

Анатолий Постолит

Python, Django и PyCharm

для начинающих



Анатолий Постолит

Python, Django и PyCharm

для начинающих

Санкт-Петербург

«БХВ-Петербург»

2021

УДК 004.43
ББК 32.973.26-018.1
П63

Постолит А. В.

П63 Python, Django и PyCharm для начинающих. — СПб.: БХВ-Петербург, 2021. — 464 с.: ил. — (Для начинающих)

ISBN 978-5-9775-6779-4

Книга посвящена вопросам разработки веб-приложений с использованием языка Python, фреймворка Django и интерактивной среды разработки PyCharm. Рассмотрены основные технологии и рабочие инструменты создания приложений, даны основы языка Python. Описаны фреймворк Django и структура создаваемых в нем веб-приложений. На простых примерах показаны обработка и маршрутизация запросов пользователей, формирование ответных веб-страниц. Рассмотрено создание шаблонов веб-страниц и форм для пользователей. Показано взаимодействие пользователей с различными типами баз данных через модели. Описана работа с базами данных через встроенные в Django классы без использования SQL-запросов. Приведен пошаговый пример создания сайта от формирования шаблона до его администрирования и развертывания в сети Интернет. Электронный архив на сайте издательства содержит коды всех примеров.

Для программистов

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Людмила Гауль</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Дизайн серии	<i>Мариной Дамбиевой</i>
Оформление обложки	<i>Кариной Соловьевой</i>

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-5-9775-6779-4

© ООО "БХВ", 2021
© Оформление. ООО "БХВ-Петербург", 2021

Оглавление

Предисловие	9
Глава 1. Веб-технологии и инструментальные средства для разработки веб-приложений 14	
1.1. Базовые сведения о веб-технологиях	15
1.1.1. Технологии клиентского программирования	18
1.1.2. Технологии серверного программирования	18
1.2. Базовые сведения о HTML.....	19
1.2.1. Теги для представления текста на HTML-страницах.....	21
1.2.2. Списки.....	24
1.2.3. Таблицы	26
1.2.4. Гиперссылки	29
1.3. Каскадные таблицы стилей (CSS)	30
1.4. Возможности использования JavaScript	32
1.5. Интерпретатор Python	34
1.5.1. Установка Python в Windows.....	34
1.5.2.Установка Python в Linux	37
1.5.3. Проверка интерпретатора Python.....	38
1.6. Интерактивная среда разработки программного кода PyCharm.....	39
1.6.1. Установка PyCharm в Windows	40
1.6.2. Установка PyCharm в Linux.....	42
1.6.3. Проверка PyCharm	43
1.7. Установка пакетов в Python с использованием менеджера пакетов pip	45
1.7.1. Репозиторий пакетов программных средств PyPI.....	46
1.7.2. pip — менеджер пакетов в Python.....	46
1.7.3. Использование менеджера пакетов pip	47
1.8. Фреймворк Django для разработки веб-приложений.....	49
1.9. Фреймворк SQLiteStudio для работы с базами данных	53
1.10. Краткие итоги.....	55
Глава 2. Основы языка программирования Python 56	
2.1. Первая программа в среде интерпретатора Python.....	57
2.2. Базовые операторы языка Python	61
2.2.1. Переменные	62

2.2.2. Функции	64
2.2.3. Массивы	69
2.2.4. Условия и циклы	70
Условия	70
Циклы	72
2.2.5. Классы и объекты	74
Классы	76
Объекты	78
2.2.6. Создание классов и объектов на примере автомобиля	80
2.2.7. Программные модули	82
Установка модуля	83
Подключение и использование модуля	84
2.3. Краткие итоги	84

Глава 3. Знакомимся с веб-фреймворком Django	86
3.1. Общие представления о Django	86
3.2. Структура приложений на Django	89
3.3. Первый проект на Django	91
3.4. Первое приложение на Django	99
3.5. Краткие итоги	104

Глава 4. Представления и маршрутизация	105
4.1. Обработка запросов пользователей	105
4.2. Маршрутизация запросов пользователей в функциях <i>path</i> и <i>re_path</i>	109
4.3. Очередность маршрутов	111
4.4. Основные элементы синтаксиса регулярных выражений	111
4.5. Параметры представлений	113
4.5.1. Определение параметров через функцию <i>re_path()</i>	113
4.5.2. Определение параметров через функцию <i>path()</i>	117
4.5.3. Определение параметров по умолчанию в функции <i>path()</i>	118
4.6. Параметры строки запроса пользователя	120
4.7. Переадресация и отправка пользователю статусных кодов	123
4.7.1. Переадресация	123
4.7.2. Отправка пользователю статусных кодов	125
4.8. Краткие итоги	126

Глава 5. Шаблоны	127
5.1. Создание и использование шаблонов	127
5.2. Класс <i>TemplateResponse</i>	136
5.3. Передача данных в шаблоны	137
5.4. Передача в шаблон сложных данных	140
5.5. Статичные файлы	142
5.5.1. Основы каскадных таблиц стилей	142
5.5.2. Использование статичных файлов в приложениях на Django	147
5.5.3. Использование класса <i>TemplateView</i> для вызова шаблонов HTML-страниц	153
5.5.4. Конфигурация шаблонов HTML-страниц	158
5.5.5. Расширение шаблонов HTML-страниц на основе базового шаблона	160

5.6. Использование специальных тегов в шаблонах HTML-страниц.....	163
5.6.1. Тег для вывода текущих даты и времени.....	163
5.6.2. Тег для вывода информации по условию.....	165
5.6.3. Тег для вывода информации в цикле.....	167
5.6.4. Тег для задания значений переменным.....	169
5.7. Краткие итоги.....	170

Глава 6. Формы 171

6.1. Определение форм.....	171
6.2. Использование в формах POST-запросов.....	176
6.3. Использование полей в формах Django	178
6.3.1. Настройка среды для изучения полей разных типов.....	178
6.3.2. Типы полей в формах Django и их общие параметры	179
6.3.3. Поле <i>BooleanField</i> для выбора решения: да\нет	184
6.3.4. Поле <i>NullBooleanField</i> для выбора решения: да\нет	185
6.3.5. Поле <i>CharField</i> для ввода текста	186
6.3.6. Поле <i>EmailField</i> для ввода электронного адреса.....	187
6.3.7. Поле <i>GenericIPAddressField</i> для ввода IP-адреса.....	189
6.3.8. Поле <i>RegexField</i> для ввода текста.....	189
6.3.9. Поле <i>SlugField</i> для ввода текста	190
6.3.10. Поле <i>URLField</i> для ввода универсального указателя ресурса (URL)	191
6.3.11. Поле <i>UUIDField</i> для ввода универсального уникального идентификатора UUID	192
6.3.12. Поле <i>ComboField</i> для ввода текста с проверкой соответствия заданным форматам	193
6.3.13. Поле <i>FilePathField</i> для создания списка файлов	194
6.3.14. Поле <i>FileField</i> для выбора файлов.....	196
6.3.15. Поле <i>ImageField</i> для выбора файлов изображений	198
6.3.16. Поле <i>DateField</i> для ввода даты	198
6.3.17. Поле <i>TimeField</i> для ввода времени	200
6.3.18. Поле <i>DateTimeField</i> для ввода даты и времени	201
6.3.19. Поле <i>DurationField</i> для ввода промежутка времени	201
6.3.20. Поле <i>SplitDateTimeField</i> для раздельного ввода даты и времени	202
6.3.21. Поле <i>IntegerField</i> для ввода целых чисел.....	203
6.3.22. Поле <i>DecimalField</i> для ввода десятичных чисел	204
6.3.23. Поле <i>FloatField</i> для ввода чисел с плавающей точкой	206
6.3.24. Поле <i>ChoiceField</i> для выбора данных из списка.....	206
6.3.25. Поле <i>TypedChoiceField</i> для выбора данных из списка.....	207
6.3.26. Поле <i>MultipleChoiceField</i> для выбора данных из списка	209
6.3.27. Поле <i>TypedMultipleChoiceField</i> для выбора данных из списка	210
6.4. Настройка формы и ее полей.....	211
6.4.1. Изменение внешнего вида поля с помощью параметра <i>widget</i>	211
6.4.2. Задание начальных значений полей с помощью свойства <i>initial</i>	213
6.4.3. Задание порядка следования полей на форме.....	213
6.4.4. Задание подсказок к полям формы	215
6.4.5. Настройки вида формы	216
6.4.6. Проверка (валидация) данных	217

6.4.7. Детальная настройка полей формы	222
6.4.8. Присвоение стилей полям формы.....	226
6.5. Краткие итоги.....	231
Глава 7. Модели данных Django.....	232
7.1. Создание моделей и миграции базы данных	233
7.2. Типы полей в модели данных Django	237
7.3. Манипуляция с данными в Django на основе CRUD	240
7.3.1. Добавление данных в БД.....	241
7.3.2. Чтение данных из БД.....	241
Метод <i>get()</i>	241
Метод <i>get_or_create()</i>	242
Метод <i>all()</i>	242
Метод <i>filter()</i>	242
Метод <i>exclude()</i>	242
Метод <i>in_bulk()</i>	243
7.3.3. Обновление данных в БД	243
7.3.4. Удаление данных из БД.....	245
7.3.5. Просмотр строки SQL-запроса к базе данных	245
7.4. Пример работы с объектами модели данных (чтение и запись информации в БД)	245
7.5. Пример работы с объектами модели данных: редактирование и удаление информации из БД	249
7.6. Организация связей между таблицами в модели данных.....	255
7.6.1. Организация связей между таблицами «один-ко-многим».....	255
7.6.2. Организация связей между таблицами «многие-ко-многим».....	261
7.6.3. Организация связей между таблицами «один-к-одному»	265
7.7. Краткие итоги.....	268
Глава 8. Пример создания веб-сайта на Django	269
8.1. Создание структуры сайта при помощи Django	269
8.2. Разработка структуры моделей данных сайта «Мир книг».....	280
8.3. Основные элементы моделей данных в Django	282
8.3.1. Поля и их аргументы в моделях данных	283
8.3.2. Метаданные в моделях Django	285
8.3.3. Методы в моделях Django	286
8.3.4. Методы работы с данными в моделях Django	287
8.4. Формирование моделей данных для сайта «Мир книг»	289
8.4.1. Модель для хранения жанров книг	290
8.4.2. Модель для хранения языков книг	290
8.4.3. Модель для хранения авторов книг	291
8.4.4. Модель для хранения книг	292
8.4.5. Модель для хранения отдельных экземпляров книг и их статуса.....	295
8.5. Административная панель Django Admin.....	303
8.5.1. Регистрация моделей данных в Django Admin	303
8.5.2. Работа с данными в Django Admin.....	304
8.6. Изменение конфигурации административной панели Django	314
8.6.1. Регистрация класса ModelAdmin	315
8.6.2. Настройка отображения списков	316

8.6.3. Добавление фильтров к спискам.....	317
8.6.4. Формирование макета с подробным представлением элемента списка.....	321
8.6.5. Разделение страницы на секции с отображением связанной информации	322
8.6.6. Встроенное редактирование связанных записей.....	324
8.7. Краткие итоги.....	328

Глава 9. Пример создания веб-интерфейса для пользователей сайта

«Мир книг».....	329
9.1. Последовательность создания пользовательских страниц сайта «Мир книг».....	329
9.2. Определение перечня и URL-адресов страниц сайта «Мир книг»	330
9.3. Создание главной страницы сайта «Мир книг»	331
9.3.1. Создание URL-преобразования.....	331
9.3.2. Создание представления (view).....	333
9.3.3. Создание базового шаблона сайта и шаблона для главной страницы сайта «Мир книг»	334
9.4. Отображение списков и детальной информации об элементе списка	342
9.5. Краткие итоги.....	357

Глава 10. Расширение возможностей для администрирования сайта

«Мир книг» и создание пользовательских форм	358
10.1. Сессии в Django	359
10.2. Аутентификация и авторизация пользователей в Django	363
10.2.1. Немного об аутентификации пользователей в Django.....	363
10.2.2. Создание отдельных пользователей и групп пользователей.....	364
10.2.3. Создание страницы регистрации пользователя при входе на сайт.....	369
10.2.4. Создание страницы для сброса пароля пользователя	375
10.3. Проверка подлинности входа пользователя в систему.....	381
10.4. Формирование страниц сайта для создания заказов на книги	383
10.5. Работа с формами	392
10.5.1. Краткий обзор форм в Django.....	393
10.5.2. Управление формами в Django	394
10.5.3. Форма для ввода и обновления информации об авторах книг на основе класса <i>Form()</i>	396
10.5.4. Форма для ввода и обновления информации о книгах на основе класса <i>ModelForm()</i>	406
10.6. Краткие итоги.....	414

Глава 11. Публикация сайта в сети Интернет.....

415

11.1. Подготовка инфраструктуры сайта перед публикацией в сети Интернет	415
11.1.1. Окружение развертывания сайта в сети Интернет.....	416
11.1.2. Выбор хостинг-провайдера.....	417
11.2. Подготовка веб-сайта к публикации	418
11.3. Пример размещения веб-сайта на сервисе Heroku.....	420
11.3.1. Создание репозитория приложения на GitHub.....	421
11.3.2. Подключение веб-сервера Gunicorn.....	431
11.3.3. Пакеты для обеспечения доступа к базе данных на Heroku	433
11.3.4. Настройка доступа к базе данных Postgres на Heroku	435

11.3.5. Подключение библиотеки WitheNoise	437
11.3.6. Задание требований к Python	439
11.3.7. Настройка среды выполнения.....	440
11.3.8. Получение аккаунта на Heroku	442
11.3.9. Создание и запуск приложения на Heroku.....	446
11.4. Другие ресурсы для публикации веб-сайта	449
11.5. Краткие итоги.....	449
Послесловие.....	450
Список источников и литературы.....	451
Приложение. Описание электронного архива.....	453
Предметный указатель	458

Предисловие

С развитием цифровых технологий и уходом в прошлое ряда популярных ранее специальностей молодые люди все чаще встают перед выбором перспективной, интересной и актуальной профессии. Международное интернет-сообщество считает, что одним из самых перспективных вариантов такого выбора является профессия веб-программиста. Именно эти специалисты формируют облик сети Интернет и создают технологические тренды будущего.

Каждую секунду в Интернете появляется от 3 до 5 новых сайтов, а каждую минуту — 80 новых интернет-пользователей. Все это технологическое «цунами» управляемся разумом и умелыми руками веб-разработчиков. Зарплата этих специалистов вполне соответствует важности и актуальности их деятельности. Даже начинающие программисты на отечественном рынке могут рассчитывать на заработную плату от 50 тыс. рублей в месяц, а опытные программисты в России и за рубежом имеют доход, превышающий указанную цифру в десятки раз.

Специалисты ИТ-технологий уже много лет подряд занимают первые позиции в рейтингах кадровых агентств, как представители наиболее востребованных профессий на отечественном и мировом рынке труда. Постоянная нехватка квалифицированных программистов дает высокий шанс молодым и целеустремленным новичкам для входа в эту профессию.

Согласно данным Ассоциации предприятий компьютерных и информационных технологий (АПКИТ) спрос на ИТ-специалистов ежегодно остается на высоком уровне, однако кандидатов по-прежнему не хватает. В ближайшие несколько лет дефицит будет только нарастать, так как ИТ-специалисты нужны повсеместно. Если раньше ими интересовались сугубо профильные компании (разработчики софта и онлайн-платформ), то сейчас квалифицированные кадры ИТ-профилей требуются практически везде. И речь здесь идет не о дефиците, а о катастрофическом дефиците. Уже много лет подряд сервис по поиску работы SuperJob фиксирует огромный неудовлетворенный спрос на ИТ-специалистов.

Рассматриваемый в этой книге язык программирования Python — один из самых популярных языков программирования, и области его применения только расширяются. Последние несколько лет он входит в ТОП-3 самых востребованных язы-

ков. Задействуя Python, можно решать множество научных проблем и задач в области бизнеса. На Западе к нему обращаются ученые (математики, физики, биологи), так как изучить его не слишком сложно. Он используется при реализации нейронных сетей, для написания веб-сайтов и мобильных приложений. В целом это универсальный язык, входящий в тройку языков для анализа больших данных. В течение последних 5 лет Python-разработчики востребованы на рынке труда, специалистов в этой сфере до сих пор не хватает.

Еще одна причина, по которой следует обратить внимание на использование языка Python и фреймворка Django для веб-разработки, — это развитие систем искусственного интеллекта. На Python реализовано большое количество библиотек, облегчающих создание программного обеспечения для систем искусственного интеллекта, а Django обеспечивает развертывание приложений в сети Интернет. С использованием этого тандема можно создавать приложения для беспилотных автомобилей, для интеллектуальных транспортных систем, для телемедицины, систем обучения, распознавания объектов в системах безопасности и т. п. Это очень востребованные и актуальные сферы, где наблюдается еще большая потребность в специалистах. Здесь молодые, талантливые и целеустремленные молодые люди могут найти как возможность самореализации, так и достойную оплату своего труда.

Каждая организация, принимающая на работу ИТ-специалиста, требует знаний, которые будут полезны именно в ее работе. Однако общее правило таково, что чем больше популярных и необходимых языков программирования, фреймворков и баз данных вы знаете (Js, HTML, C#, C++, Python, PHP, Django, SQL) и чем больше ваш опыт работы, тем выше шансы на удачное трудоустройство и достойную зарплату.

Перед теми, кто решил освоить специальность веб-программиста, встает непростой выбор — с чего же правильно начать. Конечно, всегда существует возможность получить полноценное ИТ-образование в одном из ведущих технических вузов ранга МГУ, МГТУ им. Н. Баумана, СПбГУ, МФТИ, ИТМО. Но обучение в них обойдется в круглую сумму от 60 до 350 тыс. рублей в год. Существует и более быстрый и дешевый вариант стать веб-разработчиком «с нуля», пройдя краткосрочные онлайн-курсы. Однако практикующие программисты уверяют, что на начальном этапе самый правильный способ обучиться веб-программированию — освоить его самостоятельно. Так можно не только избежать серьезных расходов, но и получить только те практические навыки, которые пригодятся в будущей работе. Полученные самостоятельно базовые знания впоследствии можно расширить, обучаясь уже на соответствующих курсах или в специализированном вузе.

Эта книга предназначена как для начинающих программистов (школьников и студентов), так и для специалистов с опытом, которые планируют заниматься или уже занимаются разработкой веб-приложений с использованием Python. Сразу следует отметить, что веб-программирование требует от разработчиков больших знаний, умений и усилий, чем программирование традиционных приложений. Здесь, кроме основного языка, который реализует логику приложения, требуется еще и знание структуры HTML-документов, основ работы с базами данных, принципов взаимодействия удаленных пользователей с веб-приложением. Если некоторые разделы

книги вам покажутся трудными для понимания и восприятия, то не стоит отчаиваться. Нужно просто последовательно, по шагам повторить приведенные в книге примеры. А когда вы увидите результаты работы программного кода, появится ясность — как работает тот или иной элемент изучаемого фреймворка.

В книге рассмотрены практически все элементарные действия, которые выполняют программисты, работая над реализацией веб-приложений, приведено множество примеров и проверенных программных модулей. Рассмотрены базовые классы фреймворка Django, методы и свойства каждого из классов и примеры их использования. Книга, как уже отмечалось, предназначена как для начинающих программистов, приступивших к освоению языка Python, так и специалистов, имеющих опыт программирования на других языках. В этом издании в меньшей степени освещены вопросы дизайна веб-приложений, а больше внимания уделено технологиям разработки веб-приложений на практических примерах.

Первая глава книги посвящена знакомству с веб-технологиями, рассмотрены принципиальные различия между приложениями, работающими на сервере и на стороне клиента. Приводятся базовые сведения о HTML-страницах, их структуре и основных тегах языка HTML, позволяющих выводить и форматировать информацию. Представлены перечень и краткие описания языков программирования, позволяющих создавать веб-приложения. Здесь же показана последовательность шагов по формированию инструментальной среды пользователя для разработки веб-приложений (установка и настройка программных средств). Это в первую очередь интерпретатор Python, интерактивная среда разработки программного кода PyCharm, фреймворк Django и менеджер работы с базами данных SQLite.

Во *второй главе* рассматриваются базовые элементы языка Python: переменные, функции, массивы, условия и циклы, классы и объекты, созданные на основе этих классов. Это самые простые команды, которых вполне хватит для понимания примеров, приведенных в последующих главах. Однако этих сведений не достаточно для полноценной работы с Python, да и целью книги является не столько описание самого Python, сколько специализированных библиотек для создания веб-приложений. А для более глубокого изучения Python необходимо воспользоваться специальной литературой.

Третья глава посвящена основным понятиям и определениям, которые используются в фреймворке Django. В ней также описана структура приложений на Django. С использованием интерактивной среды PyCharm мы создадим здесь простейший первый проект на Django и сформируем в этом проекте приложение.

В *четвертой главе* приводятся основные понятия о представлениях (view) и маршрутизации запросов пользователей. Это весьма важные компоненты, которые определяют, что должно делать веб-приложение при поступлении различных запросов от удаленных пользователей. Здесь же описан синтаксис так называемых *регулярных выражений*, которые преобразуют запросы пользователей в адреса перехода к тем или иным страницам сайта.

В *пятой главе* рассмотрены вопросы создания шаблонов HTML-страниц. На конкретных примерах показано, как передать в шаблоны простые и сложные данные.

Приведены примеры использования статичных файлов в приложениях на Django. Рассмотрена возможность расширения шаблонов HTML-страниц на основе базового шаблона — это, по сути, основной прием, который применяется практически на всех сайтах. Приведены также примеры использования в шаблонах HTML-страниц специальных тегов (для вывода текущей даты и времени, вывода информации по условию, вывода информации в цикле и для задания значений переменным).

В *шестой главе* сделан обзор пользовательских форм. Формы — это основной интерфейс, благодаря которому удаленные пользователи имеют возможность вносить информацию в удаленную базу данных. Из этой главы вы узнаете, как использовать в формах POST-запросы, в ней также подробно описаны поля, которые можно использовать для ввода данных, и приводятся короткие примеры применения каждого типа поля. Кроме того, рассмотрены примеры изменения внешнего вида полей с помощью виджетов (widget), настроек вида полей, валидации данных и стилизации полей на формах Django.

Седьмая глава посвящена изучению моделей Django. Модели, по своей сути, представляют собой описание таблиц и полей базы данных (БД), используемой веб-приложением. На основе моделей будут автоматически созданы классы, через свойства и методы которых приложение станет взаимодействовать с базой данных. Соответственно, через классы будут выполняться все процедуры работы с данными (добавление, модификация, удаление записей из таблиц БД), при этом отпадает необходимость в написании SQL-запросов — Django будет создавать их самостоятельно и действовать при манипулировании данными. Кроме того, с использованием процедуры миграции Django в автоматическом режиме создаст необходимые таблицы в различных СУБД (SQLite, PostgreSQL, MySQL, Oracle). Из этой главы вы также узнаете, что в дополнение к официально поддерживаемым базам данных существуют сторонние серверы, которые позволяют использовать с Django другие базы данных (SAP SQL, Anywhere, IBM DB2, Microsoft SQL Server, Firebird, ODBC).

В *восьмой главе* мы создадим модели для достаточно простого «учебного» сайта «Мир книг» и соответствующие таблицы в системе управления базами данных (СУБД) SQLite. Возможность работы с данными этого сайта здесь будет показана через встроенную в Django административную панель.

В *девятой главе* приведен пример создания веб-интерфейса для пользователей сайта «Мир книг». Мы определим для этого сайта перечень страниц и их интернет-адресов (URL), создадим представления (views) для обработки запросов пользователей, базовый шаблон сайта и шаблоны прочих страниц сайта. Это простейшие страницы для получения информации из БД на основе запросов пользователей.

В *десятой главе* показано, как можно расширить возможности администрирования сайта «Мир книг» и обеспечить ввод и редактирование данных со стороны удаленных пользователей через формы Django. Здесь рассмотрено такое понятие, как *сессия*, представлены процедуры создания страниц авторизация пользователей, проверки подлинности входа пользователей в систему, изменение внешнего вида страниц для зарегистрированных пользователей. Подробно на примерах показано использование классов `Form` и `ModelForm`.

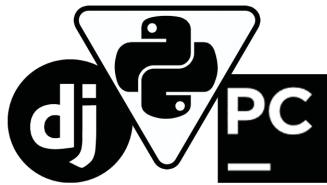
Заключительная, одиннадцатая глава посвящена теме публикации сайтов, разработанных на Django, в сети Интернет. Эта, казалось бы, несложная процедура требует от разработчика определенной квалификации и навыков. На удаленном сервере потребуется создать соответствующее окружение для Python, подгрузить необходимые библиотеки и модули, подключить нужную СУБД, настроить пути к статичным файлам рисунков и стилей. В качестве примера мы выполним процедуру публикации «учебного» сайта «Мир книг» на бесплатном сетевом ресурсе Heroku. В этой же главе вы найдете ссылки на инструкции по публикации сайтов, разработанных на Python и Django, на российских хостинг-ресурсах.

На протяжении всей книги раскрываемые вопросы сопровождаются достаточно упрощенными, но полностью законченными примерами. Ход решения той или иной задачи показан на большом количестве иллюстративного материала. Желательно изучение тех или иных разделов осуществлять, сидя непосредственно за компьютером, — тогда вы сможете последовательно повторять выполнение тех шагов, которые описаны в примерах, и тут же видеть результаты своих действий, что в значительной степени облегчит восприятие материала книги. Наилучший способ обучения — это практика. Все листинги программ приведены на языке Python, а шаблоны веб-страниц — в виде HTML-файлов. Это прекрасная возможность познакомиться с языком программирования Python и понять, насколько он прост и удобен в использовании.

ЭЛЕКТРОННЫЙ АРХИВ

Сопровождающий книгу электронный архив содержит программный код ряда наиболее важных листингов книги (см. [приложение](#)). Архив доступен для закачки с FTP-сервера издательства «БХВ» по ссылке <ftp://ftp.bhv.ru/9785977567794.zip>, ссылка на него также ведет со страницы книги на сайте <https://bhv.ru>.

Итак, если вас заинтересовали вопросы создания веб-приложений с использованием Python, Django и PyCharm, то самое время перейти к изучению материалов этой книги.



ГЛАВА 1

Веб-технологии и инструментальные средства для разработки веб-приложений

В настоящее время веб-технологии стремительно развиваются, проникая в самые разнообразные сферы нашей жизни. Для компаний присутствие в сети Интернет — это возможность рассказать о своих товарах и услугах, найти потенциальных партнеров и клиентов, снизить издержки за счет интернет-торговли и использования «облачных» сервисов. Рядовые пользователи активно пользуются интернет-магазинами, интернет-банкингом, общаются в социальных сетях, могут получать через Интернет государственные услуги.

Для разработки веб-приложений существует множество языков программирования, каждый из которых имеет свои особенности. Но из них хочется выделить Python, как популярную универсальную среду разработки программного кода с тридцатилетней историей.

Python — интерпретируемый язык программирования — в конце 1989 года создал Гвидо Ван Россум, и он очень быстро стал популярным и востребованным у программистов. В подтверждение этого можно упомянуть компании-гиганты: Google, Microsoft, Facebook, Yandex и многие другие, которые используют Python для реализации глобальных проектов.

Область применения Python очень обширна — это и обработка научных данных, и системы управления жизнеобеспечением, а также игры, веб-ресурсы, системы искусственного интеллекта. За все время существования Python плодотворно использовался и динамично развивался. Для него создавались стандартные библиотеки, обеспечивающие поддержку современных технологий — например, работы с базами данных, протоколами Интернета, электронной почтой, машинным обучением и многим другим.

Для ускорения процесса написания программного кода удобно использовать специализированную инструментальную среду — так называемую *интегрированную среду разработки* (IDE, Integrated Development Environment). Эта среда включает полный комплект средств, необходимых для эффективного программирования на Python. Обычно в состав IDE входят текстовый редактор, компилятор или интер-

претатор, отладчик и другое программное обеспечение. Использование IDE позволяет увеличить скорость разработки программ (при условии предварительного обучения работе с такой инструментальной средой).

Возможность создавать веб-приложения не встроена в основные функции Python. Для реализации задач подобного класса на Python нужен дополнительный инструментарий. Таким инструментарием является веб-фреймворк Django. Django считается лучшим веб-фреймворком, написанным на Python. Этот инструмент удобно использовать для создания сайтов, работающих с базами данных, он делает веб-разработку на Python очень качественной и удобной.

Из материалов этой главы вы получите базовые сведения о языке HTML, а также узнаете:

- что такое веб-технологии;
- в чем различия технологий клиентского и серверного программирования;
- какие теги можно использовать для представления текста на HTML-страницах;
- что такое каскадные таблицы стилей;
- какие возможности таит в себе скриптовый язык JavaScript;
- что такое интерпретатор Python и как его установить;
- как начать использовать интерактивную среду разработки программного кода PyCharm;
- как можно установить различные дополнительные пакеты в Python с использованием менеджера пакетов pip;
- как можно установить и начать использовать веб-фреймворк Django;
- как установить фреймворк SQLite Studio для работы с базами данных.

1.1. Базовые сведения о веб-технологиях

Под веб-технологиями в дальнейшем мы будем понимать всю совокупность средств для организации взаимодействия пользователей с удаленными приложениями. Поскольку в каждом сеансе взаимодействуют две стороны: сервер и клиент, то и веб-приложения можно разделить на две группы: приложения на стороне сервера (*server-side*) и приложения на стороне клиента (*client-side*). Благодаря веб-приложениям удаленному пользователю доступны не только статические документы, но и сведения из базы данных. Использование баз данных в сети Интернет привнесло огромную популярность и практически стало отдельной отраслью компьютерной науки.

Но для начала разберемся с основными понятиями веб-технологий: что такое веб-сайт и веб-страница. Часто неопытные пользователи их неправомерно смешивают. *Веб-страница* — это минимальная логическая единица в сети Интернет, которая представляет собой документ, однозначно идентифицируемый уникальным интернет-адресом (URL, Uniform Resource Locator, унифицированным указателем ресурс-

са). *Веб-сайт* — это набор тематически связанных веб-страниц, находящихся на одном сервере и принадлежащих одному владельцу. В частном случае веб-сайт может состоять из единственной веб-страницы. Сеть Интернет является совокупностью всех веб-сайтов.

Для формирования веб-страниц используется язык разметки гипертекста HTML (Hyper Text Markup Language). С его помощью осуществляется логическая (смысловая) разметка документа (веб-страницы). Для целей управления внешним видом веб-страниц используются *каскадные таблицы стилей* (от англ. Cascading Style Sheets, CSS). Сформированные на сервере HTML-документы можно просмотреть на компьютере клиента с помощью специальной программы — *браузера*.

Как было отмечено ранее, совокупность связанных между собой веб-страниц представляет собой веб-сайт. Сайты можно условно разделить на статические и динамические.

□ **Статический сайт** — это сайт с неизменным информационным наполнением. Основу статического сайта составляют неизменные веб-страницы, разработанные с использованием стандартной HTML-технологии. Страницы сайта хранятся в виде HTML-кода в файловой системе сервера. Естественно, на таком сайте могут присутствовать и различные видеоролики и анимация. Основная отличительная особенность статического сайта заключается в том, что веб-страницы такого сайта создаются заранее. Для редактирования содержимого страниц и обновления сайта страницы модифицируют вручную с применением HTML-редактора и затем заново загружают на сайт. Схема взаимодействия клиента со статическим сайтом приведена на рис. 1.1.

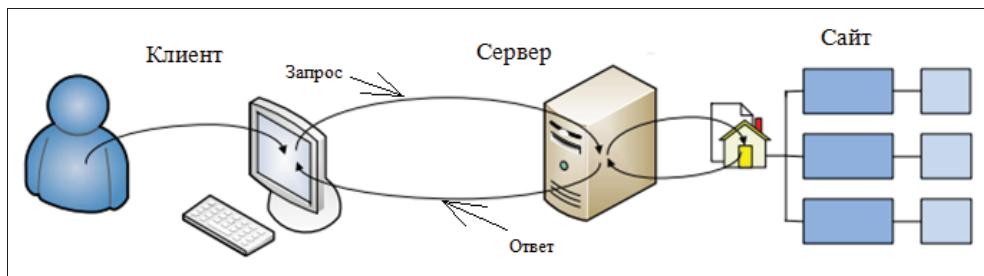


Рис. 1.1. Схема взаимодействия клиента со статическим сайтом

Такая схема приемлема в том случае, когда содержимое сайта довольно постоянно и изменяется сравнительно редко. Если же информация, размещенная на статическом сайте, требует постоянной актуализации и обновления, для его поддержания неизбежны внушительные трудозатраты. Таким образом, статический сайт дешевле в разработке и технической поддержке, но эти достоинства могут нивелироваться серьезными недостатками, связанными с необходимостью оперативного обновления актуальной информации и достаточно высокой трудоемкостью модификации.

