Антонио Меле

Django 2 в примерах

Создавайте мощные и надежные веб-приложения на Python с нуля

ДМК Пресс
издательство

Антонио Меле

Django 2 в примерах

Antonio Melé

Django 2 by Example

Build powerful and reliable Python web applications from scratch

Packt>
BIRMINGHAM – MUMBAI

Антонио Меле

Django 2 в примерах

Создавайте мощные и надежные веб-приложения Python с нуля

ДМК Пресс
издательство
Москва, 2019

УДК 004.42
ББК 32.972
М47

Меле А.
М47 Django 2 в примерах / пер. с анг. Д. В. Плотниковой. – М.: ДМК Пресс, 2019. – 408 с.: ил.

ISBN 978-5-97060-746-6

Django — это мощный Python-фреймворк для веб-приложений, который поощряет быстрое развитие и чистый, прагматичный дизайн, предлагает относительно простое обучение. Это делает его привлекательным как для новичков, так и для опытных разработчиков.

В рамках данной книги вы пройдете весь путь создания полноценных веб-приложений с помощью Django. Вы научитесь работать не только с основными компонентами, предоставляемыми фреймворком, но и узнаете, как интегрировать в проект популярные сторонние инструменты. В книге описано создание приложений, которые решают реальные задачи, используют лучшие практики разработки. После прочтения этой книги у вас будет понимание того, как работает Django, как создавать практичные веб-приложения и расширять их с помощью дополнительных инструментов.

Издание будет полезно всем разработчикам приложений.

УДК 004.42
ББК 32.972

ISBN 978-1-78847-248-7 (анг.)
ISBN 978-5-97060-746-6 (рус.)

Посвящается моей сестре

Содержание

Об авторе

Антонио Меле – технический директор компании Exo Investing и основатель Zenx IT. Он занимается разработкой Django-приложений для клиентов из различных отраслей с 2006 года. Антонио работал в качестве технического директора и консультанта для множества технологических стартапов, управлял командами разработчиков при реализации проектов для цифрового бизнеса, получил степень магистра в области компьютерных наук в Университете Потифисия Комильяс. Его увлечение программированием началось с отца, который поощрял и вдохновлял Антонио.

О рецензентах

Норберт Мате – веб-разработчик, начал свою карьеру в 2008 г. Первым языком программирования для него стал PHP, затем он перешел на JavaScript/Node.js и Python/Django/Django REST Framework. Он увлечен построением архитектуры программного обеспечения, паттернами проектирования, чистым кодом. Норберт участвовал в рецензировании еще одной книги о Django, Django RESTful Web Services, от издательства Pack Publishing.

Я бы хотел поблагодарить свою жену за ее поддержку.

Предисловие

Django – это мощный Python-фреймворк для веб-приложений, который поощряет быстрое развитие и чистый, прагматичный дизайн, предлагает относительно простое обучение. Это делает его привлекательным как для новичков, так и для опытных разработчиков.

В рамках данной книги вы пройдете весь путь создания полноценных веб-приложений с помощью Django. Вы научитесь работать не только с основными компонентами, предоставляемыми фреймворком, но и узнаете, как интегрировать в проект популярные сторонние инструменты.

В книге описано создание приложений, которые решают реальные задачи, используют лучшие практики разработки. Код разбирается в виде пошаговых инструкций, которым легко следовать.

После прочтения этой книги у вас будет понимание того, как работает Django, как создавать практичные веб-приложениия и расширять их с помощью дополнительных инструментов.

Для кого эта книга

Книга предназначена для разработчиков, которые имеют базовые знания Python и хотят изучить Django на практике. Возможно, вы совсем новичок в веб-разработке или уже немного знакомы с Django, но хотите погрузиться глубже. Эта книга даст вам возможность поработать с различными подсистемами фреймворка, применить их в реальных проектах. Для комфортного изучения необходимо знать основы программирования и иметь представление об HTML и JavaScript.

О чем эта книга

Глава 1 «Создание приложения блога» познакомит вас с фреймворком. Вы создадите свое первое Django-приложение – опишете модели, обработчики, шаблоны для отображения статей блога. Узнаете, как Django взаимодействует с базами данных, научитесь работать с Django ORM и запустите сайт администрирования.

В главе 2 «Добавление продвинутых функций в блог» вы узнаете, как обрабатывать формы и модельные формы, отправлять электронные письма пользователям, и подключите к проекту сторонние библиотеки. Добавите возможность поделиться статьей по электронной почте и комментировать статьи, создадите систему тегов.

В главе 3 «Расширение приложения блога» вы научитесь создавать собственные шаблонные теги и фильтры. В этой главе мы узнаем, как добавить карту

сайта и RSS-рассылку. Здесь вы закончите реализацию блога, добавив полнотекстовый поиск с помощью PostgreSQL.

В главе 4 «Создание социальной сети» мы начнем работать над новым проектом – социальной сетью. Вы будете использовать подсистему аутентификации Django, создадите собственную модель профиля пользователя и добавите возможность входить на сайт через аккаунты других социальных сетей.

В главе 5 «Совместное использование содержимого сайта» мы познакомимся с таким понятием, как букмарклет, – вы добавите в свою социальную сеть возможность сохранять картинки с других сайтов. В этой главе вы поработаете с отношениями между моделями вида «многие ко многим», реализуете AJAX-букмарклет и собственные декораторы, научитесь генерировать миниатюры изображений.

Глава 6 «Отслеживание действий пользователей» описывает создание системы подписок в социальной сети. Вы добавите новостную ленту пользователей, узнаете, как оптимизировать обращения к базе данных, зачем нужны сигналы Django и как их применяют в проектах, подключите хранилище Redis.

Глава 7 «Создание онлайн-магазина» начнется с создания нового проекта – интернет-магазина. Вы определите необходимые модели и классы для товаров и корзины, которые используют подсистему сессий Django, добавите контекстный процессор и научитесь отправлять асинхронные задачи в Celery.

В главе 8 «Управление заказами и платежами» мы рассмотрим, как подключить к магазину платежную систему, как доработать сайт администрирования – добавить возможность экспортировать заказы в формат CSV, а также научитесь формировать PDF-документы для предоставления покупателям счетов.

Глава 9 «Расширение онлайн-магазина» описывает процесс создания системы купонов и скидок. Кроме этого, вы узнаете, как добавить переводы на несколько языков для сайта, и реализуете рекомендательную систему, используя Redis.

В главе 10 «Создание платформы для онлайн-обучения» мы начнем с нового проекта – интернет-платформы для обучения. Узнаем, что такое фикстуры, и применим их в проекте, рассмотрим способы наследования моделей в Django, реализуем собственное поле модели, настроим доступ к разделам сайта с помощью разрешений и группы пользователей. Вы научитесь работать с системой управления содержимым сайта и обрабатывать наборы форм.

Глава 11 «Отображение и кеширование содержимого курсов» описывает, как реализовать систему регистрации участников, систему управления доступом к курсам. Вы будете формировать различное содержимое уроков и подключите подсистему кеширования.

В главе 12 «Реализация API» мы познакомимся с мощным инструментом для создания RESTful API – Django REST Framework.

