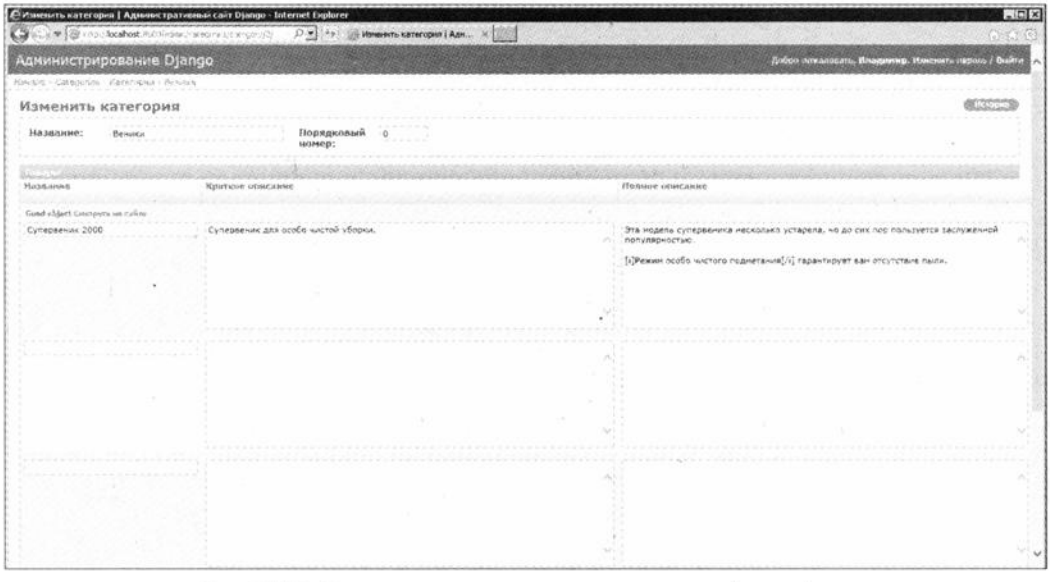


Рис. 31.10. Страница правки записи с вложенным набором форм (использовался класс-потомок `TabularInline`)

подходит, поскольку пользователю в процессе работы с записями вторичной модели (товарами) придется постоянно прокручивать страницу по горизонтали.

В обоих случаях в верхнем левом углу каждой формы вложенного набора будет присутствовать флажок **Удалить**. Если его установить, то при сохранении введенных данных помеченная им запись будет удалена.

Прочие настройки

По умолчанию для каждого поля, создающего межтабличную связь (класс `ForeignKey`, подробнее о таких полях говорилось в *главе 5*) или имеющего предопределенный набор допустимых значений (параметр `choices`, см. там же), Django выводит на страницу раскрывающийся список. Однако мы можем указать, чтобы для определенных полей вместо списков выводились группы переключателей. И поможет нам в этом свойство `radio_fields`.

Значением этого поля будет словарь. Ключами его элементов станут имена полей, для которых следует указать группы переключателей в качестве выводимых элементов управления, а значениями — одна из объявленных в модуле `django.contrib.admin` следующих переменных:

- `HORIZONTAL` — переключатели в группе располагаются по горизонтали;
- `VERTICAL` — переключатели в группе располагаются по вертикали.

Вот пример:

```
class GoodAdmin(admin.ModelAdmin):
  . . .
  radio_fields = {"category": admin.VERTICAL}
```

Здесь мы задаем для поля `category` модели `Good` вывод группы переключателей (рис. 31.11).

Рис. 31.11. Группа переключателей, служащих для указания значения поля

На страницах правки записей выводятся три кнопки, выполняющие сохранение данных: **Сохранить**, **Сохранить и продолжить редактирование** и **Сохранить и добавить другой объект**. Но если мы занесем в свойство `save_as` значение `True`, то вместо последней кнопки будет выведена кнопка **Сохранить как новый объект**. При нажатии этой кнопки введенные в форму данные будут сохранены в новой записи:

```
class GoodAdmin(admin.ModelAdmin):
  . . .
  save_as = True
```

По умолчанию кнопки сохранения данных и кнопка **Удалить** выводятся в самом низу страницы, что в случае наличия в модели большого количества полей может оказаться неудобным, т. к. пользователю придется прокручивать страницу в самый низ, чтобы добраться до этих кнопок. Задав свойству `save_on_top` значение `True`, мы укажем Django вывести их в верхней части страницы:

```
Что дальше?class GoodAdmin(admin.ModelAdmin):
  . . .
  save_on_top = True
```

Что дальше?

В этой главе мы настраивали встроенный административный сайт Django. Часть внутренних данных нашего сайта (записи гостевой книги и список рассылки) будут правиться исключительно его средствами, и предоставляемые Django инструменты настройки встроенного сайта нам совсем не помешают.

Что ж, сайт фирмы «Веник-Торг» полностью закончен и готов к публикации в Интернете. Не пора ли этим заняться?

ГЛАВА 32

Публикация Web-сайта

В предыдущей главе мы занимались встроенным административным сайтом Django, настраивали его внешний вид и поведение под свои нужды. Теперь мы знаем, как сделать так, чтобы пользователю, которому предстоит работать с записями гостевой книги и списка рассылки, было это делать комфортнее.

Что ж, сайт практически готов и полностью работоспособен. Пришло время опубликовать его в Интернете, сделав тем самым доступным для всех потенциальных клиентов фирмы «Веник-Торг». Чем мы и займемся в этой, последней главе книги.

Подготовка сайта к публикации

Сначала нам следует определенным образом подготовить сайт к публикации. Процесс подготовки включает в себя удаление временных файлов, в частности, сгенерированных библиотекой easy-thumbnails миниатюр, а также файлов, которые не нужны для работы сайта, замену в списке сайтов и коде приложений локального домена **http://localhost:8000/** тем доменом, на котором будет опубликован сайт, и внесение кое-каких правок в настройки проекта.

Настоятельно рекомендуется сделать копию проекта и все манипуляции по подготовке сайта выполнять над ним и его же впоследствии опубликовать в Сети. Код изначального проекта мы сохраним на тот случай, если потом захотим что-либо в нем переделать.

Удаление временных и ненужных файлов

В процессе разработки и, в особенности, отладки в составе сайта может появиться множество файлов, которые либо не нужны для нормального его функционирования, либо являются временными хранилищами какой-либо информации. Эти файлы перед публикацией сайта рекомендуется удалить, чтобы уменьшить занимаемый им на диске объем.

Давайте вкратце рассмотрим виды временных и ненужных файлов и их назначение и решим, стоит их удалять или нет.