□ **Динамический сайт** — это сайт с динамическим информационным наполнением. Динамические страницы также формируются с помощью HTML, но такие

страницы обновляются постоянно, нередко при каждом новом обращении к ним. Динамические сайты основываются на статических данных и HTML-разметке, но дополнительно содержат программную часть (скрипты), а также базу данных (БД), благодаря которым страница «собирается» из отдельных фрагментов в режиме реального времени. Это позволяет обеспечить гибкость в подборе и представлении информации, соответствующей конкретным запросам посетителей сайта.

Таким образом, динамический сайт состоит из набора различных блоков: шаблонов страниц, информационного наполнения (контента) и скриптов, хранящихся в виде отдельных файлов. Иными словами, динамическая веб-страница формируется из страницы-шаблона и добавляемых в шаблон динамических данных. Какие данные будут загружены в шаблон, будет зависеть от того, какой запрос сделал клиент. Схема взаимодействия клиента с динамическим сайтом приведена на рис. 1.2.

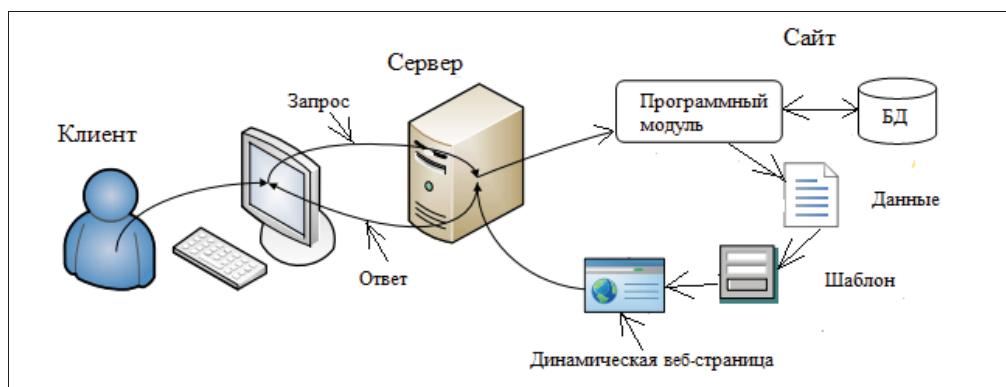


Рис. 1.2. Схема взаимодействия клиента с динамическим сайтом

Динамичность сайта заключается в том, что для изменения страницы достаточно изменить ее информационное наполнение, а сам механизм формирования и вывода страницы остается тем же. Кроме получения различных данных, удаленный клиент может изменить и содержание информационной части сайта. Для этого используются *веб-формы*. Клиент заполняет веб-форму своими данными, и эта информация вносится в БД сайта.

Динамические сайты различаются в зависимости от используемых технологий и процесса получения динамических страниц. Динамические страницы могут быть получены следующими способами:

- генерацией страницы на стороне сервера, осуществляющей серверными скриптами на языках PHP, Perl, ASP.NET, Java, Python и др. При этом информационное наполнение страниц хранится в базах данных;
- генерацией страницы на стороне клиента (JavaScript);
- комбинированной генерацией. Чаще всего на практике встречается именно комбинация первых двух способов.

1.1.1. Технологии клиентского программирования

Простейшим средством «оживления» веб-страниц, добавления динамических эффектов и задания реакции на пользовательские действия является скриптовый язык программирования JavaScript. Сценарии (скрипты) JavaScript внедряются непосредственно в веб-страницу или связываются с ней и после загрузки страницы с сервера выполняются браузером на стороне клиента. Все современные браузеры имеют поддержку JavaScript.

JavaScript — это компактный объектно-ориентированный язык для создания клиентских веб-приложений. Этот язык применяется для обработки событий, связанных с вводом и просмотром информации на веб-страницах. JavaScript обычно используется в клиентской части веб-приложений, в которых клиентом выступает браузер, а сервером — удаленный веб-сервер. Обмен информацией в веб-приложениях происходит по сети. Одним из преимуществ такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения технологии клиентского программирования представляют собой кросс-платформенные сервисы. Язык сценариев JavaScript позволяет создавать интерактивные веб-страницы и содержит средства управления окнами браузера, элементами HTML-документов и стилями CSS.

1.1.2. Технологии серверного программирования

Без использования серверного программирования нельзя обойтись, если необходимо изменять и сохранять какую-либо информацию, хранящуюся на сервере (например, организовать прием и сохранение отправляемых пользователями сообщений). Без серверных скриптов невозможно представить себе гостевые книги, форумы, чаты, опросы (голосования), счетчики посещений и другие программные компоненты, которые активно взаимодействуют с базами данных. Серверное программирование позволяет решать такие задачи, как регистрация пользователей, авторизация пользователей и управление аккаунтом (в почтовых веб-системах, социальных сетях и др.), поиск информации в базе данных, работа интернет-магазина и т. п. Серверные языки программирования открывают перед программистом большие функциональные возможности. Современные сайты зачастую представляют собой чрезвычайно сложные программно-информационные системы, решающие значительное количество разнообразных задач, и теоретически могут дублировать функции большинства бизнес-приложений.

Работа серверных скриптов зависит от платформы, т. е. от того, какие технологии поддерживаются сервером, на котором расположен сайт. Например, основной технологией, поддерживаемой компанией Microsoft, является ASP.NET (Active Server Pages, активные серверные страницы). Другие серверы могут поддерживать языки Perl, PHP, Python.

- **Perl** — высокоуровневый интерпретируемый динамический язык программирования общего назначения. Он является одним из наиболее старых языков, используемых для написания серверных скриптов. По популярности Perl сейчас

уступает более простому в освоении языку PHP. В настоящее время используются бесплатные технологии EmbPerl и mod_perl, позволяющие обрабатывать HTML-страницы со вставленными скриптами на языке Perl.

- **PHP** (Personal Home Page) — язык сценариев общего назначения, в настоящее время интенсивно применяемый для разработки веб-приложений. PHP-код может внедряться непосредственно в HTML. Это язык с открытым кодом, крайне простой для освоения, но вместе с тем способный удовлетворить запросы профессиональных веб-программистов, т. к. имеет большой набор специализированных встроенных средств.
- **Python** — универсальный язык программирования, применимый в том числе и для разработки веб-приложений. Важно, что в Python приложение постоянно находится в памяти, обрабатывая множество запросов пользователей без «перезагрузки». Таким образом поддерживается правильное предсказуемое состояние приложения. В зависимости от того, какой фреймворк будет использоваться совместно с Python, взаимодействия могут существенно упрощаться. Например, Django, который сам написан на Python, имеет систему шаблонов для написания специальных HTML-файлов, которые могут включать код Python и взаимодействовать с данными из СУБД.

1.2. Базовые сведения о HTML

Как было отмечено ранее, язык разметки гипертекста HTML является основой для формирования веб-страниц. С помощью HTML осуществляется логическое форматирование документа, и он может использоваться только для этих целей.

HTML-документы строятся на основе тегов, которые структурируют документ. Обычно теги бывают парными, т. е. состоят из открывающего и закрывающего тега, хотя бывают и исключения. Имена открывающих тегов заключаются в угловые скобки `< ... >`, а закрывающие теги еще содержат знак слеш `</ ... >`.

Весь HTML-документ обрамляется парными тегами `<html>...</html>`. Кроме того, для обеспечения корректного отображения документа современный стандарт требует использования одиночного тега `<!DOCTYPE>`, который может иметь следующую структуру:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

В самом простом случае этот одиночный тег выглядит так:

```
<!DOCTYPE html>
```

Сами HTML-документы состоят из заголовка и тела. Заголовок документа описывается парой тегов `<head>...</head>`, а тело документа обрамляется парными тегами `<body>...</body>`. Таким образом, каркас HTML-документа будет иметь следующую структуру:

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

```

<head>
    СОДЕРЖАНИЕ ЗАГОЛОВКА ДОКУМЕНТА
</head>
<body>
    СОДЕРЖАНИЕ ТЕЛА ДОКУМЕНТА
</body>
</html>

```

Заголовок может включать в себя несколько специализированных тегов, основными из которых являются `<title>...</title>` и `<meta>...</meta>`.

Тег `<title>` содержит информацию, которая будет выводиться в заголовочной части окна браузера пользователя. Например, если в этом теге имеется следующий текст:

```
<title>Мир книг</title>
```

то он следующим образом отобразится в окне браузера (рис. 1.3).

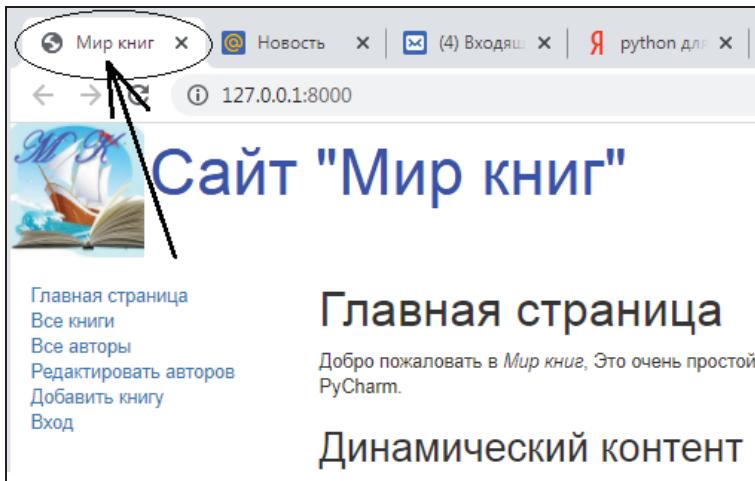


Рис. 1.3. Вывод содержимого тега `<title>` в браузере пользователя

Тег `<meta>` содержит специальную информацию:

- тип кодировки:

```
<meta charset="utf-8" />
```

- или список ключевых слов:

```
<meta name="keywords" content="СУБД, БД, Базы данных">
```

В первом случае этот тег обеспечивает поддержку необходимой кодировки, а во втором — позволяет поисковым машинам производить корректное индексирование страниц сайта по ключевым словам. Следует заметить, что современные поисковые системы могут игнорировать ключевые слова, но это не отменяет возможность использования этого атрибута.

В рассмотренных тегах фрагменты `name="keywords"` и `content="список ключевых слов"` представляют собой *атрибуты* (параметры) тегов, которые конкретизируют их.

Например, атрибуты могут указывать, что текст, заключенный в том или ином теге, при отображении должен выравниваться по центру. Атрибуты записываются сразу после имени тега, причем значения атрибутов заключаются в кавычки. Атрибутов у тега может быть несколько, но могут они и вовсе отсутствовать.

Приведем пример простейшей веб-страницы:

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Веб-страница</title>
    <meta charset="windows-1251">
    <meta name="keywords" content="веб-страница первая">
  </head>
  <body>
    Моя первая веб-страница
  </body>
</html>
```

Текст страницы может быть набран в любом редакторе и сохранен в файле с расширением `htm` или `html`. Такой файл можно открыть при помощи какого-либо браузера.

1.2.1. Теги для представления текста на HTML-страницах

Текст, содержащийся в теле документа, обычно обрамляется специальными тегами. Наиболее часто используются следующие виды тегов.

- Тег `<hi>` — для вывода заголовков, где *i* имеет значения от 1 до 6. Например:

```
<h1> ... </h1>
<h2> ... </h2>
<h3> ... </h3>
<h4> ... </h4>
<h5> ... </h5>
<h6> ... </h6>
```

Заголовки отображаются жирными шрифтами. При этом у текста заголовка в теге `<h1>` наибольший размер шрифта, а у тега `<h6>` — наименьший. К этому тегу может быть добавлен параметр `align`, который определяет правило выравнивания заголовка относительно одной из сторон документа. Этот параметр может принимать следующие значения:

- `left` — выравнивание по левому краю;
- `center` — по центру;
- `right` — по правому краю;
- `justify` — выравнивание по обоим краям.

- Тег **<P>** — для вывода параграфов (абзацев). К этому тегу также может быть добавлен параметр `align`. Каждый новый тег **<P>** с этим параметром устанавливает выравнивание параграфа относительно одной из сторон документа.
- Тег **
** — для перевода строки.
- Тег **<HR>** — для вывода горизонтальной линии. Параметр `align` этого тега определяет выравнивание линии относительно одной из сторон документа. Этот тег имеет еще несколько параметров:
 - `size` — высота линии;
 - `width` — ширина линии;
 - `color` — цвет линии;
 - `noshade` — отключает отбрасывание тени.
- Представленные далее теги меняют следующие параметры текста:
 - **** — жирный текст;
 - **<I>** — наклонный текст;
 - **<U>** — подчеркнутый текст;
 - **<TT>** — моноширинный текст.
- Тег **<PRE>** — выводит заранее отформатированный текст. Используется он для того, чтобы текст с пробелами, символами перехода на новые строки и символами табуляции корректно отображался браузером. В секциях **<PRE>** могут присутствовать гипертекстовые ссылки, однако применять другие HTML-теги нельзя.
- Тег **** — задает некоторые свойства шрифта. Эти свойства определяются следующими параметрами:
 - `size` — размер шрифта от 1 до 7;
 - `face` — начертание шрифта;
 - `color` — цвет шрифта.

Приведем пример использования некоторых тегов:

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="windows-1251">
    <title>Моя первая веб-страница</title>
  </head>
  <body>
    <p>Моя первая веб-страница</p>
    <h1>Заголовок h1</h1>
    <h2>Заголовок h2</h2>
    <h3>Заголовок h3</h3>
    <h4>Заголовок h4</h4>
```

```
<h5>Заголовок h5</h5>
<h6>Заголовок h6</h6>
</body>
</html>
```

Открывающие теги могут содержать дополнительную информацию в виде параметров (атрибутов), которые существенно расширяют возможности представления информации. Параметры в открывающем теге записываются после названия тега в виде параметр="значение" и разделяются пробелами. Порядок следования параметров в теге не произвольный. Если параметр отсутствует, его значение принимается по умолчанию согласно спецификации.

Вот основные параметры тега <BODY>:

- text — устанавливает цвет текста документа, используя значение цвета в виде RRGGBB (например, text="000000" — черный цвет);
- link — устанавливает цвет гиперссылок, используя значение цвета в виде RRGGBB (например, text="FF0000" — красный цвет);
- vlink — устанавливает цвет гиперссылок, на которых пользователь уже побывал (например, text="00FF00" — зеленый цвет);
- alink — устанавливает цвет гиперссылок при нажатии (например, text="0000FF" — синий цвет);
- bgcolor — устанавливает цвет фона документа, используя значение цвета в виде RRGGBB (например, text="FFFFFF" — белый цвет);
- background — устанавливает изображение фона документа (например, background="bg.gif");
- topmargin — устанавливает величину верхнего поля документа (например, topmargin="0");
- leftmargin — устанавливает величину левого поля документа (например, leftmargin="10").

ПРИМЕЧАНИЕ

В HTML-коде цвет определяется 6-значным кодом RRGGBB (красный, красный, зеленый, зеленый, синий, синий), а в JavaScript или в CSS — 6-значным кодом RRGGBB или английским названием цвета. Со значениями кодов, обозначающих цвета, можно ознакомиться по следующей ссылке: <https://www.rapidtables.com/web/color/html-color-codes.html>.

Вот пример использования в теге <BODY> указанных параметров:

```
<BODY text="black" bgcolor="white">
```

Несмотря на то что представленные здесь параметры форматирования еще весьма широко используются, нужно воздерживаться от их применения, т. к. для этих целей предназначены средства каскадных таблиц стилей, о чем речь пойдет в последующих разделах.

1.2.2. Списки

Современным стандартом HTML предусмотрены три основных вида списков:

- маркированные списки (unordered list);
- нумерованные списки (ordered list);
- списки определений (definition list).
- *Маркированные списки* названы так потому, что перед каждым пунктом таких списков устанавливается тот или иной *маркер*. Иногда, прибегая к дословному переводу, их также называют *неупорядоченными списками* (unordered list). Маркированные списки задаются при помощи тегов `...`. Для задания элементов списка (item list) используются теги `...`. Например:

```
<ul>
<li>Пункт 1</li>
<li>Пункт 2</li>
<li>Пункт 3</li>
</ul>
```

Помимо элементов списка внутри тегов `...` можно размещать и другие теги — например, теги заголовков:

```
<ul>
<h3>Маркированный список</h3>
<li>Пункт 1</li>
<li>Пункт 2</li>
<li>Пункт 3</li>
</ul>
```

Тип маркера может задаваться с помощью атрибута `type`. Три его возможных значения: `circle` (незакрашенный кружок), `disk` (закрашенный кружок) и `square` (закрашенный квадрат). По умолчанию используется тип `disk`. Следует отметить, что различные модели браузеров могут по-разному отображать маркеры. Далее приводится пример использования маркера типа `circle`:

```
<ul type="circle">
<li>Пункт 1</li>
<li>Пункт 2</li>
<li>Пункт 3</li>
</ul>
```

В *нумерованных списках* (ordered list), которые иногда называют *упорядоченными*, каждому пункту присваивается номер. Создаются такие списки при помощи тегов `...`. Для элементов нумерованных списков, как и в случае маркированных списков, также используются теги `...`. В таких списках доступны пять типов маркеров, которые, так же как и в маркированных списках, определяются при помощи атрибута `type`, который может принимать следующие значения:

- 1 — арабские цифры;
- i — строчные римские цифры;

- I — прописные римские цифры;
- a — строчные латинские буквы;
- A — прописные латинские буквы.

Далее приведены примеры нумерованных списков:

```
<ol>
<h3>Нумерованный список</h3>
<li>Пункт 1</li>
<li>Пункт 2</li>
<li>Пункт 3</li>
</ol>
<ol type="I">
<li>Пункт 1</li>
<li>Пункт 2</li>
<li>Пункт 3</li>
</ol>
```

Списки определений (definition list) применяются для того, чтобы организовать текст по примеру словарных статей. Они задаются с помощью тегов `<dl>...</dl>`, а определяемый термин или понятие (definition term) помещается в теги `<dt>...</dt>`. Определение понятия (definition description) заключается в тегах `<dd>...</dd>`. В тексте, содержащемся внутри тегов `<dt>...</dt>`, не могут использоваться теги уровня блока, такие как `<p>` или `<div>`. Как и в предыдущих случаях, внутри списков определений могут использоваться теги заголовков и прочие теги:

```
<dl>
<h3>Список определений</h3>
<dt>Понятие 1</dt>
<dd>Определение понятия 1</dd>
<dt>Понятие 2</dt>
<dd>Определение понятия 2</dd>
</dl>
```

Как маркированные, так и нумерованные списки можно вкладывать друг в друга, причем допускается произвольное вложение различных типов списков. При вложении друг в друга маркированных и нумерованных списков следует быть внимательным. Далее приведен пример вложенных списков:

```
<ul>
<h3>Пример вложенных списков</h3>
<li>Пункт 1</li>
<ol>
<li>Пункт 1.1</li>
<li>Пункт 1.2</li>
</ol>
<li>Пункт 2</li>
<ol type="i">
<li>Пункт 2.1</li>
```

```
<li>Пункт 2.2</li>
<li>Пункт 2.3</li>
</ol>
<li>Пункт 3</li>
<ol type="I">
<li>Пункт 3.1</li>
</ol>
</ul>
```

1.2.3. Таблицы

Таблицы являются одной из основных структур, используемых для структурирования информации в HTML-документах. Кроме того, таблицы часто задействуются для организации структуры страницы, и хотя сейчас такое использование таблиц признано устаревшим и не рекомендуемым, оно до сих пор применяется многими веб-дизайнерами. Таблица создается при помощи тега `<TABLE>`. Этот тег может иметь следующие параметры:

- align — задает выравнивание таблицы (`align=left/center/right`);
- border — задает толщину линий таблицы (в пикселях);
- bgcolor — устанавливает цвет фона документа;
- background — устанавливает изображение фона документа;
- cellpadding — задает ширину промежутков между содержимым ячейки и ее границами (в пикселях), т. е. задает поля внутри ячейки;
- cellspacing — задает ширину промежутков между ячейками (в пикселях);
- width — задает ширину таблицы в пикселях, процентах или частях.

Для создания заголовка таблицы служит тег `<CAPTION>`. По умолчанию заголовки центрируются и размещаются либо над (`<CAPTION align="top">`), либо под таблицей (`<CAPTION align="bottom">`). Заголовок может состоять из любого текста и изображений. Текст будет разбит на строки, соответствующие ширине таблицы. Каждая новая строка таблицы создается тегом `<TR>` (Table Row), который может иметь дополнительные параметры:

- align — задает горизонтальное выравнивание информации в ячейках (`align=left/center/right/justify`);
- valign — задает вертикальное выравнивание информации в ячейках (`valign=top/middle/bottom`).

Внутри строки таблицы размещаются ячейки с данными, создаваемые тегами `<TD>`. Число тегов `<TD>` в строке определяет число ячеек (столбцов) таблицы. Тег `<TD>` может иметь следующие дополнительные параметры:

- align — задает горизонтальное выравнивание информации в ячейке (`align=left/center/right/justify`);

- valign — задает вертикальное выравнивание информации в ячейке (valign=top/middle/bottom);
- colspan — объединение столбцов в строке (например, colspan=2);
- rowspan — объединение строк в столбце (например, rowspan=3);
- width — задает ширину ячейки в пикселях или процентах;
- nowrap — запрещение перехода текста в ячейке на новую строку.

Для задания заголовков столбцов и строк таблицы служит тег заголовка <TH> (Table Header). Этот тег аналогичен тегу <TD> с той лишь разницей, что текст в теге <TH> по умолчанию выделяется жирным шрифтом и располагается по центру ячейки.

Далее приведен пример простой таблицы:

```
<table border="3" cellpadding="7" cellspacing="3" height="80" width="50%">
<caption>Пример простой таблицы</caption>
<tr align="center">
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
</tr>
<tr align="center">
<td align="center">2.1</td>
<td align="right">2.2</td>
<td>2.3</td>
</tr>
</table>
```

Благодаря наличию большого числа параметров, а также возможности создания границ нулевой толщины таблица может выступать в роли невидимой модульной сетки, относительно которой добавляется текст, изображения и другие элементы, организуя информацию на странице.

□ Возможности табличной верстки:

- создание колонок;
- создание «резинового» макета;
- «склейка» изображений;
- включение фоновых рисунков;
- выравнивание элементов.

□ Недостатки табличной верстки:

- долгая загрузка;
- громоздкий код;
- плохая индексация поисковиками;
- нет разделения содержимого и оформления;
- несоответствие стандартам.

Рассмотрим некоторые типовые модульные сетки.

- **Двухколонная модульная сетка** часто применяется на небольших информационных сайтах. Как правило, в первой колонке располагается логотип и меню сайта, а во второй — основной материал. Пример такой модульной сетки приведен на рис. 1.4.

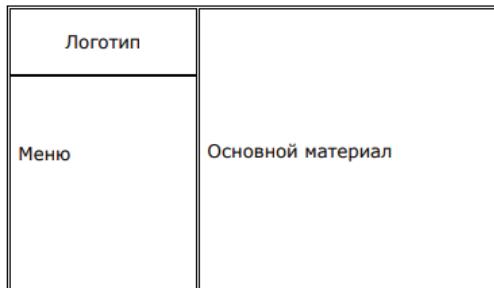


Рис. 1.4. Двухколонная модульная сетка

Далее приведен HTML-код, реализующий эту структуру:

```

<TABLE width="760" cellpadding="10" cellspacing="10" border="1">
<TR>
<TD align="center">Логотип</TD>
<TD rowspan="2">Основной материал</TD>
</TR>
<TR>
<TD>Меню</TD>
</TR>
</TABLE>
    
```

- **Трехколонная модульная сетка** «резинового» макета применяется на крупных порталах с множеством информационных сервисов. Как правило, в первой колонке располагается меню сайта и дополнительная информация, во второй — основной материал, в третьей — дополнительные функции. Часто в модульную сетку сайта добавляют блоки «Заголовок сайта» и «Окончание сайта» (этот блок еще называют «подвалом»). Пример такой модульной сетки приведен на рис. 1.5.

Далее приведен HTML код, реализующий эту структуру:

```

<TABLE width="100%" cellpadding="10" cellspacing="10" border="0">
<TR align="center">
<TD colspan="3">Заголовок сайта</TD>
</TR>
<TR>
<TD width="200px">Меню</TD>
<TD width="*">Основной материал</TD>
<TD width="200px">Дополнительные функции</TD>
</TR>
<TR align="center">
    
```

```
<TD colspan="3">Окончание сайта</TD>
</TR>
</TABLE>
```

Заголовок сайта		
Меню	Основной материал	Дополнительные функции
Окончание сайта		

Рис. 1.5. Трехколонная модульная сетка

1.2.4. Гиперссылки

Для связи различных документов HTML предусматривает использование *ссылок*. Сам термин HTML (Hyper Text Markup Language) подразумевает их широкое использование. Для реализации ссылок в HTML служит тег `<a>...`, который, как и большинство HTML-тегов, является контейнерным. Основной атрибут этого тега — `href`, собственно и содержащий адрес ресурса, на который указывает ссылка. Внутри тега `<a>...` помещается текст ссылки.

В ссылках может использоваться как относительная, так и абсолютная адресация. В случае абсолютной адресации атрибуту `href` присваивается абсолютный URL-адрес ресурса:

```
<a href="http://server.com/doc3.htm">Ссылка на документ с абсолютным адресом  
http://server.com/doc3.htm</a> .
```

В случае относительной адресации указывается путь к документу относительно текущей страницы:

```
<a href="doc1.htm">Ссылка на документ с относительным адресом doc1.htm</a> .
```

Если в заголовочной части документа использован тег `<base>`, то отсчет будет вестись от адреса, заключенного в этом теге.

Помимо веб-страниц, допускается ссылаться и на другие интернет-ресурсы: e-mail, ftp, Gopher, WAIS, Telnet, newsgroup. Далее приведен пример ссылки на адрес электронной почты:

```
<a href="mailto:sss@mail.ru">Ссылка на адрес электронной почты sss@mail.ru</a> .
```

В роли ссылок могут выступать и рисунки. Для этого сначала надо вставить рисунок с помощью тега ``. У атрибута `src` этого тега устанавливается значение, соответствующее имени файла рисунка:

```
 .
```

Далее, рисунок «обертывается» в тег ссылки:

```
<a href="link2.htm"></a> .
```

1.3. Каскадные таблицы стилей (CSS)

Если HTML используется для логического форматирования документа, то для управления его отображением на экране монитора или выводом на принтер применяются каскадные таблицы стилей (CSS). Технология CSS реализует концепцию «Документ — представление» и позволяет отделять оформление HTML-документа от его структуры. Кроме того, CSS существенно расширяет возможности представления (оформления) документов, внося множество новых возможностей.

Для того чтобы таблица стилей влияла на вид HTML-документа, она должна быть подключена к нему. Подключение каскадных таблиц стилей может быть выполнено с использованием внешних, внутренних или локальных таблиц стилей:

- внешние таблицы стилей, оформленные в виде отдельных файлов, подключаются к HTML-документу при помощи тега в заголовке документа. Например:

```
<LINK rel="stylesheet" href="style.css" type="text/css"> .
```

- внутренние таблицы стилей в составе HTML-документа помещаются в заголовок страницы при помощи тега <STYLE>. Например:

```
<HEAD>
<STYLE>
    P {color: #FF0000}
</STYLE>
</HEAD>
```

- локальные таблицы стилей объявляются непосредственно внутри тега, к которому они относятся, при помощи параметра style. Например:

```
<P style="color: #FF0000">Каскадные таблицы стилей</p> .
```

Таблицы стилей записываются в виде последовательности тегов, для каждого из которых в фигурных скобках указывается его свойства по схеме параметр: свойство. Параметры внутри фигурных скобок разделяются точкой с запятой. Например:

```
P {color: #CCCCCC; font-size: 10px}
H1{color: #FF0000} .
```

С помощью каскадных таблиц стилей можно менять свойства следующих элементов: фона и цвета, шрифта, текста, полей и отступов, границ.

- Для задания свойств фона и цвета служат следующие параметры:

- background-color — цвет заднего плана;
- background-image — изображение заднего плана;
- background-repeat — дублирование изображения заднего плана (значения: repeat/repeat-x/repeat-y/no-repeat);

- background-attachment — фиксация изображения заднего плана (значения: scroll/fixed);
- background-position — начальное положение изображения заднего плана.

□ Для задания свойств шрифта служат следующие параметры:

- font-style — стиль шрифта (значения: normal / italic);
- font-weight — начертание шрифта (значения: normal / bold);
- font-size — размер шрифта;
- font-family — список имен шрифтов в порядке их приоритета.

□ Для задания свойств текста служат следующие параметры:

- word-spacing — установка промежутка между словами;
- letter-spacing — установка высоты строки;
- text-indent — установка абзацного отступа;
- text-align — выравнивание текста;
- vertical-align — установка вертикального выравнивания текста;
- text-decoration — преобразование текста (значение: none/underline/overline/line-through/blink).

□ Для задания свойств полей и отступов служат следующие параметры:

- margin-top — установка верхнего поля;
- margin-right — установка правого поля;
- margin-bottom — установка нижнего поля;
- margin-left — установка левого поля;
- margin — установка всех полей (например: margin: 10px 10px 10px 0px);
- padding-top — установка верхнего отступа;
- padding-right — установка правого отступа;
- padding-bottom — установка нижнего отступа;
- padding-left — установка левого отступа;
- padding — установка всех отступов (например: padding: 10px 10px 10px 0px).

□ Для задания свойств границ служат следующие параметры:

- border-width — установка ширины границы;
- border-color — установка цвета границы;
- border-style — установка стиля границы (значения: none/dotted/dashed/solid/double).

Для придания управлению элементами HTML большей гибкости используются *классы*, которые позволяют задавать различные стили для одного и того же тега.

В таблицах стилей имя класса записывается после тега и отделяется от него точкой. Например:

```
P.green {color: #00FF00}  
P.blue {color: #0000FF}
```

В HTML-коде соответствие тега определенному классу указывается при помощи параметра `class`. Например:

```
<p class="green">Зеленый текст</p>  
<p class="blue">Синий текст</p>
```

1.4. Возможности использования JavaScript

Чаще всего JavaScript используется как язык, встраиваемый в веб-страницы для получения программного доступа к их элементам. Сценарии JavaScript являются основой клиентской части веб-приложений. Они загружаются с сервера вместе с веб-страницами и выполняются браузером на компьютере пользователя. Обработка сценариев JavaScript осуществляется встроенным в браузер интерпретатором. Что может делать JavaScript?

- В первую очередь JavaScript умеет отслеживать действия пользователя (например, реагировать на щелчок мыши или нажатие клавиши на клавиатуре, на перемещение курсора, на скроллинг).
- Менять стили, добавлять (удалять) HTML-теги, скрывать (показывать) элементы.
- Посыпать запросы на сервер, а также загружать данные без перезагрузки страницы (эта технология называется AJAX).

JavaScript — это объектно-ориентированный язык. Он поддерживает несколько встроенных объектов, а также позволяет создавать или удалять свои собственные (пользовательские) объекты. Объекты JavaScript могут наследовать свойства непосредственно друг от друга, образуя цепочку объект — прототип.

Сценарии JavaScript бывают встроенные, т. е. их содержимое является частью документа, и внешние, хранящиеся в отдельном файле с расширением js. Сценарии можно внедрить в HTML-документ следующими способами:

- в виде гиперссылки;
- в виде обработчика события;
- внутри элемента `<script>`.

Рассмотрим эти способы подробнее.

- Если подключать JavaScript в виде гиперссылки, то для этого код скрипта нужно разместить в отдельном файле, а ссылку на файл включить либо в заголовок:

```
<head>  
  <script src="script.js"></script>  
</head>
```

либо в тело страницы:

```
<body>
  <script src="script.js"></script>
</body>
```

Этот способ обычно применяется для сценариев большого размера или сценариев, многократно используемых на разных веб-страницах.

- Можно подключать JavaScript к обработчику события. Дело в том, что каждый HTML-элемент может иметь JavaScript-события, которые срабатывают в определенный момент. Нужно добавить необходимое событие в HTML-элемент как атрибут, а в качестве значения этого атрибута указать требуемую функцию. Функция, вызываемая в ответ на срабатывание события, и станет обработчиком события. В результате срабатывания события исполнится связанный с ним код. Этот способ применяется в основном для коротких сценариев — например, можно установить по нажатию на кнопку смену цвета фона :

```
<script>
var colorArray = ["#5A9C6E", "#A8BF5A", "#FAC46E", "#FAD5BB",
                  "#F2FEFF"]; // создаем массив с цветами фона
var i = 0;

function changeColor(){
  document.body.style.background = colorArray[i];
  i++;
  if( i > colorArray.length - 1){
    i = 0;
  }
}
</script>
<button onclick="changeColor()">Сменить цвет фона</button>
```

- Код внутри элемента `<script>` может вставляться в любое место документа. Код, который выполняется сразу после прочтения браузером или содержит описание функции, которая выполняется в момент ее вызова, располагается внутри тега. Описание функции можно располагать в любом месте — главное, чтобы к моменту ее вызова код функции уже был загружен.