В заключительной главе 13 «Запуск в боевом режиме» вы узнаете, как запустить проект в боевой среде с помощью uWSGI и NGINX, как защитить его с помощью SSL. В этой главе вы также создадите собственный промежуточный слой и команду управления Django.

Чтобы получить максимальную пользу от этой книги

Эта книга окажется максимально полезна для вас, если вы достаточно знакомы с Python, понимаете базовые конструкции языка, принципы объектно-ориентированного программирования в Python. Также желательно уметь работать с HTML и JavaScript. Перед прочтением книги рекомендуем ознакомиться с первыми тремя разделами официальной документации Django на странице https://docs.djangoproject.com/en/2.0/intro/tutorial01/.

Принятые обозначения

В книге используются специальные выделения шрифтом для важных фрагментов, рекомендуем ознакомиться с ними.

`CodeInText` – так показаны фрагменты кода в тексте, названия баз данных и таблиц, каталогов и файлов, расширения файлов, пути в системе, пользовательский ввод. Вот пример такого выделения: «Вы можете деактивировать окружение, для этого выполните команду `deactivate`».

Листинг выглядит следующим образом:

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Когда во фрагменте кода мы хотим обратить ваше внимание на конкретные строки, они будут выделены жирным:

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

Ввод и вывод команд, выполняемых в консоли, будет выглядеть так:

```
$ python manage.py startapp blog
```

Полужирное начертание используется для выделения важных слов или фраз, которые вы будете видеть в браузере. Например, тексты в меню, названия кнопок или блоков: «Заполните форму и нажмите кнопку **SAVE**».

Термины будут выделяться *курсивом*, так вы сможете легко найти определение, если захотите к нему вернуться.

Так будут оформляться предупреждения и важные примечания.

Так будут оформляться советы или рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Код также находится в публичном репозитории на GitHub: https://github.com/PacktPublishing/Django-2-by-Example, в случае если появятся доработки в коде, они будут добавлены в этом репозитории.

У нас есть и другие интересные примеры кода для книг и видео из каталога, вы можете найти их на https://github.com/PacktPublishing/.

Список опечаток

Хотя мы приняли все возможные меры, для того чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Создание приложения блога

Благодаря этой книге мы научимся создавать полноценные Django-приложения, готовые для промышленного использования. В первой главе описано создание простого блога с помощью Django. Мы получим общее представление о том, как работает фреймворк, поймем, как взаимодействуют его компоненты, и научимся создавать Django-приложения со стандартными возможностями. Мы увидим, как выстраивается полноценный проект, не вдаваясь в детали реализации каждого компонента. Более углубленно отдельные части фреймворка рассмотрим в последующих главах.

В этой главе мы изучим такие темы:

- установка Django и создание первого проекта;
- проектирование моделей и генерация миграций;
- реализация сайта для администрирования приложения;
- работа с QuerySet'ами и менеджерами моделей;
- реализация обработчиков, шаблонов и URL'ов;
- добавление постраничного вывода для списков;
- использование классов в качестве обработчиков запросов.

Установка Django

Если у вас уже установлен Django, можете пропустить этот раздел и сразу перейти к разделу «Создание первого проекта». Поскольку Django является Python-пакетом, он может быть установлен в любое окружение с Python. Для начала давайте настроим окружение для разработки.

Django 2.0 совместим с версиями Python, начиная с 3.4. Во всех примерах мы будем использовать Python 3.6.5. Если вы работаете с Linux или macOSX, скорее всего, у вас уже установлен Python. Если вы используете Windows, скачайте дистрибутив по ссылке https://www.python.org/downloads/windows.

Проверить, установлен ли Python на вашем компьютере, можно, напечатав `python` в консоли. Если увидите что-то подобное, значит, Python установлен:

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Если установленная версия ниже, чем 3.4, или если Python не установлен, скачайте и установите Python 3.6.5 с сайта https://www.python.org/downloads/.

Так как мы будем использовать Python 3, нет необходимости отдельно устанавливать базу данных. В эту версию языка уже встроена система управления базами данных (СУБД) SQLite. SQLite – легкая СУБД, которую можно использовать вместе с Django для быстрой разработки. Если в дальнейшем вы планируете развертывать ваше приложение в боевой среде, рекомендуем использовать более развитую и мощную СУБД, например PostgreSQL, MySQL или Oracle. Подробное описание того, как настроить конкретную СУБД на работу с Django, можно найти на https://docs.djangoproject.com/en/2.0/topics/install/#database-installation.

Создание изолированного Python-окружения

При разработке на Python рекомендуем использовать `virtualenv` для создания изолированного окружения. Так мы сможем использовать различные версии пакетов для разных проектов, что гораздо более практично, чем установка версий пакетов непосредственно в систему. Другим важным достоинством использования `virtualenv` является то, что для установки Python-пакетов пользователь не обязан иметь права администратора. Выполните следующую команду для установки `virtualenv`:

```
pip install virtualenv
```

После установки создайте изолированное окружение с помощью команды:

```
virtualenv my_env
```

Мы создали каталог `my_env/` для Python-окружения. Любая Python-библиотека, установленная при активированном окружении, будет сохраняться в папку `my_env/lib/python3.6/site-packages`.

Если в операционной системе изначально установлен Python 2.X и вы также установили Python 3.X, нужно будет явно указать `virtualenv`, чтобы утилита использовала версию 3.X.

Вы можете вывести путь до каталога, в который установлен Python 3, и использовать его для создания виртуального окружения с помощью команд:

```
zenx$ which python3
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3
zenx$ virtualenv my_env -p
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3
```

Для активации виртуального окружения выполним команду:

```
source my_env/bin/activate
```

При работе в консоли появится имя активированного виртуального окружения в скобках:

```
(my_env)laptop:~ zenx$
```

Для того чтобы в любой момент деактивировать виртуальное окружение, выполните команду `deactivate`.

Больше информации об утилите virtualenv можно найти на странице https://virtualenv.pypa.io/en/latest/.

Поверх `virtualenv` можно использовать `virtualenvwrapper`. Этот инструмент предоставляет обертку, которая упрощает создание и управление виртуальными окружениями. Для того чтобы узнать подробнее об этой утилите, перейдите по ссылке https://virtualenvwrapper.readthedocs.io/en/latest/.

Установка Django через pip

Использование менеджера пакетов `pip` – это предпочтительный способ установки Django. `pip` уже установлен в Python 3.6, но вы также можете найти инструкции по установке `pip` на https://pip.pypa.io/en/stable/installing/.

Выполним следующую команду в консоли, для того чтобы установить Django с помощью `pip`:

```
pip install Django==2.0.5
```

Django будет установлен в папку `site-packages/` нашего виртуального окружения.

Давайте убедимся, что установка Django прошла успешно. Для этого запустите интерпретатор, выполнив команду `python` в консоли, импортируйте Django и проверьте его версию:

```
>>> import django
>>> django.get_version()
'2.0.5'
```

Если вы получили такой же вывод в результате выполнения команд, значит, установка выполнена успешно.