- **Файлы миниатюр** — они автоматически генерируются библиотекой easy-thumbnails (см. *главу 17*) и сохраняются в папке, которую мы для них указали, для последующего использования, чтобы не генерировать их каждый раз заново. Если сайт достаточно велик, а процесс его разработки был достаточно продолжителен, файлов миниатюр может оказаться очень много.

  Файлы миниатюр могут быть безболезненно удалены. Но тогда при последующем выводе библиотеке easy-thumbnails придется выполнить их генерацию, что отнимет некоторое время. Однако, если файлов миниатюр много, и занимаемый ими объем существенен, лучше их все же удалить.

- **Файлы журналов** — автор не думает, что они окажутся слишком большими, но в готовом сайте делать им совершенно нечего.

- **Неиспользуемые модули** — это, прежде всего, модули `tests`, в которых пишется код для встроенной в Django подсистемы тестирования. Даже если мы и задействовали таковую при проверке работоспособности сайта (хотя, по мнению автора, большой пользы от нее нет), в опубликованном сайте они также не нужны.

  Можно удалить и модули `admin` приложений, модели которых не должны быть доступны через встроенный административный сайт Django (об административном сайте и его настройке рассказывалось в *главе 31*), модули `models` в приложениях, не содержащих в составе модели, и модули `urls` приложений, в которых привязка интернет-адресов выполнена полностью на уровне проекта (о привязке интернет-адресов к контроллерам говорилось в *главе 6*).

- **Все прочие файлы**, по той или иной причине оказавшиеся в папке проекта, но не нужные для работы сайта: текстовые, изображения, базы данных, модули Python. Их также следует удалить.

Принцип здесь очень прост — все, что не нужно, подлежит безоговорочному удалению. Так мы заметно уменьшим объем информации, который потребуется записывать на носители или передавать по сети, возможно, по медленному каналу.

Правка кода приложений и указание целевого домена

Часто возникает необходимость перед публикацией сайта внести правки непосредственно в его код. Скажем, в шаблоне отправляемых по электронной почте уведомлений comment_notification_email.txt мы поместили указание на домен **http://localhost:8000/**, по которому работает отладочный Web-сервер Django (выделено полужирным шрифтом):

```
. . .
- страница: http://localhost:8000{{ content_object.get_absolute_url }};
. . .
```

Разумеется, вместо него следует подставить указание на домен, на котором будет опубликован наш сайт.

После отключения режима отладки (как его отключить, мы скоро узнаем) встроенный Web-сервер Django перестанет обрабатывать статичные файлы уровня проекта и выгруженные файлы. Поэтому мы сразу же можем удалить и закомментировать в модуле `urls` пакета проекта выражения, задействующие их обработку:

```
# from django.conf import settings
# from django.conf.urls.static import static
# urlpatterns += static(settings.MEDIA_URL,↪
document_root = settings.MEDIA_ROOT)
```

Помимо этого, следует внести целевой домен в позицию встроенного в Django списка сайтов, соответствующую нашему сайту. (Мы уже вносили туда правки в *главе 28*, когда проверяли, генерируются ли каналы RSS и Atom.) Для этого следует запустить отладочный Web-сервер, зайти на встроенный административный сайт, найти на его главной странице таблицу с заголовком **Sites** и щелкнуть на единственной присутствующей в ней гиперссылке **Сайты**. На странице списка сайтов нужно щелкнуть на единственном пункте этого списка — **example.com**. А когда откроется страница правки этого сайта, останется ввести имя целевого домена в поле ввода **Доменное имя** и сохранить сделанные изменения.

Внесение изменений в настройки сайта

Осталось внести кое-какие правки в настройки проекта — а именно: отключить режим отладки, указать список разрешенных доменов и, возможно, изменить параметры используемой базы данных, указать пути к папкам статичных файлов уровня проекта и выгруженных файлов. Всем этим мы сейчас и займемся.

При создании проекта Django включает в его настройках режим отладки. В этом режиме при возникновении в коде сайта ошибки будет выводиться знакомая нам страница с сообщением (см. рис. 30.1). Помимо этого, Django в режиме отладки сохраняет в памяти каждый запрос к базе данных на случай возникновения проблем и игнорирует список допустимых доменов, указанный в переменной `ALLOWED_HOSTS` (о ней будет сказано чуть позже).

Однако перед публикацией сайта настоятельно рекомендуется отключить режим отладки. Это, во-первых, заблокирует вывод страницы с сообщением об ошибке (вместо нее будет выводиться страница с сообщением об ошибке 500 протокола HTTP), во-вторых, подавит сохранение в памяти запросов к базе данных (что положительно скажется на производительности), а в-третьих, введет в действие некоторые настройки, связанные с безопасностью.

Отключить режим отладки очень просто. Достаточно найти в модуле `settings` пакета проекта переменную `DEBUG` и заменить присваиваемое ей значение с `True` на `False`:

```
DEBUG = False
```

Также можно на всякий случай дополнительно отключить вывод сообщений об ошибках в коде шаблонов, заменив присваиваемое переменной `TEMPLATE_DEBUG` значение с `True` на `False`:

```
TEMPLATE_DEBUG = False
```

Однако делать это необязательно, т. к. отключение режима отладки также отключит и вывод сообщений об ошибках шаблонов.

> ***Отключение режима отладки***
>
> Не забываем, что после отключения режима отладки встроенный Web-сервер Django перестает обрабатывать статичные и выгруженные файлы. Поэтому отладку следует отключать лишь после завершения тестирования сайта.

Далее нужно задать список допустимых доменов, с которых можно будет зайти на наш сайт. Как минимум, этот список должен содержать наш домен.

Список разрешенных доменов указывается в переменной `ALLOWED_HOSTS`. Каждый его элемент должен представлять собой строку с именем домена. Если нужно разрешить доступ к сайту не только с этого домена, но и с его поддоменов, перед именем домена следует поставить точку:

```
ALLOWED_HOSTS = ["www.veniktorg.ru"]
```

Так мы разрешаем доступ к сайту только с домена **www.veniktorg.ru**.

```
ALLOWED_HOSTS = [".veniktorg.ru"]
```

А так — с домена **veniktorg.ru** и его поддоменов: **www.veniktorg.ru**, **goods.veniktorg.ru**, **blog.veniktorg.ru** и др.

```
ALLOWED_HOSTS = ["veniktorg.ru", "веникторг.рф"]
```

Так мы разрешаем доступ к сайту с доменов **www.veniktorg.ru** и **веникторг.рф**.

> ***Обращение к сайту с домена, не указанного в списке***
>
> При попытке обращения к сайту с домена, не указанного в списке, будет сгенерировано сообщение об ошибке 400 протокола HTTP.