Обычно код JavaScript размещается в заголовке документа (в элементе `<head>`) или после открывающего тега `<body>`. Если скрипт используется после загрузки страницы (например, код счетчика), то его лучше разместить в конце документа:

```
<footer>
  <script>
    document.write("Введите свое имя");
  </script>
</footer>
</body>
```

В примерах этой книги мы не будем задействовать возможности JavaScript, однако полезно знать о наличии этого языка и его возможностях при создании веб-страниц.

1.5. Интерпретатор Python

Язык программирования Python является весьма мощным инструментальным средством для разработки различных систем. Однако наибольшую ценность представляет даже не столько сам этот язык программирования, сколько набор подключаемых библиотек, на уровне которых уже реализованы все необходимые процедуры и функции. Разработчику достаточно написать несколько десятков строк программного кода, чтобы подключить требуемые библиотеки, создать набор необходимых объектов, передать им исходные данные и отобразить итоговые результаты.

Для установки интерпретатора Python на компьютер прежде всего надо загрузить его дистрибутив. Скачать дистрибутив Python можно с официального сайта, перейдя по ссылке: <https://www.python.org/downloads/> (рис. 1.6).

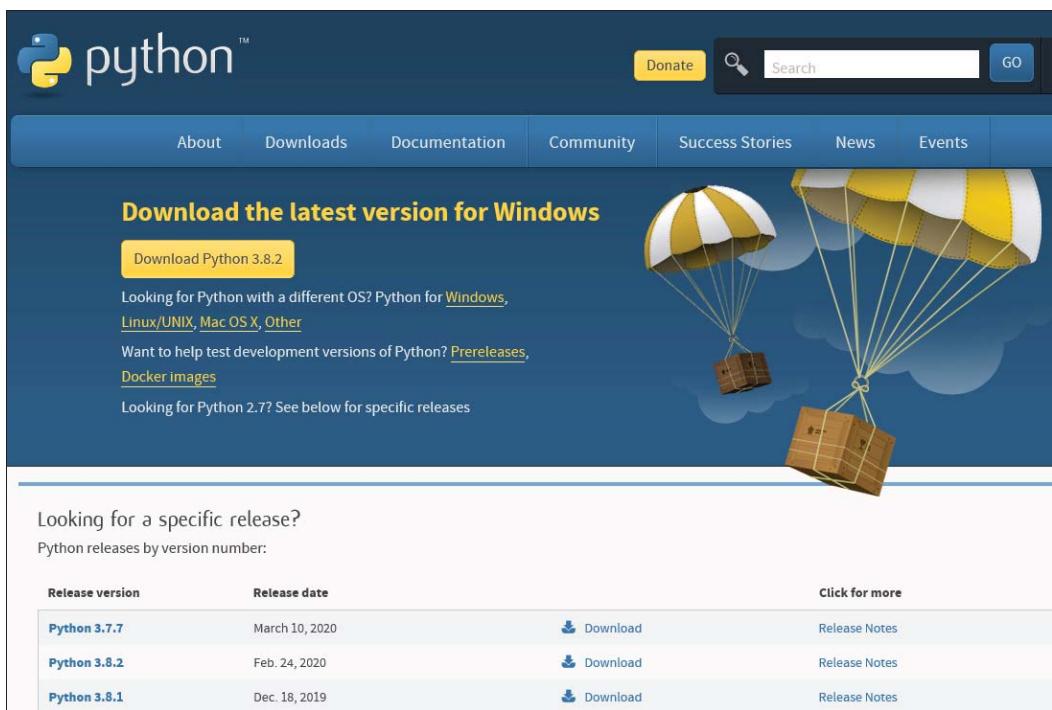


Рис. 1.6. Сайт для скачивания дистрибутива языка программирования Python

1.5.1. Установка Python в Windows

Для операционной системы Windows дистрибутив Python распространяется либо в виде исполняемого файла (с расширением `exe`), либо в виде архивного файла

(с расширением zip). На момент подготовки этой книги была доступна версия Python 3.8.3.

Порядок установки Python в Windows следующий:

1. Запустите скачанный установочный файл.
2. Выберите способ установки (рис. 1.7).

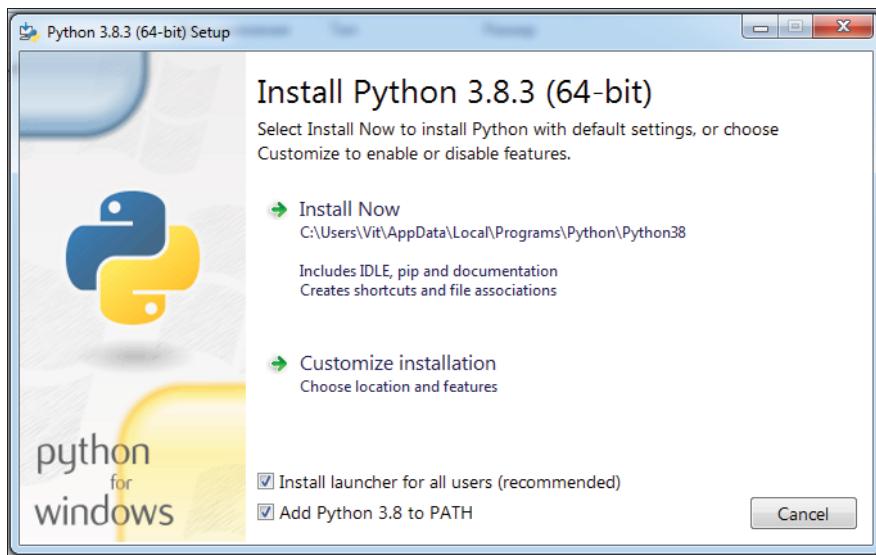


Рис. 1.7. Выбор способа установки Python

В открывшемся окне предлагаются два варианта: **Install Now** и **Customize installation**:

- при выборе **Install Now** Python установится в папку по указанному в окне пути. Помимо самого интерпретатора будут инсталлированы IDLE (интегрированная среда разработки), pip (пакетный менеджер) и документация, а также созданы соответствующие ярлыки и установлены связи (ассоциации) файлов, имеющих расширение py, с интерпретатором Python;
 - **Customize installation** — это вариант настраиваемой установки. Опция **Add Python 3.8 to PATH** нужна для того, чтобы появилась возможность запускать интерпретатор без указания полного пути до исполняемого файла при работе в командной строке.
3. Отметьте необходимые опции установки, как показано на рис. 1.8 (доступно при выборе варианта **Customize installation**).
- На этом шаге нам предлагается отметить дополнения, устанавливаемые вместе с интерпретатором Python. Рекомендуется выбрать как минимум следующие опции:
- **Documentation** — установка документации;
 - **pip** — установка пакетного менеджера pip;



Рис. 1.8. Выбор опций установки Python

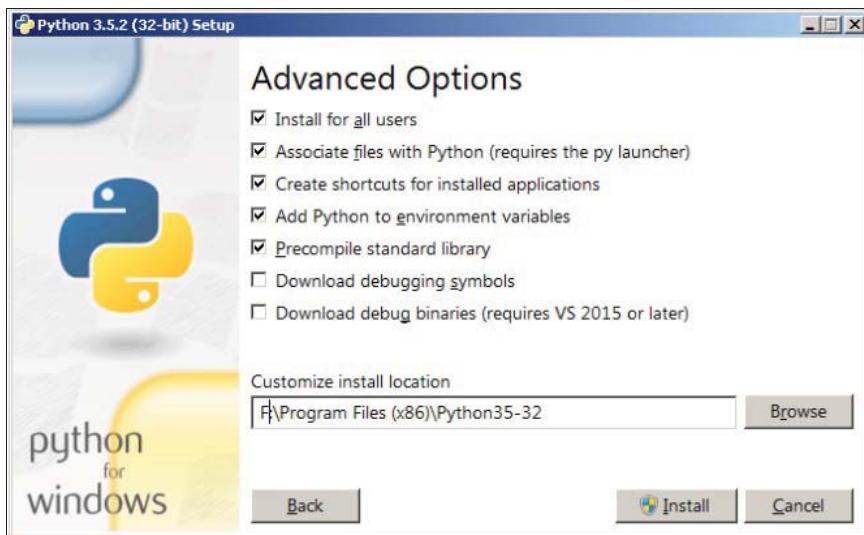


Рис. 1.9. Выбор места установки Python

- **tcl/tk and IDLE** — установка интегрированной среды разработки (IDLE) и библиотеки для построения графического интерфейса (tkinter).
4. На следующем шаге в разделе **Advanced Options** (Дополнительные опции) выберите место установки, как показано на рис. 1.9 (доступно при выборе варианта **Customize installation**).

Помимо указания пути, этот раздел позволяет внести дополнительные изменения в процесс установки с помощью опций:

- **Install for all users** — установить для всех пользователей. Если не выбрать эту опцию, то будет предложен вариант инсталляции в папку пользователя, устанавливающего интерпретатор;
- **Associate files with Python** — связать файлы, имеющие расширение `py`, с Python. При выборе этой опции будут внесены изменения в Windows, позволяющие Python запускать скрипты по двойному щелчку мыши;
- **Create shortcuts for installed applications** — создать ярлыки для запуска приложений;
- **Add Python to environment variables** — добавить пути до интерпретатора Python в переменную PATH;
- **Precompile standard library** — провести перекомпиляцию стандартной библиотеки.

Последние два пункта связаны с загрузкой компонентов для отладки, их мы устанавливать не будем.

5. После успешной установки Python вас ждет следующее сообщение (рис. 1.10).

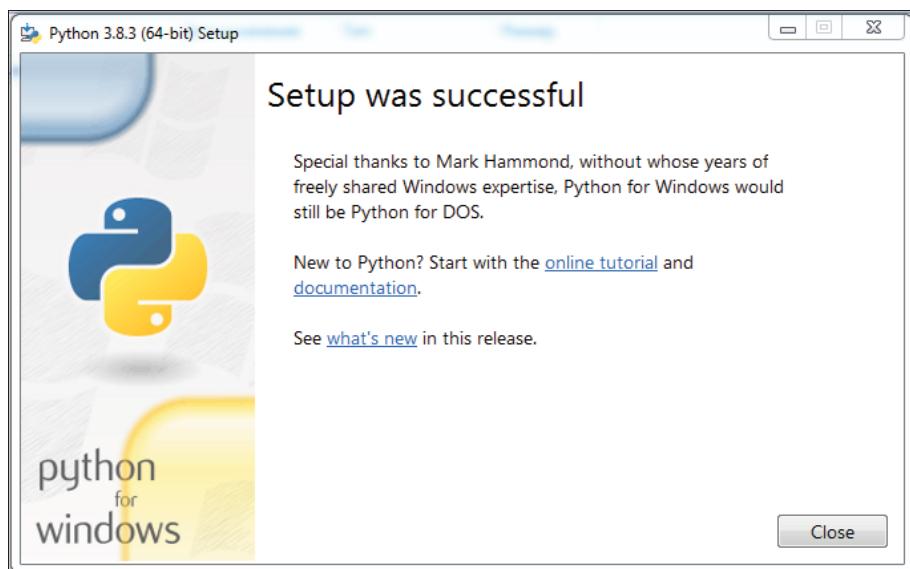


Рис. 1.10. Финальное сообщение после установки Python

1.5.2. Установка Python в Linux

Чаще всего интерпретатор Python уже входит в состав дистрибутива Linux. Это можно проверить, набрав в окне терминала команду:

> python

или

> python3

В первом случае вы запустите Python 2, во втором — Python 3. В будущем, скорее всего, во все дистрибутивы Linux, включающие Python, будет входить только третья его версия. Если у вас при попытке запустить Python выдается сообщение о том, что он не установлен или установлен, но не тот, что вы хотите, то у вас есть возможность взять его из репозитория.

Для установки Python из репозитория Ubuntu воспользуйтесь командой:

```
> sudo apt-get install python3
```

1.5.3. Проверка интерпретатора Python

Для начала протестируем интерпретатор в командном режиме. Если вы работаете в Windows, то нажмите комбинацию клавиш <Win>+<R> и в открывшемся окне введите: python. В Linux откройте окно терминала и в нем введите: python3 (или python).

В результате Python запустится в командном режиме. Выглядеть это будет примерно так, как показано на рис. 1.11 (иллюстрация приведена для Windows, в Linux результат будет аналогичным).

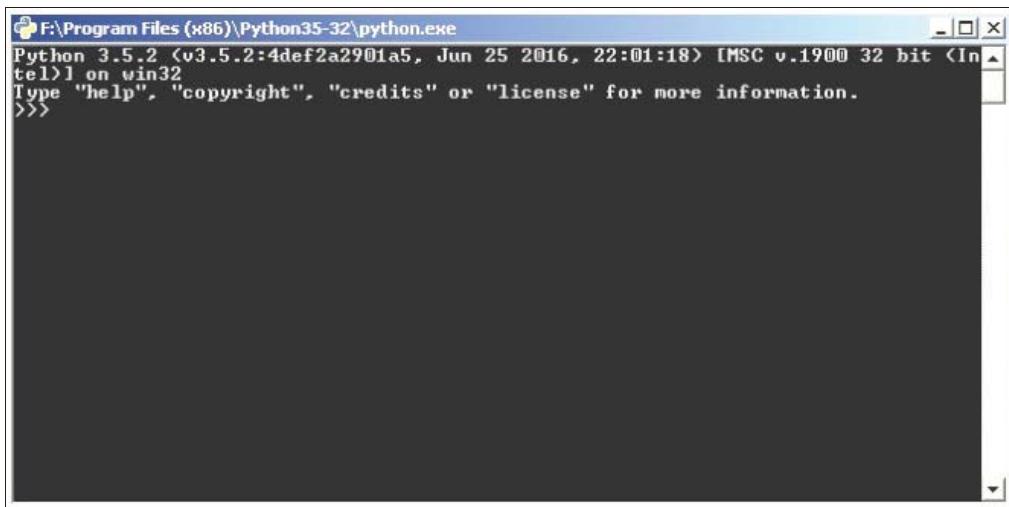


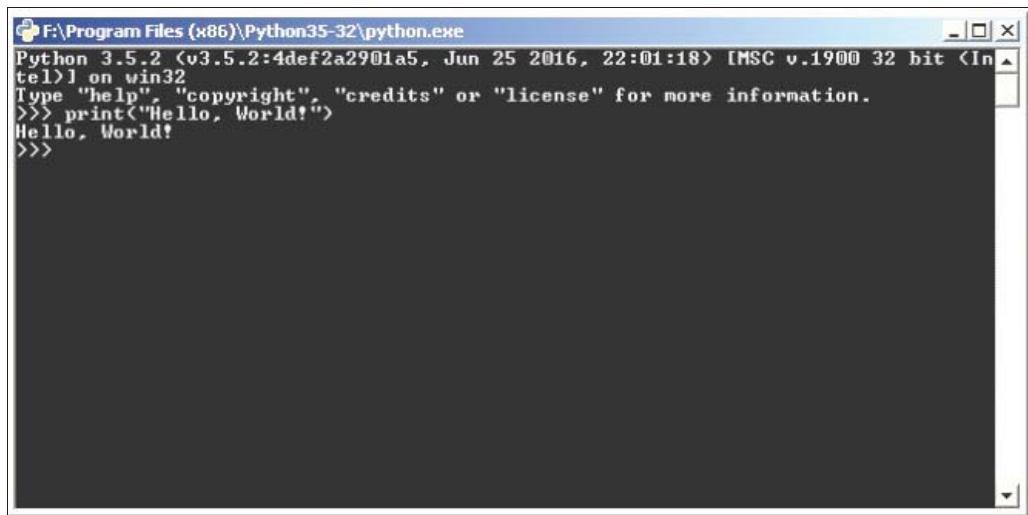
Рис. 1.11. Результат запуска интерпретатора Python в окне терминала

В этом окне введите программный код следующего содержания:

```
print("Hello, World!")
```

В результате вы увидите следующий ответ (рис. 1.12).

Получение такого результата означает, что установка интерпретатора Python прошла без ошибок.



```
F:\Program Files (x86)\Python35-32\python.exe
Python 3.5.2 |v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18| [MSC v.1900 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
>>>
```

Рис. 1.12. Результат работы программы на Python в окне терминала

1.6. Интерактивная среда разработки программного кода PyCharm

В процессе разработки программных модулей удобнее работать в интерактивной среде разработки (IDE), а не в текстовом редакторе. Для Python одним из лучших вариантов считается IDE PyCharm от компании JetBrains. Для скачивания его дистрибутива перейдите по ссылке: <https://www.jetbrains.com/pycharm/download/> (рис. 1.13).

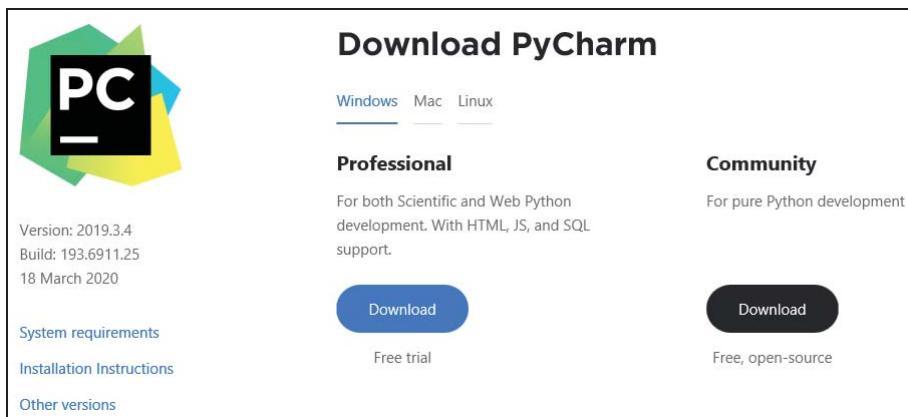


Рис. 1.13. Главное окно сайта для скачивания дистрибутива PyCharm

Эта среда разработки доступна для Windows, Linux и macOS. Существуют два вида лицензии PyCharm: **Professional** и **Community**. Мы будем использовать версию **Community**, поскольку она бесплатная и ее функционала более чем достаточно

для наших задач. На момент подготовки этой книги была доступна версия PyCharm 2020.1.2.

1.6.1. Установка PyCharm в Windows

1. Запустите на выполнение скачанный дистрибутив PyCharm (рис. 1.14).
2. Выберите путь установки программы (рис. 1.15).



Рис. 1.14. Начальная заставка при инсталляции PyCharm



Рис. 1.15. Выбор пути установки PyCharm

3. Укажите ярлыки, которые нужно создать на рабочем столе (запуск 32- или 64-разрядной версии PyCharm), и отметьте флажком опцию **.py** в области **Create associations**, если требуется ассоциировать с PyCharm файлы с расширением **.py** (рис. 1.16).
4. Выберите имя для папки в меню **Пуск** (рис. 1.17).
5. Далее PyCharm будет установлен на ваш компьютер (рис. 1.18).

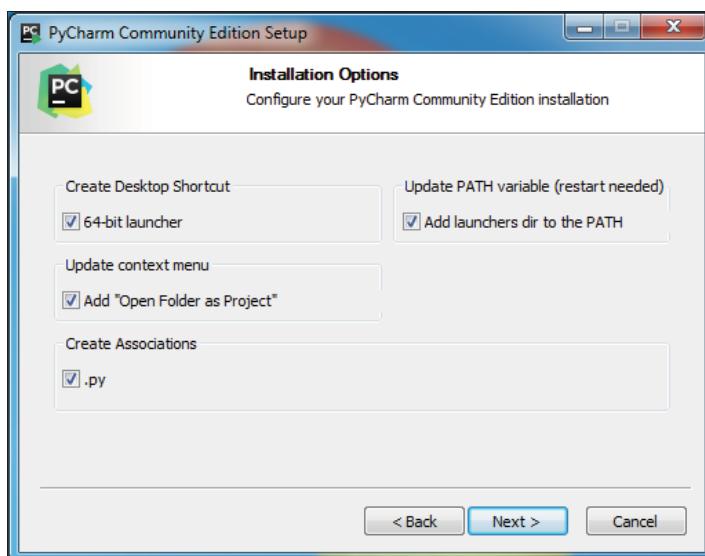


Рис. 1.16. Выбор разрядности устанавливаемой среды разработки PyCharm

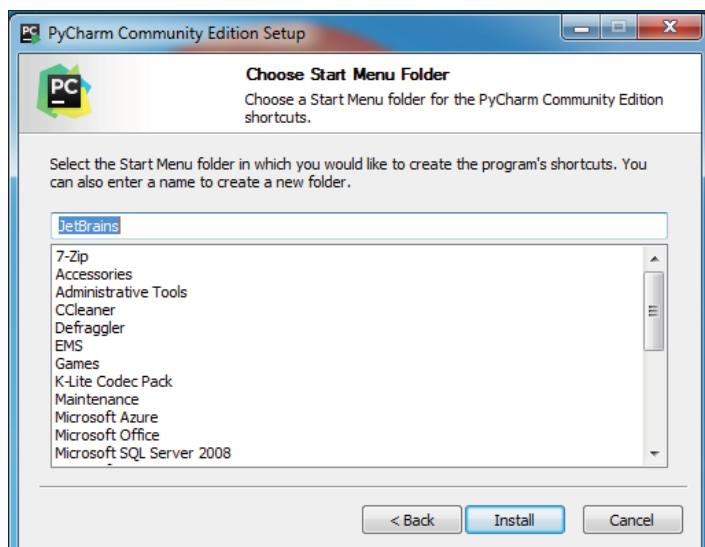


Рис. 1.17. Выбор имени папки для PyCharm в меню Пуск

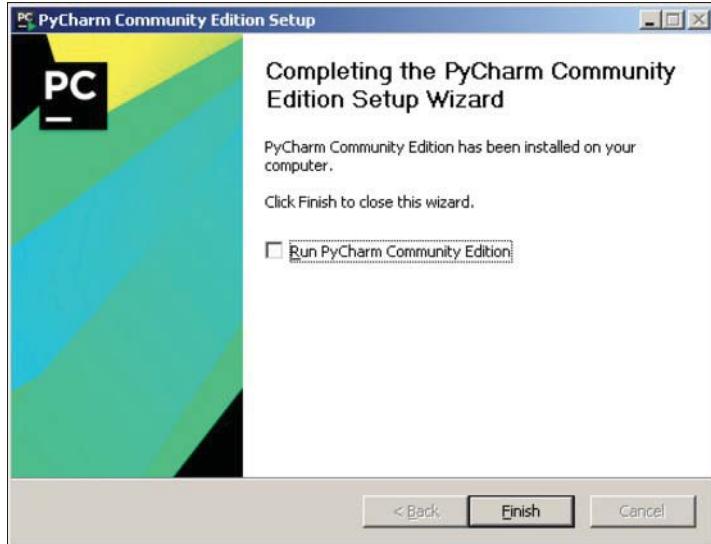


Рис. 1.18. Финальное окно установки пакета PyCharm

1.6.2. Установка PyCharm в Linux

1. Скачайте с сайта программы ее дистрибутив на свой компьютер.
2. Распакуйте архивный файл, для чего можно воспользоваться командой:

```
> tar xvf имя_архива.tar.gz
```

Результат работы этой команды представлен на рис. 1.19.

A screenshot of a terminal window titled "Terminal - user@user-VirtualBox: ~/Downloads/pycharm". The window contains the following command-line session:

```
File Edit View Terminal Tabs Help
user@user-VirtualBox:~/Downloads/pycharm$ ls
pycharm-community-5.0.4.tar.gz
user@user-VirtualBox:~/Downloads/pycharm$ tar xvf pycharm-community-5.0.4.tar.gz
```

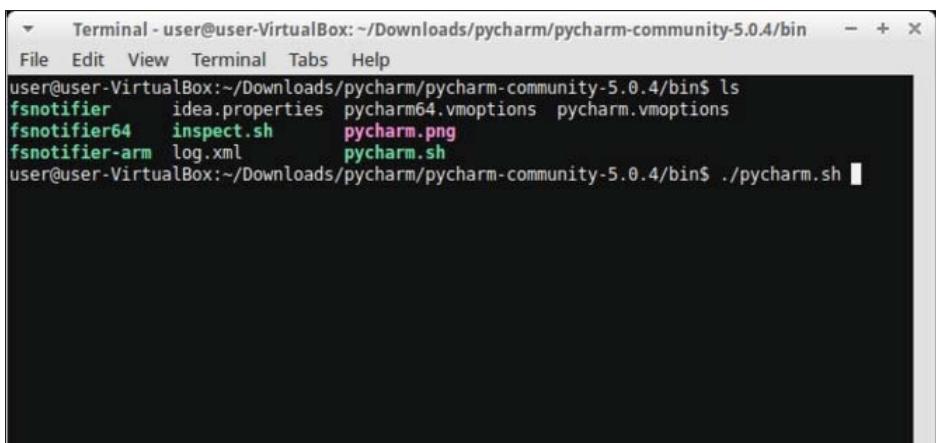
The terminal window has a standard Linux-style interface with a menu bar and tabs.

Рис. 1.19. Результат работы команды распаковки архива PyCharm

3. Перейдите в каталог, который был создан после распаковки дистрибутива, найдите в нем подкаталог bin и зайдите в него. Запустите установку PyCharm командой:

```
> ./pycharm.sh
```

Результат работы этой команды представлен на рис. 1.20.



```
Terminal - user@user-VirtualBox:~/Downloads/pycharm/pycharm-community-5.0.4/bin - + x
File Edit View Terminal Tabs Help
user@user-VirtualBox:~/Downloads/pycharm/pycharm-community-5.0.4/bin$ ls
fsnotifier    idea.properties pycharm64.vmoptions pycharm.vmoptions
fsnotifier64   inspect.sh     pycharm.png
fsnotifier-arm log.xml       pycharm.sh
user@user-VirtualBox:~/Downloads/pycharm/pycharm-community-5.0.4/bin$ ./pycharm.sh
```

Рис. 1.20. Результаты работы команды инсталляции PyCharm

1.6.3. Проверка PyCharm

1. Запустите PyCharm и выберите вариант **Create New Project** в открывшемся окне (рис. 1.21).

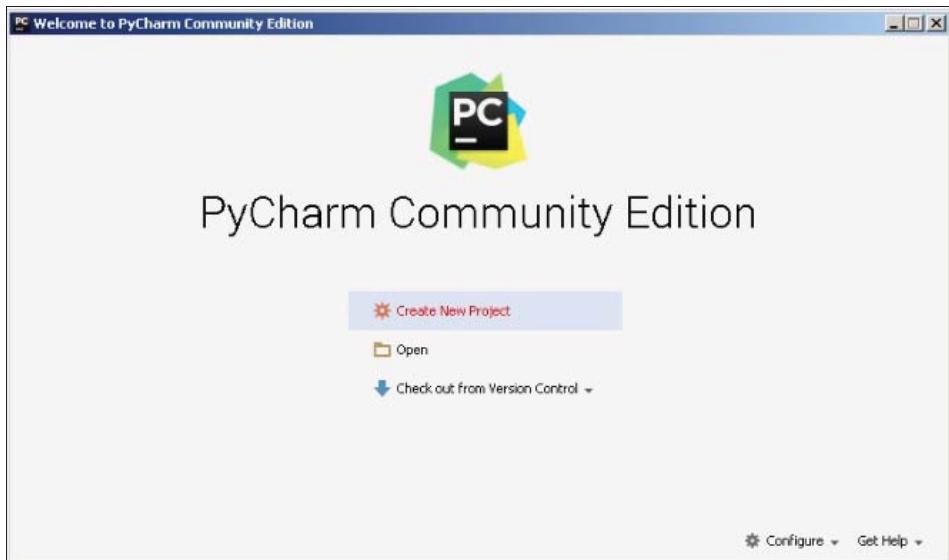


Рис. 1.21. Создание нового проекта в среде разработки PyCharm

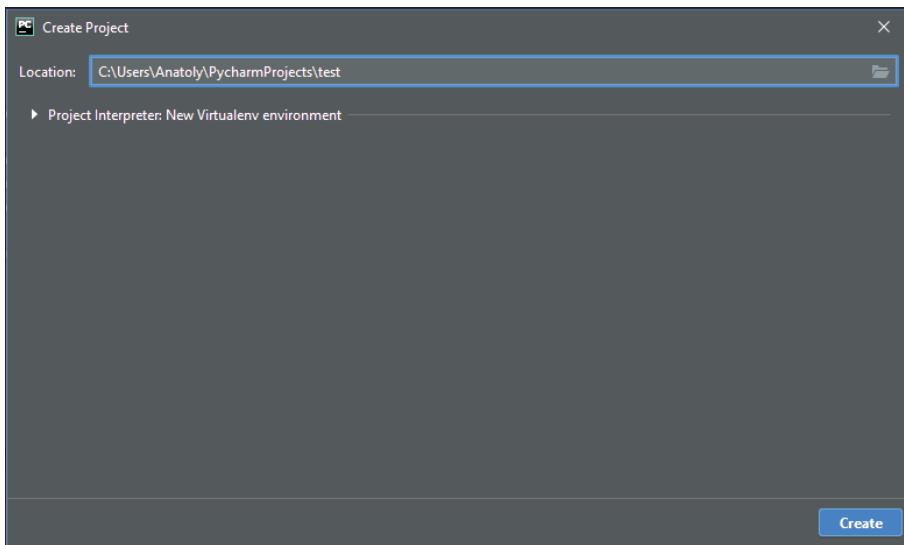


Рис. 1.22. Указание пути до проекта в среде разработки PyCharm

2. Укажите путь до создаваемого проекта Python и интерпретатор, который будет использоваться для его запуска и отладки (рис. 1.22).
3. Добавьте в проект файл, в котором будет храниться программный код Python (рис. 1.23).
4. Введите одну строчку кода программы (рис. 1.24).

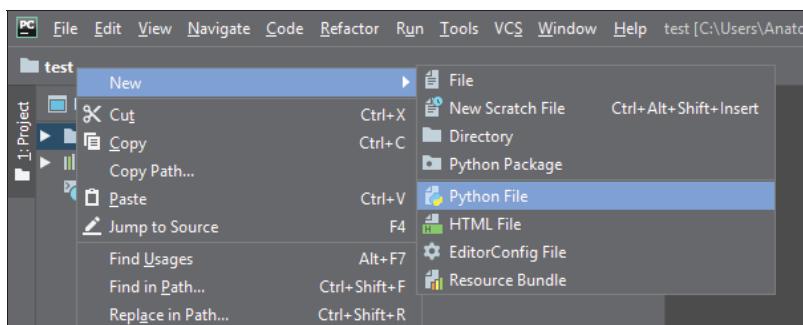


Рис. 1.23. Добавление в проект файла для программного кода на Python

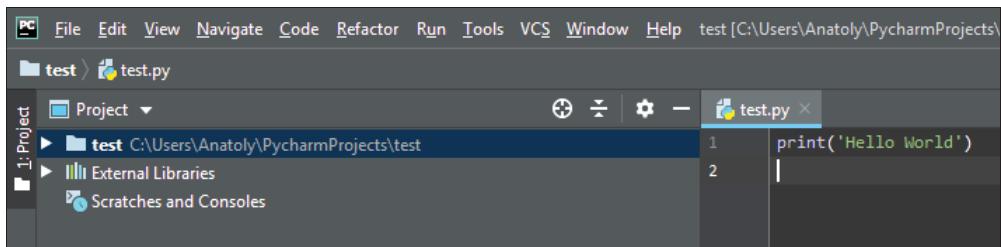


Рис. 1.24. Одна строка программного кода на Python в среде разработки PyCharm

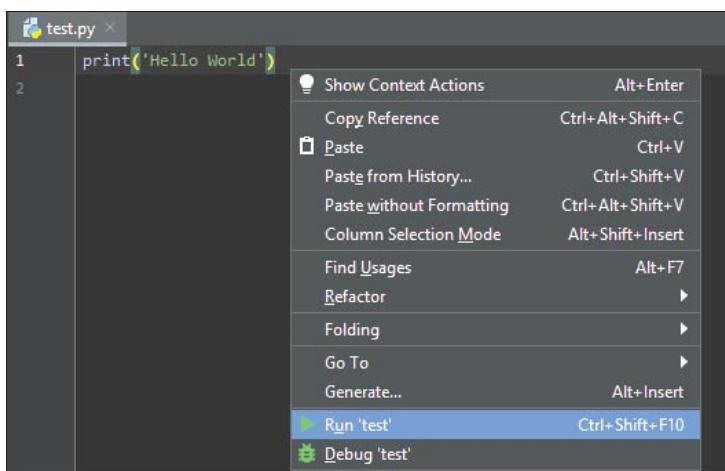


Рис. 1.25. Запуск строки программного кода на Python в среде разработки PyCharm

5. Запустите программу командой **Run** (рис. 1.25).

В результате в нижней части экрана должно открыться окно с выводом результатов работы программы (рис. 1.26).