Для установки Django могут использоваться и другие способы. Ознакомиться с полной инструкцией по настройке фреймворка можно на странице https://docs.djangoproject.com/en/2.0/topics/install/.

Создание первого проекта

Нашим первым проектом будет полноценный блог. Django предоставляет команду, которая поможет нам создать базовую структуру файлов и каталогов. Выполните ее в консоли:

```
django-admin startproject mysite
```

Благодаря этому мы создадим Django-проект с названием `mysite`.

Для исключения конфликтов имен избегайте таких названий для ваших проектов, которые могут совпадать со стандартными пакетами Python или Django.

Давайте посмотрим на структуру сгенерированного проекта:

```
mysite/
  manage.py
  mysite/
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

Каждый из этих файлов имеет свое значение:

- `manage.py` – утилита командной строки, используемая для управления проектом. Это минимальная обертка над файлом `django-admin.py`. Мы не будем редактировать этот файл;
- `mysite/` – это папка нашего проекта, которая содержит файлы:
- `__init.py` – пустой файл, который говорит Python о том, что `mysite` является Python-пакетом;
- `settings.py` – содержит конфигурацию нашего проекта, в нем уже заданы базовые настройки;
- `urls.py` – здесь будут храниться *шаблоны адресов* (Uniform Resource Locator – URL). Каждый URL, определенный в этом файле, будет связан с конкретным обработчиком;
- `wsgi.py` – конфигурация для запуска проекта как WSGI-приложения.

Сгенерированный файл `settings.py` содержит настройки приложения, включая базовую конфигурацию доступа к СУБД SQLite 3 и список `INSTALLED_APPS`, который описывает общие настройки Django-приложения, добавленные в проект по умолчанию. Мы изучим эту часть более подробно в разделе «Настройки проекта».

Для завершения первоначальной установки необходимо создать таблицы в базе данных для всех приложений из списка `INSTALLED_APPS`. Откройте консоль и выполните команды:

```
cd mysite
python manage.py migrate
```

Вы увидите вывод:

```
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying sessions.0001_initial... OK
```

Это говорит о том, что миграции успешно применены к базе данных (были созданы таблицы для стандартных приложений нашего проекта). Более подробно о миграциях и команде `migrate` мы узнаем чуть позже, в разделе «Создание и применение миграций».

Запуск сервера для разработки

Django поставляется с веб-сервером для быстрого запуска нашего кода, благодаря чему нет необходимости тратить время на настройку стороннего сервера. Когда мы запускаем сервер разработки Django, он начинает отслеживать изменения в коде и автоматически перезапускает сервер, освобождая от необходимости делать это вручную после внесения правок. Но при некоторых действиях нам все-таки придется перезапускать сервер самостоятельно, например при добавлении новых файлов в проект.

Запустите сервер разработки, выполнив команду из корневого каталога проекта:

```
python manage.py runserver
```

Вы должны будете увидеть что-то подобное:

```
Performing system checks...

System check identified no issues (0 silenced).
May 06, 2018 – 17:17:31
Django version 2.0.5, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Теперь откройте в вашем браузере `http://127.0.0.1:8000/`. Вы должны увидеть страницу с информацией о том, что проект успешно запущен, как показано на рис. 1.1:

Если вы посмотрите в консоль, то увидите, что был обработан один GET-запрос, поступивший от браузера:

```
[06/May/2018 17:20:30] "GET / HTTP/1.1" 200 16348
```

Сервер разработки логирует в консоли каждый HTTP-запрос. Любая произошедшая ошибка также будет выведена в консоль.

Вы можете указывать Django, какие порт и адрес использовать для запуска сервера для разработки или какой файл конфигурации применить, с помощью флагов:

```
python manage.py runserver 127.0.0.1:8001
--settings=mysite.settings
```

Когда мы имеем дело с несколькими виртуальными окружениями, которые требуют разных конфигураций, то можем создать несколько файлов настроек для каждого окружения.

Стоит отметить, что этот веб-сервер пригоден только для разработки и не подходит для запуска и применения на реальном проекте. Для того чтобы за-

пустить Django-приложение в боевом окружении, необходимо запустить его как WSGI-приложение и использовать полноценный веб-сервер, например Apache, Gunicorn и WSGI. Подробная информация о том, как запускать Django-приложения в боевой среде, приведена по адресу https://docs.djangoproject.com/en/2.0/howto/deployment/wsgi/.

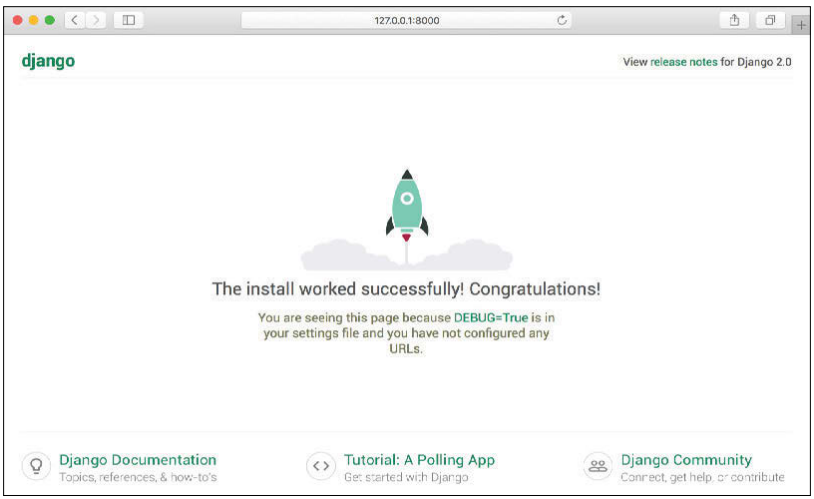

Рис. 1.1 ❖ Страница приветствия Django

С помощью главы 13 «Запуск в боевом режиме» мы узнаем, как настроить боевое окружение для наших Django-проектов.

Настройки проекта

Давайте откроем `settings.py` и посмотрим на конфигурацию проекта. В этот файл уже добавлено несколько настроек, но это только часть из тех, которые поддерживает Django. Полный список всех переменных для конфигурации приложения и их значения по умолчанию вы сможете найти на странице https://docs.djangoproject.com/en/2.0/ref/settings/. Стоит обратить внимание на следующие настройки:

- `DEBUG` – булевое значение, которое включает и отключает режим отладки проекта. Если оно равно `True`, Django будет отображать подробные страницы с ошибками при выбрасывании исключений в приложении. Когда мы будем разворачивать приложение на боевом сервере, нужно установить эту настройку в `False`. Никогда не публикуйте проект с включенным режимом отладки, т. к. пользователям станут доступны секретные данные конфигурации приложения;
- `ALLOWED_HOSTS` – не используется при включенной отладке и запуске тестов. Но как только мы развернем приложение и установим флаг `DEBUG`

в `False`, необходимо добавить домен сайта в эту настройку, для того чтобы Django мог с ним работать;