После этого нам, возможно, понадобится внести в настройки проекта следующие изменения:

- исправить параметры используемой базы данных — если мы перенесли файл базы в другое место или же вообще хотим использовать другую базу, может быть, другого формата. (О параметрах базы данных рассказывалось в *главе 4*, а о работе с базами данных других форматов будет сказано в конце этой главы);
- исправить пути к папкам статичных и выгруженных файлов — если перенесли их в другое место (см. *главы 8* и *13*);
- изменить параметры отправки почты — если собираемся отправлять письма через другой SMTP-сервер (см. *главу 15*);
- удалить код, задающий настройки журналирования — чтобы использовать параметры журналирования по умолчанию (см. *главу 30*);

- задать списки администраторов и модераторов сайта, которым будут отправляться уведомления об ошибках и вновь добавленных комментариях (см. *главы 15* и *30*).

И, наконец, проверим, все ли мы сделали, не упустили ли чего из виду и не удалили ли чего лишнего.

Создание страниц сообщений об ошибках

При возникновении ошибки в работе сайта Django выведет на экран соответствующую страницу с предупреждением. Коды наиболее часто встречающихся ошибок и ситуации, в которых возникают эти ошибки, приведены в табл. 32.1.

***Таблица 32.1.** Ошибки протокола HTTP*

Код ошибки	Ситуация, при которой возникает ошибка
400	Была произведена попытка получить доступ к сайту с домена, не перечисленного в списке допустимых (переменная ALLOWED_HOSTS, описанная ранее)
403	Пользователь, не имеющий необходимых прав, пытается добавить, изменить или удалить запись модели
404	Выполнен вызов несуществующего контроллера или загрузка несуществующего шаблона
500	При выполнении кода контроллера или шаблона возникла ошибка

По умолчанию страница с сообщением об ошибке будет сгенерирована на основе шаблона, встроенного в Django. Но мы можем указать для этого свои собственные шаблоны.

Файлы шаблонов страниц, сообщающих об ошибках протокола HTTP, должны иметь имена, совпадающие с кодами соответствующих ошибок (см. табл. 32.1), — так, файл шаблона страницы ошибки 404 должен иметь имя 404.html. Все эти файлы помещаются непосредственно в папку шаблонов уровня проекта.

В контексте данных шаблона 404.html будет присутствовать переменная `request_path`, хранящая интернет-адрес, по которому был выполнен запрос. Контекст данных остальных шаблонов будет пуст.

Мы можем создать шаблоны для страниц с сообщениями об ошибках, сделав их потомками написанного в *главе 20* шаблона login_logout.html. Так, код шаблона 404.html может быть таким:

```
{% extends "login_logout.html" %}
{% block title %}Ошибка 404{% endblock %}
{% block ll %}
  <h2>Ошибка 404</h2>
  <p>Страница с интернет-адресом {{ request_path }} не существует.</p>
{% endblock %}
```

На его основе мы можем создать шаблоны 400.html, 403.html и 500.html, просто заменив текст с сообщением об ошибке. В шаблон 403.html неплохо было бы поместить гиперссылку, указывающую на страницу входа, чтобы пользователь сразу смог войти на сайт, а в шаблон 500.html — гиперссылку на страницу контактов, где находятся, помимо всего прочего, контакты для связи с разработчиками сайта.

Публикация сайта

Итак, все подготовительные действия мы выполнили. Настает волнующий миг публикации сайта в Интернете и ожидания привлеченных им первых клиентов.

Мы рассмотрим процесс публикации Django-сайта на нашем собственном компьютере и на серверах стороннего хостинг-провайдера. И поскольку, как говорится, своя рубашка ближе к телу, начнем с первого случая.

Публикация сайта на нашем собственном компьютере

Как мы узнали еще в *главе 4*, встроенный отладочный Web-сервер Django использовать для публикации сайта категорически не рекомендуется. Нам придется подобрать стороннюю программу Web-сервера, и обычно в этой роли выступает популярный Web-сервер Apache.

«Домашний» сайт программного пакета Apache находится по интернет-адресу **http://projects.apache.org/projects/http_server.html**. Загрузить его можно и с популярных интернет-архивов программ — например, с **http://www.softodrom.ru/**. Установка Apache, как правило, не вызывает особых сложностей.

Нам также понадобится дополнительный модуль mod_wsgi, подключаемый к Apache и обеспечивающий работу Django-сайтов. Его «домашний» сайт расположен по интернет-адресу **http://code.google.com/p/modwsgi/** — там можно найти полную документацию по этой программе. Однако дистрибутивный комплект модуля mod_wsgi следует искать по интернет-адресу **http://www.lfd.uci.edu/~gohlke/pythonlibs/#mod_wsgi** — там представлены модификации для разных версий Apache и Python.

Дистрибутивный комплект mod_wsgi представляет собой обычный архив ZIP, в который упакован один-единственный файл mod_wsgi.so, собственно и представляющий собой программу. Его следует поместить в папку modules, находящуюся в папке, в которую был установлен Apache (по умолчанию это папка \Program Files\Apache Software Foundation\Apache<*номер версии Apache*>). В этом и заключается процесс установки модуля.

Подключим теперь только что установленный модуль mod_wsgi к Web-серверу. Для этого найдем в подпапке conf папки, где был установлен Apache, файл httpd.conf и откроем его в текстовом редакторе. В файле httpd.conf описываются параметры конфигурации Apache, в которые нам сейчас требуется внести некоторые изменения.

Редактирование файла httpd.conf

Папка *Program Files* в версиях Windows, начиная с Vista, защищена с помощью технологии UAC (User Access Control, управление доступом пользователя), и при сохранении в ней любого файла могут возникнуть проблемы. Поэтому лучше скопировать файл *httpd.conf* в любую другую папку, не защищенную UAC, исправить эту копию, а потом переместить в папку *conf*.

Отыщем в файле конфигурации группу строк, начинающихся со слова `LoadModule`, и добавим к ней следующую строку:

```
LoadModule wsgi_module modules/mod_wsgi.so
```

Эта строка укажет Apache при запуске загружать и инициализировать модуль mod_wsgi.

А теперь — внимание! В папке пакета проекта находится файл wsgi.py. Он формируется самой Django при создании любого проекта и содержит код, обеспечивающий взаимодействие Web-сервера Apache (посредством модуля mod_wsgi) и этого проекта. Удалять его не следует ни в коем случае.

Вернемся к файлу httpd.conf. Нам понадобится добавить в самый его конец довольно много строк, задающих различные параметры, и эти строки мы рассмотрим по частям.