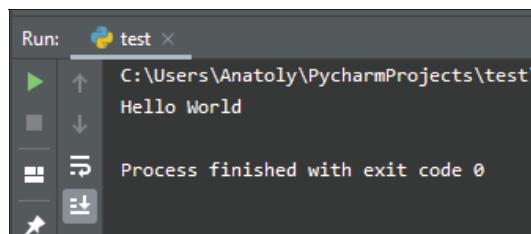


Рис. 1.26. Вывод результатов работы программы на Python в среде разработки PyCharm

1.7. Установка пакетов в Python с использованием менеджера пакетов pip

В процессе разработки программного обеспечения на Python часто возникает необходимость воспользоваться пакетом (библиотекой), который в текущий момент отсутствует на вашем компьютере.

В этом разделе вы узнаете о том, откуда можно взять нужный вам дополнительный инструментарий для разработки ваших программ. В частности:

- где взять отсутствующий пакет;
- как установить pip — менеджер пакетов в Python;
- как использовать pip;
- как установить пакет;
- как удалить пакет;

- как обновить пакет;
- как получить список установленных пакетов;
- как выполнить поиск пакета в репозитории.

1.7.1. Репозиторий пакетов программных средств PyPI

Необходимость в установке дополнительных пакетов возникнет довольно часто, поскольку решение практических задач обычно выходит за рамками базового функционала, который предоставляет Python. Это, например, создание веб-приложений, обработка изображений, распознавание объектов, нейронные сети и другие элементы искусственного интеллекта, геолокация и т. п. В таком случае необходимо узнать, какой пакет содержит функционал, который вам необходим, найти его, скачать, разместить в нужном каталоге и начать использовать. Все указанные действия можно выполнить вручную, однако этот процесс поддается автоматизации. К тому же скачивать пакеты с неизвестных сайтов может быть весьма опасно.

В рамках Python все эти задачи автоматизированы и решены. Существует так называемый Python Package Index (PyPI) — репозиторий, открытый для всех разработчиков на Python, в котором вы можете найти пакеты для решения практически любых задач. При этом у вас отпадает необходимость в разработке и отладке сложного программного кода — вы можете воспользоваться уже готовыми и проверенными решениями огромного сообщества программистов на Python. Вам нужно просто подключить нужный пакет или библиотеку к своему проекту и активировать уже реализованный в них функционал. В этом и заключается преимущество Python перед другими языками программирования, когда небольшим количеством программного кода можно реализовать решение достаточно сложных практических задач. Там также есть возможность выкладывать свои пакеты. Для скачивания и установки нужных модулей в ваш проект используется специальная утилита, которая называется `pip`. Сама аббревиатура, которая на русском языке звучит как «пип», фактически раскрывается как «установщик пакетов» или «предпочитаемый установщик программ». Это утилита командной строки, которая позволяет устанавливать, переустанавливать и деинсталлировать PyPI пакеты простой командой `pip`.

1.7.2. `pip` — менеджер пакетов в Python

`pip` — утилита консольная (без графического интерфейса). После того, как вы ее скачаете и установите, она пропишется в PATH и будет доступна для использования. Эту утилиту можно запускать как самостоятельно — например, через терминал Windows или в терминальном окне PyCharm командой:

```
> pip <аргументы>
```

`pip` можно запустить и через интерпретатор Python:

```
> python -m pip <аргументы>
```

Ключ `-m` означает, что мы хотим запустить модуль (в нашем случае `pip`).

При развертывании современной версии Python (начиная с Python 2.7.9 и более поздних версий) pip устанавливается автоматически. В PyCharm проверить наличие модуля pip достаточно просто — для этого нужно войти в настройки проекта через меню **File | Settings | Project Interpreter**. Модуль pip должен присутствовать в списке загруженных пакетов и библиотек (рис. 1.27).

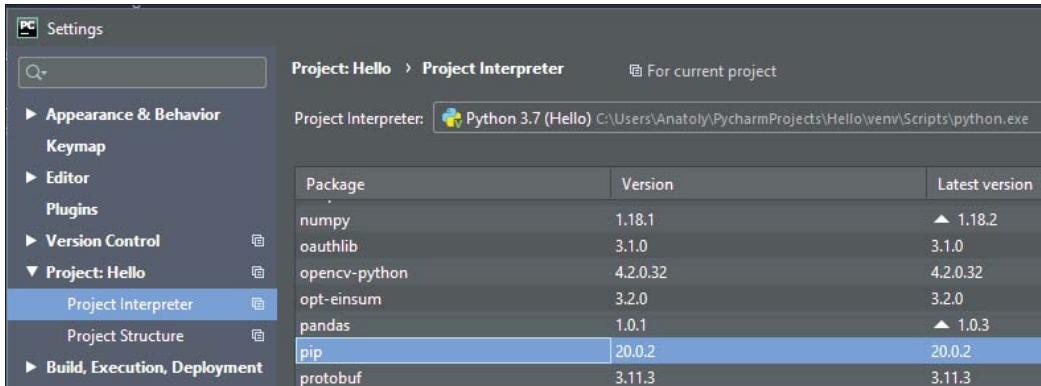


Рис. 1.27. Проверка наличия в проекте модуля pip

В случае отсутствия в списке этого модуля последнюю его версию можно загрузить, нажав на значок + в правой части окна и выбрав модуль pip из списка (рис. 1.28).

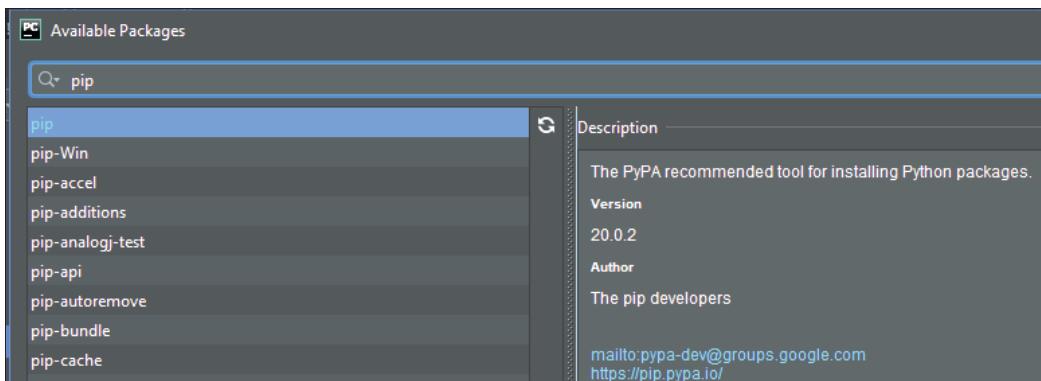


Рис. 1.28. Загрузка модуля pip

1.7.3. Использование менеджера пакетов pip

Здесь мы рассмотрим основные варианты использования pip: установку пакетов, удаление и обновление пакетов.

Pip позволяет установить самую последнюю версию пакета, конкретную версию или воспользоваться логическим выражением, через которое можно определить, что вам, например, нужна версия не ниже указанной. Также есть поддержка уста-

новки пакетов из репозитория. Рассмотрим, как использовать эти варианты (здесь Name — это имя пакета).

□ Установка последней версии пакета:

```
> pip install Name
```

□ Установка определенной версии:

```
> pip install Name==3.2
```

□ Установка пакета с версией не ниже 3.1:

```
> pip install Name>=3.1
```

□ Для того чтобы удалить пакет, воспользуйтесь командой:

```
> pip uninstall Name
```

□ Для обновления пакета используйте ключ --upgrade:

```
> pip install --upgrade Name
```

□ Для вывода списка всех установленных пакетов служит команда:

```
> pip list
```

□ Если вы хотите получить более подробную информацию о конкретном пакете, то используйте аргумент show:

```
> pip show Name
```

□ Если вы не знаете точного названия пакета или хотите посмотреть на пакеты, содержащие конкретное слово, то вы можете это сделать, используя аргумент search:

```
> pip search "test" .
```

Если вы запускаете pip в терминале Windows, то терминальное окно автоматически закроется после того, как эта утилита завершит свою работу. При этом вы просто не успеете увидеть результаты ее работы. Чтобы терминальное окно не закрывалось автоматически, команды pip нужно запускать в нем с ключом /k. Например, запуск процедуры установки пакета tensorflow должен выглядеть так, как показано на рис. 1.29.

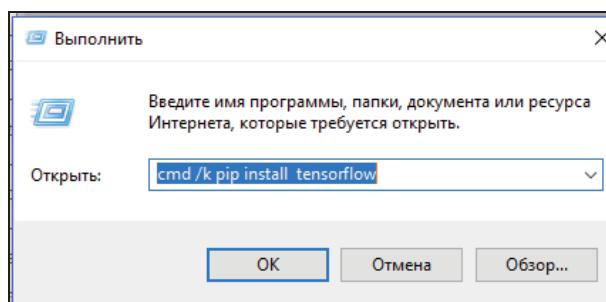


Рис. 1.29. Выполнение команды модуля pip в терминальном окне Windows

Если же пакет `pip` запускается из терминального окна PyCharm, то в использовании дополнительных ключей нет необходимости, т. к. терминальное окно после завершения работы программы не закрывается. Пример выполнения той же команды в терминальном окне PyCharm показан на рис. 1.30.

The screenshot shows the PyCharm interface with a terminal window open. The terminal tab is selected, showing the text "Terminal: Local". The terminal content displays the Windows version information and the command "pip install tensorflow" being run in a virtual environment named "(venv)". The PyCharm navigation bar at the bottom includes tabs for Run, TODO, Terminal, and Python Console.

Рис. 1.30. Выполнение команды модуля `pip` в терминальном окне PyCharm

1.8. Фреймворк Django для разработки веб-приложений

Итак, основной инструментарий для разработки программ на языке Python установлен, и мы можем перейти к установке дополнительных библиотек. В этом разделе мы установим веб-фреймворк Django, который позволяет разрабатывать веб-приложения.

Запустим среду разработки PyCharm и создадим в ней новый проект `Web_1`. Для этого в главном меню среды выполните команду **File | New Project** (рис. 1.31).

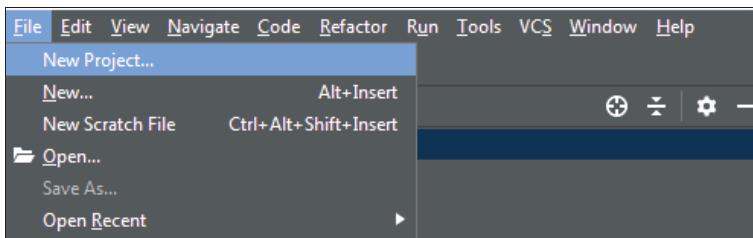


Рис. 1.31. Создание нового проекта в среде разработки PyCharm

Откроется окно, где вы можете задать имя создаваемому проекту, определить виртуальное окружение для этого проекта и указать каталог, в котором находится интерпретатор Python. Задайте новому проекту имя `Web_1` и нажмите кнопку **Create** (рис. 1.32).

Будет создан новый проект. Это, по сути дела, шаблон проекта, в котором пока еще ничего нет (рис. 1.33).

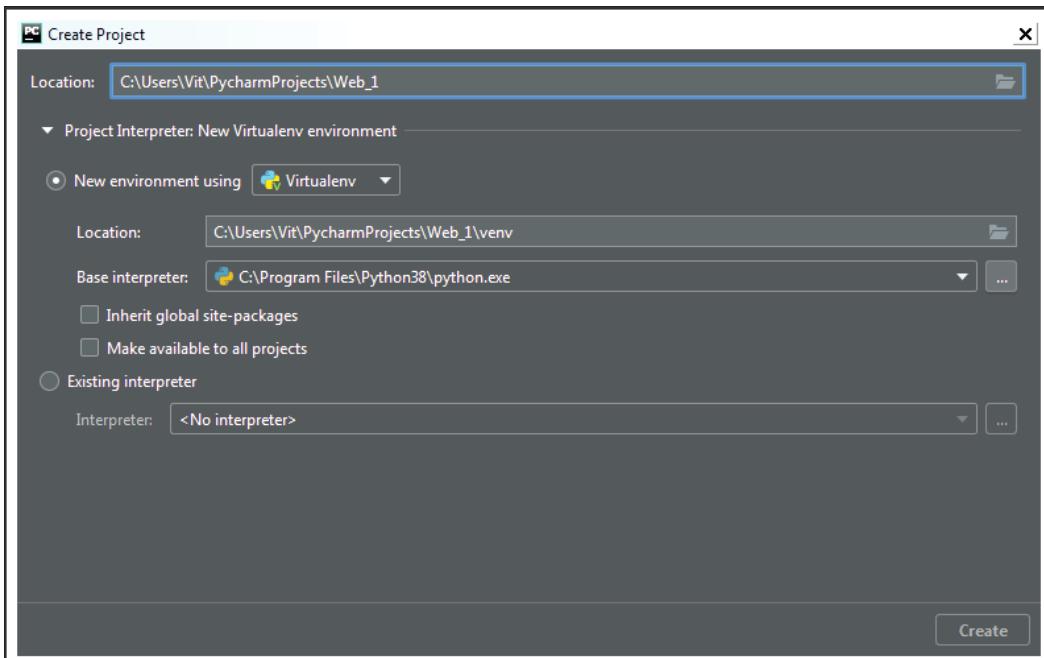


Рис. 1.32. Задаем новому проекту имя Web_1 в среде разработки PyCharm

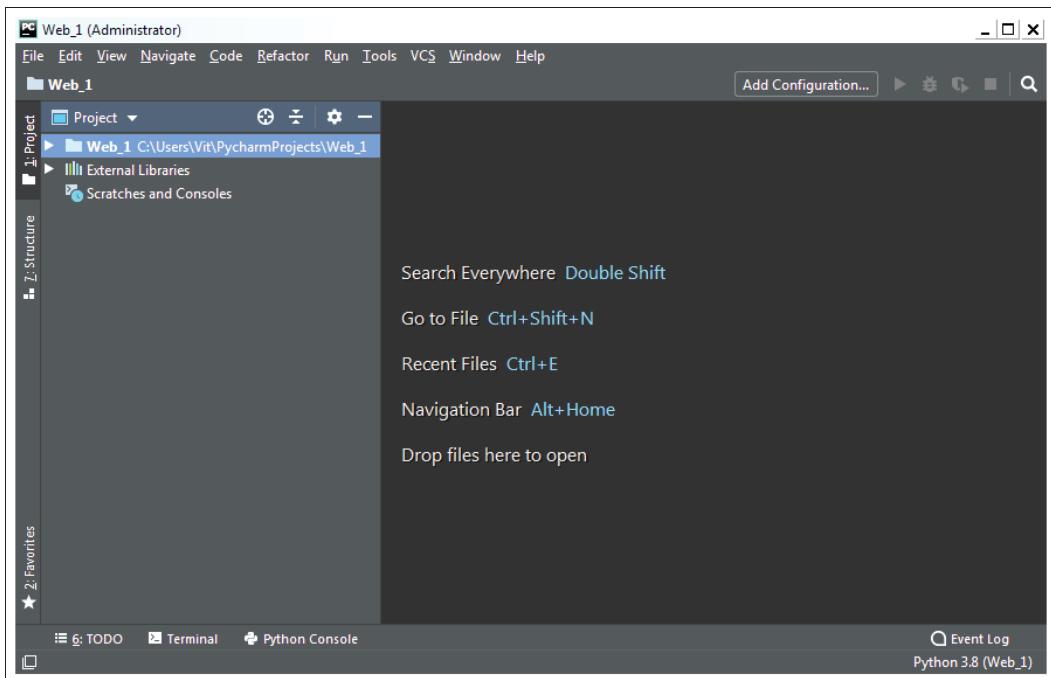


Рис. 1.33. Внешний вид интерфейса PyCharm с пустым проектом

Фреймворк Django — это фактически дополнительная библиотека к Python, и установить этот инструментарий можно так же, как и любую другую библиотеку. Для установки библиотеки Django можно в меню **File** выбрать опцию **Settings** (рис. 1.34).

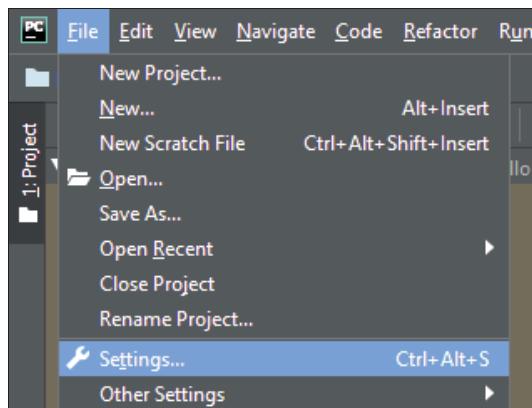


Рис. 1.34. Вызов окна **Settings** настройки параметров проекта

В левой части открывшегося окна настроек выберите опцию **Project Interpreter**, и в правой части окна будет показана информация об интерпретаторе языка Python и подключенных к нему библиотеках (рис. 1.35).

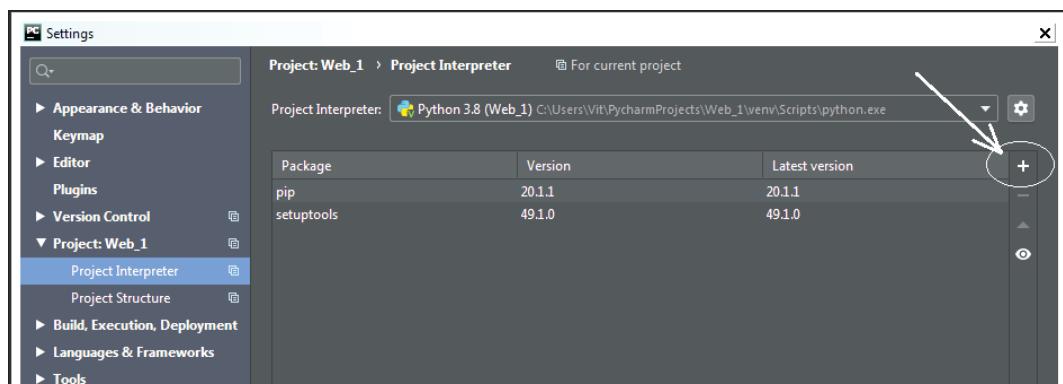


Рис. 1.35. Информация об интерпретаторе языка Python

Чтобы добавить новую библиотеку, нужно нажать на значок в правой части окна, после чего будет отображен полный список доступных библиотек. Здесь можно либо пролистать весь список и найти библиотеку Django, либо набрать наименование этой библиотеки в верхней строке поиска, и она будет найдена в списке (рис. 1.36).

Нажмите теперь на кнопку **Install Package**, и выбранная библиотека будет добавлена в ваш проект (рис. 1.37).

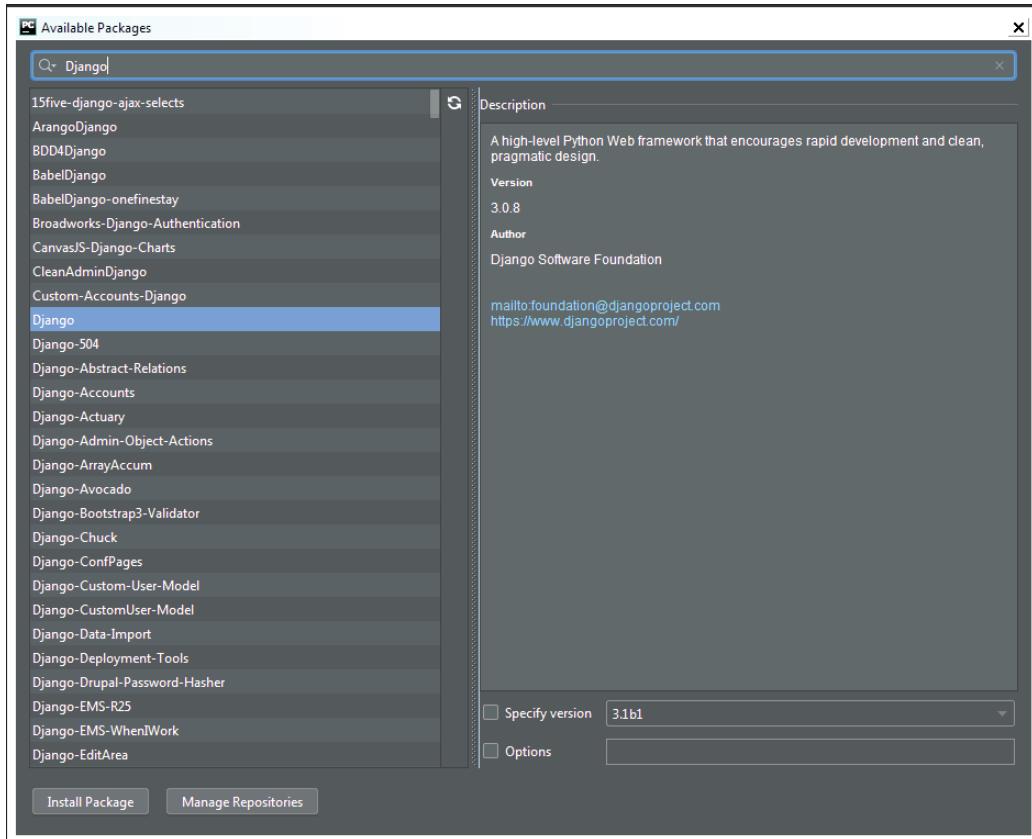


Рис. 1.36. Поиск библиотеки Django в списке доступных библиотек

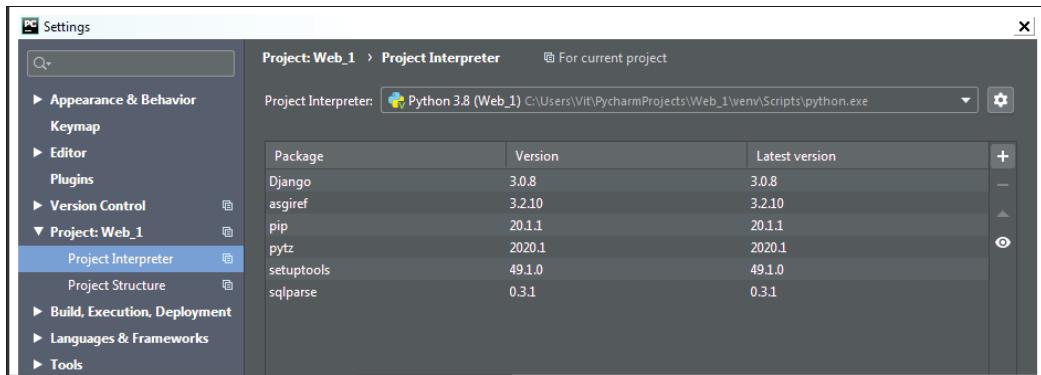


Рис. 1.37. Библиотека Django добавлена в список подключенных библиотек

Теперь у нас есть минимальный набор инструментальных средств, который необходим для разработки веб-приложений на языке Python. Впрочем, в процессе рассмотрения конкретных примеров нам понадобится загрузка еще ряда дополнительных пакетов и библиотек. Их описание и процедуры подключения будут представлены в последующих главах.

1.9. Фреймворк SQLiteStudio для работы с базами данных

Поскольку мы планируем создавать сайты, которые взаимодействуют с базами данных (БД), то нам понадобится программное средство, позволяющее взаимодействовать с такими базами. По умолчанию Django использует СУБД SQLite. Скачать установщик менеджера баз данных SQLite для ПК с ОС Windows можно по ссылке: <https://www.filehorse.com/download-sqlitestudio/>.

После запуска загрузочного файла мы увидим, что это программное средство имеет русскоязычный интерфейс (рис. 1.38).

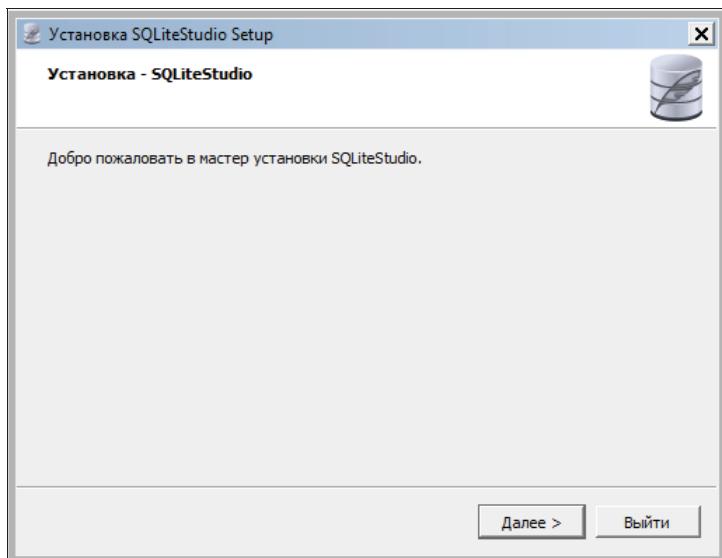


Рис. 1.38. Начальное окно установки менеджера SQLiteStudio

В следующих окнах, которые мы пропустим, следует выполнять рекомендации мастера установки. В завершающем окне нам будет предложено сразу запустить менеджер SQLiteStudio (рис. 1.39).

Нажмите здесь на кнопку **Завершить**, и откроется окно выбора языка интерфейса (рис. 1.40).

Выбираем русский язык и нажимаем на кнопку **OK**. Если процесс установки прошел корректно, то на экран будет выведено главное окно менеджера SQLiteStudio (рис. 1.41).

Как можно видеть, несмотря на выбор русского языка в окне выбора языка интерфейса (см. рис. 1.40), меню в главном окне менеджера SQLiteStudio выведено в английском исполнении. Это, скорее всего, не совсем корректная локализация именно этой версии программного продукта — часть интерфейса осталась не локализованной на русский язык.

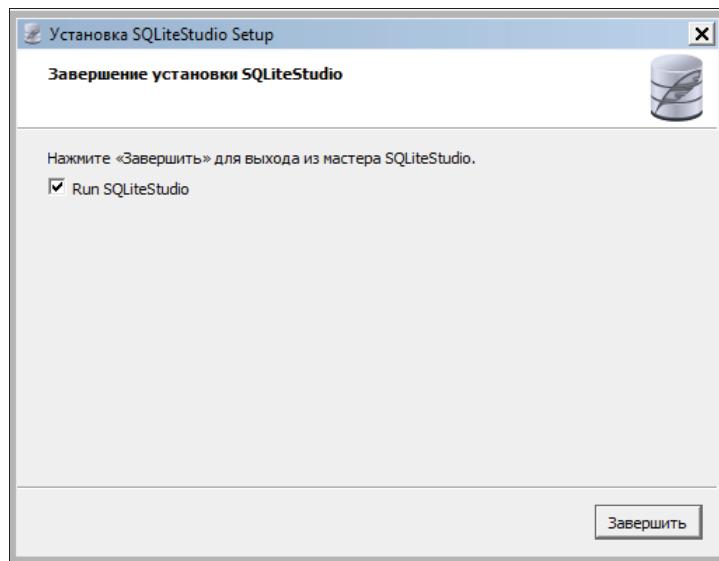


Рис. 1.39. Завершающее окно установки менеджера SQLiteStudio

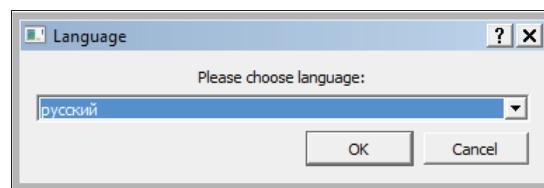


Рис. 1.40. Окно выбора языка интерфейса менеджера SQLiteStudio

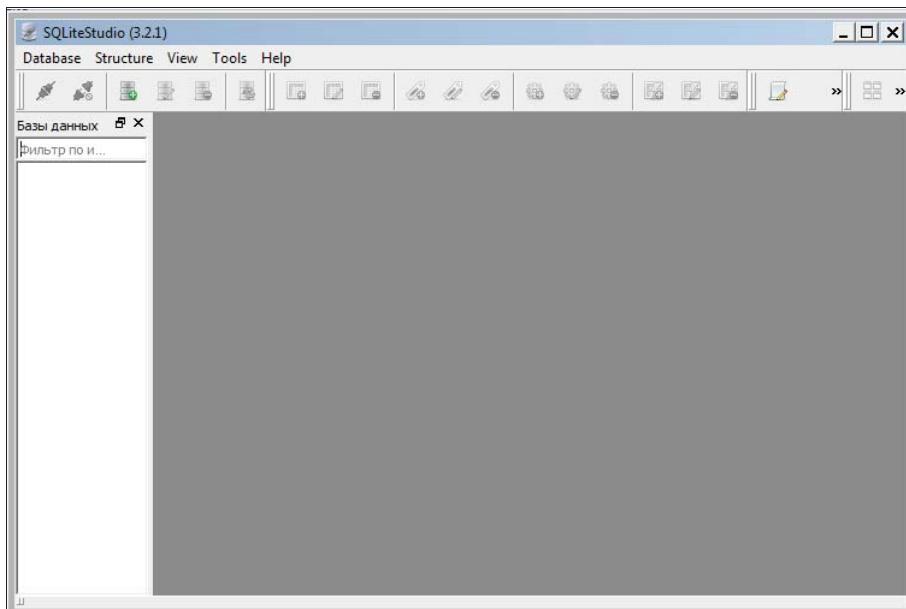


Рис. 1.41. Главное окно менеджера SQLiteStudio

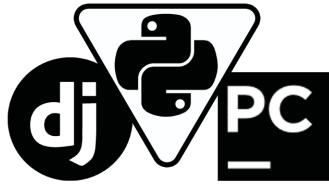
1.10. Краткие итоги

В этой главе мы познакомились с базовыми сведениями о веб-программировании и основными инструментальными средствами, с помощью которых можно разрабатывать веб-приложения. Это интерпретатор Python, интерактивная среда разработки программного кода PyCharm, фреймворк Django для разработки веб-приложений и менеджер работы с базами данных SQLiteStudio. Установив на свой компьютер эти инструментальные средства, теоретически можно приступать к написанию программного кода и созданию HTML-страниц. Однако наличие инструмента — это необходимое, но не достаточное условие для того, чтобы приступить к программированию. Нужно еще и уметь использовать этот инструментарий.

Те специалисты, которые имеют достаточный опыт программирования на Python, могут пропустить следующую главу, т. к. в ней даются самые элементарные представления об этом языке программирования. Для тех же, кто только начинает знакомиться с миром Python, следующую главу рекомендуется не просто прочитать, а сесть за компьютер и самостоятельно создать небольшие программные модули, о которых там пойдет речь. А после того, как будут получены элементарные навыки работы с Python, постараться получить более глубокие знания и навыки работы с этим языком программирования, обратившись к предназначенной для этих целей специальной литературе.

Итак, переходим к следующей главе и знакомимся с Python.

ГЛАВА 2



Основы языка программирования Python

Согласно индексу TIOBE (Ежемесячный индикатор популярности языков программирования на базе подсчета результатов поисковых запросов) Python три раза определялся языком года: в 2007, 2010 и 2018 годах. Эта награда присуждается тому языку программирования, который имеет самый высокий рост рейтинга за год. И действительно, по многочисленным обзорам и рейтингам язык занимает высокие позиции. Так, по данным DOU (Direct Observation Unit, Блок прямого наблюдения за рейтингами языков программирования), он находится на общем четвертом месте и занимает третью позицию в веб-технологиях. Для решения большинства задач и, в частности, для веб-разработки, для обработки научных данных, для создания нейронных сетей и машинного обучения, для работы с большими данными — это один из лучших языков. Как и любой другой язык программирования, Python имеет плюсы и минусы. Однако количество разработчиков, увлеченных им, растет, как и число проектов, взаимно требующих Python-специалистов. Python развивается и будет развиваться еще долго.

Как уже отмечалось ранее, Python — это объектно-ориентированный язык программирования общего назначения, который разработан с целью повышения производительности работы программистов. Он имеет следующие плюсы:

- низкий порог входления — синтаксис Python достаточно понятен для новичков;
- это логичный, лаконичный и понятный язык программирования (разве что Visual Basic может сравниться с ним по этим параметрам);
- это кроссплатформенный язык, который подходит для разных платформ (Linux и Windows);
- есть реализация интерпретаторов для мобильных устройств и непопулярных систем;
- имеет широкое применение — используется для разработки веб-приложений, игр, математических вычислений, машинного обучения, в области Интернета вещей);
- пользуется сильной поддержкой в Интернете со стороны своих приверженцев;

- имеет мощную поддержку компаний-гигантов IT-индустрии (Google, Facebook, Dropbox, Spotify, Quora, Netflix);
- обеспечивает высокую потребность в программах своего направления на рынке труда;
- в мире Python много качественных библиотек, так что не приходится «изобретать велосипед», если надо срочно решить какую-то коммерческую задачу;
- для обучения Python имеется достаточно много литературы — в первую очередь, правда, на английском языке, впрочем, и в переводе также издано большое количество книг, пишут учебники по Python и многие отечественные авторы;
- Python отличается строгими требованиями к оформлению кода (требует отступов), что, несомненно, является преимуществом — язык способствует писать код организованно и красиво.