- `INSTALLED_APPS` – настройка, которую мы будем изменять во всех наших проектах. Она указывает Django, какие приложения активны на нашем сайте. По умолчанию Django подключает такие приложения, как:
  - `django.contrib.admin` – сайт администрирования;
  - `django.contrib.auth` – подсистема аутентификации;
  - `django.contrib.contenttypes` – подсистема для работы с типами объектов системы;
  - `django.contrib.sessions` – подсистема сессий;
  - `django.contrib.messages` – подсистема сообщений;
  - `django.contrib.staticfiles` – подсистема для управления статическим содержимым сайта;
- `MIDDLEWARE` – список подключенных промежуточных слоев;
- `ROOT_URLCONF` – указывает на Python-модуль, который содержит корневые шаблоны URL'ов приложения;
- `DATABASES` – представляет собой словарь, содержащий настройки для всех баз данных проекта. Тут всегда должна быть указана хотя бы одна база данных. По умолчанию подключена СУБД SQLite3;
- `LANGUAGE_CODE` – определяет код языка по умолчанию для Django-сайта;
- `USE_TZ` – указывает Django на необходимость поддержки временных зон. В Django включена возможность использовать объекты дат, учитывающие временные зоны. Эта настройка устанавливается в `True`, когда мы создаем проект с помощью команды `startproject`.

Не переживайте, если пока вам понятно не все из того, что вы увидели. Мы подробно разберем параметры настройки Django в следующих главах.

Проекты и приложения

На протяжении всей книги мы будем постоянно сталкиваться с терминами «проект» и «приложение». В Django *проект* – это код, созданный с использованием Django и содержащий некоторые настройки. *Приложение* – это набор модулей, описывающих модели, обработчики запросов, шаблоны и конфигурации URL'ов. Приложение взаимодействует с фреймворком, предоставляя некоторую функциональность, и может быть многократно использовано в других проектах. Мы можем сопоставить проект с сайтом, который состоит из нескольких приложений (блога, раздела вопросов, форума), каждое из которых может быть использовано и в других проектах.

Создание приложения

Давайте начнем работать над нашим первым Django-приложением. Мы будем создавать блог с нуля. Выполните следующую команду из корневого каталога проекта:

```
python manage.py startapp blog
```

Так мы создадим базовую структуру приложения, которая выглядит следующим образом:

```
blog/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

Давайте рассмотрим назначение этих модулей:

- `admin.py` – здесь мы регистрируем модели для добавления их в систему администрирования Django (использование сайта администрирования Django не является обязательным);
- `apps.py` – файл, содержащий основную конфигурацию приложения `blog`;
- `migrations` – папка, содержащая миграции базы данных приложения. Миграции позволяют Django отслеживать изменения моделей и синхронизировать их со схемой данных базы;
- `models.py` – модели данных приложения. В любом Django-приложении должен быть этот файл, но он может оставаться пустым;
- `tests.py` – этот файл предназначен для создания тестов для приложения;
- `views.py` – вся логика приложения описывается здесь. Каждый обработчик получает HTTP-запрос, обрабатывает его и возвращает ответ.

Проектирование схемы данных для блога

Мы начнем создавать схему данных нашего блога с описания моделей. *Модель* – это Python-класс, который является наследником `django.db.models.Model`. Каждый атрибут представляет собой поле в базе данных. Django создает таблицу в базе данных для каждой модели, определенной в `models.py`. Когда мы создаем модель, Django предоставляет удобный *интерфейс* (ApplicationProgrammingInterface – API) для формирования запросов в базу данных.

Для начала мы определим модель `Post`, добавив следующий фрагмент кода в `models.py` приложения `blog`:

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

class Post(models.Model):
    STATUS_CHOICES = (
        ('draft', 'Draft'),
        ('published', 'Published'),
    )
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250, unique_for_date='publish')
```

```
    author = models.ForeignKey(User, on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default='draft')

    class Meta:
        ordering = ('-publish',)

    def __str__(self):
        return self.title
```

Это модель данных для статей блога. Давайте рассмотрим поля, которые мы только что определили для данной модели:

- `title` – это поле заголовка статьи. Оно определено как тип `CharField`, который соответствует типу `VARCHAR` в базе данных;
- `slug` – это поле будет использоваться для формирования URL'ов. *Слаг* – короткое название, содержащее только буквы, цифры и нижние подчеркивания или дефисы. Мы будем использовать `slug` для построения *семантических URL'ов* (friendly URLs) для статей. Мы также добавили параметр `unique_for_date`, поэтому сможем формировать уникальные URL'ы, используя дату публикации статей и `slug`. Django будет предотвращать создание нескольких статей с одинаковым слагом в один и тот же день;
- `author` – это поле является внешним ключом и определяет *отношение* «*один ко многим*». Мы указываем, что каждая статья имеет автора, причем каждый пользователь может быть автором любого количества статей. Для этого поля Django создаст в базе данных внешний ключ, используя первичный ключ связанной модели. В этом случае мы обращаемся к модели `User` подсистемы аутентификации Django. Параметр `on_delete` определяет поведение при удалении связанного объекта. Эта особенность не специфична для Django, а взята из стандарта SQL. Используя `CASCADE`, мы говорим, чтобы при удалении связанного пользователя база данных также удаляла написанные им статьи. Вы можете посмотреть все доступные опции на странице https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.ForeignKey.on_delete. Мы также указали имя обратной связи от `User` к `Post`– параметр `related_name`. Так мы с легкостью получим доступ к связанным объектам автора. Позже мы более подробно изучим эту тему;
- `body` – основное содержание статьи. Это текстовое поле, которое будет сохранено в столбце с типом `TEXT` в SQL базе данных;
- `publish` – поле даты, которое сохраняет дату публикации статьи. Мы используем функцию Django `now` для установки значения по умолчанию. Она возвращает текущие дату и время. Вы можете рассматривать ее как стандартную функцию `datetime.now` из Python, но с учетом временной зоны;

- `created` – это поле даты указывает, когда статья была создана. Так как мы используем параметр `auto_now_add`, дата будет сохраняться автоматически при создании объекта;
- `updated` – дата и время, указывающие на период, когда статья была отредактирована. Так как мы используем параметр `auto_now`, дата будет сохраняться автоматически при сохранении объекта;
- `status` – это поле отображает статус статьи. Мы использовали параметр `CHOICES`, для того чтобы ограничить возможные значения из указанного списка.

В Django определены различные типы полей, которые мы можем использовать для создания моделей. Полное их описание и примеры использования можно найти на сайте https://docs.djangoproject.com/en/2.0/ref/models/fields/.

Класс `Meta` внутри модели содержит метаданные. Мы указали Django порядок сортировки статей по умолчанию – по убыванию даты публикации, поля `publish`. О том, что порядок убывающий, говорит префикс «-». Таким образом, только что опубликованные статьи будут первыми в списке.

Метод `__str__()` возвращает отображение объекта, понятное человеку. Django использует его во многих случаях, например на сайте администрирования.

Если ранее вы использовали Python 2.X, обратите внимание, что в Python 3 все строки рассматриваются как Юникод-строки, поэтому мы используем только метод `__str__()`. Метод `__unicode__()` устарел.