Итак, добавляем первый фрагмент:

```
Alias <префикс для интернет-адреса выгруженных файлов, ↵
заданный в переменной MEDIA_URL модуля settings> <полный путь к папке ↵
выгруженных файлов>
Alias <префикс для интернет-адреса статичных файлов, ↵
заданный в переменной STATIC_URL модуля settings>/admin/ <папка, ↵
где был установлен Python>/Lib/site-packages/django/contrib/admin/↵
static/admin/
Alias <префикс для интернет-адреса статичных файлов, ↵
заданный в переменной STATIC_URL модуля settings> <полный путь к папке ↵
статичных файлов уровня проекта>
```

Эти строки создают виртуальные папки, ссылающиеся на физические папки:

- выгруженных файлов;
- статичных файлов встроенного административного сайта Django;
- статичных файлов нашего проекта.

Особенность написания путей к файлам и папкам

В файле конфигурации *httpd.conf* при написании путей к файлам и папкам следует использовать символ прямого слеша (/), а не обратного (\), какой обычно применяется в Windows.

Кроме того, пути к папкам, заданные в приведенных ранее строках, обязательно следует завершать символом прямого слеша. Если этого не сделать, Apache не сможет правильно обработать эти пути.

Добавляем следующий фрагмент:

```
<Directory <полный путь к папке выгруженных файлов>>
Order deny,allow
Allow from all
</Directory>
<Directory <папка, где был установлен Python>/Lib/site-packages/django/
contrib/admin/static/admin>
Order deny,allow
Allow from all
</Directory>
<Directory <полный путь к папке статичных файлов уровня проекта>>
Order deny,allow
Allow from all
</Directory>
```

Эти строки указывают Apache разрешить доступ ко всем файлам в папках выгруженных файлов и статичных файлов проекта и встроенного административного сайта Django и запрещают просмотр их содержимого в виде списка файлов.

В этих строках и также во всех последующих уже не нужно замыкать пути символом прямого слеша.

Добавляем следующий фрагмент:

```
WSGIScriptAlias / <полный путь к файлу wsgi.py, находящемуся в папке
пакета проекта>
WSGIPythonPath <полный путь к папке проекта>
```

Первая строка здесь задает файл wsgi.py в качестве «корня» сайта. Вторая папка указывает самому Python местоположение папки проекта.

Добавляем следующий фрагмент:

```
<Directory <полный путь к папке пакета проекта>>
<Files wsgi.py>
Order deny,allow
Allow from all
</Files>
</Directory>
```

Эти строки разрешают Apache доступ к файлу wsgi.py, в том числе и на запуск.

> ***Для Apache 2.4 или более поздних версий***
>
> Если используется Apache 2.4 или более поздней версии, строки:
>
> ```
> Order deny,allow
> Allow from all
> ```
>
> везде следует заменить строкой:
>
> ```
> Require all granted
> ```

В результате всех произведенных изменений у автора книги, установившего Python в папку C:\Python33, а проект, носящий имя `broomtrade`, поместившего в папку C:\Work\!Temp\, получилась следующая конфигурация:

```
Alias /media/ C:/Work/!Temp/broomtrade/uploads/
Alias /static/admin/ C:/Python33/Lib/site-packages/django/↳
contrib/admin/static/admin/
Alias /static/ C:/Work/!Temp/broomtrade/static/

<Directory C:/Work/!Temp/broomtrade/static>
Order deny,allow
Allow from all
</Directory>

<Directory C:/Python33/Lib/site-packages/django/contrib/admin/↳
static/admin>
Order deny,allow
Allow from all
</Directory>

<Directory C:/Work/!Temp/broomtrade/uploads>
Order deny,allow
Allow from all
</Directory>

WSGIScriptAlias / C:/Work/!Temp/broomtrade/broomtrade/wsgi.py
WSGIPythonPath C:/Work/!Temp/broomtrade

<Directory C:/Work/!Temp/broomtrade/broomtrade>
<Files wsgi.py>
Order deny,allow
Allow from all
</Files>
</Directory>
```

Приведенный здесь код можно использовать как основу для написания своих конфигураций.

Внеся все указанные правки, сохраним файл httpd.conf и запустим Apache. Попытаемся войти на сайт под целевым доменом. Мы также можем попробовать обратиться к сайту через локальный домен **http://localhost/** (или **http://localhost:8080/**, если мы выполнили установку Apache для использования в режиме разработки), однако тогда сначала придется внести этот домен в список разрешенных (описанная ранее переменная `ALLOWED_HOSTS`, объявленная в модуле `settings`).

Если все настройки были введены правильно, сайт должен работать. И тогда мы можем поздравить себя с первым успешно написанным и опубликованным в Интернете Django-сайтом.

Публикация сайта на сервере стороннего хостинг-провайдера

Если же мы собираемся опубликовать сайт на серверах стороннего хостинг-провайдера, нам следует выяснить следующее:

- установлен ли на целевом сервере программный пакет языка Python той же версии, на которой мы вели разработку сайта;
- установлена ли на целевом сервере библиотека Django той же версии;
- установлены ли на целевом сервере остальные библиотеки, перечисленные в *приложении 1* к книге, и те ли это их версии.

Если программный пакет Python или какая-либо из библиотек не установлена или установлена не та ее версия, возможно, нам придется переделывать под нее программный код сайта. А это может быть весьма длительным и трудоемким процессом.

Процесс публикации сайта на стороннем сервере описывается в документации, размещенной на сайте хостинг-провайдера, которому принадлежит этот сервер.

Использование баз данных других форматов

В процессе разработки сайта мы использовали для хранения его внутренних данных базу формата SQLite. Программа СУБД, работающая с базами этого формата, поставляется в составе Python, имеет вполне приемлемую производительность и богатые возможности и, к тому же, для нормальной работы не требует установки никаких дополнительных библиотек.

Однако для использования в уже опубликованных и, в особенности, тяжелонагруженных сайтах этот формат годится не очень. Встроенная СУБД SQLite написана на языке Python, а последний, как и все интерпретируемые языки программирования, не может похвастаться высоким быстродействием. К тому же, он не предусматривает никаких средств разграничения доступа — к базе SQLite может подключиться кто угодно и делать в ней что угодно. А это открывает широкий простор для деятельности злоумышленников.

Поэтому часто для хранения указанных сайтов применяют другие форматы баз, которые обрабатываются внешними по отношению к Python СУБД. Обычно это популярные пакеты MySQL и PostgreSQL — мощные серверные СУБД, реализующие отказоустойчивое и безопасное хранение данных с возможностью разграничения доступа и предоставляющие дополнительные возможности в плане работы с данными, такие как хранимые процедуры. Поскольку эти пакеты поставляются в откомпилированном в машинный код виде и работают отдельно от Python, их быстродействие много выше, чем у SQLite.

Работа с базами данных этих форматов выполняется теми же средствами, что мы изучили в *главе 5* и применяли на протяжении всей книги. В этом большая заслуга Django, предоставляющей для работы с данными стандартизированные механизмы

и скрывающей от нас, программистов, детали реализации конкретной СУБД и конкретного формата баз данных.