В этой главе мы не будем основательно знакомиться со всеми тонкостями и возможностями Python, а остановимся только на тех операторах и процедурах, которые нам понадобятся в процессе создания примеров, реализующих веб-приложения. В частности, мы рассмотрим следующие его базовые элементы:

- переменные;
- функции;
- массивы;
- условия и циклы;
- классы и объекты (с примером создания собственного класса и объектов на его основе);
- программные модули;
- оконная форма как основа интерфейса.

Мы также выполним подключение Windows-формы к программе на Python и сборку исполняемого файла на Python под Windows.

Итак, начинаем наше небольшое погружение в мир Python и при этом воспользуемся инструментальным средством PyCharm, позволяющим облегчить написание программного кода.

2.1. Первая программа в среде интерпретатора Python

В этом разделе описывается последовательность создания простейшего приложения с использованием программной оболочки PyCharm.

Для создания приложения выполните следующие действия.

1. Запустите PyCharm и в строке главного меню **File** выберите опцию создания нового проекта: **New Project** (рис. 2.1).
2. В открывшемся окне в строке **Location** задайте имя проекта: **MyProject** (рис. 2.2).

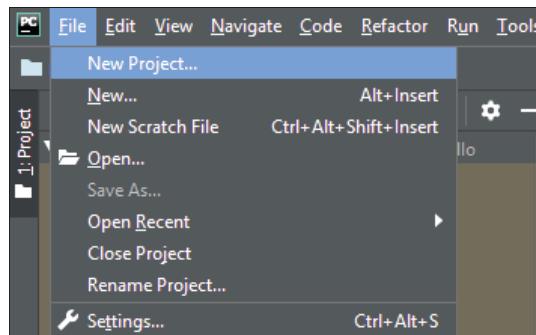


Рис. 2.1. Создание нового проекта в среде программирования PyCharm

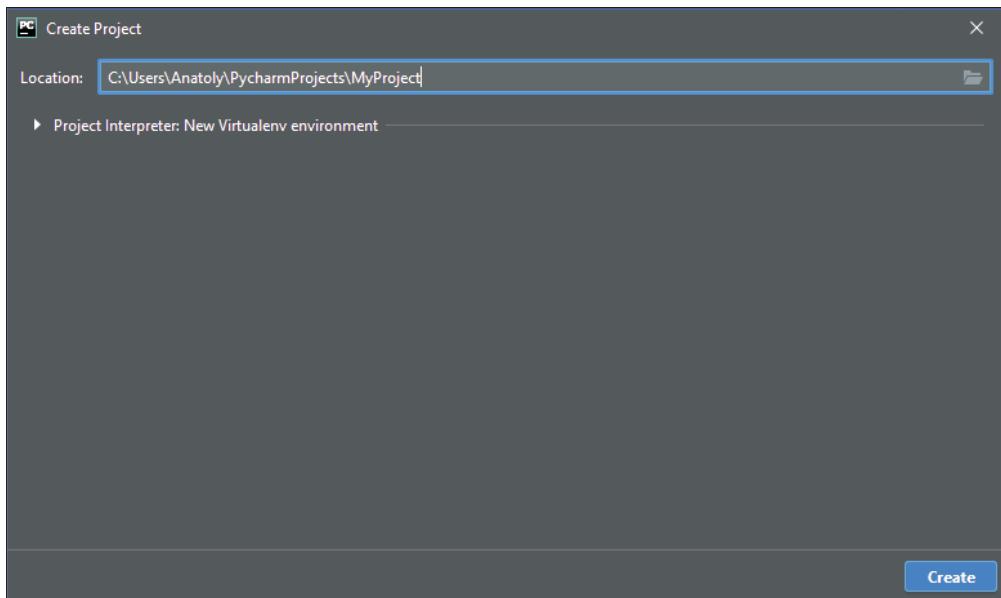


Рис. 2.2. Задание имени новому проекту в среде программирования PyCharm

3. Раскройте строку **Project interpreter: New virtualenv environment** и убедитесь, что в создаваемой виртуальной среде программирования определен интерпретатор языка Python. В нашем конкретном случае мы видим, что это интерпретатор языка Python версии 3.7 (рис. 2.3).
4. Нажмите кнопку **Create**, и будет создан новый проект (рис. 2.4).
5. На следующем шаге необходимо создать файл, в котором будет содержаться программный код на языке Python. Для этого щелкните правой кнопкой мыши в строке **MyProject** и из открывшегося меню выполните команду **New | Python File** (рис. 2.5).
6. В открывшемся окне (рис. 2.6) выберите опцию **Python file** и задайте имя создаваемой программы. Назовем ее `FirstProgram`.

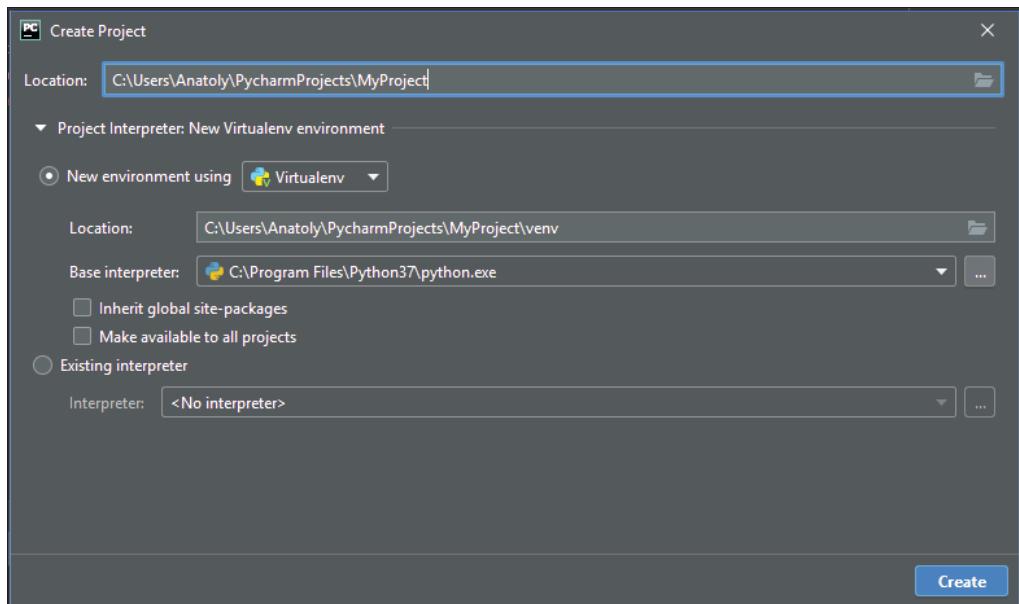


Рис. 2.3. Задание интерпретатора языка Python в виртуальной среде программирования

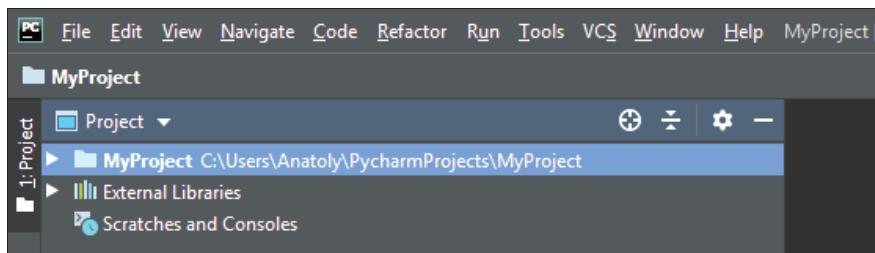


Рис. 2.4. Новый проект в среде программирования PyCharm

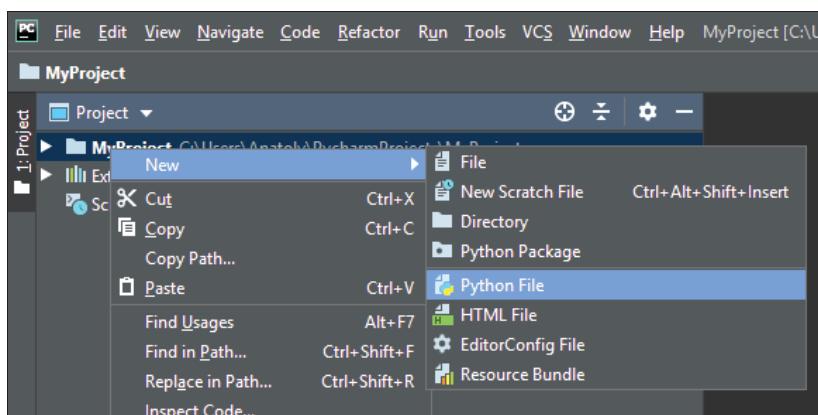


Рис. 2.5. Создание файла для программного кода на языке Python

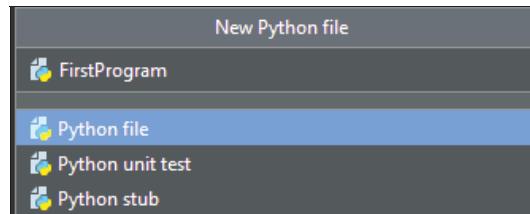


Рис. 2.6. Задание имени файла с программным кодом на языке Python

7. Нажмите на клавишу <Enter> — будет создан соответствующий файл и откроется окно для ввода и редактирования программного кода (рис. 2.7).

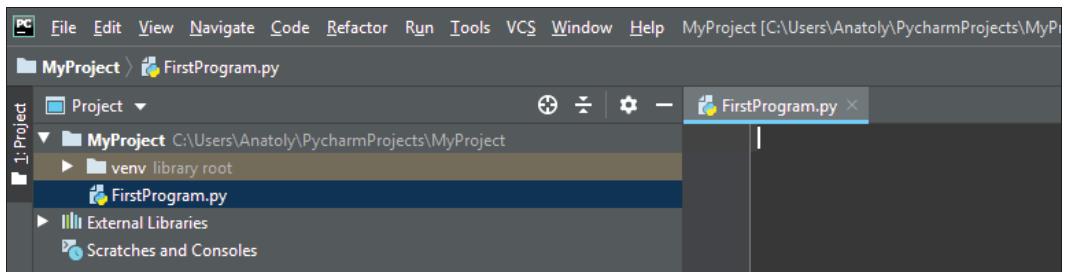


Рис. 2.7. Окно для ввода и редактирования программного кода на языке Python

Напишите в окне редактора одну строку программного кода на языке Python:

```
print("Привет Python")
```

Запустить программу на исполнение можно двумя способами:

- выполнить из главного меню команду **Run | Run 'FirstProgram'** (рис. 2.8);

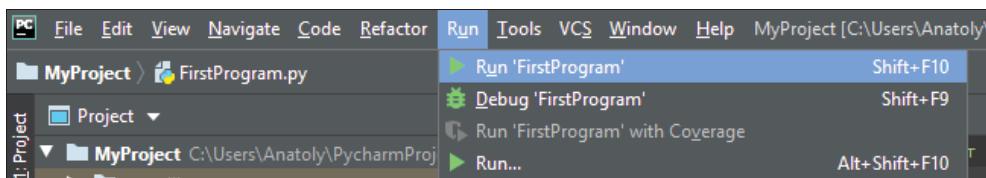


Рис. 2.8. Запуск программы через главное меню PyCharm

- щелкнуть правой кнопкой мыши в окне редактора программного кода и из выпадающего меню выполнить команду **Run 'FirstProgram'** (рис. 2.9).

В обоих случаях программный код будет запущен на исполнение и в нижней части экрана будут показаны результаты его работы (рис. 2.10).

Итак, мы сделали первые шаги по освоению языка программирования Python: загрузили на свой компьютер пакет необходимых инструментальных средств и написали первую программу.

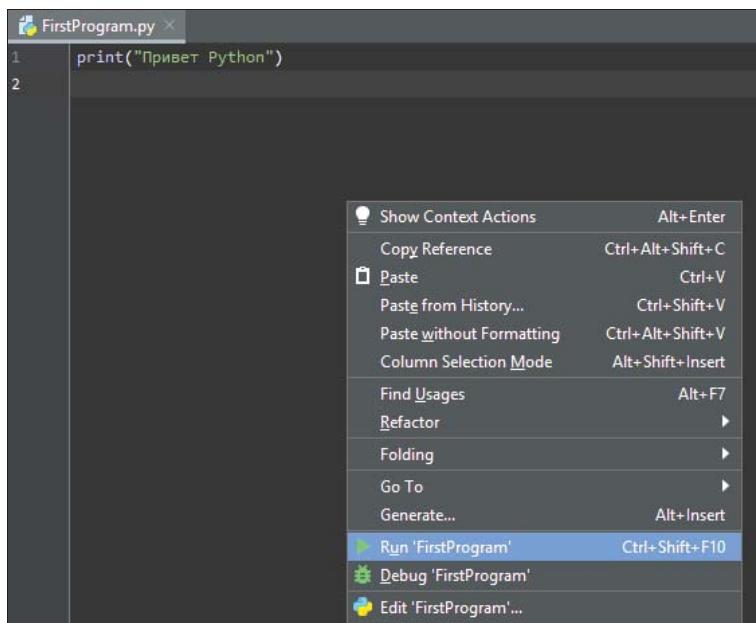


Рис. 2.9. Запуск программы через выпадающее меню PyCharm

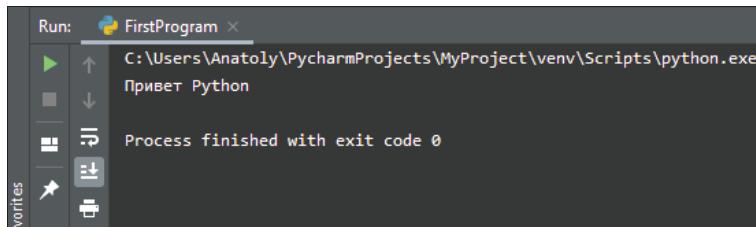


Рис. 2.10. Результаты работы первой программы на языке Python

2.2. Базовые операторы языка Python

Веб-приложения с Django базируются на языке Python. Поэтому очень важно иметь основные представления о том, как писать на нем программы. В этом разделе мы остановимся только на базовых операторах и функциях языка Python — т. е. на тех, которые нам понадобятся для подключения различных пакетов и библиотек, организации циклов и разветвлений, создания объектов, вызова функций и вывода итогов работы программ. Этого будет вполне достаточно для того, чтобы с минимальными трудозатратами на написание программного кода создать нужный функционал, основанный на уже готовых библиотеках и пакетах, созданных сообществом программистов на Python.

Приведенный здесь материал рассчитан на людей, только начинающих знакомиться с языками программирования. Программисты с опытом работы могут его пропустить.

2.2.1. Переменные

Переменная — ключевое понятие в любом языке программирования. Проще всего представить переменную в виде коробки с ярлыком. В этой коробке хранится что-то (число, матрица, объект и т. п.). Например, мы хотим создать две переменные: *x* и *y*, которые должны хранить значения 2 и 3. На Python код создания этих переменных будет выглядеть так:

```
x=2
y=3
```

В левой части этих выражений мы объявляем две переменные с именами: *x* и *y*. Это равносильно тому, что мы бы взяли две коробки и приклеили на них именные ярлыки. Далее идет знак равенства и числа 2 и 3. Знак равенства здесь играет необычную роль. Он не означает, что «*x* равно 2, а *y* равно 3». Равенство здесь означает, что в коробку с именем *x* положили 2 предмета, а в коробку с именем *y* — 3 предмета (рис. 2.11).

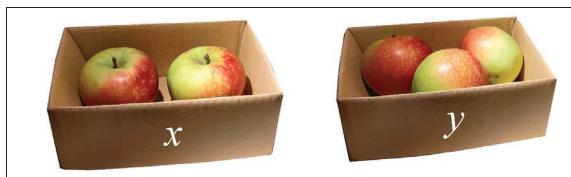


Рис. 2.11. Присвоение значений переменным *x* и *y*

Если говорить более корректно, то мы *присваиваем* переменной *x* число 2, а переменной *y* — число 3. Теперь в программном коде мы можем обращаться к этим переменным и выполнять с ними различные действия. Можно, например, просто вывести значения этих переменных на экран. Программный код в этом случае будет выглядеть следующим образом (листинг 2.1).

Листинг 2.1

```
x = 2
y = 3
print(x, y)
```

Здесь команда `print(x, y)` представляет собой вызов *функции*. Функции мы будем рассматривать чуть позже, а сейчас нам важно понять, что эта функция выводит в терминальное окно то, что расположено внутри скобок (рис. 2.12, *вверху*). Внутри скобок у нас стоят переменные: *x* и *y*. Вспомним, что ранее мы переменной *x* присвоили значение 2, а переменной *y* — значение 3. Соответственно, именно эти значения и будут выведены в окне терминала, если вы выполните эту программу (рис. 2.12, *внизу*).

С переменными, которые хранят числа, можно выполнять различные простейшие действия: складывать, вычитать, умножать, делить и возводить в степень. Текст программы выполнения этих действий приведен в листинге 2.2.

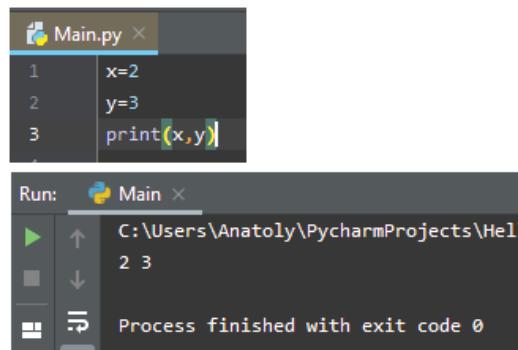


Рис. 2.12. Вывод значений переменных x и y в окне терминала PyCharm

Листинг 2.2

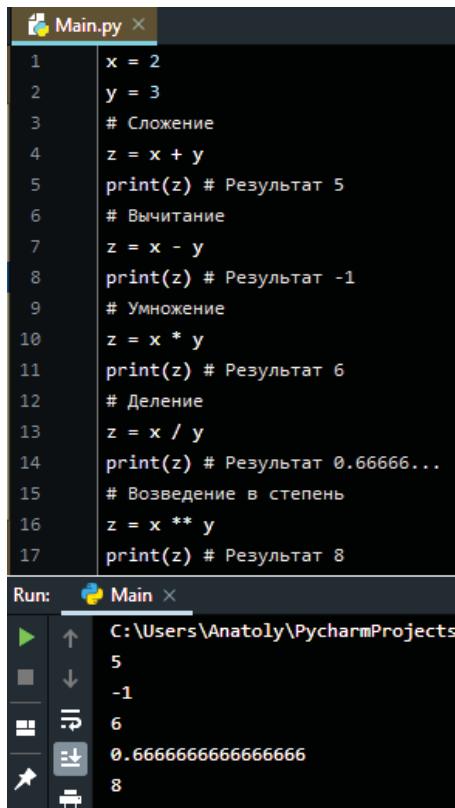
```
x = 2
y = 3
# Сложение
z = x + y
print(z) # Результат 5
# Вычитание
z = x - y
print(z) # Результат -1
# Умножение
z = x * y
print(z) # Результат 6
# Деление
z = x / y
print(z) # Результат 0.66666...
# Возведение в степень
z = x ** y
print(z) # Результат 8
```

Обратите внимание, что в тексте этой программы имеются и невыполняемые строки — т. е. текстовая информация, содержащая некоторые *комментарии*, помогающие пояснить смысл некоторых команд и выполняемых в программе действий. В языке Python любую строку можно закомментировать (превратить в комментарий) символом `#`. Эти строки не выполняются при работе программ, а служат только для облегчения понимания смысла фрагментов программного кода.

Выполнив программу, приведенную в листинге 2.2 (рис. 2.13, *вверху*), мы получим следующий результат (рис. 2.13, *внизу*).

Как можно видеть, в этом программном коде мы вначале создаем две переменные, содержащие значения 2 и 3. Затем создаем переменную z , в которую заносим результат операции с переменными x и y , и выводим значение переменной z в кон-

соль. На этом примере хорошо видно, что переменная может многократно менять свое значение в ходе выполнения программы. Так, наша переменная *z* меняет здесь свое значение пять раз.



```
Main.py
1 x = 2
2 y = 3
3 # Сложение
4 z = x + y
5 print(z) # Результат 5
6 # Вычитание
7 z = x - y
8 print(z) # Результат -1
9 # Умножение
10 z = x * y
11 print(z) # Результат 6
12 # Деление
13 z = x / y
14 print(z) # Результат 0.66666...
15 # Возведение в степень
16 z = x ** y
17 print(z) # Результат 8

Run: Main
C:\Users\Anatoly\PycharmProjects\Main
5
-1
6
0.6666666666666666
8
```

Рис. 2.13. Вывод значений различных действий с переменными *x* и *y* в окне терминала PyCharm

2.2.2. Функции

Функция — это программный код, в котором записаны некоторые часто повторяющиеся действия. Различают системные функции и функции пользователя. К *системным функциям* относятся те, которые уже имеются в структуре языка программирования. В приведенном ранее примере мы использовали системную функцию `print()`.

Кроме системных функций, программист имеет возможность написать свои — пользовательские — функции и многократно обращаться к ним из любой точки программного кода. Их основное назначение заключается в том, чтобы избежать многократного повторения одного и того же программного кода. Задается *функция пользователя* с помощью ключевого слова `def`. За ним следует название функции, потом скобки и двоеточие. В скобках через запятую можно привести *параметры*, которые будут принимать функция. Далее с отступом надо записать действия, которые будут выполнены при вызове функции. Если функция возвращает результат

вычислений, то последней ее строкой должен быть оператор `return` с указанием возвращаемого результата. Для вызова функции нужно указать ее имя и в скобках привести передаваемые в нее *аргументы*. В листинге 2.3 приведен пример программного кода описания и вызова функции.

Листинг 2.3

```
def NameFun(p1, p2):  
    ...  
    ...  
    return result  
  
Itog = NameFun(a1, a2)
```

Здесь `a1` и `a2` — это аргументы, передаваемые в функцию `NameFun`. А в самой функции `NameFun` переменные `p1` и `p2` — это принимаемые параметры. Другими словами, переменные, которые мы передаем функции при ее вызове, называются *аргументами*. А вот внутри функции эти переданные переменные называются *параметрами*. По сути, это два названия одного и того же, но путать их не стоит. Кроме того, следует иметь в виду, что функция может как иметь, так и не иметь параметров. Может возвращать, а может и не возвращать результаты своих действий. Если функция не возвращает результатов, то в ней будет отсутствовать последний оператор `return`.

В примере из предыдущего раздела (см. листинг 2.2) мы уже использовали функцию `print()`. Она относится к системным функциям, которые уже встроены в язык программирования. Мы самостоятельно не писали программный код, который обеспечивает вывод данных, а просто обратились к готовому программному коду. Вернемся еще раз к этому примеру (листинг 2.4).

Листинг 2.4

```
x = 2  
y = 3  
# Сложение  
z = x + y  
print(z) # Результат 5  
# Вычитание  
z = x - y  
print(z) # Результат -1  
# Умножение  
z = x * y  
print(z) # Результат 6
```

Здесь мы многократно меняем значение переменной `z`, а затем передаем это значение в функцию `print` в виде аргумента, т. е. пишем: `print(z)`. В результате функция `print()` многократно выводит пользователю рассчитанное значение переменной `z`.

Сам по себе программный код функции не выполняется до тех пор, пока не будет вызван на исполнение с помощью специальной команды.

Для того чтобы выдавать более осмысленные результаты вычислений, программный код этого примера можно дополнить соответствующими пояснениями (листинг 2.5).

Листинг 2.5

```
x = 2
y = 3

# Сложение
z = x + y
print('Мы выполнили сложение переменных X и Y')
print("Результат Z=", z) # Результат 5

# Вычитание
z = x - y
print('Мы выполнили вычитание переменных X и Y')
print("Результат Z=", z) # Результат -1

# Умножение
z = x * y
print('Мы выполнили умножение переменных X и Y')
print("Результат Z=", z) # Результат 6

# Деление
z = x / y
print('Мы выполнили деление переменных X и Y')
print("Результат Z=", z) # Результат 0.6

# Возвведение в степень
z = x ** y
print('Мы выполнили возвведение в степень переменных X и Y')
print("Результат Z=", z) # Результат 8
```

Еще раз убедимся, что для каждого действия над переменными `x` и `y` повторяется один и тот же программный код вызова функции `print` с разными аргументами. Значит, имеет смысл создать собственную пользовательскую функцию, в которой будут выполняться эти действия, — создадим такую функцию с именем `MyPrint` (листинг 2.6).

Листинг 2.6

```
def MyPrint(d,r):
    print('В функции мы выполнили печать ', d, 'переменных X и Y')
    print('Результат Z=', r)
```

Эта функция принимает два параметра: `d` и `r` — и в следующих двух строках выполняет вывод значений этих двух параметров с помощью команды `print()`. Обратиться к функции можно, указав ее имя и записав в скобках те аргументы, которые нужно передать в функцию. В листинге 2.7 приведен фрагмент программы обращения к функции `MyPrint`. Полный ее код показан на рис. 2.14, *вверху*, а результаты работы функции `MyPrint` — на рис. 2.14, *внизу*.

The screenshot shows the PyCharm IDE interface. The top window is titled "Main.py" and contains the following Python code:

```
1 def MyPrint(d,r):
2     print('В функции мы выполнили печать ', d, 'переменных X и Y')
3     print('Результат Z=', r)
4
5 x = 2
6 y = 3
7 z = x + y
8 MyPrint('сложения', z)
9 z = x - y
10 MyPrint('вычитания', z)
11 z = x * y
12 MyPrint('умножения', z)
13 z = x / y
14 MyPrint('деления', z)
15 z = x ** y
16 MyPrint('возведения в степень', z)
```

The bottom window is titled "Run: Main" and shows the output of the program execution:

```
C:\Users\Anatoly\PycharmProjects\Hello\venv\Scripts\python.exe C:/Users
В функции мы выполнили печать сложения переменных X и Y
Результат Z= 5
В функции мы выполнили печать вычитания переменных X и Y
Результат Z= -1
В функции мы выполнили печать умножения переменных X и Y
Результат Z= 6
В функции мы выполнили печать деления переменных X и Y
Результат Z= 0.6666666666666666
В функции мы выполнили печать возведения в степень переменных X и Y
Результат Z= 8

Process finished with exit code 0
```

Рис. 2.14. Вывод значений различных действий с переменными `X` и `Y` с использованием функции

Листинг 2.7

```
x = 2
y = 3
```

```
z = x + y
MyPrint('сложения', z)
z = x - y
MyPrint('вычитания', z)
z = x * y
MyPrint('умножения', z)
z = x / y
MyPrint('деления', z)
z = x ** y
MyPrint('возведения в степень', z)
```

Как можно видеть, программный код получился короче, а отображение итогов работы программы более понятны конечному пользователю.

Функция может и не иметь параметров — в этом случае после ее имени в скобках ничего не указывается. Например, если в созданную нами функцию не нужно было бы передавать параметры, то ее определение выглядело бы так: `def MyPrint()`.

Написанная нами функция `MyPrint()` выполняет только вывод информации на печать и не возвращает никаких значений. Просто повторять в функции некоторые действия, конечно, удобно. Но это еще не все. Иногда требуется не только передать в функцию значения переменных, но и вернуть из нее результаты сделанных там операций. Таким образом, функция может не только принимать данные и использовать их в процессе выполнения команд, но и возвращать результат.

Напишем простую функцию, которая складывает два переданных ей числа и возвращает результат (листинг 2.8).

Листинг 2.8

```
def f_sum(a, b):
    result = a + b
    return result
```

Первая строка выглядит почти так же, как и в ранее рассмотренных функциях. Внутри скобок находятся две переменные: `a` и `b` — это параметры функции. Наша функция имеет два параметра, т. е. принимает значения двух переменных. Параметры можно использовать внутри функции как обычные переменные. Во второй строке мы создаем переменную `result`, которая равна сумме параметров `a` и `b`, а в третьей строке — возвращаем значение переменной `result`.

Теперь в программном коде мы можем писать обращение к этой функции (листинг 2.9).

Листинг 2.9

```
s = f_sum(2, 3)
print(s)
```

Здесь мы вызываем функцию `f_sum` и передаем ей два аргумента: 2 и 3. Аргумент 2 становится значением переменной `a`, а аргумент 3 — значением переменной `b`. Наша функция складывает переменные: `a + b`, присваивает итог этой операции переменной `result` и возвращает рассчитанное значение в точку вызова. То есть в точке вызова переменной `s` будет присвоен результат работы функции, который оператором `print()` будет выведен пользователю. Выполним этот программный код (рис. 2.15, *вверху*) и посмотрим на результат (рис. 2.15, *внизу*).

The screenshot shows the PyCharm interface. The top window is titled 'my_sum.py' and contains the following Python code:

```
1 def f_sum(a, b):
2     result = a + b
3     return result
4
5
6 s = f_sum(2, 3)
7 print(s)
```

The bottom window is titled 'Run' and shows the output of the script:

```
C:\Users\Anatoly\PycharmProjects>Hello
5
Process finished with exit code 0
```

Рис. 2.15. Результаты работы функции `f_sum`

В языке Python важно строго соблюдать порядок описания и обращения к функции. Сначала функция должна быть описана оператором `def`, и только после этого можно к ней обращаться. Если обращение к функции будет выполнено до ее описания, то мы получим сообщение об ошибке.

2.2.3. Массивы

Если переменную можно представить как коробку, которая что-то хранит (не обязательно число), то *массивы* по этой аналогии можно представить в виде шкафа со множеством полок (рис. 2.16).

Этот шкаф (в нашем случае — массив) носит имя `array`, а каждая его полка — свой номер от 0 до 5 (нумерация полок начинается с нуля). Разместим на всех этих полках различную информацию. На языке Python это будет выглядеть следующим образом:

```
array = [10, 11, 12, 13, 14, "Это текст"]
```

В рассматриваемом случае массив содержит разные элементы: пять чисел и текстовую информацию. Попробуем вывести какой-нибудь элемент массива:

```
array = [10, 11, 12, 13, 14, "Это текст"]
print(array[1])
```

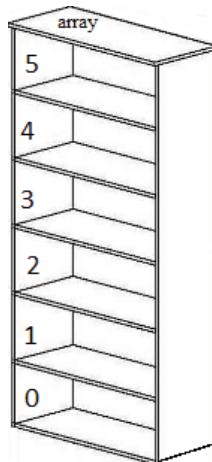


Рис. 2.16. Образное представление массива из шести элементов

В консоли будет выведено число 11. Но почему 11, а не 10? Все дело в том, что в Python, как и во многих других языках программирования, нумерация элементов массивов начинается с 0. Поэтому `array[1]` дает нам второй элемент массива, а не первый. Для вызова первого надо было написать `array[0]`.

Если выполнить следующий программный код:

```
array = [10, 11, 12, 13, 14, "Это текст"]
print(array[5])
```

в консоли будет выведено: **Это текст**.

Иногда бывает очень полезно получить количество элементов в массиве. Для этого можно использовать функцию `len()` — она сама подсчитает количество элементов и вернет их число:

```
array = [10, 11, 12, 13, 14, "Это текст"]
print(len(array))
```

В консоли будет выведено число **6**.

2.2.4. Условия и циклы

По умолчанию любые программы выполняют все команды подряд — от первой строки до последнего оператора. Но есть ситуации, когда необходимо проверить какое-либо условие, и в зависимости от того, правдиво оно или нет, выполнить разные действия. Кроме того, часто возникает необходимость много раз повторить практически одинаковую последовательность команд. В первой ситуации помогают операторы, обеспечивающие выполнение различных фрагментов программы, исходя из соблюдения или несоблюдения некоторого *условия*, а во второй — организация многократно повторяющихся *циклов*.

Условия

Условия нужны, чтобы выполнить два разных набора действий в зависимости от того, истинно или ложно проверяемое утверждение (рис. 2.17).

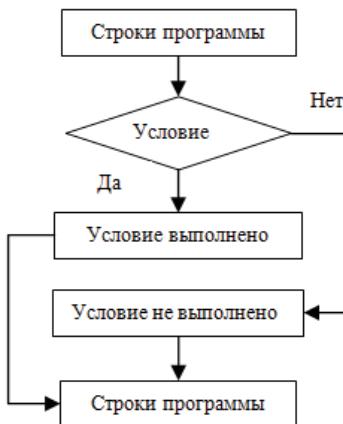


Рис. 2.17. Условный переход к разным частям программного кода

В Python условия можно записывать с помощью конструкции:

```
if: ... else: ...
```

Пусть у нас есть некая переменная x , которой в ходе работы программы было присвоено некоторое значение. Если x меньше 10, то мы делим его на 2. Если же x больше или равно 10, то мы умножаем его на 2. Вот так будет выглядеть программный код при $x = 8$ (листинг 2.10).

