Лабораторная работа 1. Разработка приложения для ведения блога

Установка языка Python

Django 4.1 поддерживает язык Python версий 3.8, 3.9 и 3.10. В приведенных в этой книге примерах мы будем использовать Python 3.10.6. Если вы применяете Linux или macOS, то у вас, вероятно, Python уже установлен. Если же вы используете Windows, то на странице https://www.python.org/downloads/windows/ можно скачать установщик Python. Откройте оболочку командной строки своего компьютера. Если вы используете macOS, то откройте каталог /Applications/Utilities в файловом менеджере **Finder**, затем дважды кликните по **Terminal**. Если вы используете Windows, то откройте меню **Пуск** и в поле поиска наберите cmd. Затем кликните по приложению **командной строки**, чтобы его открыть.
Проверьте, что на вашем компьютере Python установлен, набрав следующую ниже команду в командной оболочке:

```
python
```

Если вы видите что-то вроде следующего ниже, то Python на вашем компьютере установлен:

```
Python 3.10.6 (v3.10.6:9c7b4bd164, Aug  1 2022, 17:13:48) [Clang 13.0.0
(clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

Если установленная версия Python ниже 3.10 или если Python на вашем компьютере не установлен, то скачайте Python 3.10.6 на странице https://www.python.org/downloads/ и следуйте инструкциям по его установке. На странице скачиваний находится установщик Python для Windows, macOS и Linux.
На протяжении всей этой лабораторной работы, когда в командной оболочке упоминается Python, мы будем использовать команду python, хотя в некоторых системах, возможно, потребуется применение команды python3. Если вы работаете на Linux или macOS, а в вашей системе используется Python 2, то для того, чтобы задействовать установленную вами версию Python 3, вам нужно будет использовать команду python3

В Windows команда python представляет исполняемый файл установки Python, которая используется по умолчанию, тогда как команда py – программу быстрого запуска языка Python. Программа быстрого запуска языка Python для Windows была впервые представлена в Python версии 3.3. Она выясняет версии языка Python, которые были установлены на вашем компьютере, и автоматически переключается на последнюю версию. Если вы работаете с Windows, то рекомендуется заменить команду python командой py.

Создание виртуальной среды Python

При написании приложений на Python обычно используются пакеты и модули, которые не включены в стандартную библиотеку Python. При этом некоторым приложениям Python может требоваться другая версия одного и того же модуля. Однако в масштабах всей системы можно устанавливать только определенную версию модуля. Если обновить версию модуля приложения, то это может привести к нарушению работы других приложений, которым требуется более старая версия этого модуля.

В целях решения указанной проблемы используются виртуальные среды Python. С помощью виртуальных сред модули Python можно устанавливать в изолированном месте, не устанавливая их глобально. Каждая виртуальная среда имеет свой собственный двоичный файл Python и свой собственный независимый набор пакетов Python, расположенных в каталоге *site-packages*.

Начиная с версии 3.3 Python идет в комплекте с библиотекой *venv*, которая обеспечивает поддержку создания облегченных виртуальных сред. Применяя модуль Python *venv* для создания изолированных сред Python, можно использовать разные версии пакетов для разных проектов. Еще одним преимуществом работы с *venv* является то, что для установки пакетов Python не понадобятся какие-либо административные привилегии.

Если вы работаете с Linux или macOS, то следующей ниже командой создайте изолированную среду:

```
python -m venv my_env
```

В случае если ваша система идет в комплекте с Python 2 и вы установили Python 3, то не забудьте применить команду python3 вместо python. Если вы используете Windows, то вместо этого примените следующую ниже команду:

```
py -m venv my_env
```

При этом будет использоваться программа быстрого запуска Python в Windows. Приведенная выше команда создаст среду Python в новом каталоге с именем my_env/. Любые библиотеки Python, которые устанавливаются вами, пока ваша виртуальная среда является активной, будут помещаться в каталог *my_env/lib/python3.10/site-packages*.

Если вы используете Linux или macOS, то выполните следующую ниже команду, чтобы активировать свою виртуальную среду:

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

Если вы используете Windows, то вместо этого привлеките следующую ниже команду:

```
.\my_env\Scripts\activate
```

Приглашение командной оболочки будет содержать имя активной виртуальной среды, заключенное в круглые скобки, как показано ниже:

```
(my_env) zenx@pc:~ zenx$
```

Свою среду можно деактивировать в любое время с помощью команды deactivate.

Установка веб-фреймворка Django

Если вы уже установили Django 4.1, то этот раздел можно пропустить и перейти непосредственно к разделу «Создание первого проекта». Django поставляется в виде модуля Python, и, следовательно, его можно устанавливать в любой среде Python. Если вы еще не установили Django, то ниже приведено краткое руководство по его установке на компьютер.

Установка Django с помощью pip

Система управления пакетами *pip* является предпочтительным методом установки *Django*. Python 3.10 идет в комплекте с предустановленной pip, однако инструкция по установке *pip* находится на странице *https://pip.pypa.io/en/stable/installing/*.

Выполните следующую ниже команду в командной оболочке, чтобы установить *Django* с помощью pip:

```
pip install Django~=4.1.0
```

Она установит последнюю версию *Django* 4.1 в каталог Python *site-packages/* вашей виртуальной среды.

Теперь мы проверим успешность установки *Django*. Выполните следующую ниже команду в командной оболочке:

```
python -m django --version
```

Если вы получите результат 4.1.X, то, значит, Django был успешно установлен на вашем компьютере. Если вы получаете сообщение No module named Django, то, значит, Django на вашем компьютере не установлен.

Создание первого проекта

Ваш первый проект Django будет состоять из приложения для ведения блога. Мы начнем с создания проекта Django и приложения Django для ведения блога. Затем создадим модели данных и синхронизируем их с базой данных. Django предоставляет команду, которая позволяет создавать изначальную файловую структуру проекта. Выполните следующую ниже команду в командной оболочке:

```
django-admin startproject mysite
```

Она создаст проект Django с именем mysite.

Давайте взглянем на сгенерированную структуру проекта:

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

Внешний каталог mysite/ является контейнером проекта. Он содержит следующие ниже файлы:

- manage.py: это утилита командной строки, используемая для взаимодействия с проектом. Редактировать этот файл не требуется;
- mysite/: это пакет проекта на языке Python; пакет состоит из следующих ниже файлов:

– _ _init__.py: пустой файл, который сообщает Python, что каталог mysite нужно трактовать как модуль Python;
– asgi.py: конфигурация для выполнения проекта в качестве приложения, работающего по протоколу интерфейса шлюза асинхронного сервера (ASGI) с ASGI-совместимыми веб-серверами. ASGI – это новый стандарт Python для асинхронных веб-серверов и приложений;
– settings.py: здесь указаны настроечные параметры и конфигурация проекта и содержатся изначальные параметры со значениями, используемыми по умолчанию;
– urls.py: место, где располагаются ваши шаблоны URL-адресов. Каждый URL-адрес, который определен здесь, соотносится с представлением;
– wsgi.py: конфигурация для выполнения проекта в качестве приложения, работающего по протоколу интерфейса шлюза веб-сервера (WSGI)1 с WSGI-совместимыми веб-серверами.

Применение первоначальных миграций базы данных

Для того чтобы хранить данные, приложениям Django требуется база данных. Упомянутый выше файл settings.py содержит конфигурацию базы данных проекта в настроечном параметре DATABASES. Изначально конфигурацией предусматривается использование базы данных SQLite3, если не указана иная. SQLite идет в комплекте с Python 3 и может применяться в любом приложении Python. SQLite – это облегченная база данных, которую можно использовать с Django для разработки. Если вы планируете развернуть свое приложение в производственной среде, то вам следует использовать полнофункциональную базу данных, такую как PostgreSQL, MySQL или Oracle.

Файл settings.py также содержит настроечный параметр INSTALLED_APPS со списком, содержащим распространенные приложения Django, которые добавляются в ваш проект по умолчанию. Мы рассмотрим эти приложения позже в разделе «Настроечные параметры проекта».
Приложения Django содержат модели данных, которые соотносятся с таблицами базы данных. В разделе «Создание моделей данных блога» вы создадите свои собственные модели. Для того чтобы завершить настройку проекта, необходимо создать таблицы, ассоциированные с моделями стандартных приложений Django, включенных в состав параметра INSTALLED_APPS. Django поставляется вместе с системой, которая помогает управлять миграциями баз данных.
Откройте приглашение командной оболочки и выполните следующие ниже команды:

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

Вы увидите результат, который заканчивается следующими ниже строками:

```
(base) C:\Users\asus\mysite>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... 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 auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Показанные выше строки – это применяемые веб-фреймворком Django миграции базы данных. В результате применения изначальных настроек в базе данных создаются таблицы для приложений, перечисленных в настроечном параметре INSTALLED_APPS.

Запуск и выполнение сервера разработки

Django идет в комплекте вместе с облегченным веб-сервером с целью быстрого выполнения вашего исходного кода без необходимости тратить время на настройку производственного сервера. Во время работы сервера разработки он непрерывно проверяет наличие изменений в исходном коде. Он автоматически перезагружается, освобождая от необходимости перезагружать его вручную после изменения кода. Однако есть случаи, когда он может не замечать некоторые действия, такие как добавление новых файлов в проект, поэтому в подобных случаях приходится перезапускать сервер вручную. Запустите сервер разработки, набрав следующую ниже команду в командной оболочке:

```
python manage.py runserver
```