Использование баз данных MySQL

Бесплатную версию серверной СУБД MySQL можно загрузить с сайта **http://dev.mysql.com/**. Установка ее обычно не вызывает никаких проблем, поэтому описываться здесь не будет. (Если же при установке все же возникнут проблемы, можно обратиться к документации по этой СУБД, расположенной на том же сайте.)

> ***Хранение файлов баз данных***
>
> Для хранения файлов самих баз данных следует указывать папку, расположенную по пути, который не защищен системой UAC, т. е. не в папке *Program Files* и не в папке, где установлена Windows. Автор рекомендует помещать эту папку в корне диска.

В составе MySQL поставляется библиотека MySQL Connector/Django, посредством которой Django может взаимодействовать с этим сервером. По умолчанию она устанавливается на компьютер вместе с собственно сервером и вспомогательными программами. Нам остается лишь ее использовать.

> ***Сторонняя библиотека MySQLdb***
>
> Документация по Django рекомендует применять для взаимодействия с сервером данных MySQL стороннюю библиотеку MySQLdb. Однако последняя на текущий момент версия этой библиотеки предназначена для работы под Python 2.7, и когда выйдет новая версия, поддерживающая Python 3, неизвестно.

Первое, что нам необходимо сделать, — создать базу данных для сайта, поскольку Django этого за нас не сделает. Нам также следует создать нового пользователя, от имени которого Django будет подключаться к этой базе, — новый пользователь должен иметь полные права на работу с созданной нами базой данных, в том числе права на создание таблиц. Выполнить все это можно в среде административной утилиты MySQL Workbench, поставляемой в составе MySQL.

Создав базу данных и пользователя, мы можем записать в настройки проекта сведения о вновь созданной базе и подключении к ней. Они указываются в модуле `settings` пакета проекта, в переменной `DATABASES`, описанной в *главе 4*.

Здесь нам понадобятся параметры базы данных, перечисленные далее. Их значения всегда указываются в виде строк:

- `ENGINE` — в качестве значения этого параметра мы укажем пакет `mysql.connector.django`, входящий в состав упомянутой ранее библиотеки MySQL Connector/Django;
- `NAME` — имя базы, где будут храниться данные сайта;
- `USER` — имя пользователя, под которым будет выполняться подключение к базе;
- `PASSWORD` — пароль для этого пользователя;
- `HOST` — интернет-адрес сервера. Если MySQL установлен на том же компьютере, на котором размещается сайт, следует указать интернет-адрес **localhost** или **127.0.0.1**;

> ***Интернет-адрес для параметра HOST***
>
> В параметре HOST следует указать тот же интернет-адрес, что был задан в параметрах пользователя, от имени которого Django будет подключаться к базе. Так, если в параметрах пользователя был указан интернет-адрес **127.0.0.1**, то и в параметре HOST следует указать тот же адрес. В противном случае сервер MySQL может отказать в подключении к базе.

- PORT — номер порта, через который осуществляется взаимодействие с сервером. Указывается только в том случае, если сервер данных работает через порт, отличный от используемого по умолчанию.

Вот пример:

```
DATABASES = {
  'default': {
    'ENGINE': 'mysql.connector.django',
    'NAME': 'site',
    'USER': 'site',
    'PASSWORD': 'site',
    'HOST': '127.0.0.1',
  }
}
```

Здесь мы указываем базу данных site, обрабатываемую сервером, который располагается на том же компьютере, где развернут сайт, и задаем для подключения к ней пользователя site с паролем site.

Внеся в модуль settings параметры новой базы, проверим, работает ли в настоящий момент сам сервер MySQL, после чего выполним синхронизацию с базой данных. Затем мы можем начать работу с сайтом, занося в него все необходимые для работы данные: категории, товары, новости и статьи блога.

Использование баз данных PostgreSQL

Дистрибутивный комплект бесплатного сервера данных PostgreSQL можно загрузить с сайта, расположенного по интернет-адресу **http://www.postgresql.org/**. Его установка также не вызывает проблем, так что мы не будем ее рассматривать.

> ***Не вторгайтесь на пути, защищенные системой UAC***
>
> И в этом случае папка, где будут храниться сами файлы базы данных, не должна находиться по пути, защищенном системой UAC.

Загрузим и установим библиотеку Psycopg, служащую интерфейсом между Django и PostgreSQL. Ее дистрибутивный комплект находится по интернет-адресу **http://initd.org/psycopg/**.

Подготовка сайта к работе с базой данных PostgreSQL выполняется так же, как и в случае базы MySQL. Мы создаем базу данных, пользователя, от имени которого будем к ней подключаться, и который имеет полные права на работу с этой базой, вносим правки в модуль settings пакета проекта и выполняем синхронизацию

с базой. Для работы с базами данных и пользователями удобно использовать поставляющуюся в составе PostgreSQL утилиту pgAdmin III.

Единственное отличие будет заключаться в значении параметра `ENGINE` конфигурации базы данных. В нашем случае мы должны присвоить ему строку с именем пакета `django.db.backends.postgresql_psycopg2` — с его помощью Django и «общается» с базами данных формата PostgreSQL:

```
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.postgresql_psycopg2',
    'NAME': 'site',
    'USER': 'site',
    'PASSWORD': 'site',
    'HOST': '127.0.0.1',
  }
}
```

Этим мы перепишем представленную ранее конфигурацию для работы с базой PostgreSQL.

> ***Работу с базами данных других форматов***
>
> Django также поддерживает работу с базами данных форматов Oracle, Microsoft SQL Server, Firebird, IBM DB2, SAP SQL Anywhere, а также использование универсальных механизмов доступа к данными Microsoft ODBC и ADSDB. Как настроить проект на работу с такими базами, и какие сторонние библиотеки для этого применяются, описано в документации по библиотеке, расположенной на «домашнем» сайте Django.

На этом позвольте закончить книгу, посвященную разработке Web-сайтов на языке Python с применением библиотеки Django. Автор прощается с вами, уважаемые читатели. Удачи в работе!

Заключение

Вот и закончена книга, рассказывающая о разработке Web-сайтов на языке программирования Python с применением библиотеки Django. Мы изучили все, что необходимо для создания сайтов общего назначения, и даже сделали в качестве практического занятия полнофункциональный сайт гипотетической фирмы «Веник-Торг». И теперь можем с уверенностью и гордостью именовать себя настоящими программистами!

Чем мы займемся дальше? Разработкой других, уже реально действующих сайтов — это безусловно. А также более углубленным изучением языка Python и библиотеки Django.