Листинг 2.10

```
x = 8
if (x < 10):
    x = x / 2
else:
    x = x * 2
print(x)
```

А результатом работы программы будет значение $x = 4$.

Теперь изменим этот программный код и зададим значение $x = 12$ (листинг 2.11).

Листинг 2.11

```
x = 12
if (x < 10):
    x = x / 2
else:
    x = x * 2
print(x)
```

В этом случае результатом работы программы будет значение $x = 24$.

Рассмотрим этот программный код подробнее. После создания переменной `x` и присвоения ей некоторого значения записывается *условие*. Начинается все с ключевого слова `if` (в переводе с английского «если»). В скобках мы указываем проверяемое выражение. В нашем случае мы хотим убедиться, действительно ли наша переменная `x` меньше 10. Если она на самом деле меньше 10, то мы делим эту переменную на 2. Затем идет ключевое слово `else`, после которого начинается блок действий, которые будут выполнены, если выражение в скобках после `if` — ложное. Если значение переменной `x` больше или равно 10, то мы умножаем эту переменную на 2. И последним оператором выводим значение `x` в консоль.

Циклы

Циклы нужны для многократного повторения действий. Предположим, мы хотим вывести таблицу квадратов первых 10 натуральных чисел. Это можно сделать так (листинг 2.12).

Листинг 2.12

```
print("Квадрат 1 равен " + str(1**2))
print("Квадрат 2 равен " + str(2**2))
print("Квадрат 3 равен " + str(3**2))
print("Квадрат 4 равен " + str(4**2))
print("Квадрат 5 равен " + str(5**2))
print("Квадрат 6 равен " + str(6**2))
print("Квадрат 7 равен " + str(7**2))
print("Квадрат 8 равен " + str(8**2))
print("Квадрат 9 равен " + str(9**2))
print("Квадрат 10 равен " + str(10**2))
```

Здесь в каждой строчке программного кода мы формируем текстовую строку вида "Квадрат 1 равен " и добавляем к ней знаком + новую строку вида: `str(1**2)`. В этом коде используется функция `str`, преобразующая числа в текстовую информацию, — в ее скобках мы возводим указанное число в квадрат. И таких строчек у нас 10.

Рис. 2.18. Результат работы программы возведения в квадрат 10 натуральных чисел

Если мы запустим эту программу на выполнение, то получим следующий результат (рис. 2.18).

А что, если нам надо вывести квадраты первых 100 или 1000 чисел? Неужели нужно будет писать 100 или даже 1000 строк программного кода. Совсем нет, именно для таких случаев и существуют циклы. Всего в Python два вида циклов: `while` и `for`. Рассмотрим их по очереди.

Цикл `while`

Цикл `while` повторяет необходимые команды до тех пор, пока остается истинным некоторое условие. Программный код, приведенный в листинге 2.12, с использованием циклов будет выглядеть следующим образом (листинг 2.13).

Листинг 2.13

```
x = 1
while x <= 100:
    print("Квадрат числа " + str(x) + " равен " + str(x**2))
    x = x + 1
```

Здесь мы сначала создаем переменную `x` и присваиваем ей значение 1. Затем организуем цикл `while` и проверяем, меньше (или равен) ли 100 наш `x`. Если меньше (или равен), то мы выполняем два действия:

- возводим `x` в квадрат;
- увеличиваем `x` на 1.

После выполнения второй команды программа возвращается к проверке условия. Если условие снова истинно, то мы опять выполняем эти два действия. И так до тех пор, пока `x` не станет равным 101. Тогда условие вернет значение ложь, и цикл перестанет выполняться. Результат работы программы возведения в квадрат 100 натуральных чисел с использованием цикла приведен на рис. 2.19 (первые 10 строк — слева, последние — справа).

```
C:\Users\Anatoly\PycharmProjects\my_sum>
Квадрат числа 1 равен 1
Квадрат числа 2 равен 4
Квадрат числа 3 равен 9
Квадрат числа 4 равен 16
Квадрат числа 5 равен 25
Квадрат числа 6 равен 36
Квадрат числа 7 равен 49
Квадрат числа 8 равен 64
Квадрат числа 9 равен 81
Квадрат числа 10 равен 100
...
Квадрат числа 91 равен 8281
Квадрат числа 92 равен 8464
Квадрат числа 93 равен 8649
Квадрат числа 94 равен 8836
Квадрат числа 95 равен 9025
Квадрат числа 96 равен 9216
Квадрат числа 97 равен 9409
Квадрат числа 98 равен 9604
Квадрат числа 99 равен 9801
Квадрат числа 100 равен 10000
```

Рис. 2.19. Результат работы программы возведения в квадрат 100 натуральных чисел с использованием цикла `while`

Цикл `for`

Цикл `for` предназначен для того, чтобы перебирать массивы. Запишем тот же пример с формированием таблицы квадратов первой сотни натуральных чисел (см. листинг 2.13), но уже через цикл `for` (листинг 2.14).

Листинг 2.14

```
for x in range(1, 101):
    print("Квадрат числа " + str(x) + " равен " + str(x**2))
```

Посмотрим на первую строку этого кода. Мы используем здесь для создания цикла ключевое слово `for`. Затем указываем, что хотим повторить определенные действия для всех `x` в диапазоне от 1 до 100. При этом функция `range (1,101)` создает массив из 100 чисел, начиная с 1 и заканчивая 100.

Вот еще пример перебора массива с помощью цикла (листинг 2.15).

Листинг 2.15

```
for i in [1, 10, 100, 1000]:
    print(i * 2)
```

Приведенный здесь код выводит 4 цифры: **2, 20, 200 и 2000** — как результат умножения на 2 элементов массива `[1, 10, 100, 1000]`, сформированного в первой строке. Тут наглядно видно, что в первой строке программы в цикле `for` последовательно перебираются элементы этого массива, а во второй строке они умножаются на 2. Результат работы этой программы представлен на рис. 2.20.

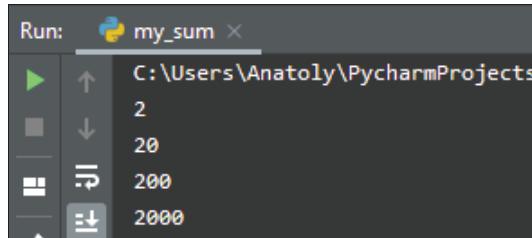


Рис. 2.20. Результат работы программы выполнения действий над элементами массива с использованием цикла `for`

2.2.5. Классы и объекты

В реальной жизни мы оперируем не переменными или функциями, а *объектами*. Светофор, автомобиль, пешеход, кошка, собака, самолет — это все объекты. Взглянем более подробно на объект «кошка». Абсолютно все кошки обладают некоторыми характерными свойствами или параметрами. К ним можно отнести: наличие

хвоста, четырех лап, шерсти, усов и т. п. Но это еще не все. Помимо определенных внешних параметров, кошка может выполнять свойственные ей действия: мурлыкать, шипеть, царапаться, кусаться.

Теперь посмотрим на объект «автомобиль». Практически все автомобили имеют: кузов, колеса, фары, двигатель, тормоза и т. п. Помимо этих внешних параметров, автомобиль может выполнять свойственные ему действия: двигаться, тормозить, издавать звуковой сигнал, выпускать выхлопные газы и т. п.

Только что мы схематически описали некие общие свойства всех кошек и всех автомобилей в целом. Подобное описание характерных свойств и действий какого-либо объекта на различных языках программирования называется *классом*. То есть мы описали здесь два класса: класс «кошки» и класс «автомобили». Класс — просто набор переменных и функций, которые описывают какой-либо объект.

Если все кошки (как класс) имеют общую характеристику — наличие, например, шерсти, то каждая конкретная кошка (как объект) будет иметь свою только ей присущую шерсть, ее цвет и блеск. То же самое касается и автомобиля — все автомобили (как класс) имеют кузов, но каждый конкретный автомобиль (как объект) имеет свою форму кузова, свой его цвет. Все автомобили имеют колеса, но на каждом конкретном автомобиле стоят шины определенного размера и завода-изготовителя. Исходя из сказанного, очень важно понимать разницу между *классом* и конкретным *объектом* этого класса. Класс — это некая схема, которая описывает объект в целом. Объект класса — это, если можно так сказать, материальное воплощение некоторого конкретного элемента из этого класса. Класс Кошка — это описание ее свойств и действий, он всегда только один. А объектов класса Кошка может быть великое множество. По аналогии класс Автомобиль — это описание свойств и действий автомобиля, и такой класс всегда только один. А объектов класса Автомобиль может также великое множество (рис. 2.21).

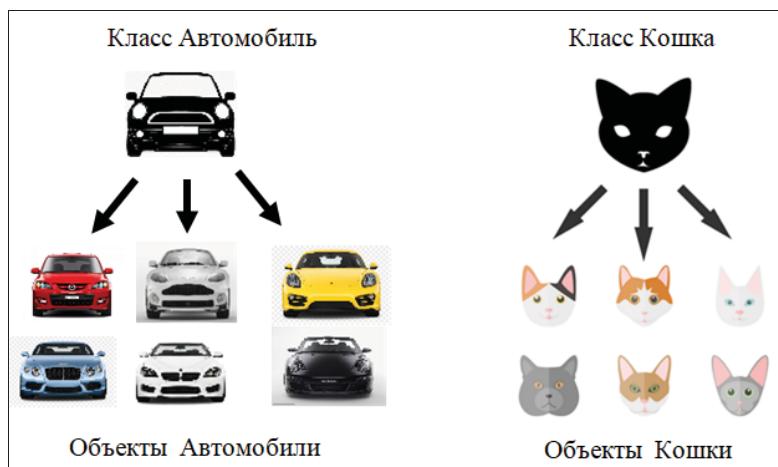


Рис. 2.21. Демонстрация различий между классом и объектом класса

Классы

Для создания класса надо написать ключевое слово `class` и затем указать имя этого класса. Создадим класс Кошки с именем `Cat`:

```
class Cat:
```

Затем нам нужно указать действия, которые будет способен совершать наш класс (в нашем случае — возможные действия кошки). Такие действия реализуются в виде функций, которые описываются внутри класса. Функции, описанные внутри класса, называются *методами*. Словесно мы можем описать следующие методы кошки (действия, которые она может совершать): мурлыкать (`purr`), шипеть (`hiss`), царапаться (`scrabble`). Теперь реализуем эти методы в классе Кошки (`Cat`) на языке Python (листинг 2.16).

Листинг 2.16

```
# Класс кошки
class Cat:

    # Мурлыкать
    def purr(self):
        print("Муррр!")

    # Шипеть
    def hiss(self):
        print("Шшшшш!")

    # Царапаться
    def scrabble(self):
        print("Цап-царап!")
```

Все достаточно просто! Мы создали класс с именем `Cat` и внутри него определили три обычные функции с именами `purr`, `hiss` и `scrabble`. Как видно из приведенного программного кода, каждая из трех описанных функций имеет единственный параметр `self`. Рассмотрим, зачем он нужен и что означает параметр `self` в функциях Python, описанных внутри класса.

Классам нужен способ, чтобы ссылаться на самих себя. В методах класса первый параметр функции по соглашению именуют `self`, и он представляет собой ссылку на сам объект этого класса. Помещать этот параметр нужно в каждую функцию, чтобы иметь возможность вызвать ее на текущем объекте. Таким образом, параметр `self` заменяет идентификатор объекта. Все наши кошки (объект `Cat`) умеют мурлыкать (имеют метод `purr`). Теперь мы хотим, что бы замурлыкал наш конкретный объект Кошка, созданный на основе класса `Cat`. Как это сделать? Допустим, мы в Python напишем следующую команду:

```
Cat.purr()
```

Если мы запустим на выполнение эту команду, то станут мурлыкать сразу все кошки и коты на свете. А если мы воспользуемся командой:

```
self.purr ()
```

то мурлыкать станет только та кошка, на которую укажет параметр `self`, т. е. именно наша.

Как видите, обязательный для любого метода параметр `self` позволяет нам обращаться к методам и переменным самого класса! Без этого аргумента выполнить подобные действия мы бы не смогли.

Кроме того, что кошки могут шипеть, мурлыкать и царапаться, у них есть еще и ряд свойств: рост, вес, цвет шерсти, длина усов и пр. Давайте теперь в классе Кошки зададим им ряд свойств и, в частности: цвет шерсти, цвет глаз, кличку. А также зададим статический атрибут *наименование класса* — `Name_Class`. Как это сделать? Мы можем создать переменную и занести в нее наименование класса. А также в абсолютно любом классе нужно определить функцию `__init__()`. Эта функция вызывается всегда, когда мы создаем реальный объект на основе нашего класса (обратите внимание: здесь используются символы двойного подчеркивания). Итак, задаем нашим кошкам наименование класса — `Name_Class`, и три свойства: цвет шерсти — `wool_color`, цвет глаз — `eyes_color`, кличку — `name` (листинг 2.17).

Листинг 2.17

```
class Cat:  
    Name_Class = "Кошки"  
  
    # Действия, которые надо выполнять при создании объекта "Кошка"  
    def __init__(self, wool_color, eyes_color, name):  
        self.wool_color = wool_color  
        self.eyes_color = eyes_color  
        self.name = name
```

В приведенном здесь методе `__init__()` мы задаем переменные, в которых будут храниться свойства нашей кошки. Как мы это делаем? Обязательно используем параметр `self` — чтобы при создании конкретного объекта Кошка сразу (по умолчанию) задать нашей кошке три нужных свойства. Затем определяем в этом методе три переменные, отвечающие за цвет шерсти, цвет глаз и кличку. Как мы задаем кошке такое свойство, как цвет шерсти. Вот эта строчка:

```
self.wool_color = wool_color
```

В левой части выражения мы создаем атрибут с именем `wool_color` для задания цвета шерсти нашей кошки. Этот процесс практически не отличается от обычного создания переменной. Присутствует только одно отличие — аргумент `self`, который указывает на то, что переменная относится к классу `Cat`. Затем мы присваиваем этому атрибуту значение. Аналогичным образом формируем еще два свойства: цвет глаз и кличку. Программный код описания класса Кошка теперь будет выглядеть следующим образом (листинг 2.18).

Листинг 2.18

```
Class Cat:
    Name_Class = "Кошки"

    # Действия, которые надо выполнять при создании объекта "Кошка"
    def __init__(self, wool_color, eyes_color, name):
        self.wool_color = wool_color
        self.eyes_color = eyes_color
        self.name = name

    # Мурлыкать
    def purr(self):
        print("Муррр!")

    # Шипеть
    def hiss(self):
        print("Шмммм!")

    # Царапаться
    def scrabble(self):
        print("Цап-царап!")
```

Объекты

Мы создали класс Кошки. Теперь давайте создадим на основе этого класса реальный объект — кошку:

```
my_cat = Cat('Цвет шерсти', 'Цвет глаз', 'Кличка')
```

В этой строке мы создаем переменную `my_cat`, а затем присваиваем ей объект класса `Cat`. Выглядит это все так, как вызов некоторой функции `Cat(...)`. На самом деле так оно и есть. Этой записью мы вызываем метод `__init__()` класса `Cat`. Функция `__init__()` в нашем классе принимает четыре аргумента: сам объект класса — `self`, который указывать не надо, а также еще три аргумента, которые затем становятся атрибутами нашей кошки.

Итак, с помощью приведенной строки мы создали реальный объект — нашу собственную кошку. Для этой кошки зададим следующие атрибуты или свойства: белую шерсть, зеленые глаза и кличку Мурка. Давайте выведем эти атрибуты в консоль с помощью следующего программного кода (листинг 2.19).

Листинг 2.19

```
my_cat = Cat('Белая', 'Зеленые', 'Мурка')
print("Наименование класса - ", my_cat.Name_Class)
print("Вот наша кошка:")
print("Цвет шерсти- ", my_cat.wool_color)
```

```
print("Цвет глаз- ", my_cat.eyes_color)
print("Кличка- ", my_cat.name)
```

То есть обратиться к атрибутам объекта мы можем, записав имя объекта, поставив точку и указав имя желаемого атрибута. Результат работы этой программы представлен на рис. 2.22.

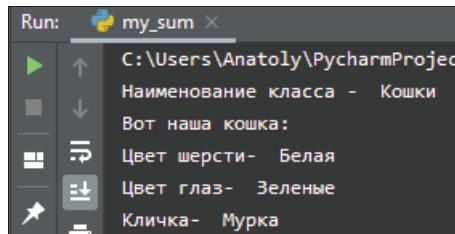


Рис. 2.22. Параметры объекта Кошка (`my_cat`), созданные по умолчанию

Атрибуты кошки можно менять. Например, давайте сменим кличку нашей кошки и поменяем ее цвет (сделаем кота Ваську черного цвета). Для этого изменим наш программный код следующим образом (листинг 2.20).

Листинг 2.20

```
my_cat = Cat('Белая', 'Зеленые', 'Мурка')
my_cat.name = "Васька"
my_cat.wool_color = "Черный"
print("Наименование класса - ", my_cat.Name_Class)
print("Вот наша кошка:")
print("Цвет шерсти- ", my_cat.wool_color)
print("Цвет глаз- ", my_cat.eyes_color)
print("Кличка- ", my_cat.name)
```

И в итоге получим следующий результат (рис. 2.23).

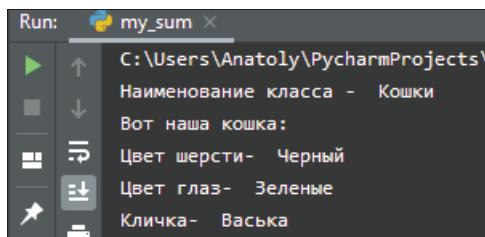


Рис. 2.23. Измененные параметры объекта Кошка (`my_cat`)

Теперь вспомним, что в нашем классе Кошки запрограммирована возможность выполнять некоторые действия. Попросим нашего кота Ваську мяукнуть. Для этого добавим в программу всего одну строку:

```
my_cat.purr()
```

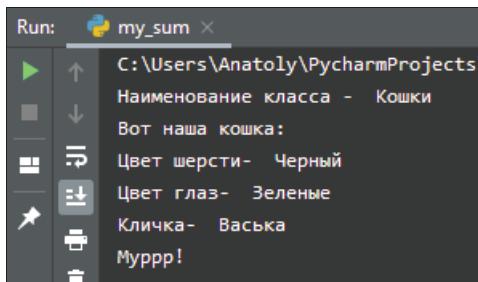


Рис. 2.24. Демонстрация вызова метода объекта Кошка

Выполнение этой команды выведет в консоль текст **Муррр!** (рис. 2.24).

Как видите, обращаться к методам объекта так же просто, как и обращаться к его атрибутам.

2.2.6. Создание классов и объектов на примере автомобиля

В настоящее время активно развивается беспилотный транспорт и, в частности, автомобили. Программные средства, которые управляют беспилотными автомобилями, строятся с использованием элементов искусственного интеллекта. К таким элементам относятся нейронные сети, машинное обучение, системы распознавания и идентификации объектов, программные средства приема и обработки большого количества данных, поступающих от различных датчиков и видеокамер, модули привода в действие таких элементов автомобиля, как тормоза, рулевое управление, акселератор и т. п. Исходя из этого, попробуем создать простейший класс `Car`, описывающий автомобиль (листинг 2.21).

Листинг 2.21

```
class Car(object):
    # Наименование класса
    Name_class = "Автомобиль"

    def __init__(self, brand, weight, power):
        self.brand = brand    # Марка, модель автомобиля
        self.weight = weight  # Вес автомобиля
        self.power = power    # Мощность двигателя

    # Метод двигаться прямо

    def drive(self):
        # Здесь команды двигаться прямо
        print("Поехали, движемся прямо!")
```

```
# Метод повернуть направо

def right(self):
    # Здесь команды повернуть руль направо
    print("Едем, поворачиваем руль направо!")

# Метод повернуть налево

def left(self):
    # Здесь команды повернуть руль налево
    print("Едем, поворачиваем руль налево!")

# Метод тормозить

def brake(self):
    # Здесь команды нажатия на педаль тормоза
    print("Стоп, активируем тормоз")

# Метод подать звуковой сигнал

def beep(self):
    # Здесь команды подачи звукового сигнала
    print("Подан звуковой сигнал")
```

В приведенном примере мы создали класс Автомобиль (`Car`) и добавили в него три атрибута и пять методов. Атрибутами здесь являются:

```
self.brand = brand      # Марка, модель автомобиля
self.weight = weight     # Вес автомобиля
self.power = power       # Мощность двигателя
```

Указанные атрибуты описывают автомобиль: его марку (модель), вес и мощность двигателя. Так же у этого класса есть пять методов. Как мы уже знаем, метод описывает, что делает класс, т. е. Автомобиль. В нашем случае автомобиль может двигаться прямо, поворачивать направо, поворачивать налево, подавать звуковой сигнал и останавливаться. Что касается аргумента под названием `self`, то его назначение подробно рассмотрено в предыдущем разделе.

Теперь на основе этого класса создадим объект — автомобиль с именем `MyCar` и атрибутами: Мерседес, вес — 1200 кг, мощность двигателя — 250 лошадиных сил и выведем параметры созданного объекта (листинг 2.22).

Листинг 2.22

```
MyCar = Car('Мерседес', 1200, 250)
print('Параметры автомобиля, созданного из класса- ', MyCar.Name_class)
print('Марка (модель)- ', MyCar.brand)
print('Вес (кг)- ', MyCar.weight)
print('Мощность двигателя (лс)- ', MyCar.power)
```

Результат работы этого программного кода представлен на рис. 2.25.

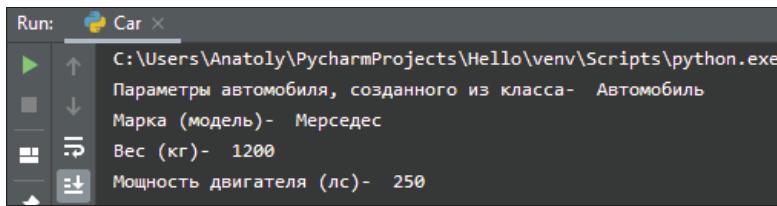


Рис. 2.25. Демонстрация создания объекта Автомобиль (MyCar) из класса Car и вывода его параметров

Теперь испытаем созданные в этом классе методы и заставим автомобиль двигаться, т. е. обратимся к соответствующим методам созданного нами объекта MyCar (листинг 2.23).

Листинг 2.23

```
MyCar.drive()      # Двигается прямо
MyCar.right()     # Поворачиваем направо
MyCar.drive()      # Двигается прямо
MyCar.left()       # Поворачиваем налево
MyCar.drive()      # Двигается прямо
MyCar.beep()        # Подаем звуковой сигнал
MyCar.brake()      # Тормозим
```

Результат работы этого программного кода представлен на рис. 2.26.

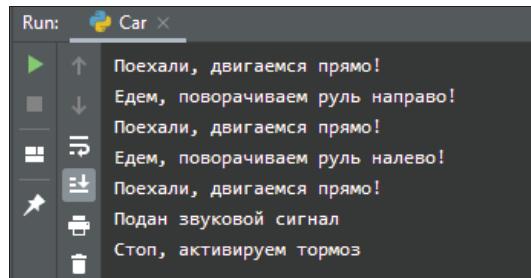


Рис. 2.26. Демонстрация работы методов объекта Автомобиль (MyCar) из класса Car

Итак, мы познакомились с базовыми понятиями о классах и объектах, создали два класса: Кошка и Автомобиль, а также заставили эти объекты выполнить некоторые элементарные действия.

2.2.7. Программные модули

Любой файл с расширением *ру* является *модулем*. Зачем они нужны? Многие программисты создают приложения с полезными функциями и классами. Другие программисты могут подключать эти сторонние модули и использовать все имеющиеся в них функции и классы, тем самым упрощая себе работу.

Например, вам не нужно тратить время и писать свой программный код для работы с матрицами. Достаточно подключить модуль `numpy` и использовать его функции и классы. К настоящему моменту программистами Python написано более 110 тыс. разнообразных модулей. Упоминавшийся ранее модуль `numpy` позволяет быстро и удобно работать с матрицами и многомерными массивами. Модуль `math` предоставляет множество методов для работы с числами: синусы, косинусы, переводы градусов в радианы и прочее, и прочее.

Установка модуля

Интерпретатор Python устанавливается вместе со стандартным набором модулей. В этот набор входит очень большое количество модулей, которые позволяют работать с математическими функциями, веб-запросами, читать и записывать файлы и выполнять другие необходимые действия.

Для того чтобы использовать модуль, который не входит в стандартный набор, необходимо его установить. Для установки модуля в Windows нужно либо нажать комбинацию клавиш `<Win>+<R>`, либо щелкнуть правой кнопкой мыши на значке главного меню Windows в левой нижней части экрана и выбрать опцию **Выполнить** (рис. 2.27)

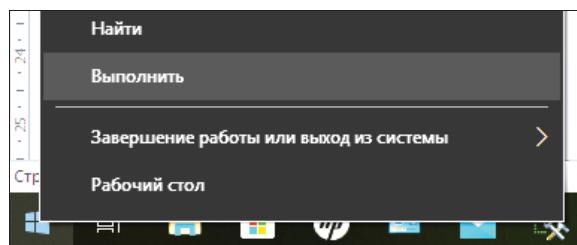


Рис. 2.27. Вызов окна запуска программ на выполнение

В результате этих действий откроется окно Windows для запуска программ на выполнение (рис. 2.28).

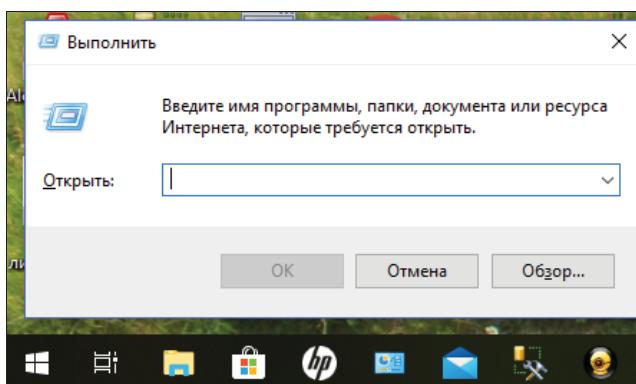


Рис. 2.28. Окно Windows для запуска программ на выполнение

В текстовом поле **Открыть** введите и запустите на исполнение команду инсталляции модуля:

```
pip install [название_модуля]
```

То же самое можно сделать и в окне терминала PyCharm (рис. 2.29).

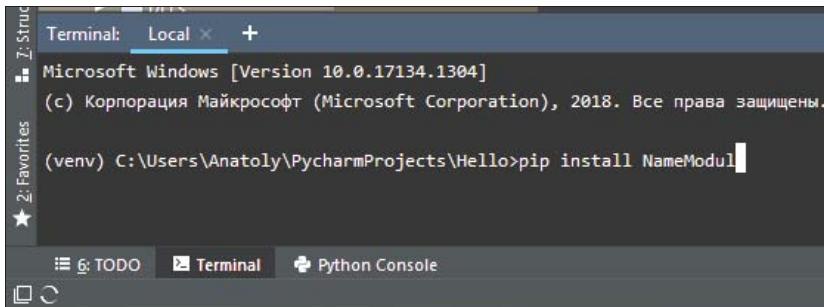


Рис. 2.29. Окно терминала PyCharm для запуска инсталляции модуля

В обоих случаях начнется процесс установки модуля. Когда он завершится, вы сможете использовать установленный модуль в своей программе.

Подключение и использование модуля

Сторонний модуль подключается достаточно просто — нужно написать всего одну короткую строку программного кода:

```
import [название_модуля]
```

Например, для импорта модуля, позволяющего работать с математическими функциями, надо написать следующее:

```
import math
```

Для обращения к какой-либо функции модуля достаточно написать название модуля, затем поставить точку и написать название функции или класса. Например, вычисление факториала числа 10 будет выглядеть так:

```
math.factorial(10)
```

Здесь мы обратились к функции `factorial(a)`, которая определена внутри модуля `math`. Это удобно, ведь нам не нужно тратить время и вручную создавать функцию, которая вычисляет факториал числа. Можно просто подключить модуль и сразу выполнить необходимое действие.

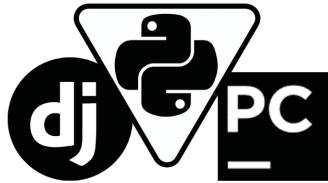
2.3. Краткие итоги

Итак, мы установили весь необходимый инструментарий, познакомились с основами языка Python и можем писать элементарные программы. Но прежде чем приступить к разработке веб-приложений с использованием Python, нужно разобраться:

- что такое фреймворк Django;
- со структурой веб-приложений на Django;
- с механизмом доступа к базам данных и HTML-страницам в приложениях на Django;

и получить практические навыки создания элементарных веб-приложений с использованием Django.

Рассмотрению этих вопросов и посвящена следующая глава.



ГЛАВА 3

Знакомимся с веб-фреймворком Django

В этой главе мы ответим на вопрос «Что такое Django?» и поясним, почему он столь привлекателен для программистов. Мы также покажем вам некоторые основные базовые блоки приложения, написанного с помощью Django (хотя у вас пока еще не будет среды разработки и тестирования). В общих чертах мы познакомимся со структурой и основными элементами Django, структурой приложений на нем, основными понятиями и определениями. В процессе изучения материала главы:

- будет создан первый проект на Django;
- рассмотрены понятия о «представлениях» и «маршрутизации» при разработке веб-приложений на Django.

Итак, приступим к знакомству с основами работы на Django.

3.1. Общие представления о Django

Django — это веб-фреймворк, написанный на Python и для Python, позволяющий быстро создавать безопасные и удобно поддерживаемые веб-сайты. Созданный опытными специалистами, Django берет на себя большую часть работы, поэтому разработчик может сосредоточиться на написании своего веб-приложения без необходимости «изобретать велосипед». Django бесплатный и с открытым исходным кодом, имеет активное сообщество, отличную документацию и множество вариантов как бесплатной, так и платной поддержки.

Django представляет собой весьма эффективный инструментарий, который имеет следующие характерные особенности.

- **Полнокомплектный инструмент.** Django следует философии «Все в одном флаконе» и предоставляет почти все необходимое, что разработчикам может понадобиться при разработке своих проектов. Поскольку все, что нужно, является частью единого продукта, все модули безупречно работают вместе в соответствии с последовательными принципами проектирования. Для него также имеется подробная документация.
- **Универсальный инструмент.** Django может быть использован для создания практически любых типов веб-сайтов — от систем управления контентом до со-

циальных сетей и новостных порталов. Он может работать с любой клиентской средой и способен доставлять контент практически в любом формате (HTML, RSS-каналы, JSON, XML и т. д.).

На Django может быть реализована практически любая функциональность, которая может понадобиться конечному пользователю, он работает с различными популярными базами данных, при необходимости его базовые модули могут быть дополнены сторонними компонентами.

- **Инструмент для разработки безопасных приложений.** Django помогает разработчикам избежать многих распространенных ошибок безопасности, предоставляя фреймворк с автоматической защитой сайта. Так, в Django реализован безопасный способ управления учетными записями пользователей и паролями, что позволяет избежать распространенных ошибок — таких, например, как размещение информации о сеансе в файлах cookie, где она уязвима. В Django файлы cookie содержат только ключ, а содержательная информация хранится отдельно в базе данных. Все пароли хранятся только в зашифрованном или хэшированном виде. Как известно, хэшированный пароль — это пароль фиксированной длины, созданный путем его обработки через криптографическую хэш-функцию. Django может проверить правильность введенного пароля, пропустив его через хэш-функцию и сравнив вывод с сохраненным значением хэша. Благодаря «одностороннему» характеру функции, даже если сохраненное хэш-значение скомпрометировано, злоумышленнику будет сложно определить исходный пароль.
- **Масштабируемые приложения.** Django использует компонентную архитектуру, при которой каждая часть создаваемого приложения независима от других частей и, следовательно, может быть заменена либо изменена. Четкое разделение между частями означает, что Django может масштабироваться при увеличении трафика путем добавления оборудования на любом уровне: серверы кэширования, серверы баз данных или серверы приложений. Одни из самых загруженных сайтов — Instagram — успешно масштабирован на Django.
- **Разработанные приложения удобны в сопровождении.** Код Django написан на основе принципов и шаблонов проектирования, которые поощряют создание поддерживаемого и повторно используемого кода. В частности, в нем задействован принцип DRY (Don't Repeat Yourself, не повторяйся), что позволяет избежать ненужного дублирования и сокращает объем кода. Django также способствует группированию связанных функциональных возможностей в повторно используемые приложения и группирует связанный программный код в модули в соответствии с концепцией Model-View-Controller (MVC).

ПРИМЕЧАНИЕ

Model-View-Controller, или MVC (можно перевести как «Модель-Представление-Контроллер», или «Модель-Вид-Контроллер»), — это схема разделения приложения на три отдельных компонента: модель (описывает структуру данных), представление (отвечает за отображение данных) и контроллер (интерпретирует действия пользователя). Таким образом, в разработанном приложении модификация каждого компонента может осуществляться независимо друг от друга.