Вы должны увидеть что-то вроде этого:

```
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).
January 01, 2022 - 10:00:00
Django version 4.0, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Теперь пройдите по URL-адресу http://127.0.0.1:8000/ в своем браузере. Вы должны увидеть страницу, на которой указано, что проект успешно выполняется, как показано на рис. 1.2.

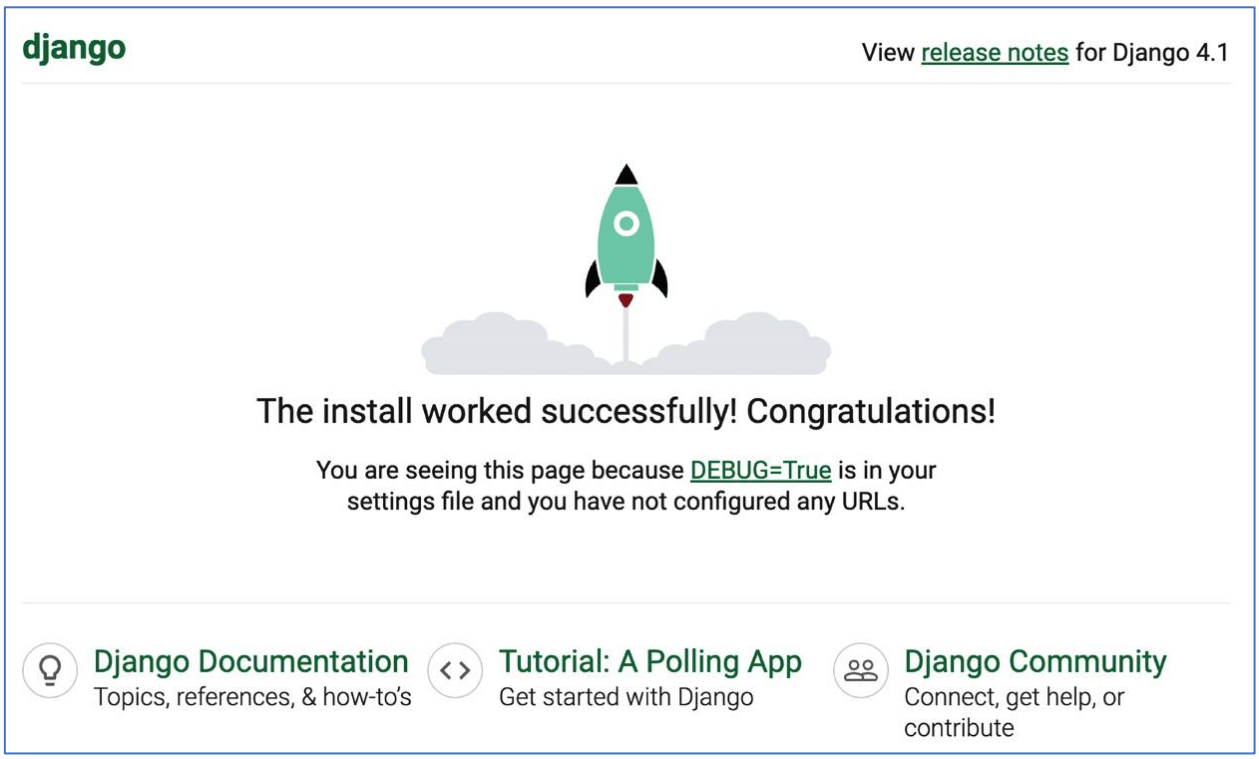

Рис. 1.2. Домашняя страница сервера разработки

Приведенный выше снимок экрана показывает, что Django работает. Если вы взглянете на свою консоль, то увидите выполняемый браузером запрос GET:

```
[01/Jan/2022 17:20:30] "GET / HTTP/1.1" 200 16351
```

Каждый HTTP-запрос регистрируется в консоли сервером разработки. Любая ошибка, возникающая во время работы сервера разработки, также будет появляться в консоли. Сервер разработки можно выполнять на конкретно-прикладном хосте и порту либо сообщать Django, что нужно загружать определенный настроечный файл, как показано ниже:

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

Когда приходится иметь дело с несколькими средами, требующими разных конфигураций, то следует создавать настроечный файл отдельно для каждой среды.

Этот сервер предназначен только для разработки и не подходит для производственного использования. Для того чтобы развернуть Django в производственной среде, необходимо его выполнять как приложение на основе WSGI с использованием такого веб-сервера, как Apache, Gunicorn или uWSGI, или же как приложение на основе ASGI с использованием такого сервера, как Daphne или Uvicorn.

Настроечные параметры проекта

Давайте откроем файл settings.py и взглянем на конфигурацию проекта. Несколько настроечных параметров уже внесены в указанный файл веб-фреймворком Django, но это лишь часть всех имеющихся параметров.

Давайте рассмотрим некоторые настроечные параметры проекта.

- DEBUG – это булев параметр, который включает и выключает режим отладки проекта. Если его значение установлено равным *True*, то *Django* будет отображать подробные страницы ошибок в случаях, когда приложение выдает неперехваченное исключение. При переходе в производственную среду следует помнить о том, что необходимо устанавливать его значение равным False. Никогда не развертывайте свой сайт в производственной среде с включенной отладкой, поскольку вы предоставите конфиденциальные данные, связанные с проектом.
- ALLOWED_HOSTS не применяется при включенном режиме отладки или привыполнении тестов. При перенесении своего сайта в производственную среду и установке параметра *DEBUG* равным *False* в этот настроечный параметр следует добавлять свои домен/хост, чтобы разрешить ему раздавать ваш сайт *Django*.
- INSTALLED_APPS – это параметр, который придется редактировать во всех проектах. Он сообщает Django о приложениях, которые для этого сайта являются активными. По умолчанию Django вставляет следующие ниже приложения:
  - *django.contrib.admin*: сайт администрирования;
  - *django.contrib.auth:* фреймворк аутентификации;
  - *django.contrib.contenttypes:* фреймворк типов контента;
  - *django.contrib.sessions*: фреймворк сеансов;
  - *django.contrib.messages:* фреймворк сообщений;
  - *django.contrib.staticfiles:* фреймворк управления статическими файлами.

- MIDDLEWARE – подлежащие исполнению промежуточные программные компоненты.
- ROOT_URLCONF указывает модуль Python, в котором определены шаблоны корневых *URL*-адресов приложения.
- DATABASES – словарь, содержащий настроечные параметры всех баз данных, которые будут использоваться в проекте. Всегда должна существовать база данных, которая будет использоваться по умолчанию.
- В стандартной конфигурации используется база данных *SQLite3*, если не указана иная.
- LANGUAGE_CODE определяет заранее заданный языковой код этого сайта Django.
- USE_TZ сообщает Django, что нужно активировать/деактивировать поддержку часовых поясов. Django поставляется вместе с поддержкой дат и времен с учетом часовых поясов. Этот настроечный параметр получает значение True при создании нового проекта с помощью команды управления *startproject*.

Не волнуйтесь, если вы многое не понимаете из того, что здесь видите.

Создание приложения

Давайте создадим первое приложение Django. Мы разработаем приложение для ведения блога с нуля. Выполните следующую ниже команду в командной оболочке из корневого каталога проекта:

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

Она создаст базовую структуру приложения, которая будет выглядеть следующим образом:

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

Ниже приведено описание этих файлов:

- *__init__.py*: пустой файл, который сообщает Python, что каталог *blog* нужно трактовать как модуль Python;
- *admin.py*: здесь вы регистрируете модели, чтобы включать их в состав сайта администрирования – этот сайт используется опционально, по вашему выбору;

- *apps.py*: содержит главную конфигурацию приложения *blog*;
- *migrations*: этот каталог будет содержать миграции базы данных приложения. Миграции позволяют *Django* отслеживать изменения модели ный каталог содержит пустой файл _ _*init*_ _.*py*;
- *models.py*: содержит относимые к приложению модели данных; все приложения *Django* должны иметь файл *models.py*, но его можно оставлять пустым;
- *tests.py*: здесь можно добавлять относимые к приложению тесты;
- *views.py*: здесь расположена логика приложения; каждое представление получает *HTTP*-запрос, обрабатывает его и возвращает ответ. Когда структура приложения готова, можно приступать к разработке моделей данных блога.

Создание моделей данных блога

- Напомним, что объект *Python* – это набор данных и методов. Класс – это концептуальная схема, которая объединяет данные и функциональности в единое целое. Создание нового класса влечет новый тип объекта, позволяя формировать экземпляры этого типа.
- Модель *Django* – это источник информации и поведения данных. Она состоит из класса *Python*, который является подклассом *django.db.models.Model*.
- Каждой модели ставится в соответствие одна таблица базы данных, где каждый атрибут класса соотносится с полем базы данных. Когда вы будете создавать модель, *Django* будет предоставлять практичный *API*, чтобы легко запрашивать объекты в базе данных.
- Сначала мы определим модели баз данных для приложения *blog*. Затем сгенерируем для этих моделей миграции базы данных, чтобы создать соответствующие таблицы базы данных. При применении миграций *Django* будет создавать таблицу по каждой модели, определенной в файле *models.py* приложения.

Создание модели поста

- Сначала мы определим модель *Post*, которая позволит хранить посты блога в базе данных.
- Добавьте следующие ниже строки в файл *models.py* приложения *blog*. Новые строки выделены жирным шрифтом:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
```

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()

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

Это модель данных для постов блога. Посты будут иметь заголовок, короткую метку под названием slug и тело поста. Давайте взглянем на поля указанной модели:

- *title*: поле заголовка поста. Это поле с типом *CharField*, которое транслируется в столбец *VARCHAR* в базе данных *SQL*;
- *slug*: поле *SlugField*, которое транслируется в столбец *VARCHAR* в базе данных *SQL*. *Слаг* – это короткая метка, содержащая только буквы, цифры, знаки подчеркивания или дефисы. Пост с заголовком «Django Reinhardt: A legend of Jazz» мог бы содержать такой заголовок: «django-reinhardtlegend-jazz»;
- *body*: поле для хранения тела поста. Это поле с типом *TextField*, которое транслируется в столбец *Text* в базе данных *SQL*.

В модельный класс также добавлен метод _ _*str*_ _*()*. Это метод Python, который применяется по умолчанию и возвращает строковый литерал с удобочитаемым представлением объекта. Django будет использовать этот метод для отображения имени объекта во многих местах, таких как его сайт администрирования.

Давайте посмотрим, как модель и ее поля будут транслированы в таблицу и столбцы базы данных. На следующей ниже диаграмме показана модель *Post* и соответствующая таблица базы данных, которую *Django* создаст, когда мы синхронизируем модель с базой данных.

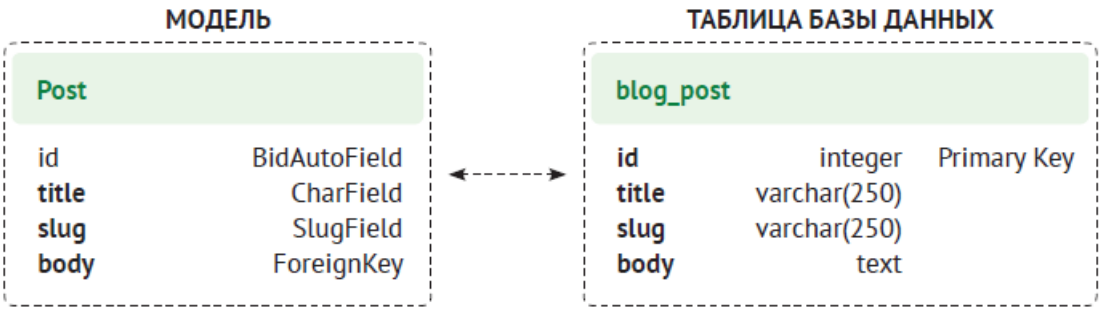

Рис. 1.4. Соответствие изначальной модели Post и таблицы базы данных

Django создаст столбец базы данных для каждого поля модели: *title*, *slug* и *body*. На рисунке хорошо видно, как каждый тип поля соответствует типу данных в базе данных. Django по умолчанию добавляет поле автоматически увеличивающего ся первичного ключа в каждую модель. Тип этого поля указывается в конфигурации каждого приложения либо глобально в настроечном параметре *DEFAULT_AUTO_FIELD*. При создании приложения командой *startapp* значение параметра *DEFAULT_AUTO_FIELD* по умолчанию имеет тип *BigAutoField*. Это 64-битное целое число, которое увеличивается автоматически в соответствии с доступными идентификаторами. Если не указывать первичный ключ своей модели, то *Django* будет добавлять это поле автоматически. В качестве первичного ключа можно также определить одно из полей модели, установив для него параметр *primary_key=True*.
Мы расширим модель *Post* дополнительными полями и поведением. После завершения мы синхронизируем ее с базой данных, создав миграцию в базе данных и применив ее.