Активация приложения

Для того чтобы Django начал отслеживать наше приложение и создал таблицы в базе данных для моделей, необходимо активировать его. Для этого отредактируйте `settings.py` и добавьте `blog.apps.BlogConfig` в настройку `INSTALLED_APPS`. Должно получиться так:

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

Класс `BlogConfig` – это конфигурация приложения. Теперь Django знает, что для этого проекта приложение блога активно, и сможет загрузить его модели.

Создание и применение миграций

Мы создали модель данных для статей блога, теперь нам необходимо создать соответствующую таблицу в базе данных. В Django встроена *подсистема миграций*, которая отслеживает изменения моделей и позволяет транслировать их

в базу данных. Команда `migrate` применяет все миграции для всех приложений из списка `INSTALLED_APPS`.Она изменяет базу данных с учетом текущих моделей и созданных миграций.

Для начала необходимо создать инициализирующую миграцию для модели `Post`. В корневом каталоге проекта выполните следующую команду:

```
python manage.py makemigrations blog
```

Вы должны увидеть такой вывод:

```
Migrations for 'blog':
  blog/migrations/0001_initial.py
    - Create model Post
```

Django только что создал файл `0001_initial.py` в папке `migrations` приложения `blog`. Вы можете открыть его, чтобы посмотреть, как выглядит файл миграции. Объект миграции определяет зависимости с другими миграциями и операции, которые необходимо выполнить для синхронизации базы данных.

Давайте посмотрим на SQL-код, который будет выполнен в базе данных для создания таблицы модели. Команда `sqlmigrate` получает на входе имя миграции и возвращает ее SQL-код, не выполняя его. Следующая команда выведет в консоль SQL нашей первой миграции:

```
python manage.py sqlmigrate blog 0001
```

Вывод должен выглядеть следующим образом:

```
BEGIN;
--
-- Create model Post
--
CREATE TABLE "blog_post" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"title" varchar(250) NOT NULL, "slug" varchar(250) NOT NULL, "body" text NOT
NULL, "publish" datetime NOT NULL, "created" datetime NOT NULL, "updated"
datetime NOT NULL, "status" varchar(10) NOT NULL, "author_id" integer NOT
NULL REFERENCES "auth_user" ("id"));
CREATE INDEX "blog_post_slug_b95473f2" ON "blog_post" ("slug");
CREATE INDEX "blog_post_author_id_dd7a8485" ON "blog_post" ("author_id");
COMMIT;
```

Вывод может отличаться, конкретные команды зависят от используемой в проекте базы данных. Представленный код генерируется для SQLite. Как вы можете заметить, Django формирует имя таблицы, используя строчные названия приложения и модели (`blog_post`), но мы можем переопределить это имя в классе `Meta` модели, используя атрибут `db_table`. Django автоматически создает первичный ключ для каждой модели, но и это можно изменить, указав `primary_key=True` для одного из полей модели. По умолчанию первичным ключом является колонка `id`, которая заполняется целыми числами с автоинкрементом. Эта колонка соответствует полю `id`, которое добавляется автоматически для всех моделей.

Давайте синхронизируем базу данных. Выполните следующую команду для применения миграций:

```
python manage.py migrate
```

Вывод в консоли закончится такой строкой:

```
Applying blog.0001_initial... OK
```

Мы только что применили миграцию для всех приложений, указанных в INSTALLED_APPS, включая blog. После применения миграций база данных полностью соответствует текущему состоянию моделей.

Каждый раз, когда мы будем редактировать models.py, добавляя, удаляя или изменяя поля существующих моделей или добавляя новые модели, мы будем создавать новую миграцию с помощью команды makemigrations. Так Django сможет отслеживать изменения в моделях. После создания миграций нужно применять их командой migrate для синхронизации базы данных.

Создание сайта администрирования

Сначала давайте создадим пользователя для управления сайтом администрирования. Выполните следующую команду:

```
python manage.py createsuperuser
```

Вы увидите запрос на ввод логина, электронной почты и пароля для нового пользователя:

```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: ********
Password (again): ********
Superuser created successfully.
```

Введите данные, и пользователь будет создан.

Сайт администрирования Django

Теперь запустите сервер разработки командой python manage.py runserver и откройте в браузере http://127.0.0.1:8000/admin/. Вы увидите страницу авторизации, как на рис. 1.2:

Войдите, используя введенные на предыдущем шаге данные пользователя. Перед вами появится главная страница сайта администрирования (рис. 1.3).

Модели Group и User, которые вы видели на предыдущем скриншоте, – часть подсистемы аутентификации Django.Они находятся в приложении django.contrib.auth. Если вы кликните на Users, то увидите всех пользователей, созданных на текущий момент. Созданная нами модель Post приложения blog связана с моделью User через поле author.

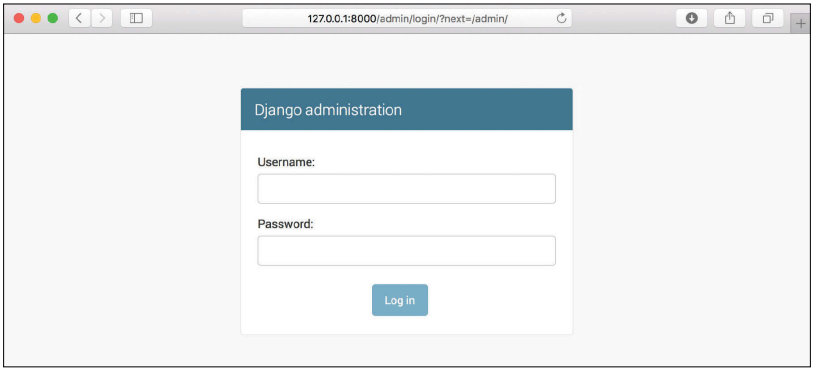

Рис. 1.2 ❖ Страница авторизации на сайте администрирования

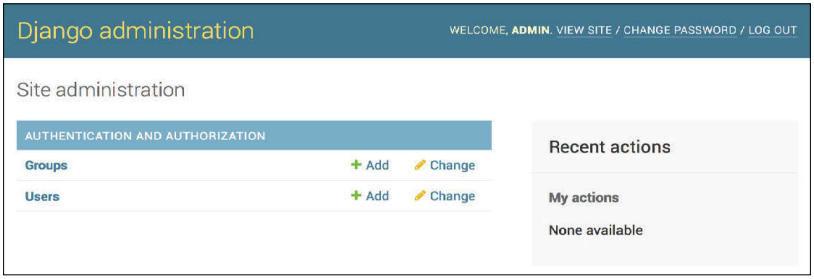

Рис. 1.3 ❖ Главная страница сайта администрирования

Добавление собственных моделей на сайт администрирования

Давайте добавим модели блога на сайт администрирования. Отредактируйте `admin.py` в приложении `blog` таким образом:

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Теперь перезагрузите страницу в браузере, и вы увидите, что модель `Post` добавлена на сайт администрирования (рис. 1.4).

Легко, не правда ли? Регистрируя модель на сайте администрирования Django, мы получаем удобный интерфейс для просмотра, редактирования, создания и удаления объектов.

Кликните на ссылку **Add** напротив пункта **Posts** для создания новой статьи. Обратите внимание на то, что Django автоматически сделал форму для создания и редактирования объектов этой модели (рис. 1.5).