- **Разработанные приложения являются кросс-платформенными.** Django написан на Python, который работает на многих платформах. Это означает, что вы не привязаны к какой-либо конкретной серверной платформе и можете запускать приложения на многих версиях Linux, Windows и macOS. Кроме того, Django хорошо поддерживается многими веб-хостингами, которые часто предоставляют определенную инфраструктуру и документацию для размещения сайтов Django. Хотя следует отметить, что развертывание готового сайта на хостинге внешнего провайдера — не совсем простая процедура и требует определенных знаний и квалификации.

Django был разработан в период с 2003 по 2005 год командой, которая занималась созданием и обслуживанием веб-сайтов для газет. Создав несколько сайтов, команда начала повторно использовать общий код и шаблоны. Этот общий код эволюционировал в Open Source проект веб-фреймворка Django в июле 2005 года.

Django продолжает расти и улучшаться с момента его первого релиза (1.0), вышедшего в сентябре 2008 года, до недавно выпущенной версии 3.0. В каждой следующей версии исправлены обнаруженные ошибки и добавлены новые функциональные возможности, начиная от поддержки новых типов баз данных, шаблонизаторов и кэширования, до добавления «общих» функций и классов, уменьшающих объем кода, который разработчики должны писать для ряда задач.

Django — это процветающий совместный проект с открытым исходным кодом, в котором заняты многие тысячи пользователей и участников. Несмотря на то что у него все еще есть некоторые особенности, отражающие его происхождение, Django превратился в универсальную среду, предоставляющую возможности разрабатывать веб-сайты любого типа.

Основываясь на количестве крупных сайтов, которые используют Django (таких, как Instagram, Mozilla, National Geographic, Open Knowledge Foundation, Pinterest и Open Stack), количестве участников и количестве специалистов, предоставляющих как бесплатную, так и платную его поддержку, можно полагать, что Django — весьма популярный фреймворк. Впрочем, нет никаких доступных и окончательных оценок популярности серверных фреймворков, хотя сайты, подобные Hot Framework, пытаются оценить их популярность, используя такие механизмы, как подсчет количества проектов на GitHub для каждой платформы.

ПРИМЕЧАНИЕ

GitHub — это система управления проектами и версиями программного кода, а также общедоступная платформа, созданная для разработчиков. GitHub позволяет программистам разместить на нем свой программный код и дает возможность каждому разработчику обмениваться своими достижениями с другими людьми по всему миру.

Все веб-фреймворки можно условно поделить на «гибкие» и «негибкие»:

- «негибкие» — это те, которые придерживаются «прямого пути» для решения какой-либо конкретной задачи. Они часто поддерживают быстрое развертывание в определенной области (решение проблем определенного типа). Однако они могут быть менее «гибкими» при решении проблем за пределами их основ-

ной сферы и, как правило, предлагают меньше вариантов того, какие компоненты и подходы они могут задействовать;

- напротив, у «гибких» фреймворков гораздо меньше ограничений на лучший способ склеивания компонентов для достижения цели или даже того, какие компоненты следует использовать. Они облегчают разработчикам применение наиболее подходящих инструментов для выполнения конкретной задачи, хотя и за счет того, что разработчику самому приходится искать эти компоненты.

Django — «умеренно гибкий» фреймворк. Он предоставляет набор компонентов для обработки большинства задач веб-разработки и один (или два) предпочтительных способа их использования. Однако такая архитектура Django означает, что вы обычно можете выбирать из нескольких имеющихся различных опций или, при необходимости, добавлять поддержку совершенно новых.

3.2. Структура приложений на Django

В типовом информационном сайте веб-приложение ожидает HTTP-запросы от веб-браузера пользователя. Получив запрос, приложение обрабатывает его и выполняет запрограммированные действия. Затем приложение возвращает ответ веб-браузеру пользователя. Возвращаемая страница может быть либо статической, либо динамической. В последнем случае некоторые данные вставляются в HTML-шаблон.

Веб-приложение, написанное на Django, разбито на четыре базовых блока, которые содержатся в отдельных файлах, не зависимых друг от друга, но при этом и работающих в связке.

Фреймвок Django реализует архитектуру Model-View-Template, или, сокращенно, MVT, которая по факту является модификацией распространенной в веб-программировании архитектуры MVC (Model-View-Controller). Схематично архитектура MVT в Django представлена на рис. 3.1.

Приложение на Django имеет следующие блоки:

- диспетчер URL-адресов (URL-mapper или картостроитель адресов);
- представления (View);
- модели (Models);
- шаблоны (Templates).

Рассмотрим эти блоки подробнее:

- **Диспетчер URL-адресов (URLs).** Хотя можно обрабатывать запросы с каждого URL-адреса с помощью одной функции, гораздо удобнее писать отдельную функцию для обработки каждого ресурса. URL-mapper используется для перенаправления HTTP-запросов в соответствующее представление (View) на основе URL-адреса запроса. URL-mapper также может извлекать данные из URL-адреса в соответствии с заданным шаблоном и передавать их в соответствующую функцию в виде аргументов.

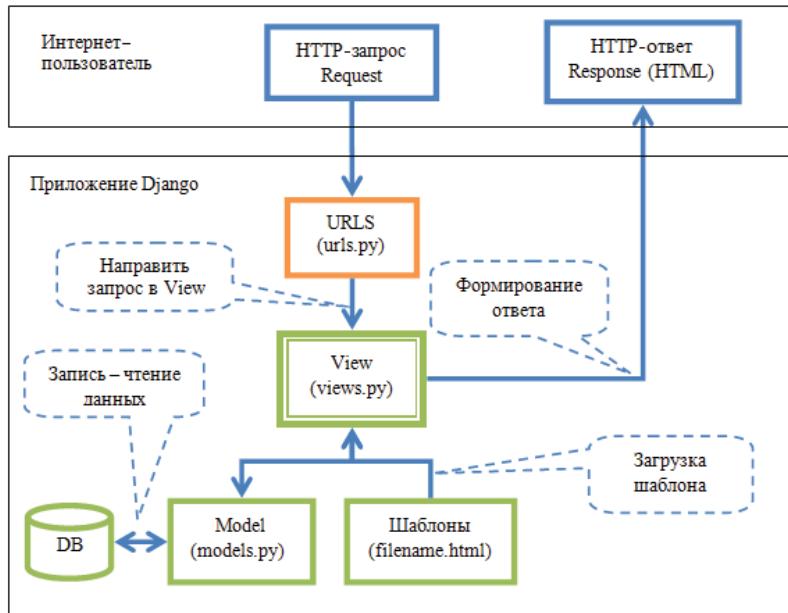


Рис. 3.1. Базовые блоки веб-приложения на Django

- **Модели (Models).** Модели представляют собой объекты Python, которые определяют структуру данных приложения и предоставляют механизмы для управления данными (добавления, изменения, удаления) и выполнения запросов к базе данных.
- **Шаблоны (Templates).** Template (шаблон) — это текстовый файл, определяющий структуру или разметку страницы (например, HTML-страницы), с полями, которые используются для подстановки актуального содержимого из базы данных или параметров, вводимых пользователем.
- **Представление (View).** Центральная роль в этой архитектуре принадлежит представлению (view). Когда к приложению Django приходит запрос от удаленного пользователя (HTTP-запрос), то URL-диспетчер определяет, с каким ресурсом нужно сопоставить этот запрос, и передает его выбранному ресурсу. Ресурсом в этом случае является представление (view), которое, получив запрос, определенным образом обрабатывает его. В процессе обработки запроса представление (view) может обращаться через модели (model) к базе данных, получать из нее данные или, наоборот, сохранять данные в нее. Результат обработки запроса отправляется обратно, и этот результат пользователь видит в своем браузере. Как правило, результат обработки запроса представляет сгенерированный HTML-код, для генерации которого применяются шаблоны (Template).

Лучшим помощником при изучении нового материала является практика. Поэтому продолжим изучение Django путем реализации «с нуля» своего первого веб-приложения.

3.3. Первый проект на Django

Следует напомнить, что мы будем работать в среде PyCharm и с проектом `Web_1`, который создали в *главе 1*. Нужно иметь в виду, что при установке Django в папке виртуальной среды автоматически устанавливается скрипт `django-admin.py`, а в Windows — также исполняемый файл `django-admin.exe`. Их можно найти в папке виртуальной среды, в которую производилась установка Django: на Windows — в подкаталоге `Scripts` (рис. 3.2), а на Linux/macOS — в каталоге `bin`.

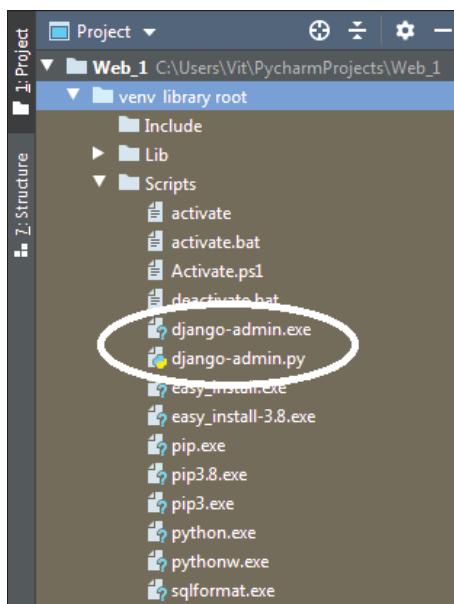


Рис. 3.2. Файлы Django
в каталоге Script
созданного проекта

Исполняемый файл `django-admin.exe` предоставляет возможность выполнения ряда команд для управления проектом Django. В частности, для создания проекта служит команда `startproject`. Этой команде в качестве аргумента передается название проекта.

Веб-приложение, или проект Django, состоит из отдельных приложений. Полноценное веб-приложение они образуют в совокупности. Каждое приложение представляет какую-то определенную функциональность или группу функций. Один проект может включать множество приложений. Это позволяет выделить группу задач в отдельный модуль и разрабатывать их относительно независимо от других. Кроме того, мы можем переносить приложение из одного проекта в другой, независимо от функциональности проекта.

Итак, загрузите PyCharm и откройте проект `Web_1`, который был создан в *главе 1*. Теперь можно войти в окно терминала PyCharm и создать новый проект Django с именем `hello`. Для этого в окне терминала выполните следующую команду (рис. 3.3):

```
django-admin startproject hello
```

```
Terminal: Local +  
Microsoft Windows [Version 6.1.7601]  
(c) Корпорация Майкрософт 2013. Все права защищены.  
(venv) C:\Users\Vit\PycharmProjects\Web_1>django-admin startproject hello
```

Рис. 3.3. Создание нового проекта Django в окне терминала PyCharm

После выполнения этой команды в папке Web_1 проекта PyCharm будет создана новая папка hello проекта Django (рис. 3.4).

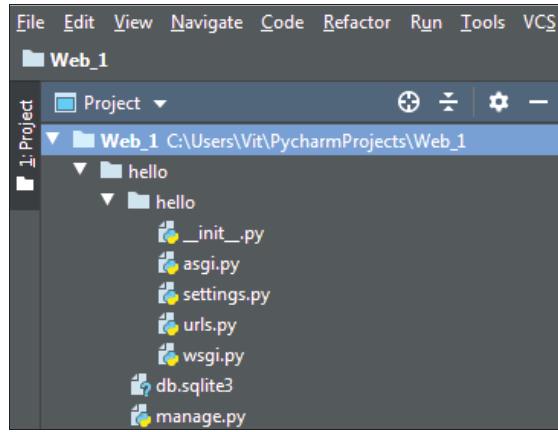
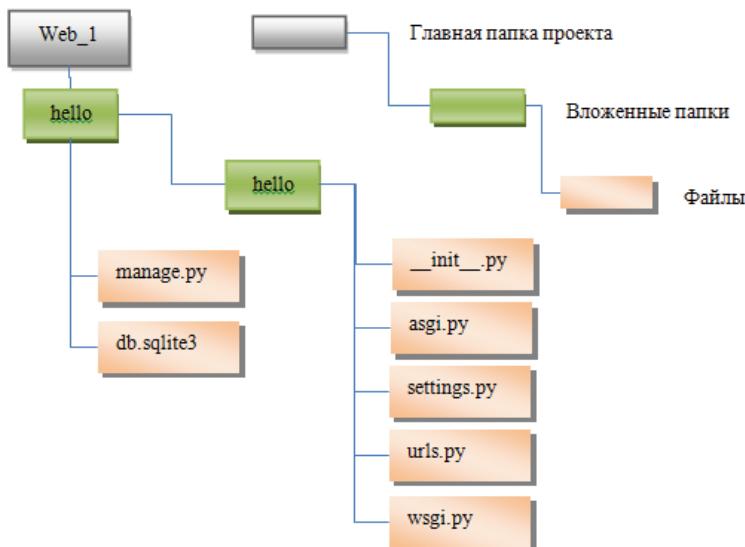


Рис. 3.4. Файловая структура проекта hello

Все автоматически созданные веб-проекты на Django имеют одинаковую структуру. Изучим более детально эту структуру на примере нашего проекта hello (рис. 3.5), последовательно рассмотрев каждую папку и файл начального проекта, который пока не содержит приложений:

- папка Web_1 — корневой каталог (папка) нашего проекта PyCharm. Он представляет собой контейнер, в котором будут создаваться наши приложения и файлы для управления проектом. Название папки ничего не значит для Django, и его можно поменять на свое усмотрение;
- папка Web_1/hello — это внешняя папка нашего веб-проекта;
- файл manage.py — инструмент управления из командной строки, при помощи которого можно управлять проектом различными путями. В частности, при помощи этого инструмента вы можете запустить на исполнение свой проект на сервере Python;
- файл db.sqlite3 — база данных SQLite. Она создается по умолчанию и используется для хранения данных;
- папка Web_1/hello/hello/ — внутренняя папка проекта. Она содержит текущий и единственный на текущий момент пакет (или, вернее, проект Python) в нашем проекте. Имя этого пакета в дальнейшем будет использовано для импорта внутренней структуры кода (к примеру, для импорта hello.urls);

Рис. 3.5. Структура проекта Django с именем `hello`

- файл `hello/__init__.py` — этот пустой файл предназначен для указания и регистрации пакетов. Наличие его в каталоге `Web_1` указывает интерпретатору Python, что текущий каталог будет рассматриваться в качестве пакета;
- файл `hello/asgi.py` — это точка входа для ASGI-совместимых веб-серверов, обслуживающих ваш проект (он потребуется при развертывании приложения на публичном сайте);
- файл `hello/settings.py` — это файл установок и конфигурации текущего проекта Django;
- файл `hello/urls.py` — это URL-декларации текущего проекта Django, или, иначе говоря, это «таблица контента» Django-проекта;
- файл `hello/wsgi.py` — это точки входа для WSGI-совместимого веб-сервера (он потребуется при развертывании приложения на публичном сайте).

Пока в проекте Django еще нет ни одного приложения. Несмотря на это, мы уже можем запустить проект на выполнение. Для этого нужно сначала войти в папку `hello`, для чего в окне терминала PyCharm выполнить следующую команду:

```
cd hello
```

После этого вы из корневой папки `Web_1` перейдете в папку `hello` проекта Django (рис. 3.6).

Теперь, находясь в этой папке, можно запустить наш проект на выполнение. Наберите в окне терминала команду запуска локального сервера:

```
python manage.py runserver
```

В результате выполнения этой команды будет запущен локальный веб-сервер разработки на вашем компьютере с адресом: <http://127.0.0.1:8000/> (рис. 3.7).

```
Terminal: Local +  
Microsoft Windows [Version 6.1.7601]  
(c) Корпорация Майкрософт 2013. Все права защищены.  
  
(venv) C:\Users\Vit\PycharmProjects\Web_1>django-admin startproject hello  
  
(venv) C:\Users\Vit\PycharmProjects\Web_1>cd hello  
  
(venv) C:\Users\Vit\PycharmProjects\Web_1\hello>
```

Рис. 3.6. Переход в папку hello проекта Django

```
Terminal: Local +  
  
(venv) C:\Users\Vit\PycharmProjects\Web_1>python manage.py runserver  
Watching for file changes with StatReloader  
Performing system checks...  
  
System check identified no issues (0 silenced).  
  
You have 17 unapplied migration(s). Your project may not work properly until  
Run 'python manage.py migrate' to apply them.  
July 08, 2020 - 09:07:18  
Django version 3.0.8, using settings 'hello.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CTRL-BREAK.
```

Рис. 3.7. Старт локального веб-сервера разработки

ПРИМЕЧАНИЕ

Если после выполнения этой команды вы получили сообщение об ошибке:

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xcf in position 5:  
invalid continuation byte
```

то это означает, что в имени вашего компьютера присутствуют символы из русского алфавита — например, **Vit-ПК**. В таком случае нужно войти в свойства компьютера и поменять его имя (убрать символы **ПК**). Для этого следует щелкнуть правой кнопкой мыши на значке **Мой компьютер**, в открывшемся меню активировать опцию **Свойства** и выбрать левой кнопкой мыши ссылку **Изменить параметры** (рис. 3.8).

В открывшемся окне **Свойства системы** на вкладке **Имя компьютера** нажмите кнопку **Изменить** — откроется панель **Изменение имени компьютера или домена**, в которой и можно убрать символы русского алфавита из имени компьютера. В данном случае имя компьютера **Vit-ПК** было заменено на **Vit** (рис. 3.9).

Если локальный сервер запустился без ошибок, щелкните левой кнопкой мыши на его ссылке: <http://127.0.0.1:8000/> (рис. 3.10).

После этого на вашем компьютере откроется веб-браузер и в него будет загружена веб-страница поздравления с успешной установкой Django (рис. 3.11).

По умолчанию страница загружается на английском языке. Однако Django имеет локализацию и может работать с разными языками.

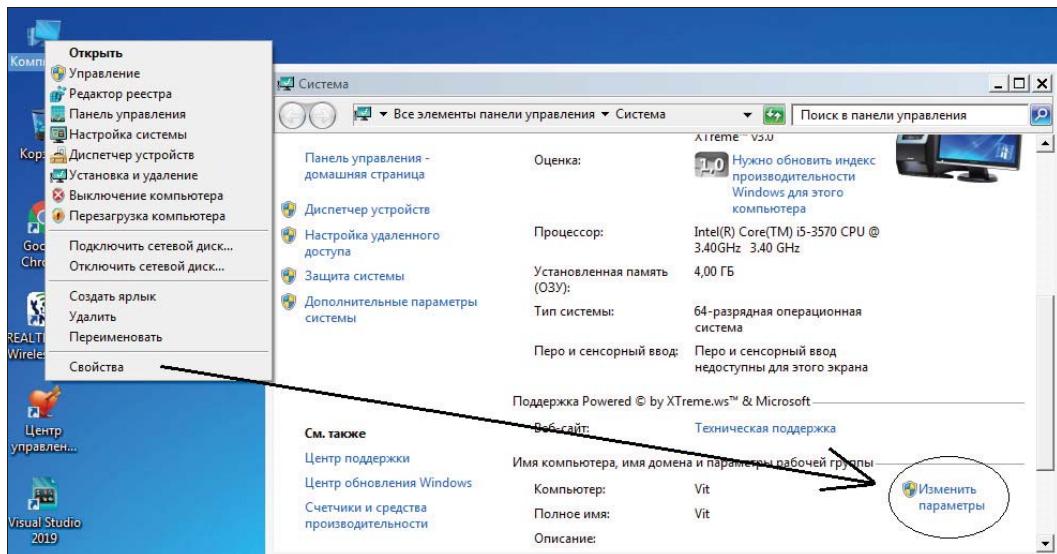


Рис. 3.8. Окно входа в режим изменения параметров компьютера

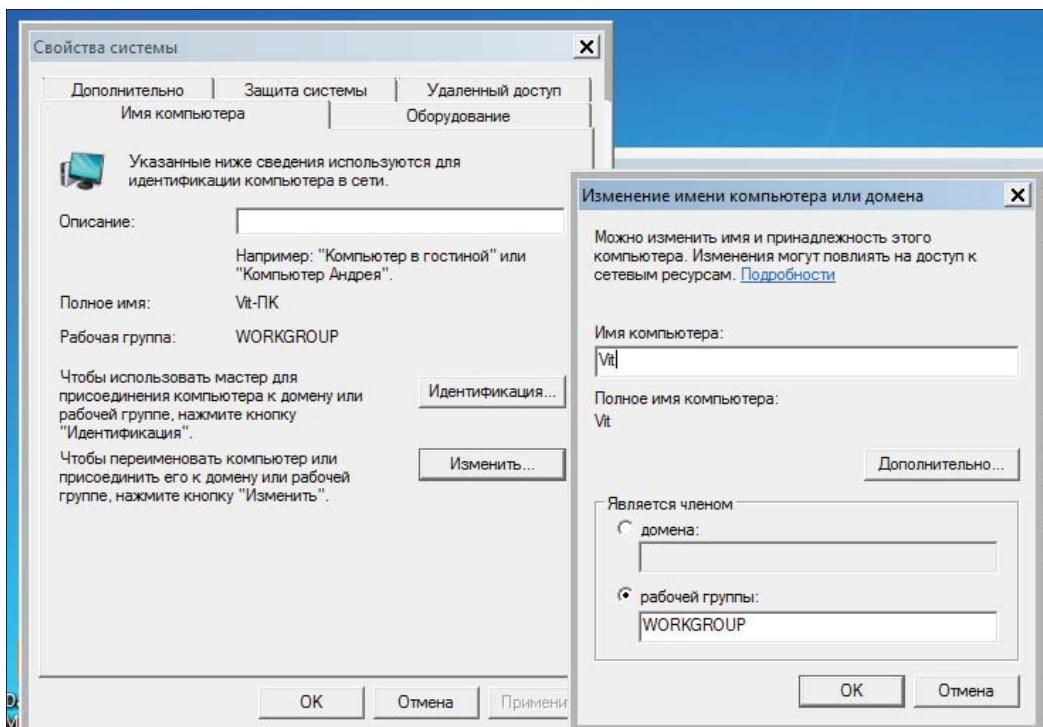
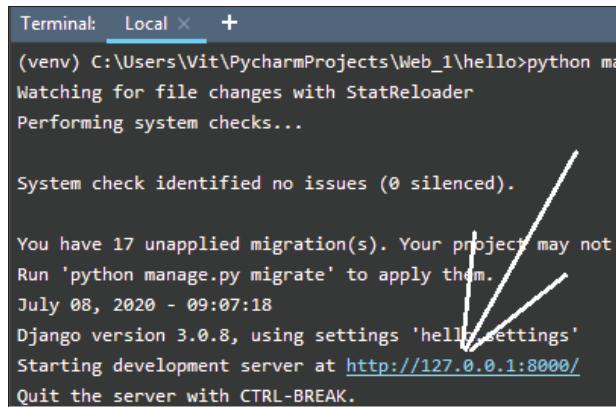


Рис. 3.9. Изменение имени компьютера



```

Terminal: Local × +
(venv) C:\Users\Vit\PycharmProjects\Web_1\hello>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not
Run 'python manage.py migrate' to apply them.
July 08, 2020 - 09:07:18
Django version 3.0.8, using settings 'hello.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

```

Рис. 3.10. Запуск локального веб-сервера разработчика

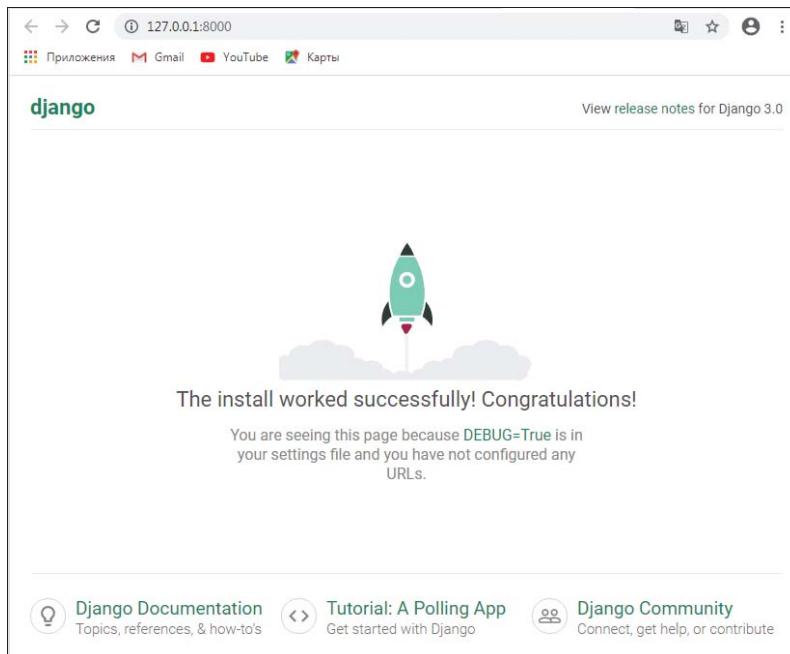


Рис. 3.11. Страница поздравления с успешной установкой Django

Остановите локальный веб-сервер нажатием комбинации клавиш <Ctrl>+<C>. Откройте файл `settings.py` и найдите фрагмент кода с установкой языка:

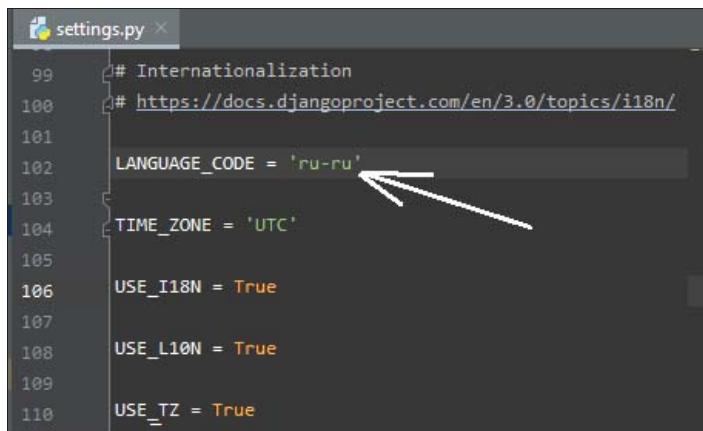
```

# Internationalization
# https://docs.djangoproject.com/en/3.0/topics/i18n/
LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'UTC'
USE_I18N = True
USE_L10N = True
USE_TZ = True

```

Как можно видеть, здесь по умолчанию задан английский язык в его американской версии (`en-us`). Задайте в этой опции русский язык (рис. 3.12):

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



```
99     # Internationalization
100    # https://docs.djangoproject.com/en/3.0/topics/i18n/
101
102    LANGUAGE_CODE = 'ru-ru' ←
103
104    TIME_ZONE = 'UTC'
105
106    USE_I18N = True
107
108    USE_L10N = True
109
110    USE_TZ = True
```

Рис. 3.12. Задание русского языка в установках Django

Снова в окне терминала запустите локальный сервер разработки:

```
python manage.py runserver
```

и вы увидите, что после этого изменения окно приветствия и поздравления с успешной установкой Django будет выдано на русском языке (рис. 3.13)

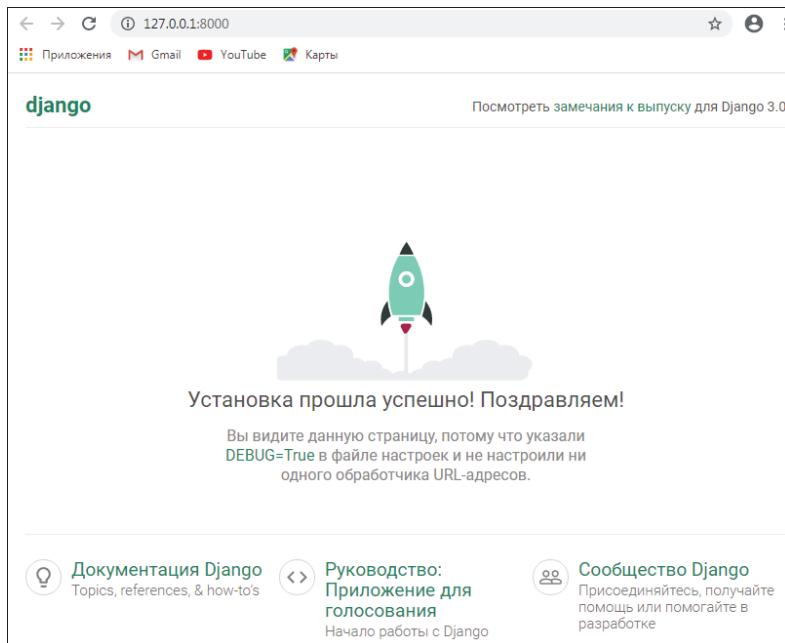


Рис. 3.13. Страница поздравления с успешной установкой Django на русском языке

Итак, нас можно поздравить с успешной установкой и пробным запуском Django! Вернитесь теперь к содержимому папки проекта `hello`. В ней имеется файл `db.sqlite3`. Как уже отмечалось ранее, это файл с базой данных SQLite (рис. 3.14).

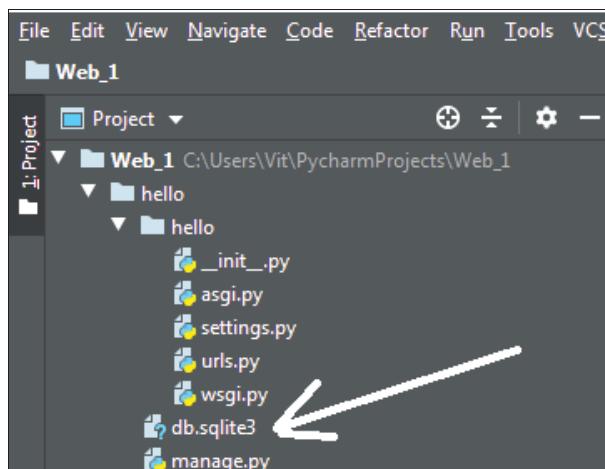


Рис. 3.14. Файл базы данных SQLite

SQLite — компактная встраиваемая реляционная база данных, исходный код которой передан в общественное достояние. Она является чисто реляционной базой данных.

Слово «встраиваемая» означает, что SQLite не использует парадигму «клиент-сервер». То есть движок SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а предоставляет библиотеку, с которой программа компонуется, а движок становится составной частью программы. При этом в качестве протокола обмена используются вызовы функций (API) библиотеки SQLite. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том компьютере, на котором исполняется программа.

В Django база данных SQLite создается автоматически (по умолчанию), формирование именно этой базы данных прописывается в файле конфигурации проекта. Если снова открыть файл `settings.py`, то в нем можно увидеть следующие строки:

```
# Database
# https://docs.djangoproject.com/en/3.0/ref/settings/#databases

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

Именно здесь можно переопределить базу данных, с которой будет работать приложение. Чтобы использовать другие системы управления базами данных (СУБД), необходимо установить соответствующий пакет (табл. 3.1).

Таблица 3.1. Основные базы данных, с которыми может работать приложение Django

СУБД	Пакет	Команда установки
PostgreSQL	psycopg2	pip install psycopg2
MySQL	mysql-python	pip install mysql-python
Oracle	cx_Oracle	pip install cx_Oracle

3.4. Первое приложение на Django

При создании проекта он уже содержит несколько приложений по умолчанию:

- django.contrib.admin;
- django.contrib.auth;
- django.contrib.contenttypes;
- django.contrib.sessions;
- django.contrib.messages;
- django.contrib.staticfiles.

Список всех приложений, созданных по умолчанию, можно найти в переменной `INSTALLED_APPS` файла проекта `settings.py`:

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

Теперь у нас все готово для работы над своими приложениями, в которых мы сможем реализовать нужный нам функционал. Создадим в нашем проекте первое приложение, для которого затем напишем собственный программный код.

Для создания нового приложения в окне терминала PyCharm выполните следующую команду (рис. 3.15):

```
python manage.py startapp firstapp
```

Имя приложения может быть любым — оно указывается после команды `startapp`.

```
Terminal: Local × +
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт 2013. Все права защищены.

(venv) C:\Users\Vit\PycharmProjects\Web_1\hello>python manage.py startapp firstapp
```

Рис. 3.15. Создания нового приложения firstapp в окне терминала PyCharm

В результате работы этой команды в проекте Django будет создано приложение firstapp, а в проекте появится новая папка, в которой будут хранить все файлы созданного приложения (рис. 3.16).