Ведь автор описал в книге далеко не все возможности этой замечательной библиотеки. Так, не были рассмотрены инструменты для работы с файлами CSV, экспорт в формат Adobe PDF, криптографические средства, средства для разработки многоязычных сайтов, использование в проекте нескольких баз данных и многое другое. «За бортом» осталась и разработка дополнений для Django: собственных классов полей моделей, дополнительных модулей шаблонизатора и пр.

Автор особо не углублялся и в рассмотрение языка Python. Он очень мало сказал об объявлении классов, ограничившись базовыми средствами, без которых мы не смогли бы объявить модель или класс-контроллер. Он опустил рассмотрение развитых языковых конструкций — они были бы слишком сложны для начинающих программистов, которыми мы в настоящий момент являемся, к тому же, в конце концов, без них можно обойтись. Да и о богатейшей стандартной библиотеке Python он практически не рассказал, ограничившись рассмотрением лишь нескольких самых необходимых классов и функций.

Как автор и предупреждал во *введении*, в книге шел разговор лишь о минимальном наборе инструментов, необходимом для разработки сайта общего назначения. Такого, как интернет-представительство фирмы «Веник-Торг», включающее перечень товаров, разбитых на категории, гостевую книгу, новости, блог, универсальное хранилище изображений, список рассылки, генератор каналов RSS и Atom, главную страницу (куда же без нее!) и несколько обычных страниц со сведениями о фирме, списком контактов и тому подобными данными.

Что касается дополнительных библиотек, то, опять же, автор рассказал лишь о тех, что были востребованы в большинстве случаев: о генераторе миниатюр, о средстве

для привязки тегов и выполнения поиска по ним и о системе форматирования текста согласно правилам BBCode. Все прочие библиотеки (а их существует огромное множество и выполняют они самые разные задачи) здесь не описывались.

Невозможно описать все, чем богаты Python, Django и все библиотеки сторонних разработчиков, в одной не очень толстой книге.

Но есть Интернет, множество сайтов, посвященных Python и Django, в том числе и российских. На этих сайтах можно найти официальную документацию, статьи о программировании, примеры кода, которые мы можем без проблем использовать в своих разработках, и самые разнообразные библиотеки. В табл. З.1 перечислены некоторые из этих сайтов.

Таблица З.1. *Интернет-адреса для получения дополнительной информации*

Интернет-адрес	Описание
http://www.python.org/	Официальный сайт языка Python. Дистрибутивы, документация, поддержка
https://pypi.python.org/pypi	Официальный, одобренный сообществом разработчиков языка Python перечень дополнительных библиотек
http://python.su/	Российский блог Python-программистов
http://python-3.ru/	Статьи по программированию на Python 3
http://pythonworld.ru/	Статьи по Python-программированию для начинающих
http://diveinto.python.ru/index.html	Статьи по Python-программированию для опытных программистов
http://habrahabr.ru/hub/python/	Подборка статей по Python на сайте «Хабрахабр». Для опытных программистов
https://www.djangoproject.com/	Официальный сайт библиотеки Django. Дистрибутивы, документация, поддержка
http://djbook.ru/	Русскоязычная документация по Django
http://habrahabr.ru/hub/django/	Подборка статей по Django на сайте «Хабрахабр» Для опытных программистов
http://www.regexr.com/	Онлайновый тестировщик регулярных выражений

Официальные сайты использованных в книге сторонних библиотек перечислены в *приложении 1*.

Остается только напомнить, что материалы разработанного нами полнофункционального сайта фирмы «Веник-Торг» содержатся в сопровождающем книгу электронном архиве, который можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977504218.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение 2*).

На этом все. Автор книги теперь уже окончательно прощается и желает вам успешного программирования.

Владимир Дронов

ПРИЛОЖЕНИЕ 1

Установка программной среды языка Python и дополнительных библиотек

В этом приложении описан процесс установки программной среды языка Python и дополнительных библиотек для него, в том числе и Django, и приведен полный список библиотек, необходимых для разработки описанного в книге сайта.

Установка Python

Дистрибутивный комплект программной среды Python можно найти на «домашнем» сайте проекта по интернет-адресу **http://www.python.org/**. Этот комплект в версии для Windows представляет собой обычный пакет формата Microsoft Installer (MSI) и доступен как в 32-, так и 64-разрядной редакциях.

Установка Python не представляет особой сложности и выполняется точно так же, как и установка любого другого Windows-приложения. Есть лишь пара моментов, о которых нужно сказать несколько слов.

Во-первых, в начале установки установщик Python спросит нас, какие компоненты пакета необходимо установить на компьютер и какие изменения в системных настройках нужно выполнить. Окно установщика при этом будет выглядеть так, как показано на рис. П1.1.

В иерархическим списке, имеющемся в этом окне, приведен полный комплект устанавливаемых компонентов и вносимых в системные настройки изменений. Все они помечены как принятые к выполнению, за исключением параметра **Add python.exe to Path** (соответствующий пункт на рис. П1.1 выбран).

Этот параметр указывает установщику добавить путь к исполняемому файлу python.exe — интерпретатору Python — в список системной переменной PATH. Нам настоятельно рекомендуется это сделать — в противном случае мы не сможем установить некоторые сторонние библиотеки.

Щелкнем на пункте **Add python.exe to Path** левой кнопкой мыши. На экране появится небольшое меню, в котором мы выберем пункт **Will be installed on local hard drive**.

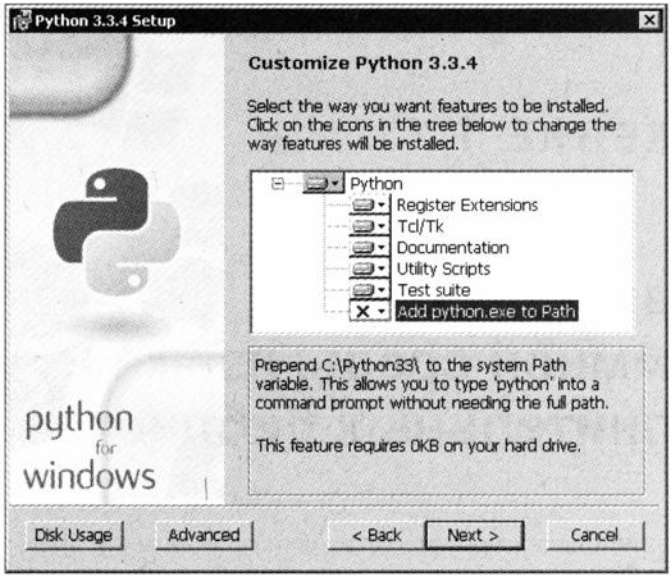


Рис. П1.1. Окно установщика Python со списком устанавливаемых компонентов

После установки Python нам также следует добавить в список путей, хранящихся в системной переменной `PATH`, путь к файлу django-admin.py. Этот файл, хранящий код административной утилиты Django, мы использовали в *главе 4* для создания проекта. Если мы не добавим путь к нему в список путей переменной `PATH`, нам придется каждый раз набирать в командной строке полное имя этого файла: `<папка, где установлен Python>\Lib\site-packages\django\bin\django-admin.py`.