Добавление полей даты/времени

Мы продолжим, добавив в модель *Post* различные поля даты/времени. Каждый пост будет публиковаться в определенную дату и время. Следовательно, необходимо иметь поле для хранения даты и времени публикации. Мы также хотим хранить дату и время создания объекта *Post* и его последнего изменения.
Отредактируйте файл models.py приложения *blog*, придав ему следующий вид. Новые строки выделены жирным шрифтом:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

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

В модель Post были добавлены следующие ниже поля:

- publish: поле с типом *DateTimeField*, которое транслируется в столбец *DATETIME* в базе данных *SQL*. Оно будет использоваться для хранения даты и времени публикации поста. По умолчанию значения поля за-даются методом *Django*

timezone.now. Обратите внимание, что для того, чтобы использовать этот метод, был импортирован модуль *timezone*.

- Метод *timezone.now* возвращает текущую дату/время в формате, зависящем от часового пояса. Его можно трактовать как версию стандартного метода *Python datetime.now* с учетом часового пояса;
- *created*: поле с типом *DateTimeField*. Оно будет использоваться для хранения даты и времени создания поста. При применении параметра *auto_now_add* дата будет сохраняться автоматически во время создания объекта;
- *updated*: поле с типом *DateTimeField*. Оно будет использоваться для хранения последней даты и времени обновления поста. При применении параметра *auto_now* дата будет обновляться автоматически во время сохранения объекта.
-

Определение предустановленного порядка сортировки

Посты блога обычно отображаются на странице в обратном хронологическом порядке (от самых новых к самым старым). В нашей модели мы определим заранее заданный порядок. Он будет применяться при извлечении объектов из базы данных, в случае если в запросе порядок не будет указан. Отредактируйте файл *models.py* приложения *blog*, придав ему следующий вид. Новые строки выделены жирным шрифтом:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['-publish']

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

Внутрь модели был добавлен Meta-класс. Этот класс определяет метаданные модели. Мы используем атрибут ordering, сообщающий Django, что он должен сортировать результаты по полю publish. Указанный порядок будет применяться по умолчанию для запросов к базе данных, когда в запросе не указан конкретный порядок. Убывающий порядок задается с помощью дефиса

перед именем поля: -publish. По умолчанию посты будут возвращаться в обратном хронологическом порядке.

Добавление индекса базы данных

Давайте определим индекс базы данных по полю *publish*. Индекс повысит производительность запросов, фильтрующих или упорядочивающих результаты по указанному полю. Мы ожидаем, что многие запросы извлекут преимущества из этого индекса, поскольку для упорядочивания результатов мы по умолчанию используем поле *publish*.
Отредактируйте файл *models.py* приложения *blog*, придав ему следующий вид. Новые строки выделены жирным шрифтом:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

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

В Meta-класс модели была добавлена опция *indexes*. Указанная опция позволяет определять в модели индексы базы данных, которые могут содержать одно или несколько полей в возрастающем либо убывающем порядке, или функциональные выражения и функции базы данных. Был добавлен индекс по полю *publish*, а перед именем поля применен дефис, чтобы определить индекс в убывающем порядке. Создание этого индекса будет вставляться в миграции базы данных, которую мы сгенерируем позже для моделей блога.

Индексное упорядочивание в MySQL не поддерживается. Если в качестве базы данных вы используете MySQL, то убывающий индекс будет создаваться как обычный индекс.

Активация приложения

Теперь необходимо активировать приложение *blog* в проекте, чтобы *Django* мог отслеживать приложение и имел возможность создавать таблицы базы данных для его моделей.

Отредактируйте файл *settings.py*, добавив *blog.apps.BlogConfig* в настроечный параметр *INSTALLED_APPS*. Это должно выглядеть, как показано ниже. Новые строки выделены жирным шрифтом:

```
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 является активным, и сможет загружать модели приложения.

Добавление поля статуса

Очень часто в функциональность ведения блогов входит хранение постов в виде черновика до тех пор, пока они не будут готовы к публикации. Мы добавим в модель поле статуса, которое позволит управлять статусом постов лога. В постах будут использоваться статусы *Draft* (Черновик) и *Published* (Опубликован).

Отредактируйте файл *models.py* приложения *blog*, придав ему следующий вид. Новые строки выделены жирным шрифтом:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
        PUBLISHED = 'PB', 'Published'
```

```
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    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=2,
                              choices=Status.choices,
                              default=Status.DRAFT)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

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

Мы определили перечисляемый класс *Status* путем подклассирования класса *models.TextChoices*. Доступными вариантами статуса поста являются *DRAFT* и *PUBLISHED*. Их соответствующими значениями выступают *DF* и *PB*, а их метками или читаемыми именами являются *Draft* и *Published*.
Django предоставляет перечисляемые типы, которые можно подклассировать, чтобы легко и просто определять варианты выбора. Они основаны на объекте *enum* стандартной библиотеки Python. Подробнее об *enum* можно почитать на странице https://docs.python.org/3/library/enum.html.

Перечисляемые типы Django имеют несколько видоизменений по сравнению с *enum*. Об этих различиях можно узнать по адресу https://docs.djangoproject.com/en/4.1/ref/models/fields/#enumeration-types.
Для того чтобы получать имеющиеся варианты, можно обращаться к вариантам статуса (*Post.Status.choices*), для того чтобы получать удобочитаемые имена – к меткам статуса (*Post.Status.labels*), и для того чтобы получать фактические значения вариантов – к значениям статуса (*Post.Status.values*).
В модель также было добавлено новое поле *status*, являющееся экземпляром типа *CharField*. Оно содержит параметр choices, чтобы ограничивать значение поля вариантами из *Status.choices*. Кроме того, применяя параметр *default*, задано значение поля, которое будет использоваться по умолчанию. В этом поле статус *DRAFT* используется в качестве предустановленного варианта, если не указан иной.

На практике неплохая идея – определять варианты внутри модельного класса и использовать перечисляемые типы. Такой подход будет позволять легко ссылаться на метки вариантов, значения или имена из любого места исходного кода. При этом можно импортировать модель Post и использовать Post.Status.DRAFT в качестве ссылки на статус *Draft* в любом месте своего исходного кода.

Давайте посмотрим, как взаимодействовать с вариантами статуса. Выполните следующую ниже команду в командной оболочке, чтобы открыть оболочку Python:

```
python manage.py shell
```

Затем наберите такие строки:

```
>>> from blog.models import Post
>>> Post.Status.choices
```

Вы получите варианты перечисления в формате пар значение–метка, подобные показанным ниже:

```
[('DF', 'Draft'), ('PB', 'Published')]
```

Наберите следующую ниже строку:

```
>>> Post.Status.labels
```

Вы получите удобочитаемые имена членов перечисления enum, как показано ниже:

```
['Draft', 'Published']
```

Наберите следующую ниже строку:

```
>>> Post.Status.values
```

Вы получите значения элементов перечисления *enum*, как показано ниже. Эти значения можно сохранить в базе данных в поле *status*:

```
['DF', 'PB']
```

Наберите такую строку:

```
>>> Post.Status.names
```

Вы получите имена вариантов, как показано ниже:

```
['DRAFT', 'PUBLISHED']
```

К конкретному искомому перечисляемому элементу можно обращаться посредством *Post.Status.PUBLISHED*, а также обращаться к его свойствам *.name* и *.value*.

Добавление взаимосвязи многие-к-одному

Посты всегда пишутся автором. В данном разделе будет создана взаимосвязь (отношение (*relation*) в реляционной базе данных – это таблица, или сущность, состоящая из строк (кортежей) и различных атрибутов (столбцов или полей). Взаимосвязь (*relationship*) – это ассоциация между двумя сущностями, основанная на реляционной модели) между пользователями и постами, которая будет указывать на конкретных пользователей и написанные ими посты. *Django* идет в комплекте с фреймворком аутентификации, который

ведет учетные записи пользователей. Встроенный в *Django* фреймворк аутентификации располагается в пакете *django.contrib.auth* и содержит модель *User* (Пользователь). Модель *User* будет применяться из указанного фреймворка аутентификации, чтобы создавать взаимосвязи между пользователями и постами. Отредактируйте файл *models.py* приложения *blog*, придав ему следующий вид. Новые строки выделены жирным шрифтом:

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

class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    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=2,
                              choices=Status.choices,
                              default=Status.DRAFT)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]


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

Мы импортировали модель *User* из модуля *django.contrib.auth.models* и добавили в модель *Post* поле *author*. Это поле определяет взаимосвязь многие-к-одному, означающую, что каждый пост написан пользователем и пользователь может написать любое число постов. Для этого поля *Django* создаст внешний ключ в базе данных, используя первичный ключ соответствующей модели.

Параметр *on_delete* определяет поведение, которое следует применять при удалении объекта, на который есть ссылка. Это поведение не относится конкретно к *Django*; оно является стандартным для *SQL*. Использование ключевого слова *CASCADE* указывает на то, что при удалении пользователя,

на которого есть ссылка, база данных также удалит все связанные с ним посты в блоге. Со всеми возможными опциями можно ознакомиться по адресу https://docs.djangoproject.com/en/4.1/ref/models/fields/#django.db.models.ForeignKey.on_delete.
Мы используем *related_name*, чтобы указывать имя обратной связи, от *User* к *Post*. Такой подход позволит легко обращаться к связанным объектам из объекта *User*, используя обозначение *user.blog_posts*.
Django содержит разные типы полей, которые можно использовать для определения своих моделей. Все типы полей находятся на странице https://docs.djangoproject.com/en/4.1/ref/models/fields/.
Теперь модель Post завершена, и сейчас можно синхронизировать ее с базой данных. Но перед этим нужно активировать приложение blog в проекте Django.

Создание и применение миграций

Теперь, когда есть модель постов блога, необходимо создать соответствующую таблицу базы данных. Django идет в комплекте с системой миграции, которая отслеживает внесенные в модели изменения и позволяет их распространять по базе данных.
Команда migrate применяет миграции ко всем приложениям, перечисленным в *INSTALLED_APPS*. Она синхронизирует базу данных с текущими моделями и существующими миграциями.
Прежде всего необходимо создать первоначальную миграцию модели Post. Выполните следующую ниже команду в командной оболочке из корневого каталога своего проекта:

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

Вы должны получить результат, аналогичный приведенному ниже:

```
Migrations for 'blog':
    blog/migrations/0001_initial.py
        - Create model Post
        - Create index blog_post_publish_bb7600_idx on field(s)
          -publish of model post
```

Внутри каталога миграций приложения *blog Django* только что создал файл *0001_initial.py*. Эта миграция содержит инструкции *SQL* по созданию таблицы базы данных для модели *Post* и определения индекса базы данных для поля *publish*.
Можно взглянуть на содержимое файла, чтобы увидеть, как определяется миграция. Миграция задает зависимости от других миграций и операций, которые необходимо выполнить в базе данных, чтобы синхронизировать ее с изменениями модели.
Давайте взглянем на исходный код *SQL*, который *Django* исполнит в базе данных, чтобы создать таблицы вашей модели. Команда *sqlmigrate* принимает имена миграций и возвращает их *SQL* без его исполнения.

Выполните следующую ниже команду из командной оболочки, чтобы проинспектировать результирующий исходный код *SQL* вашей первой миграции:

```
python manage.py sqlmigrate blog 0001
Результат должен выглядеть вот так:
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") DEFERRABLE
INITIALLY DEFERRED);
--
-- Create blog_post_publish_bb7600_idx on field(s) -publish of model post
--
CREATE INDEX "blog_post_publish_bb7600_idx" ON "blog_post" ("publish" DESC);
CREATE INDEX "blog_post_slug_b95473f2" ON "blog_post" ("slug");
CREATE INDEX "blog_post_author_id_dd7a8485" ON "blog_post" ("author_id");
COMMIT;
```

Точный результат зависит от используемой вами базы данных. Приведенный выше результат сгенерирован для *SQLite*. Из полученного результата видно, что *Django* генерирует имена таблиц, комбинируя имя приложения с именем модели, обозначенной в нижнем регистре (*blog_post*). Кроме того, существует возможность указывать своей модели имя конкретно-прикладной базы данных. Это делается в Meta-классе модели при помощи атрибута *db_table*.
Django создает автоинкрементный столбец id, используемый в каждой модели в качестве первичного ключа, указав *primary_key=True* в одном из полей модели, но это поведение можно тоже переопределять. Столбец *id* состоит из автоматически увеличивающегося целого числа. Этот столбец соответствует полю *id*, которое добавляется в модель автоматически.
Создаются следующие три индекса базы данных:

- индекс в убывающем порядке по столбцу *publish*. Мы определили этот индекс явным образом с помощью опции *indexes* Meta-класса модели;
- индекс по столбцу slug, поскольку поля типа *SlugField* по умолчанию подразумевают индекс;
- индекс по столбцу *author_id*, поскольку поля типа *ForeignKey* по умолчанию подразумевают индекс.

Давайте сравним модель Post с соответствующей ей таблицей *blog_post* базы данных.

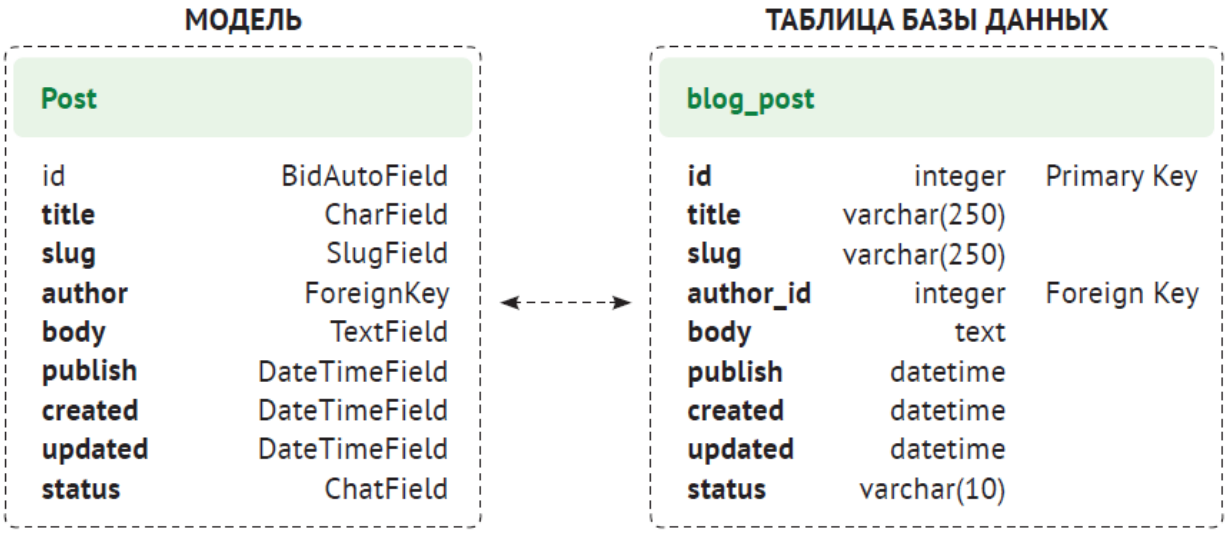

Рис. 1.5. Полное соответствие модели Post и таблицы базы данных

На рис. 1.5 показано, как поля модели соответствуют столбцам таблицы базы данных. Давайте синхронизируем базу данных с новой моделью. Примените следующую ниже команду в командной оболочке, чтобы воспользоваться существующими миграциями:

```
python manage.py migrate
```

Вы получите результат, который заканчивается следующей ниже строкой:

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

Мы только что применили миграции приложений, перечисленных в *INSTALLED_APPS*, включая приложение *blog*. После применения миграций база данных отражает текущее состояние моделей.

Если вы внесете в файл *models.py* любые правки, чтобы добавить, удалить либо изменить поля существующих моделей, либо добавите новые модели, то вам придется создать новые миграции, снова применив команду *makemigrations*.

Каждая миграция дает *Django* возможность отслеживать изменения модели. Затем нужно применить миграцию командой *migrate*, чтобы синхронизировать базу данных с моделями.

Создание сайта администрирования для моделей

Теперь, когда модель *Post* синхронизирована с базой данных, можно создать простой сайт администрирования, чтобы управлять постами блога. Django идет в комплекте со встроенным интерфейсом администрирования, который широко используется для редактирования контента. Сайт *Django* формируется динамически путем чтения метаданных моделей и предоставления готового к работе интерфейса для редактирования контента. Его можно использовать прямо «из коробки», сконфигурировав его так, чтобы ваши модели отображались в нем в том виде, в котором вы хотите. Приложение

django.contrib.admin уже вставлено в настроечный параметр *INSTALLED_APPS*, поэтому добавлять его нет необходимости.

Создание суперпользователя

Сперва необходимо создать пользователя, который будет иметь право управлять сайтом администрирования. Выполните приведенную ниже команду:

```
python manage.py createsuperuser
```

Вы увидите следующий ниже результат. Введите желаемое пользовательское имя (username), адрес электронной почты и пароль, как показано:

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

И вы увидите такое сообщение об успехе:

```
Superuser created successfully.
```

Мы только что создали пользователя-администратора с самым высоким уровнем разрешений.

Сайт администрирования

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу http://127.0.0.1:8000/admin/ в своем браузере. Вы должны увидеть страницу входа на сайт администрирования, как показано на рис. 1.6.

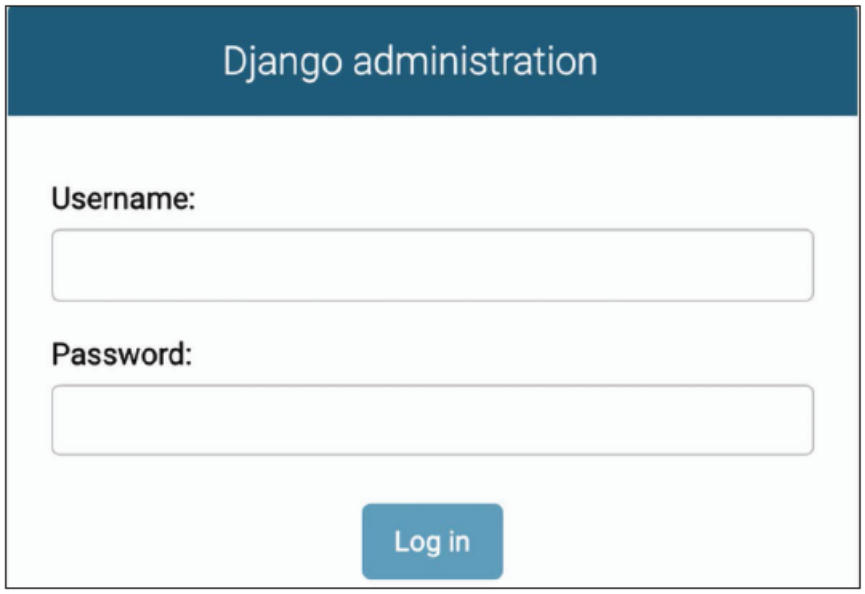

Рис. 1.6. Экран входа на сайт администрирования

Войдите на сайт администрирования, используя учетные данные пользователя, которые вы создали на предыдущем шаге. Вы увидите индексную страницу сайта администрирования, как показано на рис. 1.7.

Модели *Group* и *User*, которые вы видите на приведенном выше скриншоте, являются частью встроенного в *Django* фреймворка аутентификации, расположенного в django.contrib.auth. Если кликнуть по *Users* (Пользователи), то можно увидеть пользователя, которого вы создали ранее.

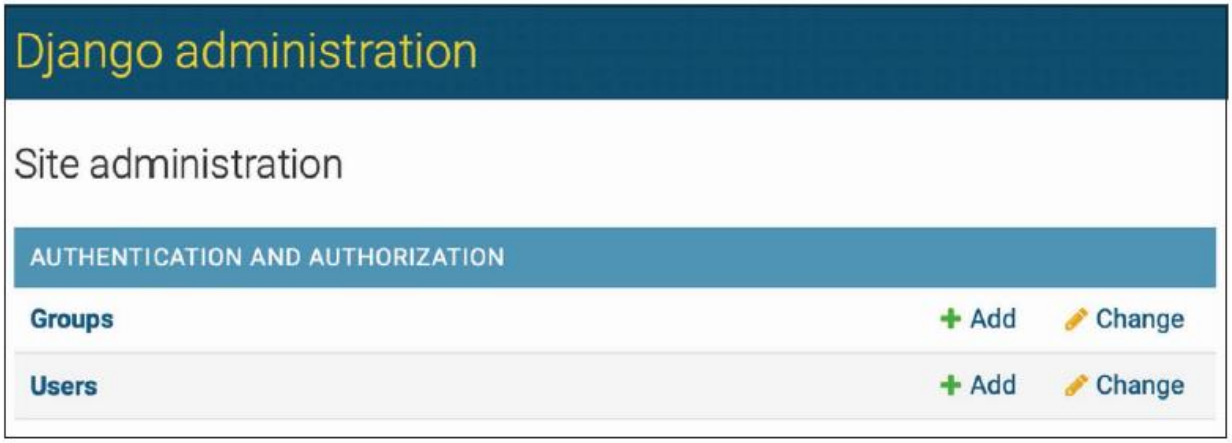

Рис. 1.7. Индексная страница сайта администрирования

Добавление моделей на сайт администрирования

Давайте добавим модели блога на сайт администрирования. Отредактируйте файл *admin.py* приложения *blog*, придав ему следующий вид. Новые строки выделены жирным шрифтом:

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

admin.site.register(Post)
```