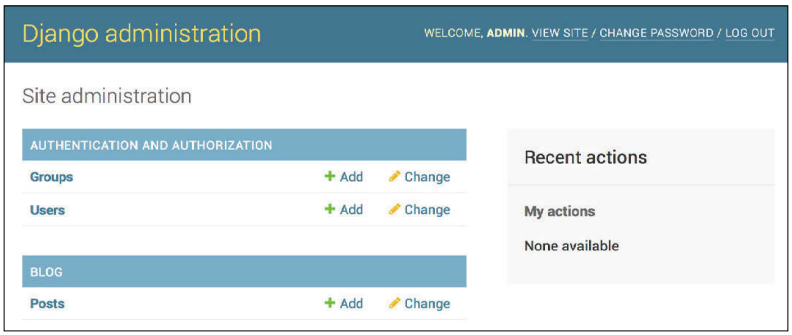

Рис. 1.4 ❖ Собственные модели на сайте администрирования

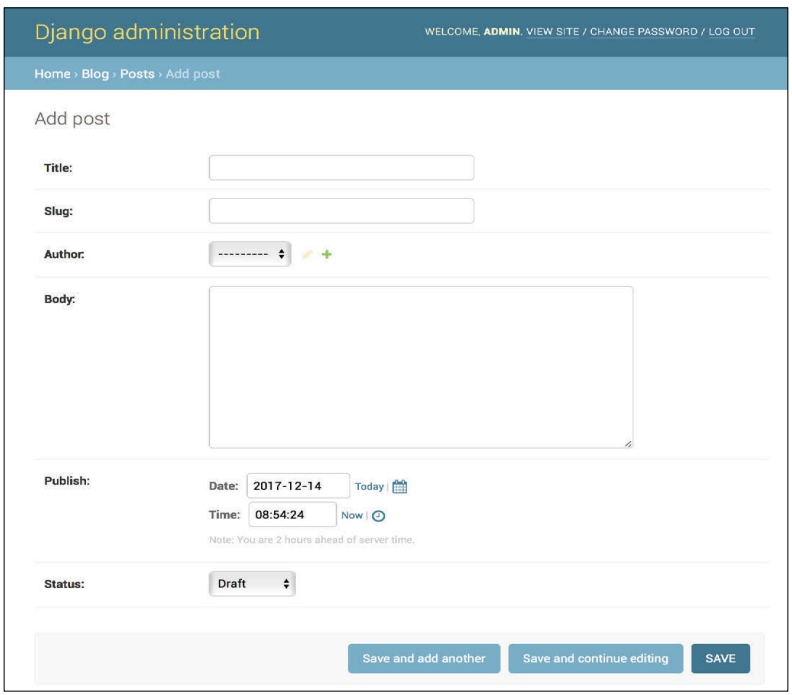

Рис. 1.5 ❖ Форма, созданная автоматически для редактирования статьи

Django использует различные виджеты в формах для каждого типа поля. Даже сложные поля, такие как `DateTimeField`, отображены в виде полей JavaScript, предназначенных для удобной работы с датой и временем.

Заполните форму и нажмите кнопку **SAVE**. Django перенаправит нас на страницу списка статей с сообщением об успешном сохранении статьи, которую мы только что создали (рис. 1.6).

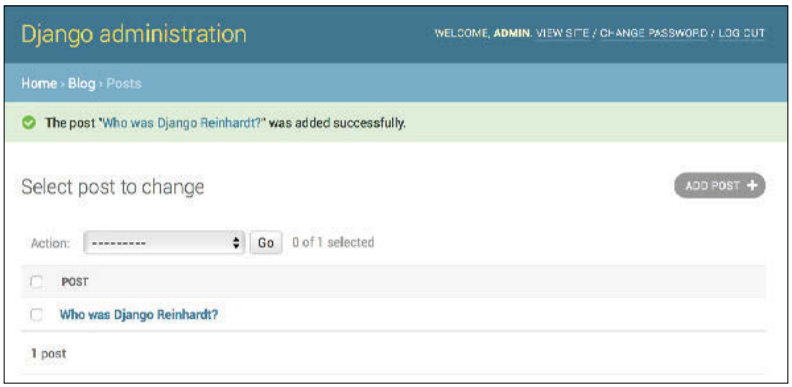

Рис. 1.6 ❖ Список статей

Настройка отображения моделей

Теперь давайте настроим отображение сайта администрирования. Отредактируйте файл `admin.py` приложения блога и добавьте следующий фрагмент:

```
from django.contrib import admin
from .models import Post

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'author', 'publish', 'status')
```

Так мы говорим Django, что наша модель зарегистрирована на сайте администрирования с помощью пользовательского класса, наследника `ModelAdmin`. В нем мы указали, как отображать модель на сайте и как взаимодействовать с ней. Атрибут `list_display` позволяет перечислить поля модели, которые мы хотим отображать на странице списка. Декоратор `@admin.register()` выполняет те же действия, что и функция `admin.site.register()`: регистрирует декорируемый класс – наследник `ModelAdmin`.

Давайте настроим отображение модели, добавив несколько других опций:

```
@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'author', 'publish', 'status')
    list_filter = ('status', 'created', 'publish', 'author')
    search_fields = ('title', 'body')
    prepopulated_fields = {'slug': ('title',)}
    raw_id_fields = ('author',)
    date_hierarchy = 'publish'
    ordering = ('status', 'publish')
```

Вернитесь в браузер и перезагрузите страницу. Теперь она должна выглядеть следующим образом:

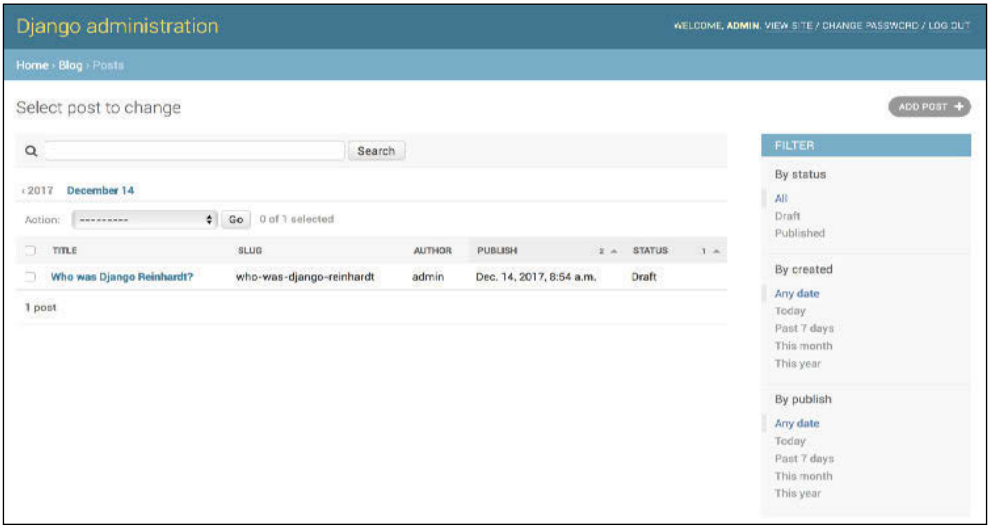

Рис. 1.7 ❖ Настроенная страница списка статей

Вы можете убедиться, что теперь в списке статей отображаются те поля, которые мы указали в атрибуте `list_display`. Справа на странице появился блок фильтрации списка, который фильтрует статьи по полям, перечисленным в `list_filter`. Также появилась строка поиска. Она добавляется для моделей, для которых определен атрибут `search_fields`. Под поиском благодаря атрибуту `date_hierarchy` добавлены ссылки для навигации по датам. По умолчанию статьи отсортированы по полям `status` и `publish`. Эта настройка задается в атрибуте `ordering`.