Добавить путь в список переменной `PATH` несложно. Сейчас мы узнаем, как это делается.

Если мы пользуемся Windows Vista или более новой версией Windows, мы запустим из Панели управления апплет **Система** и щелкнем на расположенной в его окне ссылке **Дополнительные параметры системы**. Пользователем более старых, чем Vista, версий Windows проще — им достаточно лишь запустить этот апплет.

В открывшемся на экране окне мы найдем вкладку **Дополнительно** и переключимся на нее (рис. П1.2). На этой вкладке щелкнем на кнопке **Переменные среды**, после чего на экране откроется одноименное диалоговое окно (рис. П1.3).

Здесь нам понадобится список **Системные переменные**. Отыщем в нем пункт **Path**, соответствующий нужной нам переменной, выделим его и либо нажмем расположенную под списком кнопку **Изменить**, либо просто щелкнем на этом пункте двойным щелчком.

В поле ввода **Значение переменной** открывшегося на экране диалогового окна **Изменение системной переменной** (рис. П1.4) мы увидим список уже хранящихся там путей, разделенных символами точки с запятой. Допишем туда путь к файлу django-admin.py — только путь, без имени самого файла! После чего поочередно нажмем кнопки **ОК** всех трех открытых окон, чтобы сохранить сделанные изменения.

На этом процесс установки программного пакета Python закончен.

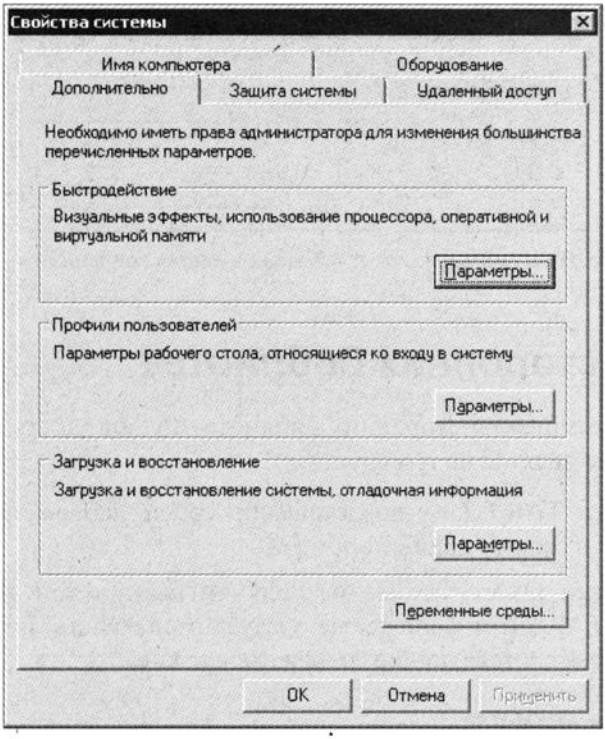


Рис. П1.2. Вкладка **Дополнительно** окна апплета **Свойства системы**

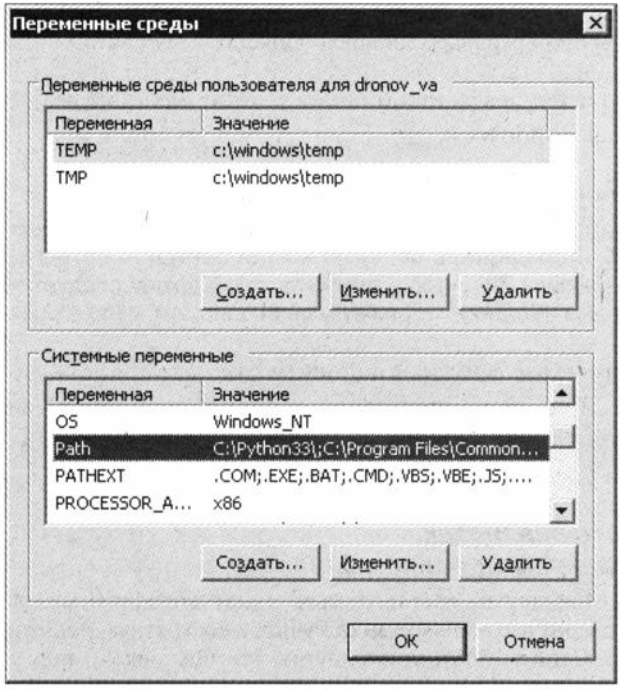


Рис. П1.3. Диалоговое окно **Переменные среды**

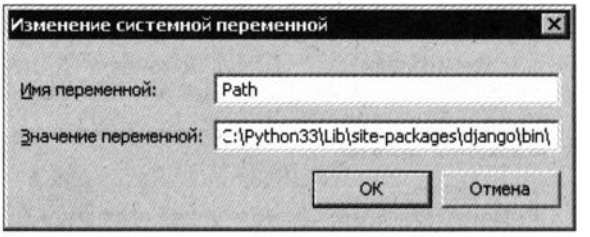


Рис. П1.4. Диалоговое окно **Изменение системной переменной**

Установка сторонних библиотек

Дистрибутивные комплекты сторонних библиотек Python, доступные в Интернете, по своим форматам делятся на три группы:

- архивы формата TAR/GZ — представляют собой наборы файлов, созданные в формате TAR и сжатые архиватором GZ.

  После распаковки такого архива мы получим папку dist, в которой находится файл формата TAR. Этот файл также следует распаковать, и в результате будут извлечены файлы, составляющие содержимое архива;

  Распаковка архивов

  Для распаковки архивов подобного рода автор рекомендует пользоваться архиваторами 7-Zip (**http://7-zip.org/**) или Bandizip (**http://www.bandisoft.com/bandizip/ru/**). Они поддерживают распаковку практически всех форматов наиболее распространенных в настоящее время архивных файлов и полностью бесплатны.

- архивы ZIP — распаковать такой архив можно архиватором, включенным в состав Проводника Windows;

  Загрузка файлов с портала Github

  Известный портал публикации исходных текстов программ и обеспечения контроля их версий Github (**https://github.com/**) позволяет загрузить опубликованный на нем код только в виде архива ZIP. Чтобы загрузить такой архив, следует перейти на страницу, где опубликованы исходные тексты нужной библиотеки, и нажать кнопку **Download ZIP**.

- обычные исполняемые файлы Windows (в формате EXE) — представляют собой дистрибутивы и после установки создают в списке апплета **Установка/удаление программ** или **Программы и компоненты** пункт, выбрав который, мы можем удалить эту библиотеку.