Теперь перезагрузите сайт администрирования в своем браузере. Вы должны увидеть свою модель *Post* на сайте, как показано ниже:

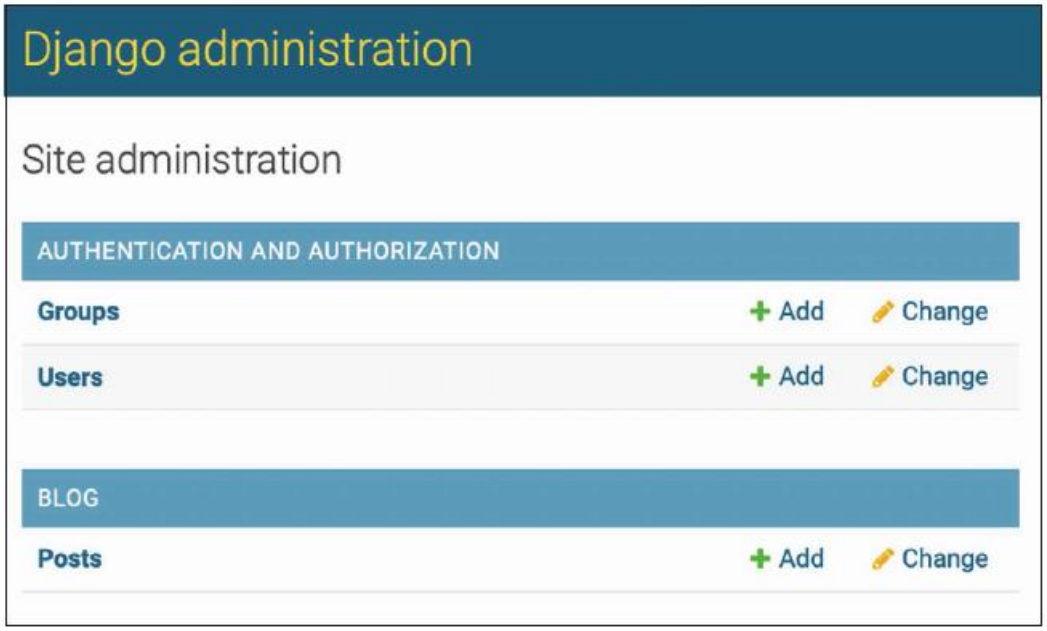

Рис. 1.8. Модель Post приложения blog,
включенная в индексную страницу сайта администрирования

Все достаточно просто, не правда ли? При регистрации модели на сайте администрирования будет получен удобный интерфейс, сгенерированный путем интроспекции созданных разработчиком моделей, позволяющий простым способом выводить списки, редактировать, создавать и удалять объекты.
Кликните по ссылке *Add* (Добавить) напротив *Posts* (Посты), чтобы добавить новый пост. Вы увидите форму, которую *Django* сгенерировал для модели динамически, как показано на рис. 1.9.

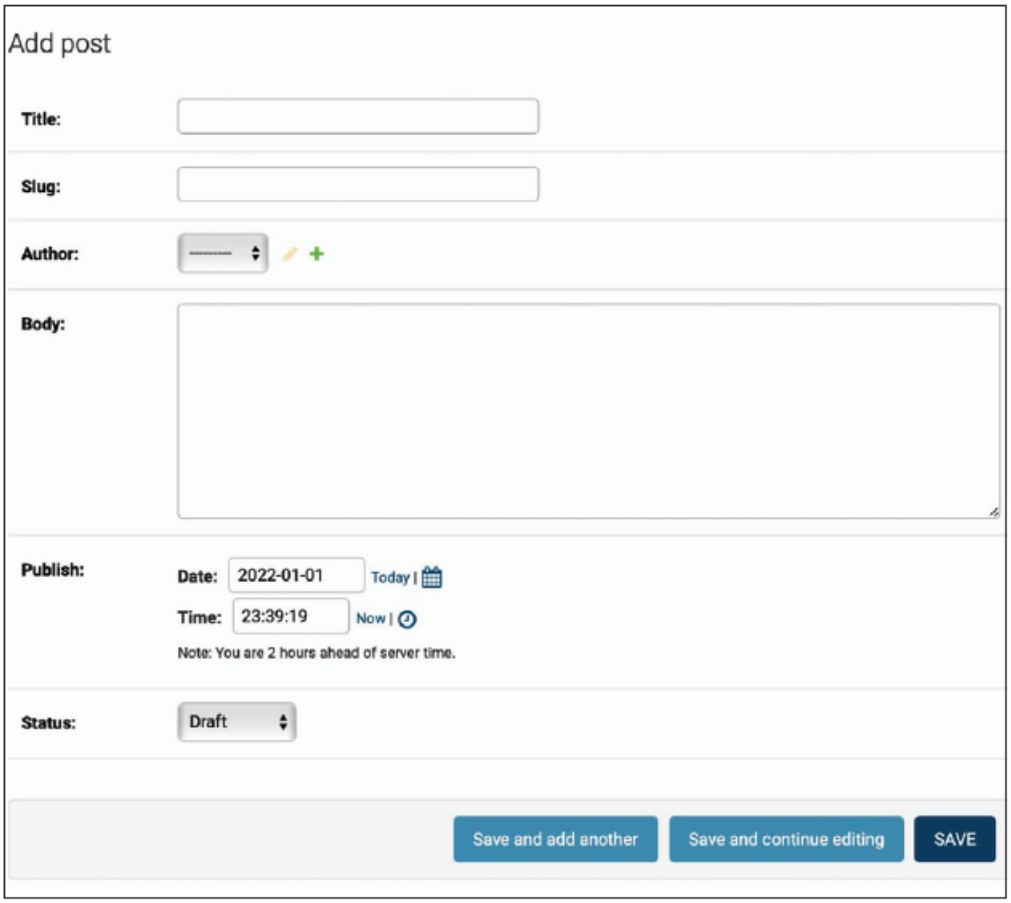

Рис. 1.9. Форма редактирования на сайте администрирования для модели Post

Для каждого типа поля *Django* использует различные виджеты форм. Даже сложные поля, такие как поле *DateTimeField*, отображаются на странице с простым интерфейсом, таким как элемент выбора даты на языке *JavaScript*. Заполните форму и кликните по кнопке *Save* (Сохранить). Вы будете перенаправлены на страницу списка постов с сообщением об успехе и только что созданным постом, как показано на рис. 1.10.

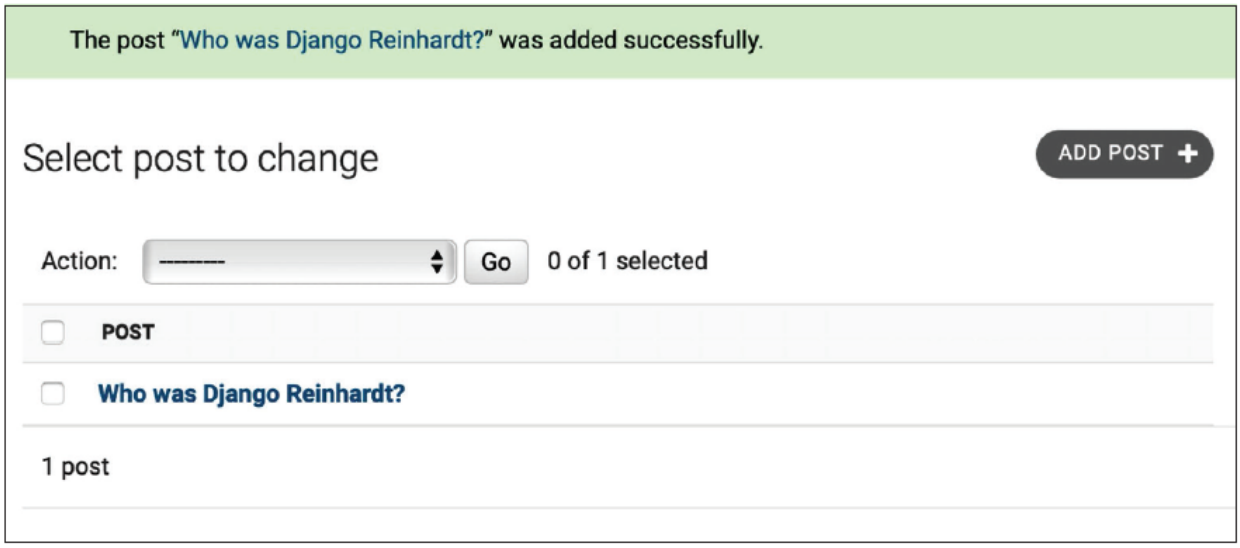

Рис. 1.10. Представление списка на сайте администрирования для модели Post с сообщением об успешном добавлении

Адаптация внешнего вида моделей под конкретно-прикладную задачу

Теперь давайте посмотрим на способы адаптации сайта администрирования под конкретно-прикладную задачу. Отредактируйте файл *admin.py* приложения *blog*, изменив его, как показано ниже. Новые строки выделены жирным шрифтом:

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

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

Мы сообщаем сайту администрирования, что модель зарегистрирована на сайте с использованием конкретно-прикладного класса, который наследует от *ModelAdmin*. В этот класс можно вставлять информацию о том, как показывать модель на сайте и как с ней взаимодействовать.

Атрибут *list_display* позволяет задавать поля модели, которые вы хотите показывать на странице списка объектов администрирования. Декоратор *@admin.register()* выполняет ту же функцию, что и функция *admin.site.register()*, которую вы заменили, регистрируя декорируемый им класс *ModelAdmin*.