Теперь кликните на ссылку **Add Post**. Здесь также есть некоторые изменения. Когда вы будете вводить заголовок, обратите внимание, что слаг заполняется автоматически. Мы настроили Django так, что `slug` генерируется автоматически из поля `title` с помощью атрибута `prepopulated_fields`. Также теперь поле `author` содержит поле поиска, что значительно упрощает выбор автора из выпадающего списка, когда в системе сотни пользователей:

Рис. 1.8 ❖ Поле поиска автора в форме редактирования статьи

С помощью всего нескольких строк мы настроили отображение модели на сайте администрирования. Существует несколько способов переопределения и расширения показа моделей, о которых мы узнаем чуть позже.

Работа с QuerySet и менеджерами

Теперь, когда у нас появился полностью настроенный сайт администрирования блога, самое время узнать, как получать информацию из базы данных и взаимодействовать с ней. Django предоставляет мощный API, позволяющий легко создавать, получать, изменять и удалять объекты. *Система объектно-реляционного отображения* Django (Object Relational Mapping – *ORM*) совместима с MySQL, PostgreSQL, SQLite и Oracle. Мы можем определить используемую в проекте СУБД в файле `settings.py` в настройке `DATABASES`. Django одновременно поддерживает работу с несколькими базами данных.

По завершении процесса создания моделей Django предоставляет простой в использовании API для управления ими. Более подробную информацию о данных моделях можно найти на странице официальной документации https://docs.djangoproject.com/en/2.0/ref/models/.

Создание объектов

Откройте терминал и выполните следующую команду в консоли Python:

```
python manage.py shell
```

Затем введите следующие строки кода:

```
>>> from django.contrib.auth.models import User
>>> from blog.models import Post
>>> user = User.objects.get(username='admin')
>>> post = Post(title='Another post',
                slug='another-post',
                body='Post body.',
                author=user)
>>> post.save()
```

Давайте рассмотрим, что делает этот код. Во-первых, мы получаем объект пользователя `user` с логином `admin`:

```
user = User.objects.get(username='admin')
```

Метод `get()` возвращает единственный объект из базы данных. При этом он ожидает, что существует только один объект, подходящий по параметрам. Если база данных не вернет объект, будет выброшено исключение `DoesNotExist`. Если будет найдено несколько подходящих объектов, то Django выбросит исключение `MultipleObjectsReturned`. Оба этих исключения являются атрибутами модели, к которой выполнялся запрос.

Во-вторых, мы создаем объект статьи `Post`, указав заголовок, слаг, контент и автора, полученного на предыдущем шаге:

```
post = Post(title='Another post', slug='another-post', body='Post body.', author=user)
```

На текущий момент этот объект находится только в памяти и пока не сохранен в базу данных.

В-третьих, мы сохраняем статью в базу данных, используя метод `save()`:

```
post.save()
```

Последняя команда формирует SQL-запрос `INSERT` в базу данных. Мы видим, как можно создать объект в памяти и затем отправить его на хранение в базу. Однако мы можем объединить создание и сохранение с помощью метода `create()`:

```
Post.objects.create(title='One more post', slug='one-more-post', body='Post body.',
author=user)
```

Изменение объектов

Давайте попробуем изменить заголовок статьи и сохранить ее:

```
>>> post.title = 'New title'
>>> post.save()
```

В этом примере метод `save()` будет преобразован в SQL-выражение `UPDATE`.

Все изменения, которые мы делаем для объекта в памяти, не применяются в базе данных до тех пор, пока не будет вызван метод `save()`.

Получение объектов

Django ORM основана на объектах запросов QuerySet. *QuerySet* – это коллекция объектов, полученных из базы данных. К ней могут быть применены фильтрация и сортировка. Каждая модель Django имеет как минимум один менеджер модели, по умолчанию называемый `objects`. С его помощью мы получаем объект запроса QuerySet. Для того чтобы получить все объекты из таблицы, мы можем использовать метод `all()` стандартного менеджера:

```
>>> all_posts = Post.objects.all()
```

Так мы создадим запрос, который вернет все объекты статей из базы данных. Отметим, что на текущий момент SQL-запрос в базу данных не выполнялся. В Django объекты запросов *ленивы*. Они выполняются только тогда, когда поступает непосредственное обращение к элементам из QuerySet. Такое поведение делает QuerySet'ы очень эффективными. Если вместо присвоения QuerySet'а переменной мы выведем его в консоль, то SQL-запрос выполнится, потому что для вывода нужно получить значения из базы данных:

```
>>> Post.objects.all()
```

Использование метода filter()

Для фильтрации выборки вы можете использовать метод менеджера `filter()`. Например, мы можем получить все статьи, опубликованные в 2017 г.:

```
Post.objects.filter(publish__year=2017)
```

Мы можем осуществить фильтрацию по нескольким полям. Например, для получения всех статей, опубликованных в 2017 г. пользователем с логином admin, выполните:

```
Post.objects.filter(publish__year=2017, author__username='admin')
```

Этот же запрос можно сформировать, выстроив фильтры по полям в цепочку:

```
Post.objects.filter(publish__year=2017) \
            .filter(author__username='admin')
```

Условия фильтрации строятся с использованием двойного подчеркивания, например publish__year. Такой же способ записи применяется и для доступа к полям связанных объектов, например author__username.

Использование метода exclude()

Кроме фильтрации, мы можем исключать некоторые записи из QuerySet'а с помощью метода exclude() менеджера модели. Например, мы можем получить все опубликованные в 2017 г. статьи, у которых заголовок не начинается с Why:

```
Post.objects.filter(publish__year=2017) \
            .exclude(title__startswith='Why')
```

Использование order_by()

Для сортировки результата запроса по разным полям мы можем использовать метод order_by() менеджера модели. Например, возможно получить статьи, отсортированные по заголовку в алфавитном порядке:

```
Post.objects.order_by('title')
```

Можно сортировать и в обратном порядке, для чего нужно добавить префикс:

```
Post.objects.order_by('-title')
```

Удаление объектов

Если вы хотите удалить объект, вызовите его метод delete():

```
post = Post.objects.get(id=1)
post.delete()
```

Отметим, что удаление объекта также удаляет зависимые от него объекты, определенные через ForeignKey и имеющие параметр on_delete, равный CASCADE.