  Особенность библиотек в виде исполняемых файлов Windows

  Дистрибутивы библиотек, поставляемые в виде исполняемых файлов Windows, создаются под конкретную версию языка Python и конкретную редакцию Windows: 32- или 64-разрядную. Если выбрать дистрибутив, не подходящий под установленную нами версию Python или используемую нами редакцию операционной системы, установить библиотеку у нас, скорее всего, не получится.

Установка библиотек, поставляющихся в виде исполняемых файлов Windows, выполняется так же, как и установка любого другого Windows-приложения. С библиотеками, поставляющимися в виде архивов, дело обстоит чуть сложнее.

Прежде всего, распакуем библиотеку, которую хотим установить. Далее запустим командную строку, в ней выполним переход в папку, где находится содержимое библиотеки. Проверим, присутствует ли в этой папке файл setup.py, содержащий код установщика этой библиотеки, и наберем следующую команду:

```
setup.py install
```

Установщик в процессе работы выведет в командную строку большое количество сообщений, указывающих, какой файл в какую папку был помещен. А если библиотека достаточно велика (как, например, Django), процесс установки может занять несколько минут.

По окончании процесса установки библиотеки мы увидим в окне командной строки приглашение на ввод очередной команды. Это значит, что библиотека полностью установлена и готова к использованию.

Список необходимых библиотек

Осталось привести полный список сторонних библиотек, необходимых для разработки сайтов согласно приведенным в книге сведениям, с их краткими описаниями и интернет-адресами «домашних» сайтов и страниц с дистрибутивами, рекомендуемыми автором книги.

Django

Предназначена для разработки серверной части Web-сайтов. Обязательна для установки в любом случае.

- «Домашний» сайт: **https://www.djangoproject.com/**.
- Страница с дистрибутивом: **https://www.djangoproject.com/download/**.
- Формат дистрибутива: TAR/GZ.

Setuptools

Содержит инструменты, предназначенные для установки библиотек и отсутствующие в комплекте поставки дистрибутива Python. Необходима для успешной установки многих сторонних библиотек.

- «Домашний» сайт: **https://pypi.python.org/pypi/setuptools**.
- Страница с дистрибутивом: **http://www.lfd.uci.edu/~gohlke/pythonlibs/#setuptools**.
- Формат дистрибутива: EXE.

Pytz

Обеспечивает правильный вывод гиперссылок для быстрой фильтрации записей по значению даты на страницах встроенного административного сайта Django (см. *главу 31*). Необходима только при использовании этой функции, но настоятельно рекомендуется к установке.

- «Домашний» сайт: **https://pypi.python.org/pypi/pytz**.
- Страница с дистрибутивом: **https://pypi.python.org/pypi/pytz**.
- Формат дистрибутива: ZIP и TAR/GZ.

Pillow

Мощная библиотека для обработки графики. Необходима в случае использования в моделях полей класса `ImageField` (см. *главу 13*) и для нормального функционирования библиотеки easy-thumbnails.

- «Домашний» сайт: **http://pillow.readthedocs.org/en/latest/**.
- Страница с дистрибутивом: **http://www.lfd.uci.edu/~gohlke/pythonlibs/#pillow**.
- Формат дистрибутива: EXE.

easy-thumbnails

Компактная, быстрая и удобная библиотека для формирования и вывода миниатюр. Требует для работы наличия установленной библиотеки Pillow.

- «Домашний» сайт: **https://github.com/SmileyChris/easy-thumbnails**.
- Страница с дистрибутивом: **https://github.com/SmileyChris/easy-thumbnails**.
- Формат дистрибутива: ZIP.

django-taggit

Обеспечивает привязку к сущностям, хранящимся в записях моделей, тегов и фильтрацию записей по ним.

- «Домашний» сайт: **http://django-taggit.readthedocs.org/en/latest/**.
- Страница с дистрибутивом: **https://github.com/alex/django-taggit**.
- Формат дистрибутива: ZIP.

django-precise-bbcode

Позволяет форматировать текст, применяя теги BBCode.

- «Домашний» сайт: **http://django-precise-bbcode.readthedocs.org/en/latest/**.
- Страница с дистрибутивом: **https://pypi.python.org/pypi/django-precise-bbcode/**.
- Формат дистрибутива: TAR/GZ.

Psycopg

Обеспечивает взаимодействие Django с сервером данных PostgreSQL. Требуется лишь в случае, если сайт для хранения данных использует базу такого формата.

- «Домашний» сайт: **http://initd.org/psycopg/**.
- Страница с дистрибутивом: **http://www.stickpeople.com/projects/python/win-psycopg/**.
- Формат дистрибутива: EXE.

ПРИЛОЖЕНИЕ 2

Описание электронного архива

Электронный архив с материалами, сопровождающими книгу, можно скачать с FTP-сервера издательства по ссылке **ftp://ftp.bhv.ru/9785977504218.zip**, а также со страницы книги на сайте **www.bhv.ru**.

В архиве находятся следующие папки и файлы:

- \!!!others — папка с дополнительными файлами;
- \broomtrade — папка с полным содержимым сайта фирмы «Веник-Торг», примерами к материалам этой книги;
- \readme.txt — пояснительный текст, обязательный к прочтению.

Предметный указатель

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

Р

С

Т

Ф

Х

Ц

Ш

Э

Django: практика создания Web-сайтов на Python

Web-программирование на Python и Django — это просто!

ДРОНОВ Владимир Александрович профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года, автор более 20 популярных компьютерных книг, в том числе «HTML 5, CSS 3 и Web 2.0. Разработка современных Web-сайтов», «PHP 5/6, MySQL 5/6 и Dreamweaver CS4. Разработка интерактивных Web-сайтов», «JavaScript и AJAX в Web-дизайне», «Windows 8: разработка Metro-приложений для мобильных устройств» 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».

Книга посвящена разработке Web-сайтов на популярном языке программирования Python с использованием библиотеки Django. Описывается создание моделей, контроллеров и шаблонов, применение форм для ввода данных и выгрузки на сайт файлов, реализация разграничения доступа, комментирование кода, работа со статичными страницами, применение сторонних библиотек для вывода миниатюр. Рассказывается о форматировании текста тегами BBCode, привязке к позициям тегов и выполнении поиска по тегам. Рассматриваются инструменты для генерирования каналов новостей RSS и Atom, рассылки электронной почты и настройка встроенного административного сайта Django под свои нужды. Детально описывается процесс разработки и публикации полнофункционального коммерческого Web-сайта, использующего, в том числе, технологию AJAX.

КАТЕГОРИЯ:
Программирование, языки, библиотеки

ISBN 978-5-9775-0421-8

БХВ-ПЕТЕРБУРГ
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