Давайте адаптируем модель *admin*, внеся в нее еще несколько опций. Отредактируйте файл admin.py приложения *blog*, изменив его, как показано ниже. Новые строки выделены жирным шрифтом:

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

@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 свой браузер и перезагрузите страницу списка постов. Теперь она будет выглядеть примерно так:

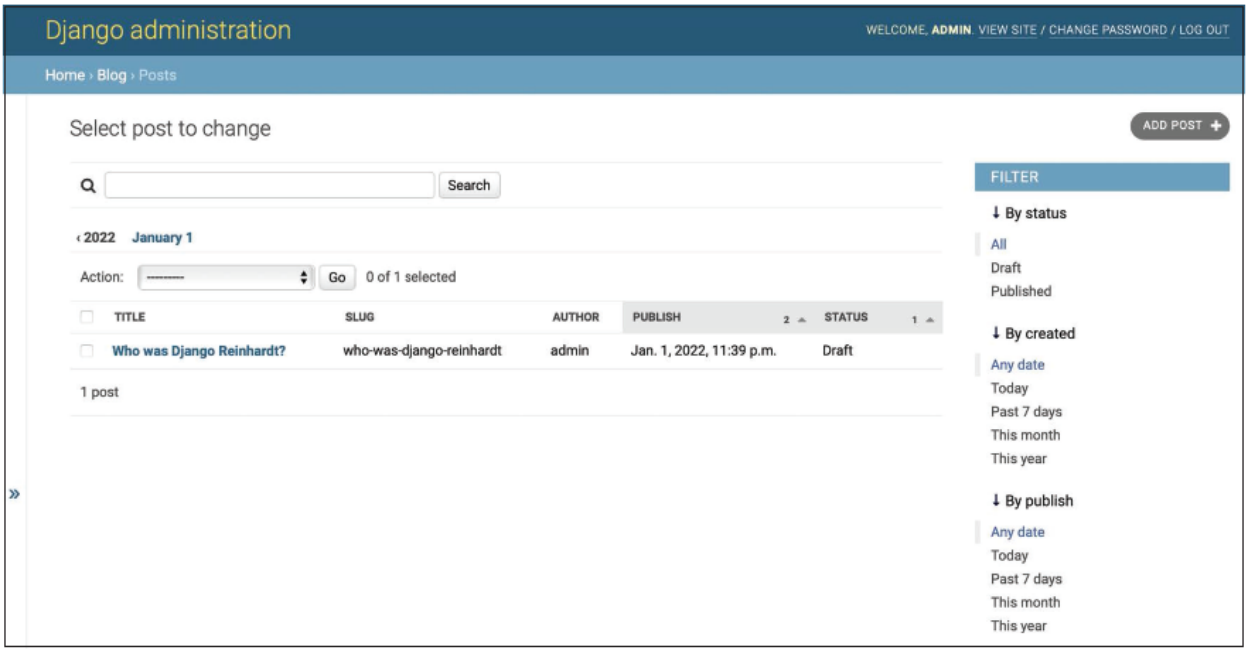

Рис. 1.11. Конкретно-прикладное представление списка на сайте администрирования для модели Post

Вы видите, что отображаемые на странице списка постов поля соответствуют тем, которые мы указали в атрибуте *list_display*. Теперь страница списка содержит правую боковую панель, которая позволяет фильтровать результаты по полям, включенным в атрибут *list_filter*.

На странице появилась строка поиска. Это вызвано тем, что мы определили список полей, по которым можно выполнять поиск, используя атрибут *search_fields*. Чуть ниже строки поиска находятся навигационные ссылки для навигации по иерархии дат; это определено атрибутом *date_hierarchy*.

Вы также видите, что по умолчанию посты упорядочены по столбцам *STATUS* (Статус) и *PUBLISH* (Опубликован). С помощью атрибута *ordering* были заданы критерии сортировки, которые будут использоваться по умолчанию.

Далее кликните по ссылке *ADD POST* (Добавить пост). Здесь вы тоже заметите некоторые изменения. При вводе заголовка нового поста поле *slug* заполняется

автоматически. Вы сообщили *Django*, что нужно предзаполнять поле *slug* данными, вводимыми в поле title, используя атрибут *prepopulated_fields*:

Рис. 1.12. Теперь модель slug автоматически предзаполняется при наборе заголовка на клавиатуре

Кроме того, теперь поле *author* отображается поисковым виджетом, который будет более приемлемым, чем выбор из выпадающего списка, когда у вас тысячи пользователей. Это достигается с помощью атрибута *raw_id_fields* и выглядит следующим образом:

Рис. 1.13. Виджет для отбора ассоциированных объектов для поля author модели Post

Всего несколькими строками исходного кода мы адаптировали отображение модели на сайте администрирования.

Работа с наборами запросов QuerySet и менеджерами

Теперь, когда у нас есть полнофункциональный сайт администрирования, чтобы управлять постами блога, самое время научиться программно читать контент из базы данных и писать его в базу данных.

Встроенный в *Django* объектно-реляционный преобразователь *ORM* (object-relational mapper) – это мощный *API* абстракции базы данных, который позволяет легко создавать, извлекать, обновлять и удалять объекты (Объектно-реляционный преобразователь – это технология преобразования объектов в таблицы реляционной базы данных и наоборот).

ORM-преобразователь дает возможность генерировать запросы на языке SQL, используя объектно-ориентированную парадигму Python. Его можно трактовать как способ взаимодействия с базой данных в Python’овском стиле вместо написания сырых SQL-запросов.

ORM-преобразователь соотносит модели с таблицами базы данных и предоставляет простой Python’овский интерфейс взаимодействия с базой данных. ORM-преобразователь генерирует SQL-запросы и соотносит результаты с объектами модели. ORM-преобразователь совместим с реляционными системами управления базами данных MySQL, PostgreSQL, SQLite, Oracle и MariaDB.

Напомним, что базу данных своего проекта можно определять в настроечном параметре *DATABASES* файла settings.py проекта. *Django* может работать с несколькими базами данных одновременно, при этом можно программировать маршрутизаторы баз данных, чтобы создавать конкретно-прикладные схемы маршрутизации данных.
После создания своих моделей данных *Django* предоставит бесплатный API для взаимодействия с ними.
Встроенный в *Django ORM*-преобразователь основан на итерируемых наборах запросов *QuerySet*. Итерируемый набор запросов *QuerySet* – это коллекция запросов к базе данных, предназначенных для извлечения объектов из базы данных. К наборам запросов можно применять фильтры, чтобы сужать результаты запросов на основе заданных параметров.

Создание объектов

Выполните следующую ниже команду в командной оболочке, чтобы открыть оболочку Python:

```
python manage.py shell
```

Затем наберите следующие ниже строки:

```
>> 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 проанализируем работу приведенного выше исходного кода. Сначала мы извлекаем объект *user* с пользовательским именем *admin*:

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

Метод *get()* позволяет извлекать из базы данных только один объект. Обратите внимание, что этот метод ожидает результат, совпадающий с запросом. Если база данных не возвращает результатов, то указанный метод вызовет исключение *DoesNotExist*, а если база данных возвращает более одного результата, то он вызовет исключение *MultipleObjectsReturned*. Оба исключения являются атрибутами модельного класса, на котором выполняется запрос.
Затем мы создаем экземпляр класса *Post* с конкретно-прикладным заголовком, слагом и телом и задаем пользователя, которого мы ранее извлекли, в качестве автора поста:

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

Этот объект находится в памяти и не сохраняется в базе данных; мы создали объект Python, который можно использовать на стадии работы программы, но который не сохраняется в базе данных.

Наконец, мы сохраняем объект Post в базе данных, используя метод save():

```
post.save()
```

Приведенное выше действие за кулисами выполняет инструкцию *SQL INSERT*. Сначала мы создали объект в памяти, а затем сохранили его в базе данных. Создавать объект и сохранять его в базе данных также можно одной операцией, используя метод *create()*. Это делается следующим образом:

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

Обновление объектов

Теперь измените заголовок поста на что-то другое и снова сохраните объект:

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

На этот раз метод *save()* исполняет инструкцию *SQL UPDATE*.

Вносимые в модельный объект изменения не сохраняются в базе данных до тех пор, пока не будет вызван метод save().

Извлечение объектов

Одиночный объект извлекается из базы данных методом *get()*. Мы применили этот метод посредством метода *Post.objects.get()*. Каждая модель *Django* имеет по меньшей мере один модельный менеджер, а менеджер, который применяется по умолчанию, называется *objects*. Набор запросов *QuerySet* можно получать с помощью модельного менеджера.

Для того чтобы извлечь все объекты из таблицы, используется метод *all()* применяемого по умолчанию менеджера objects. Например:

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

Вот как мы создаем набор запросов *QuerySet*, который возвращает все объекты базы данных. Обратите внимание, что этот *QuerySet* еще не исполнен. Наборы запросов *QuerySet* в *Django* являются ленивыми, то есть они вычисляются только тогда, когда это приходится делать. Подобное поведение придает наборам запросов *QuerySet* большую эффективность. Если не назначать набор запросов *QuerySet* переменной, а вместо этого писать его непосредственно в оболочке *Python*, то инструкция *SQL* набора запросов будет исполняться, потому что вы побуждаете ее генерировать результат:

```
Post.objects.all()
<QuerySet [<Post: Who was Django Reinhardt?>, <Post: New title>]>
```

Применение метода filter()

Для фильтрации набора запросов *QuerySet* можно использовать метод *filter()* менеджера. Например, все посты, опубликованные в 2022 году, можно получить, используя следующий ниже набор запросов:

```
>>> Post.objects.filter(publish__year=2022)
```

Фильтрация также может выполняться по нескольким полям. Например, все посты, опубликованные в 2022 году автором с пользовательским именем *admin*, можно получить следующим образом:

```
>>> Post.objects.filter(publish__year=2022, author__username='admin')
```

Это приравнивается к формированию одного и того же набора запросов *QuerySet*, соединяющего несколько фильтров в цепочку:

```
>>> Post.objects.filter(publish__year=2022) \
>>>             .filter(author__username='admin')
```

Запросы с операциями поиска в полях формируются с использованием двух знаков подчеркивания, например publish__year, но те же обозначения также используются для обращения к полям ассоциированных моделей, например author__username.

Применение метода exclude()

Определенные результаты можно исключать из набора запросов *QuerySet*, используя метод *exclude()* менеджера. Например, все посты, опубликованные в 2022 году, заголовки которых не начинаются со слова *Why* (Почему), можно получить следующим образом:

```
>>> Post.objects.filter(publish__year=2022) \
>>>             .exclude(title__startswith='Why')
```

Применение метода order_by()

Используя метод *order_by()* менеджера, можно упорядочивать результаты по разным полям. Например, можно извлечь все объекты, упорядоченные по их полю *title*, как показано ниже:

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

Подразумевается возрастающий порядок. Убывающий порядок указывается с помощью префикса с отрицательным знаком. Например:

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

Удаление объектов

Если необходимо удалить объект, то это можно сделать из экземпляра объекта, используя метод *delete()*:

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

Обратите внимание, что удаление объектов также приводит к удалению любых зависимых взаимосвязей объектов ForeignKey, в случае если параметр *on_delete* задан равным значению *CASCADE*.

Создание модельных менеджеров

По умолчанию в каждой модели используется менеджер *objects*. Этот менеджер извлекает все объекты из базы данных. Однако имеется возможность определять конкретно-прикладные модельные менеджеры.

Давайте создадим конкретно-прикладной менеджер, чтобы извлекать все посты, имеющие статус *PUBLISHED*.

Есть два способа добавлять или адаптировать модельные менеджеры под конкретно-прикладную задачу: можно добавлять дополнительные методы менеджера в существующий менеджер либо создавать новый менеджер, видоизменив изначальный набор запросов *QuerySet*, возвращаемый менеджером. Первый метод предоставляет обозначение набора запросов в виде *Post.objects.my_manager()*, а второй предоставляет обозначение набора запросов в виде *Post.my_manager.all()*.

Мы выберем второй метод, чтобы реализовать менеджер, который позволит извлекать посты, используя обозначение *Post.published.all()*. Отредактируйте файл *models.py* приложения *blog*, добавив конкретно-прикладной менеджер, как показано ниже. Новые строки выделены жирным шрифтом:

```
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset()\
                      .filter(status=Post.Status.PUBLISHED)

class Post(models.Model):

    # поля модели
```

```
    # ...

    objects = models.Manager() # менеджер, применяемый по умолчанию
    published = PublishedManager() # конкретно-прикладной менеджер

    class Meta:
        ordering = ['-publish']

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

Первый объявленный в модели менеджер становится менеджером, который используется по умолчанию. Для того чтобы указать другой такой менеджер, применяется Meta-атрибут *default_manager_name*. Если менеджер в модели не определен, то *Django* автоматически создает для нее стандартный менеджер *objects*. Если в своей модели вы объявляете какие-либо менеджеры, но также хотите сохранить менеджер objects, то вы должны добавить его в свою модель

явным образом. В приведенном выше исходном коде мы добавили в модель *Post* стандартный менеджер objects и конкретно-прикладной менеджер *published*.
Метод *get_queryset()* менеджера возвращает набор запросов *QuerySet*, который будет исполнен. Мы переопределили этот метод, чтобы сформировать конкретно-прикладной набор запросов *QuerySet*, фильтрующий посты по их статусу и возвращающий поочередный набор запросов *QuerySet*, содержащий посты только со статусом *PUBLISHED*.
Теперь, когда мы определили для модели *Post* конкретно-прикладной менеджер, давайте его протестируем!
Следующей ниже командой снова запустите сервер разработки из командной оболочки:

```
python manage.py shell
```

Теперь можно импортировать модель *Post* и извлечь все опубликованные посты, заголовки которых начинаются с *Who*, исполнив следующий ниже набор запросов *QuerySet*:

```
>>> from blog.models import Post
>>> Post.published.filter(title__startswith='Who')
```

Для того чтобы получить результаты этого набора запросов, проверьте, чтобы поле status было равным значению *PUBLISHED* в объекте *Post*, поле *title* которого начинается со слова *Who*.

Разработка представлений списка и детальной информации

Теперь, когда вы понимаете, как использовать ORM-преобразователь, вы готовы к разработке представлений приложения *blog*. Представление *Django* – это просто функция *Python*, которая получает веб-запрос и возвращает веб-ответ. Вся логика желаемого ответа находится внутри функции-представления. Сначала в своем приложении нужно создать функции-представления, затем по каждому представлению сформировать шаблон URL-адреса и, наконец, создать шаблоны *HTML*, чтобы прорисовывать сгенерированные представлениями данные. Каждое представление будет прорисовывать шаблон, передавая ему переменные, и возвращать HTTP-ответ с прорисованным результатом.

Создание представлений списка постов и детальной информации о посте

Давайте начнем с создания представления списка постов на странице. Отредактируйте файл *views.py* приложения *blog*, придав ему следующий вид. Новые строки выделены жирным шрифтом:

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

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

Это ваше самое первое представление Django. Представление *post_list* принимает объект *request* в качестве единственного параметра. Указанный параметр необходим для всех функций-представлений.
В данном представлении извлекаются все посты со статусом *PUBLISHED*, используя менеджер *published*, который мы создали ранее. Наконец, мы используем функцию сокращенного доступа (В *Django* функция сокращенного доступа (shortcut function) – это вспомогательная функция, которая «охватывает» несколько уровней *MVC*. Другими словами, такая функция ради удобства привносит управляемое сопряжение. Кроме того, такими свойствами обладают и некоторые классы) *render()*, предоставляемую *Django*, чтобы прорисовать список постов заданным шаблоном. Указанная функция принимает объект *request*, путь к шаблону и контекстные переменные, чтобы прорисовать данный шаблон. Она возвращает объект *HttpResponse* с прорисованным текстом (обычно исходным кодом HTML). Функция сокращенного доступа *render()* учитывает контекст запроса, поэтому любая переменная, установленная процессорами контекста шаблона, доступна данному шаблону. Процессоры контекста шаблона – это просто вызываемые объекты (функции, методы и классы), которые назначают контекст переменным. Давайте создадим второе представление одиночного поста на странице. Добавьте следующую ниже функцию в файл *views.py*:

```
from django.http import Http404

def post_detail(request, id):
    try:
        post = Post.published.get(id=id)
    except Post.DoesNotExist:
        raise Http404("No Post found.")

    return render(request,
                  'blog/post/detail.html',
                  {'post': post})
```

Это представление детальной информации о посте. Указанное представление принимает аргумент id поста. Здесь мы пытаемся извлечь объект *Post* с заданным *id*, вызвав метод *get()* стандартного менеджера objects. Мы создаем исключение *Http404*, чтобы вернуть ошибку *HTTP* с кодом состояния, равным 404, если возникает исключение *DoesNotExist*, то есть модель не существует, поскольку результат не найден.

Наконец, мы используем функцию сокращенного доступа *render()*, чтобы прорисовать извлеченный пост с использованием шаблона.

Применение функции сокращенного доступа get_object_or_404()

Django предоставляет функцию сокращенного доступа для вызова метода *get()* в заданном модельном менеджере и вызова исключения *Http404* вместо исключения *DoesNotExist*, когда объект не найден.

Отредактируйте файл *views.py*, импортировав функцию сокращенного доступа *get_object_or_404* и изменив представление *post_detail*, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
from django.shortcuts import render, get_object_or_404

# ...
```

```
def post_detail(request, id):
    post = get_object_or_404(Post,
                             id=id,
                             status=Post.Status.PUBLISHED)
    return render(request,
                  'blog/post/detail.html',
                  {'post': post})
```

Теперь в представлении детальной информации о посте используется функция сокращенного доступа *get_object_or_404()*, чтобы извлекать желаемый пост. Указанная функция извлекает объект, соответствующий переданным параметрам, либо исключение *HTTP* с кодом состояния, равным *404* (не найдено), если объект не найден.

Добавление шаблонов URL-адресов представлений

Шаблоны URL-адресов позволяют соотносить URL-адреса с представлениями. Шаблон URL-адреса состоит из строкового шаблона, представления и, опционально, имени, которое позволяет именовать URL-адрес в масштабе всего проекта. *Django* просматривает каждый шаблон URL-адреса и останавливается на первом, который совпадает с запрошенным URL-адресом. Затем *Django* импортирует представление, совпадающее с шаблоном URL-адреса, и исполняет его, передавая экземпляр класса *HttpRequest* и именованные или позиционные аргументы.

Внутри каталога приложения *blog* создайте файл *urls.py* и добавьте в него следующие ниже строки:

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

app_name = 'blog'

urlpatterns = [
    # представления поста
    path('', views.post_list, name='post_list'),
    path('<int:id>/', views.post_detail, name='post_detail'),
]
```

В приведенном выше исходном коде определяется именное пространство приложения с помощью переменной *app_name*. Такой подход позволяет упорядочивать URL-адреса по приложениям и при обращении к ним использовать имя. С помощью функции *path()* определяются два разных шаблона.

Первый шаблон URL-адреса не принимает никаких аргументов и соотносится с представлением *post_list*. Второй шаблон соотносится с представлением *post_detail* и принимает только один аргумент *id*, который совпадает с целым числом, заданным целым числом конвертора путей *int*.

Для захвата значений из URL-адреса используются угловые скобки. Любое значение, указанное в шаблоне URL-адреса как <*parameter*>, записывается в качестве строкового литерала. Для конкретного сопоставления и возврата целого числа используются конверторы путей, такие как <*int:year*>. Например, <*slug:post*> будет, в частности, совпадать со слагом (строковым литералом, который может содержать только буквы, цифры, подчеркивания или дефисы).

Если функции *path()* и конверторов будет недостаточно, то вместо них можно использовать *re_path()*, чтобы определять сложные шаблоны URL-адресов с помощью регулярных выражений Python.

Создание файла urls.py для каждого приложения – это наилучший способ сделать ваши приложения пригодными для реиспользования в других проектах.

Далее необходимо вставить шаблоны URL-адресов приложения *blog* в главные шаблоны URL-адресов проекта. Отредактируйте файл *urls.py*, расположенный внутри каталога *mysite* проекта, придав ему следующий вид. Новый исходный код выделен жирным шрифтом:

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

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

Новый шаблон URL-адреса, определенный с помощью функции *include*,

ссылается на шаблоны URL-адресов, определенные в приложении *blog*, чтобы они были включены в рамки пути blog/. Указанные шаблоны вставляются в рамки именного пространства blog. Именные пространства должны быть уникальными для всего проекта. Позже можно будет легко ссылаться на URL-запросы блога, используя именное пространство, за которым следует двоеточие, и имя URL-запроса, например *blog:post_list* и *blog:post_detail*.

Создание шаблонов представлений

Вы создали представления и шаблоны URL-адресов для приложения *blog*. Шаблоны URL-адресов соотносят URL-адреса с представлениями, а те, в свою очередь, решают, какие данные будут возвращаться пользователю. Шаблоны определяют способ отображения данных; обычно они пишутся на HTML в сочетании с языком шаблонов *Django*.

Давайте добавим в приложение шаблоны, чтобы отображать посты в удобном для пользователя виде.

Внутри каталога приложения *blog* создайте следующие ниже каталоги и файлы:

```
templates/
    blog/
        base.html
        post/
            list.html
            detail.html
```

Приведенная выше структура будет файловой структурой ваших шаблонов. Файл base.html будет включать в себя главную HTML-структуру веб-сайта и разделит контент на главную область содержимого и боковую панель. Файлы *list.html* и *detail.html* будут наследовать от файла *base.html*, чтобы прорисовывать представления соответственно списка постов блога и детальной информации о посте.