Когда выполняются запросы QuerySet'ов

Мы можем добавить сколько угодно фильтров в объект запроса, но непосредственно выполнение SQL-запроса произойдет тогда, когда запустится вычисление QuerySet'а. QuerySet выполняется только в этих случаях:

- первая итерация по коллекции QuerySet'а;
- когда мы делаем срез по коллекции, например Post.objects.all()[:3];

- при сериализации или кешировании;
- при вызове методов `repr()` или `len()`;
- когда мы явно вызываем функцию `list()`, передавая ее аргументом QuerySet;
- при использовании QuerySet в логических выражениях, таких как `bool()`, `or`, `and`, `if`.

Создание менеджера модели

Как мы ранее заметили, `objects` – менеджер модели по умолчанию. Он возвращает все объекты из базы. Однако мы можем создать свой менеджер. Давайте реализуем собственный менеджер для получения всех опубликованных статей.

Существуют два способа добавления собственного менеджера для модели: указать дополнительные методы или заменить менеджер модели. В первом случае мы получим расширенный API QuerySet'а – `Post.objects.my_manager()`, а во втором – `Post.my_manager.all()`. Созданный менеджер позволит нам получать статьи, используя запись `Post.published.all()`.

Отредактируйте файл `models.py` приложения `blog`, чтобы добавить свой менеджер:

```
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(status='published')

class Post(models.Model):
    # ...
    objects = models.Manager()     # Менеджер по умолчанию.
    published = PublishedManager() # Наш новый менеджер.
```

Метод менеджера `get_queryset()` возвращает QuerySet, который будет выполняться. Мы переопределили его и добавили фильтр над результирующим QuerySet'ом. Также мы описали менеджер и добавили его в модель `Post`.Теперь мы можем использовать его для выполнения запросов. Давайте попробуем это сделать.

Запустите сервер разработки и выполните команду:

```
python manage.py shell
```

С помощью следующей команды мы получим все опубликованные статьи, название которых начинается с `Who`:

```
Post.published.filter(title__startswith='Who')
```

Обработчики списка статей и страницы подробностей

Теперь у нас есть базовые знания об использовании ORM, и мы готовы к созданию страниц блога. *Обработчики Django* – это простая Python-функция, которая получает веб-запрос и возвращает веб-ответ. Вся логика, формирующая желаемый ответ, описывается внутри этой функции.

Для начала мы создадим обработчики, затем определим для них шаблоны URL'ов и, наконец, сделаем HTML-шаблоны для отображения результатов обработки. Каждый обработчик генерирует шаблон, используя переменные контекста, и возвращает HTTP-ответ со сформированной HTML-страницей.

Создание обработчиков списка и страницы подробностей

Давайте начнем с создания обработчика для отображения списка статей. Добавьте в `views.py` приложения `blog` следующий фрагмент кода:

```
from django.shortcuts import render, get_object_or_404
from .models import Post

def post_list(request):
    posts = Post.published.all()
    return render(request, 'blog/post/list.html', {'posts': posts})
```

Мы только что создали первый обработчик – `post_list`. Он получает объект `request` в качестве аргумента и является обязательным для всех обработчиков. В этой функции мы запрашиваем из базы данных все опубликованные статьи с помощью менеджера `published`.

После этого мы используем функцию `render()` для формирования шаблона со списком статей. Она принимает объект запроса `request`, путь к шаблону и переменные контекста для этого шаблона. В ответ вернется объект `HttpResponse` со сформированным текстом (обычно это HTML-код). Функция `render()` использует переданный контекст при формировании шаблона, поэтому любая переменная контекста будет доступна в шаблоне. *Процессоры контекста* – это вызываемые функции, которые добавляют в контекст переменные. Более подробно мы познакомимся с ними в главе 3 «Расширение приложения блога».

Давайте добавим второй обработчик для отображения статьи. Допишите следующий фрагмент в `views.py`:

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post, slug=post, status='published', publish__year=year,
                                publish__month=month, publish__day=day)
    return render(request,'blog/post/detail.html', {'post': post})
```

Это обработчик страницы статьи. Он принимает на вход аргументы `year`, `month`, `day` и `post` для получения статьи по указанным слагу и дате. Обратите внимание на то, что когда мы создали модель `Post`, у нее был указан атрибут `unique_for_date` для поля `slug`. Таким образом мы добавили ограничение, чтобы слаг был уникальным для статей, созданных в один день. Поэтому гарантированно сможем получить статью по комбинации этих полей. В обработчике мы используем `get_object_or_404()`, для того чтобы найти нужную статью. Эта функция возвращает объект, который подходит по указанным параметрам, или вызывает исключение HTTP 404 (объект не найден), если не найдет ни одной статьи. В конце мы используем функцию `render()` для формирования HTML-шаблона.

Добавление шаблонов URL'ов для обработчиков

Шаблоны URL'ов позволяют сопоставить адреса с обработчиками. Шаблон представляет собой комбинацию из строки, описывающей адрес, обработчика и необязательного названия, которое даст возможность обращаться к этому шаблону на всех уровнях проекта. Django проходит по порядку по всем шаблонам, пока не найдет первый подходящий, т. е. совпадающий с URL'ом запроса. Затем Django сможет импортировать соответствующий обработчик и выполнить его, передав внутрь объект запроса `HttpRequest` и ключевые слова или позиционные аргументы.

Создайте `urls.py` в папке приложения `blog` и добавьте в него следующий код:

```
from django.urls import path
from . import views

app_name = 'blog'

urlpatterns = [
    # post views
    path('', views.post_list, name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
          views.post_detail, name='post_detail'),
]
```

Мы определили пространство имен приложения в переменной `app_name`. Это позволит нам сгруппировать адреса для приложения блога и использовать их названия для доступа к ним. Мы объявили два шаблона, используя функцию `path()`. Первый шаблон не принимает никаких аргументов. Он сопоставляется с обработчиком `post_list`. Второй вызывает функцию `post_detail` и принимает в качестве параметров следующие:

- `year` – целое число, задающее год публикации статьи;
- `month` – целое число, задающее месяц;
- `day` – целое число, представляющее день публикации;
- `post` – строка, которая может содержать буквы, цифры и дефисы или нижние подчеркивания.

Мы использовали треугольные скобки для извлечения значений из URL'а. Любое значение, определенное в шаблоне как `<parameter>`, возвращается в виде строки. Мы используем конвертер, например `<int:year>`, чтобы явно указать, что год должен быть извлечен из адреса в виде целого числа; `<slug:post>` – слаг будет извлечен как строка, которая может содержать только буквы, цифры и дефисы с нижними подчеркиваниями (в соответствии со стандартом ASCII). Вы можете найти функции преобразования, предоставляемые Django на странице https://docs.djangoproject.com/en/2.0/topics/http/urls/#path-converters.

Если использование `path()` и конвертеров не подходит, можно задействовать `re_path()`. Эта функция позволяет задавать шаблоны URL'ов в виде регулярных выражений. Более подробно об этом написано на странице https://docs.djangoproject.com/en/2.0/ref/urls/#django.urls.re_path. Если раньше вы не работали с регулярными выражениями, рекомендуем познакомиться с ними на https://docs.python.org/3/howto/regex.html.