Django обладает мощным языком шаблонов, который позволяет указывать внешний вид отображения данных. Он основан на шаблонных тегах, шаблонных переменных и шаблонных фильтрах:

- шаблонные теги управляют прорисовкой шаблона и выглядят как {%tag %};
- шаблонные переменные заменяются значениями при прорисовке шаблона и выглядят как {{ variable }};
- шаблонные фильтры позволяют видоизменять отображаемые переменные и выглядят как {{ variable|filter }}.

Создание базового шаблона

Отредактируйте файл *base.html*, добавив в него следующий ниже исходный код:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
    <p>This is my blog.</p>
  </div>
</body>
</html>
```

Тег *{% load static %}* сообщает Django, что нужно загрузить статические шаблонные теги (static), предоставляемые приложением *django.contrib*. staticfiles, которое содержится в настроечном параметре *INSTALLED_APPS*. После их загрузки шаблонный тег *{% static %}* можно использовать во всем этом шаблоне. С помощью указанного шаблонного тега можно вставлять статические файлы, такие как файл *blog.css*, который находится в исходном коде данного примера в каталоге *static/* приложения *blog*.
Вы видите, что присутствуют два тега {% block %}. Они сообщают Django, что нужно определить блок в отмеченной области. Шаблоны, которые наследуют от этого шаблона, могут заполнять блоки контентом. В приведенном выше исходном коде был определен блок под названием *title* и блок под названием *content*.

Создание шаблона списка постов

Давайте отредактируем файл *post/list.html* и придадим ему следующий вид:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{% url 'blog:post_detail' post.id %}">
        {{ post.title }}
      </a>
    </h2>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
{% endblock %}
```

Шаблонный тег {% extends %} сообщает Django, что надо наследовать от шаблона *blog/base.html*. Затем заполняются блоки *title* и *content* базового шаблона. Посты прокручиваются в цикле, и их заголовок, дата, автор и тело отображаются на странице, включая ссылку в заголовке на подробный URL-адрес поста. URL-адрес формируется с использованием предоставляемого веб-фреймворком Django шаблонного тега {% url %}.
Этот шаблонный тег позволяет формировать URL-адреса динамически по их имени. Мы используем *blog:post_detail*, чтобы ссылаться на URL-адрес *post_detail* в именном пространстве *blog*. Мы передаем необходимый параметр *post.id*, чтобы сформировать URL-адрес для каждого поста.

Для формирования URL-адресов в своих шаблонах следует всегда использовать шаблонный тег {% url %}, а не писать жестко привязанные URL-адреса. Такой подход упростит техническое сопровождение URL-адресов в будущем.

В теле поста применяются два шаблонных фильтра: *truncatewords* усекает значение до указанного числа слов, а *linebreaks* конвертирует результат в разрывы строк в формате HTML. При этом можно конкатенировать столько шаблонных фильтров, сколько потребуется; каждый из них будет применен к результату, сгенерированному предыдущим.

Доступ к приложению

Откройте командную оболочку и исполните следующую ниже команду, что-бы запустить сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу http://127.0.0.1:8000/blog/ в своем браузере; вы увидите, что все работает. Обратите внимание, что для того чтобы можно было отобразить здесь посты, необходимо иметь несколько постов со статусом *PUBLISHED*. Вы должны увидеть что-то вроде этого:

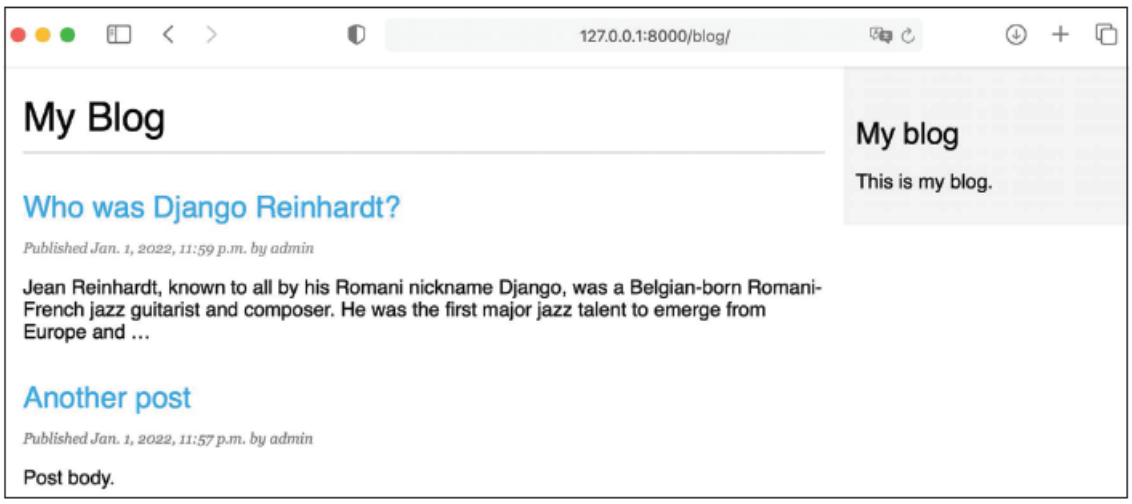

Рис. 1.14. Страница представления списка постов

Создание шаблона детальной информации о посте

Далее отредактируйте файл *post/detail.html*:

```
{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
{% endblock %}
```

Затем можно вернуться в свой браузер и кликнуть по одному из заголовков постов, чтобы просмотреть детальную информацию о посте. Вы должны увидеть что-то вроде этого:

Рис. 1.15. Страница представления детальной информации о посте

Взгляните на URL-адрес – он должен содержать автоматически генерируемый ИД поста. Например, */blog/1/*.

Цикл запроса/ответа

Давайте рассмотрим цикл запроса/ответа Django, воспользуясь приложением, которое мы разработали. Следующая ниже схема показывает упрощенный

пример того, как Django обрабатывает HTTP-запросы и генерирует HTTP-ответы (рис. 1.16).

Рассмотрим процесс запроса/ответа Django.

1. Веб-браузер запрашивает страницу по ее URL-адресу, например https://domain.com/blog/33/. Веб-сервер получает HTTP-запрос и передает его Django.

2. Django пробегает по всем шаблонам URL-адресов, определенным в конфигурации шаблонов URL-адресов. Он проверяет каждый шаблон на соответствие заданному пути URL-адреса в порядке их появления и останавливается на первом, который совпадает с запрошенным URL-адресом. В данном случае шаблон /blog/<id>/ соответствует пути /blog/33/.

3. Django импортирует представление совпавшего шаблона URL-адреса и исполняет его, передавая экземпляр класса HttpRequest и именованные либо позиционные аргументы. Представление использует модели, чтобы извлечь информацию из базы данных. С помощью встроенного в Django ORM-преобразователя наборы запросов QuerySets транслируются в SQL и исполняются в базе данных.

4. В представлении используется функция render(), которая прорисовывает шаблон HTML, передав в него объект Post в качестве контекстной переменной.

5. Прорисованный контент возвращается представлением в виде объекта HttpResponse, по умолчанию с типом контента text/html.

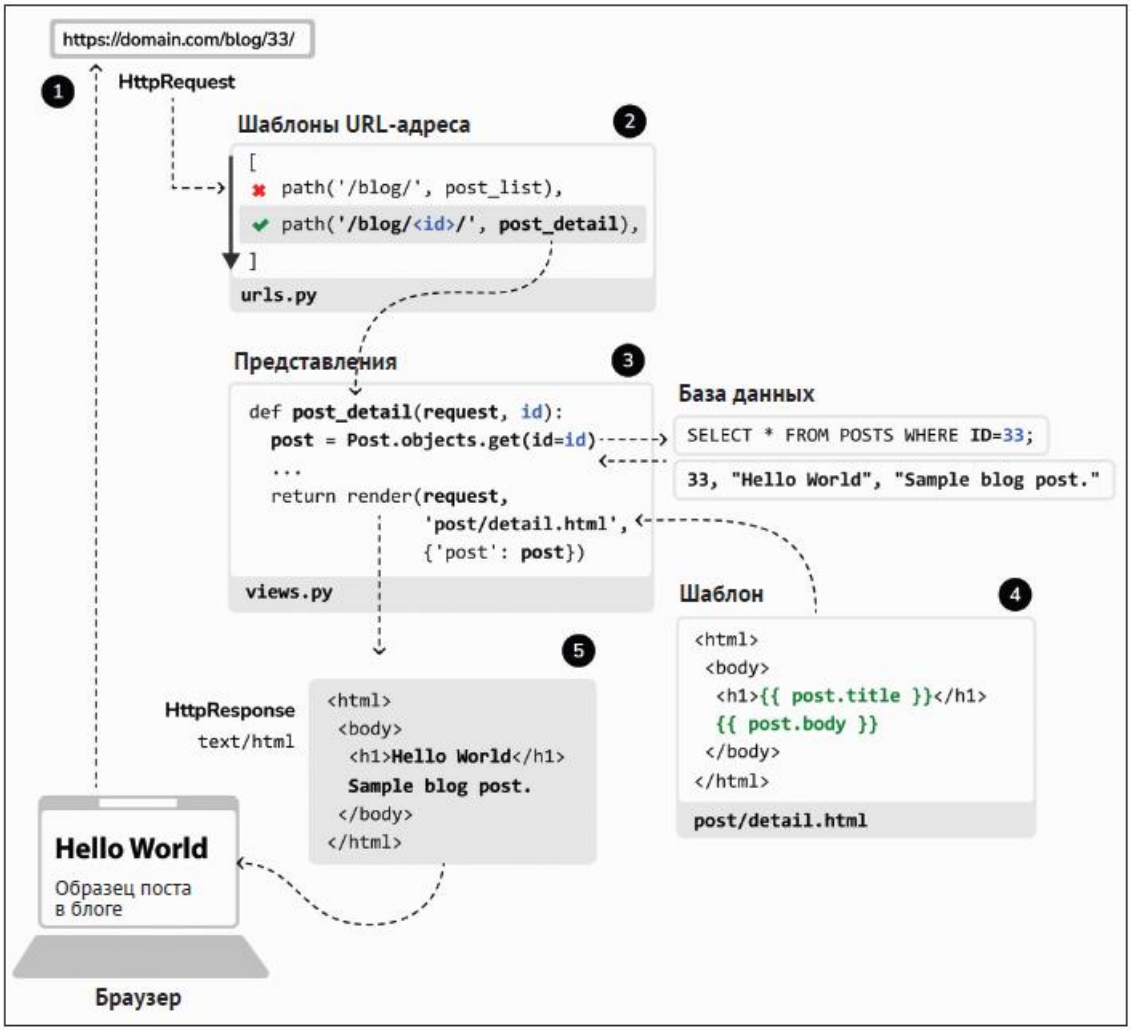

Рис. 1.16. Цикл запроса/ответа Django

Указанную схему всегда можно использовать в качестве базового ориентира в отношении того, как Django обрабатывает запросы. В целях простоты данная схема не содержит промежуточные программные компоненты Django.