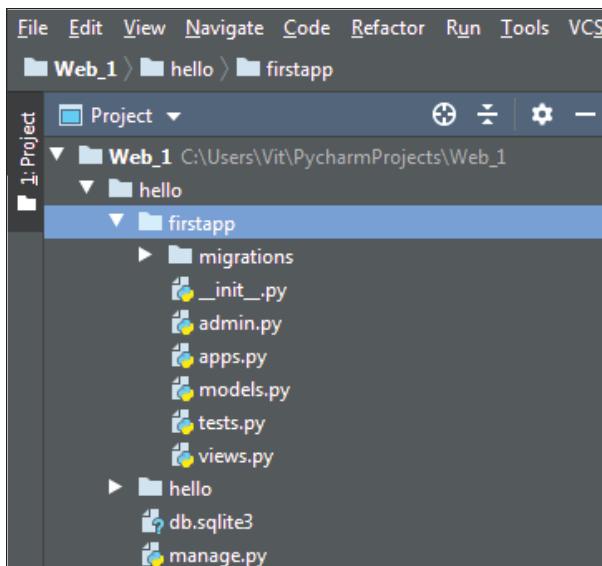


Рис. 3.16. Структура файлов приложения firstapp в окне терминала PyCharm

Рассмотрим структуру папок и файлов приложения firstapp:

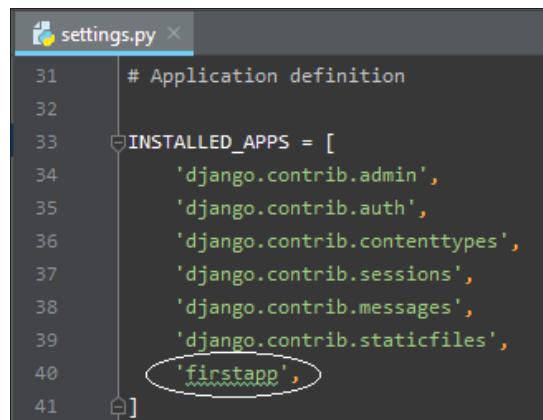
- папка migrations — хранит информацию, позволяющую сопоставить базу данных со структурой данных, описанных в модели;
- файл __init__.py — указывает интерпретатору Python, что текущий каталог будет рассматриваться в качестве пакета;
- файл admin.py — предназначен для административных функций. В частности, здесь производится регистрация моделей, которые используются в интерфейсе администратора;
- файл apps.py — определяет конфигурацию приложения;
- файл models.py — хранит определение моделей, которые описывают используемые в приложении данные;
- файл tests.py — хранит тесты приложения;

- ❑ файл `views.py` — определяет функции, которые получают запросы пользователей, обрабатывают их и возвращают ответ.

Но пока наше приложение никак не задействовано, его надо зарегистрировать в проекте Django. Для этого нужно открыть файл `settings.py` и добавить в конец массива с приложениями проекта `INSTALLED_APPS` наше приложение `firstapp` (добавляемый код выделен серым фоном и полужирным шрифтом):

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

В окне среды разработки PyCharm это будет выглядеть следующим образом (рис. 3.17).



```
settings.py ×  
31     # Application definition  
32  
33     INSTALLED_APPS = [  
34         'django.contrib.admin',  
35         'django.contrib.auth',  
36         'django.contrib.contenttypes',  
37         'django.contrib.sessions',  
38         'django.contrib.messages',  
39         'django.contrib.staticfiles',  
40         ''firstapp'',  
41     ]
```

Рис. 3.17. Окно терминала PyCharm:
регистрация приложения `firstapp` в файле `settings.py`

В проекте может быть несколько приложений, и каждое из них нужно добавлять аналогичным образом. После добавления приложения `firstapp` полная структура нашего веб-приложения будет выглядеть так, как показано на рис. 3.18.

Теперь определим какие-нибудь простейшие действия, которые будет выполнять это приложение, — например, отправлять в ответ пользователю приветствие **Hello World**.

Для этого перейдем в проекте приложения `firstapp` к файлу `views.py`, который по умолчанию должен выглядеть так:

```
from django.shortcuts import render  
# Create your views here.
```

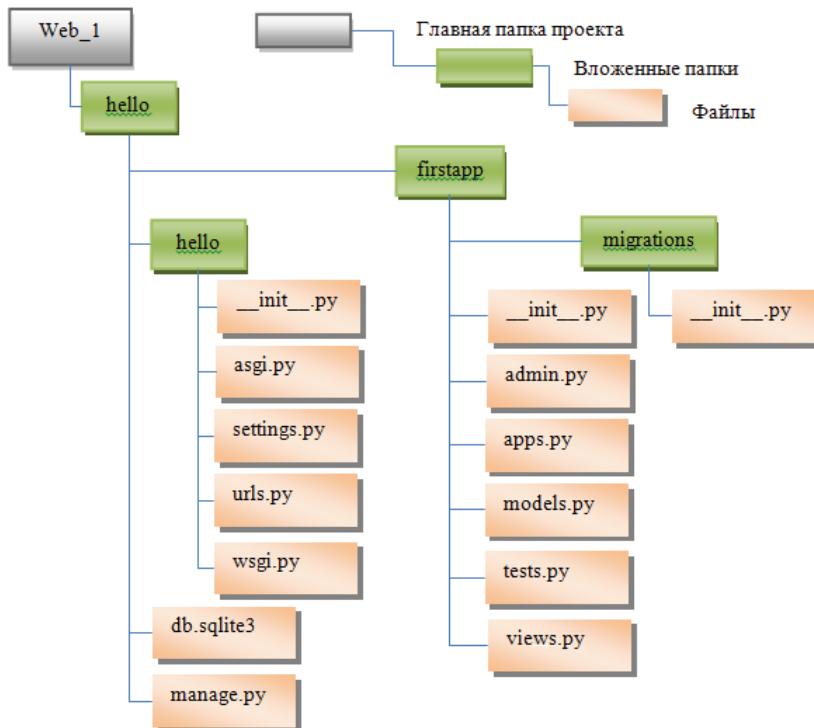


Рис. 3.18. Полная структура файлов и папок веб-приложения с проектом hello и приложением firstapp

Изменим этот код следующим образом (листинг 3.1).

Листинг 3.1

```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.

def index(request):
    return HttpResponse("Hello World! Это мой первый проект на Django!")
```

Здесь мы импортируем класс `HttpResponse` из стандартного пакета `django.http`. Затем определяем функцию `index()`, которая в качестве параметра получает объект запроса пользователя `request`. Класс `HttpResponse` предназначен для создания ответа, который отправляется пользователю. И с помощью кода `return HttpResponse` мы отправляем пользователю строку: "Hello World! Это мой первый проект на Django!".

Теперь в проекте Django откроем файл `urls.py`, который позволяет сопоставить маршруты с представлениями, обрабатывающими запрос по этим маршрутам. По умолчанию файл `urls.py` выглядит так:

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

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

В первой строке из модуля `django.contrib` импортируется класс `AdminSite`, который предоставляет возможности работы с интерфейсом администратора сайта. Во второй строке из модуля `django.urls` импортируется функция `path`. Эта функция задает возможность сопоставлять запросы пользователя (определенные маршруты) с функцией их обработки. Так, в нашем случае маршрут `'admin/'` будет обрабатываться методом `admin.site.urls`. То есть если пользователь запросит показать страницу администратора сайта (`'admin/'`), то будет вызван метод `admin.site.urls`, который вернет пользователю HTML-страницу администратора.

Но ранее мы определили в файле `views.py` функцию `index`, которая возвращает пользователю текстовую строку. Поэтому изменим файл `urls.py`, как показано в листинге 3.2 (добавляемый код выделен серым фоном и полужирным шрифтом).

Листинг 3.2

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

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

Чтобы использовать функцию `views.index`, здесь мы вначале импортируем модуль `views`. Затем сопоставляем маршрут `('')` — это, по сути, запрос пользователя к корневой странице сайта, с функцией `views.index`, а также дополнительно задаем имя для этого маршрута (`name='home'`). То есть если пользователь запросит показать главную (корневую) страницу сайта `('')`, то будет вызвана функция `index` из файла `views.py`. А в этой функции мы указали, что пользователю нужно вернуть HTML-страницу с единственным сообщением: "Hello World! Это мой первый проект на Django!".

По сути, маршрут с именем `'home'` будет сопоставляться с запросом к корню приложения.

Теперь снова запустим приложение командой:

```
python manage.py runserver
```

И вновь перейдем в браузере по адресу `http://127.0.0.1:8000/`. После чего браузер нам отобразит строку: **Hello World!** Это мой первый проект на **Django!** (рис. 3.19).

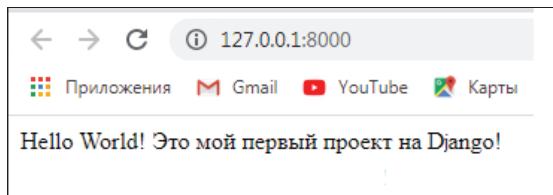
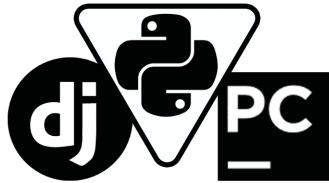


Рис. 3.19. Результаты работы приложения `firstapp`

Поскольку маршрутизация запросов пользователя является ключевым моментом при разработке веб-приложений на Django, мы изучим маршрутизацию более подробно в следующей главе.

3.5. Краткие итоги

В этой главе были даны общие представления о Django и о структуре приложений на нем. Создан первый простейший проект на Django с выводом единственной страницы. Рассмотрены базовые сведения о представлениях (view) и маршрутизации запросов пользователей. Поскольку представления и маршрутизация запросов пользователей при разработке веб-приложений являются весьма важной частью Django, то эти вопросы более детально рассмотрены в следующей главе.



ГЛАВА 4

Представления и маршрутизация

Представления (views) и маршрутизация запросов пользователя являются ключевыми элементами Django. Рассмотрению связанных с ними вопросов и посвящена эта глава, из материала которой вы узнаете:

- каким образом обрабатываются запросы, поступившие от удаленного пользователя;
- что такое маршруты, по которым обрабатываются запросы пользователей;
- что такое регулярные выражения при описании маршрутов и каков их синтаксис;
- что такое представления (views) и каковы их параметры;
- что такое параметры в строке запроса пользователя;
- что такое переадресация запросов и как выполняется отправка пользователю статусных кодов.

Для многих разработчиков, которые только осваивают веб-программирование, материал этой главы может трудно восприниматься с первого прочтения. Однако не стоит отчаиваться, нужно просто сесть за компьютер и последовательно набрать и выполнить все представленные здесь фрагменты программного кода. Это та основа, без понимания которой дальнейший материал книги просто не будет восприниматься. И это касается не только материалов этой книги, а вообще перспектив освоения технологий разработки веб-приложений. Если же вам удастся осмыслить изложенный здесь материал и разобраться, каким образом обрабатываются запросы пользователей и как на основе этих запросов происходит адресация к тем или иным страницам сайта, то можно считать, что перед вами откроются двери в сложный, но весьма увлекательный мир веб-программирования.

4.1. Обработка запросов пользователей

Центральным моментом любого веб-приложения является обработка запроса, который отправляет пользователь. В Django за обработку запроса отвечают *представления* (views). По сути, в представлениях реализованы функции обработки, которые

принимают данные запроса пользователя в виде объекта `request` и генерируют некий ответ (`response`), который затем отправляется пользователю в виде HTML-страницы. По умолчанию представления размещаются в приложении в файле `views.py`.

Взглянем, например, на проект `hello`, в который добавлено приложение `firstapp`, созданное в предыдущей главе. При создании нового проекта файл `views.py` имел следующее содержимое:

```
from django.shortcuts import render
# Create your views here.
```

Этот код пока никак не обрабатывает запросы — он только импортирует функцию `render` (оказывать, предоставлять), которая может использоваться для обработки входящих запросов.

Генерировать результат обработки запроса (ответ пользователю) можно различными способами. Одним из таких способов является использование класса `HttpResponse` (ответ HTTP) из пакета `django.http`, который позволяет отправить текстовое содержимое.

Изменим содержимое файла `views.py` следующим образом (листинг 4.1).

Листинг 4.1

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h2>Главная</h2>")

def about(request):
    return HttpResponse("<h2>О сайте</h2>")

def contact(request):
    return HttpResponse("<h2>Контакты</h2>")
```

Этим кодом мы, по сути, запрограммировали функции для выдачи заголовков трех страниц сайта: **Главная (index)**, **О сайте (about)**, **Контакты (contact)**, как показано на рис. 4.1.



Рис. 4.1. Структура страниц сайта в приложении `firstapp`

В нашем случае здесь определены три функции, которые будут обрабатывать запросы пользователя. Каждая функция принимает в качестве параметра объект `request` (запрос). Для генерации ответа в конструкторе объекта `HttpResponse` (ответ HTTP) имеется некоторая строка. Этим ответом может быть и строка кода HTML.

Чтобы эти функции сопоставлялись с запросами пользователя, надо в файле `urls.py` определить для них в проекте маршруты. В частности, изменим этот файл следующим образом (листинг 4.2).

Листинг 4.2

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

urlpatterns = [
    path('', views.index),
    path('about', views.about),
    path('contact', views.contact),
]
```

Переменная `urlpatterns` (шаблон URL) определяет набор сопоставлений запросов пользователя с функциями обработки этих запросов. В нашем случае, например, запрос пользователя к корню веб-сайта ('') будет обрабатываться функцией `index`, запрос по адресу 'about' — функцией `about`, а запрос 'contact' — функцией `contact`.

После этих изменений снова запустим приложение командой:

```
python manage.py runserver
```

и перейдем в браузере по адресу <http://127.0.0.1:8000/>. Браузер нам отобразит главную страницу сайта (рис. 4.2).

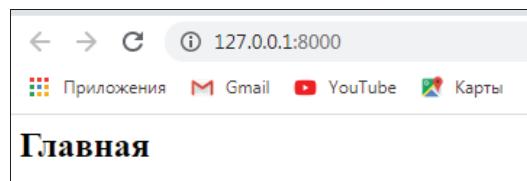


Рис. 4.2. Главная страница сайта

Если в адресной строке браузера изменить адрес на: <http://127.0.0.1:8000/about>, то мы попадем на страницу **О сайте** (рис. 4.3).

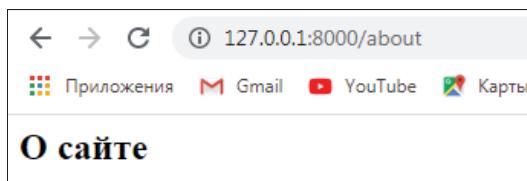


Рис. 4.3. Страница сайта О сайте

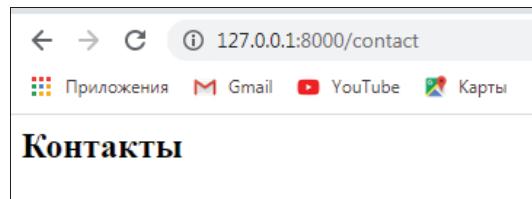


Рис. 4.4. Страница сайта Контакты

Если же в адресной строке браузера изменить адрес на: `http://127.0.0.1:8000/contact`, то мы попадем на страницу **Контакты** (рис. 4.4).

Схематично маршрутизация запросов пользователя и формирование ответов от веб-приложения представлена на рис. 4.5.

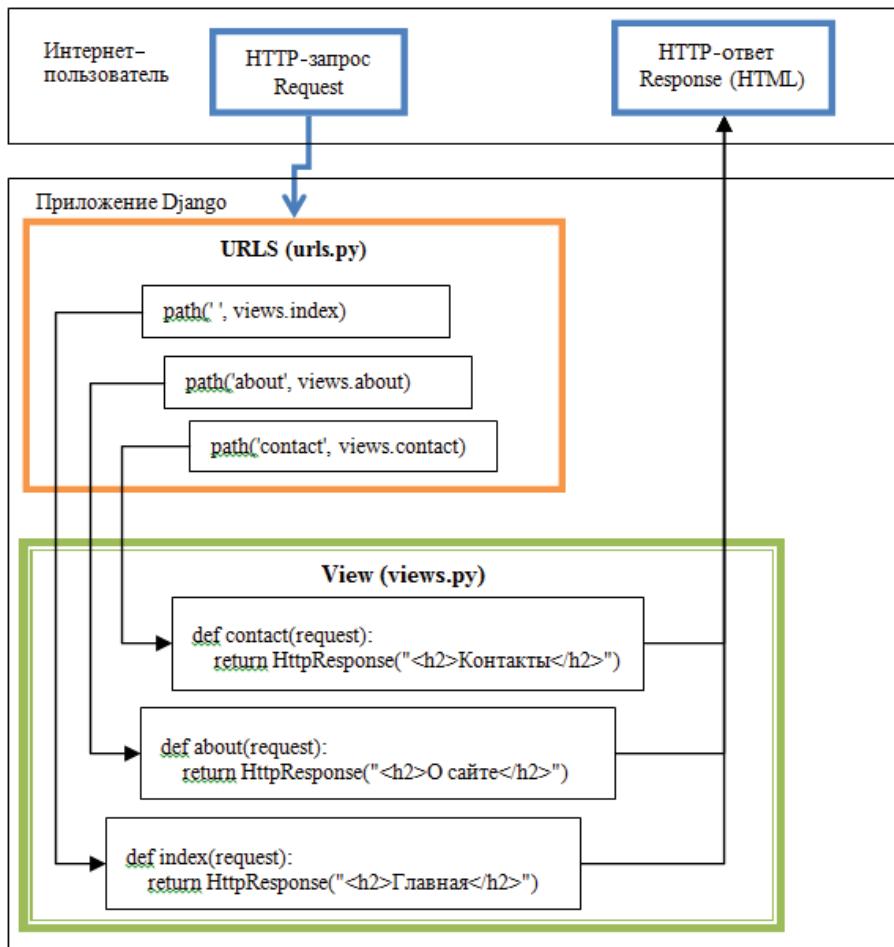


Рис. 4.5. Схема маршрутизации запросов пользователя и формирования ответов от веб-приложения

Следует отметить, что функция `path` при маршрутизации запросов пользователя при неточном описании параметров запроса может вызывать ошибки. В следующем разделе будет показано, как можно минимизировать ошибки при маршрутизации запросов пользователя.

4.2. Маршрутизация запросов пользователей в функциях `path` и `re_path`

В предыдущем разделе было рассмотрено сопоставление интернет-адресов (URL) и функций, которые обрабатывают запросы пользователей по этим адресам. В частности мы создали в файле `views.py` следующие функции (листинг 4.3).

Листинг 4.3

```
def index(request):
    return HttpResponse("<h2>Главная</h2>")

def about(request):
    return HttpResponse("<h2>О сайте</h2>")

def contact(request):
    return HttpResponse("<h2>Контакты</h2>")
```

А в файле `urls.py` они сопоставляются с адресами URL с помощью функции `path()` (листинг 4.4).

Листинг 4.4

```
urlpatterns = [
    path('', views.index),
    path('about', views.about),
    path('contact', views.contact),
]
```

Здесь мы воспользовались функцией `path()`, расположенной в пакете `django.urls` и принимающей два параметра: запрошенный адрес URL и функцию, которая обрабатывает запрос по этому адресу. Дополнительно через третий параметр можно указать имя маршрута:

```
path('', views.index, name='home'),
```

Здесь маршруту задано имя 'home'.

Однако функция `path` имеет серьезное ограничение — запрошенный путь должен абсолютно точно соответствовать указанному в маршруте адресу URL. Так, в приведенном ранее примере, функция `views.about` сможет корректно обработать запрос только в том случае, когда адрес будет в точности соответствовать значению

'about'. Например, стоит добавить к этому значению слеш ('about/'), и Django уже не сможет сопоставить путь с этим запросом, а мы в ответ получим ошибку (рис. 4.6).



Рис. 4.6. Возврат ошибки при обращении к странице about по адресу: <http://127.0.0.1:8000/about/>

В качестве альтернативы для определения маршрутов мы можем использовать функцию `re_path()`, также расположенную в пакете `django.urls`. Ее преимущество состоит в том, что она позволяет задать адреса URL с помощью *регулярных выражений*.

В качестве примера изменим содержание файла `urls.py` следующим образом (листинг 4.5).

Листинг 4.5

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

urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
]
```

Адрес в первом маршруте по-прежнему образуется с помощью функции `path` и указывает на «корень» веб-приложения.

А остальные два маршрута образуются с помощью функции `re_path()`. Причем поскольку определяется регулярное выражение, то перед строкой с шаблоном адреса URL ставится буква `r`. В самом шаблоне адреса можно использовать различные элементы синтаксиса регулярных выражений. В частности, выражение `^about` указывает на то, что адрес должен начинаться с `about`.

Однако он необязательно в точности должен соответствовать строке 'about', как это было в случае с функцией `path`. Мы можем обратиться по любому адресу — главное, чтобы он начинался с `about`, и тогда подобный запрос будет корректно обрабатываться функцией `views.about`. Если мы теперь укажем слеш в конце: ('`about/`'), то Django корректно сопоставит путь с этим запросом, и мы в ответ получим запрошенную страницу (рис. 4.7).

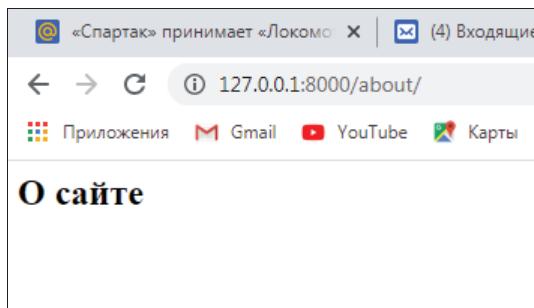


Рис. 4.7. Корректное обращение к странице `about` по адресу: `http://127.0.0.1:8000/about/` при использовании функции `re_path()`

4.3. Очередность маршрутов

Когда к приложению приходит запрос от пользователя, система проверяет соответствие запроса маршрутам по мере их следования в модуле `urls.py`: вначале сравнивается первый маршрут, если он не подходит, то сравнивается второй и т. д. Поэтому более общие маршруты должны определяться в последнюю очередь, а более конкретные маршруты — располагаться в начале этого модуля. Рассмотрим следующий пример следования маршрутов в переменной `urlpatterns`:

```
urlpatterns = [
    re_path(r'^about/contact/', views.contact),
    re_path(r'^about', views.about),
    path('', views.index),
]
```

Здесь адрес `'^about/contact/'` представляет собой более конкретный маршрут по сравнению с `'^about'`. Поэтому он определяется в первую очередь.

4.4. Основные элементы синтаксиса регулярных выражений

В предыдущем разделе мы использовали только один символ в регулярном выражении, определяющем начало адреса (^). Далее приведены некоторые базовые элементы регулярных выражений, которые можно использовать для определения адресов URL:

- ^ — начало адреса;
- \$ — конец адреса;
- + — 1 и более символов;
- ? — 0 или 1 символ;
- {n} — n символов;
- {n, m} — от n до m символов;
- . — любой символ;
- \d+ — одна или несколько цифр;
- \D+ — одна или несколько НЕ цифр;
- \w+ — один или несколько буквенных символов.

В табл. 4.1 представлено несколько возможных сопоставлений шаблонов адресов и запросов.

Таблица 4.1. Варианты возможных сопоставлений шаблонов адресов и запросов

Адрес	Запрос
r'^\$'	http://127.0.0.1/ (корень сайта)
r'^about'	http://127.0.0.1/about/ или http://127.0.0.1/about/contact
r'^about\contact'	http://127.0.0.1/about/contact
r'^products/\d+/'	http://127.0.0.1/products/23/ или http://127.0.0.1/products/6459/abc Но не соответствует запросу: http://127.0.0.1/products/abc/
r'^products/\D+/'	http://127.0.0.1/products/abc/ или http://127.0.0.1/products/abc/123 Не соответствует запросам: http://127.0.0.1/products/123/ или http://127.0.0.1/products/123/abc
r'^products/phones tablets/'	http://127.0.0.1/products/phones/1 или http://127.0.0.1/products/tablets/ Не соответствует запросу: http://127.0.0.1/products/clothes/
r'^products/\w+'	http://127.0.0.1/abc/ или http://127.0.0.1/123/ Не соответствует запросу: http://127.0.0.1/abc-123
r'^products/[-\w]+/'	http://127.0.0.1/abc-123

Таблица 4.1 (окончание)

Адрес	Запрос
r'^products/[A-Z]{2}/'	http://127.0.0.1/RU Не соответствует запросам: http://127.0.0.1/Ru или http://127.0.0.1/RUS

4.5. Параметры представлений

Функции-представления могут принимать параметры, через которые можно передавать различные данные. Такие параметры передаются в адресе URL. Например, в запросе: **http://localhost/index/3/5/** последние два сегмента (**3/5/**) могут представлять параметры URL, которые можно связать с параметрами функции-представления через систему маршрутизации.

4.5.1. Определение параметров через функцию `re_path()`

Вернемся к нашему приложению `firstapp`, которое мы создали в предыдущей главе, и определим в этом приложении в файле `views.py` следующие дополнительные функции (листинг 4.6).

Листинг 4.6

```
def products(request, productid):
    output = "<h2>Продукт № {0}</h2>".format(productid)
    return HttpResponse(output)

def users(request, id, name):
    output = "<h2>Пользователь</h2><h3>id: {0}</h3>
    Имя:{1}</h3>".format(id, name)
    return HttpResponse(output)
```

Здесь функция `products` кроме параметра `request` принимает также параметр `productid` (идентификатор товара). Отправляемый пользователю ответ содержит значение этого параметра.

Функция `users` принимает два дополнительных параметра: `id` и `name` (идентификатор и имя пользователя). И подобным же образом эти данные отправляются обратно пользователю.

Теперь изменим файл `urls.py`, чтобы он мог сопоставить эти функции с запросами пользователя (листинг 4.7).

Листинг 4.7

```
from django.urls import re_path
from firstapp import views

urlpatterns = [
    re_path(r'^products/(?P<productid>\d+)/', views.products),
    re_path(r'^users/(?P<id>\d+)/(?P<name>\D+)/', views.users),
]
```

Для представления параметра в шаблоне адреса используется выражение `?P<>`. Общее определение параметра соответствует формату:

`(?P<имя_параметра>регулярное_выражение)`

В угловых скобках помещается название параметра. После закрывающей угловой скобки следует регулярное выражение, которому должно соответствовать значение параметра.

Например, фрагмент: `?P<productid>\d+` определяет, что параметр называется `productid`, и он должен соответствовать регулярному выражению `\d+`, т. е. представлять последовательность цифр.

Во втором шаблоне адреса: `(?P<id>\d+)/(?P<name>\D+)` определяются два параметра: `id` и `name`. При этом параметр `id` должен представлять число, а параметр `name` — состоять только из буквенных символов.

Ну и также отметим, что количество и название параметров в шаблонах адресов URL соответствуют количеству и названиям параметров соответствующих функций, которые обрабатывают запросы по этим адресам.

Снова в окне терминала запустим локальный сервер разработки:

`python manage.py runserver`

По умолчанию будет запущена главная страница сайта (рис. 4.8).

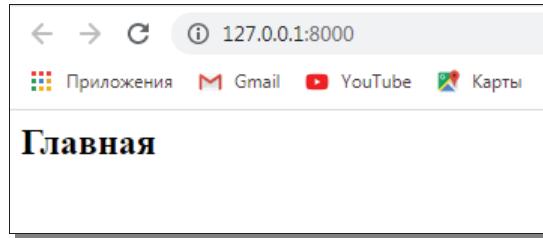


Рис. 4.8. Главная страница сайта, загруженная по умолчанию

Теперь мы можем через адресную строку передать от пользователя данные в приложение в виде запроса. Наберем в адресной строке браузера следующий текст:

`http://127.0.0.1:8000/products/5`

Таким способом пользователь может запросить на веб-сайте информацию о продукте с идентификационным номером 5. В ответ пользователю будет возвращена страница с информацией о продукте № 5 (рис. 4.9).

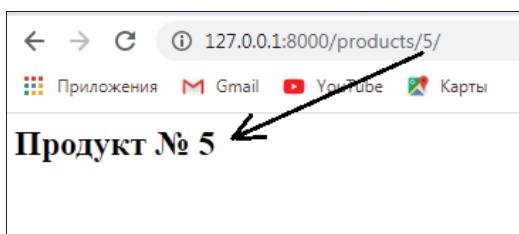


Рис. 4.9. Страница сайта, возвращенная пользователю, запросившему информацию о продукте № 5

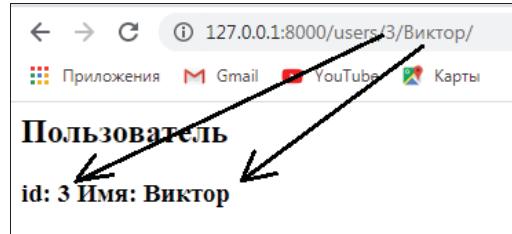


Рис. 4.10. Страница сайта, возвращенная пользователю, запросившему информацию о пользователе № 3

Теперь попробуем через адресную строку сделать в приложение запрос о пользователе. Наберем в адресной строке браузера следующий текст:

```
http://127.0.0.1:8000/users/3/Виктор/
```

Так пользователь может запросить на веб-сайте информацию о пользователе с идентификационным номером 3 по имени Виктор. В ответ пользователю будет возвращена страница с информацией о пользователе № 3 (рис. 4.10).

Однако если мы в запросе не передадим значение для параметра или передадим значение, которое не соответствует регулярному выражению, то система не сможет найти ресурс для обработки такого запроса и выдаст сообщение об ошибке. Например если пользователь запросит информацию о продукте, но при этом не укажет идентификационный номер продукта, то система выдаст в ответ следующее сообщение об ошибке (рис. 4.11).

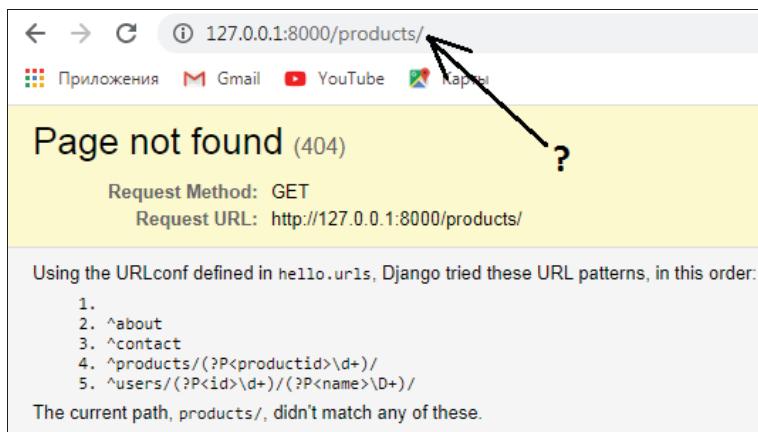


Рис. 4.11. Страница сайта, возвращенная пользователю, после ошибочного запроса информации о продукте

Для предотвращения ошибок такого рода можно в функции `products`, расположенной в файле `views.py`, определить значение параметра по умолчанию (листинг 4.8).

Листинг 4.8

```
def products(request, productid = 1):
    output = "<h2>Продукт № {0}</h2>".format(productid)
    return HttpResponse(output)
```

То есть если в функцию не было передано значение для параметра `productid`, то он получает значение по умолчанию 1.

В этом случае в файле `urls.py` надо дополнительно определить еще один маршрут:

```
re_path(r'^products/$', views.products),
```

Добавив эту строчку, мы в итоге получим следующее содержание файла `urls.py` (листинг 4.9).

Листинг 4.9

```
from django.contrib import admin
from django.urls import path
from django.urls import re_path
from firstapp import views

urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
    re_path(r'^products/$', views.products), # маршрут по умолчанию
    re_path(r'^products/(?P<productid>\d+)/', views.products),
    re_path(r'^users/(?P<id>\d+)/(?P<name>\D+)/', views.users),
]
```

Теперь если в запросе пользователя не будет указан `id` продукта, то система вернет ему не страницу с ошибкой, а страницу с номером продукта, который в функции `products` был задан по умолчанию: `productid=1` (рис. 4.12).

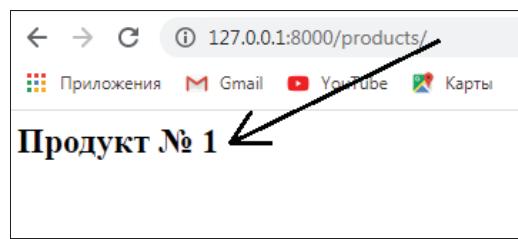


Рис. 4.12. Страница сайта, возвращенная пользователю, запросившему информацию о продукте без указания его идентификатора

4.5.2. Определение параметров через функцию *path()*

Вернемся к функциям, описанным в файле `views.py` (листинг 4.10).

Листинг 4.10

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h2>Главная</h2>")

def about(request):
    return HttpResponse("<h2>О сайте</h2>")

def contact(request):
    return HttpResponse("<h2>Контакты</h2>")

def products(request, productid=1):
    output = "<h2>Продукт № {0}</h2>".format(productid)
    return HttpResponse(output)

def users(request, id, name):
    output = "<h2>Пользователь</h2><h3>id: {0}\n        Имя: {1}</h3>".format(id, name)
    return HttpResponse(output)
```

Определение для них параметров в файле `urls.py` с помощью функции `path()` будет выглядеть следующим образом (листинг 4.11).

Листинг 4.11

```
from django.contrib import admin
from django.urls import path
from django.urls import re_path
from firstapp import views

urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
    path('products/<int:productid>/', views.products),
    path('users/<int:id>/<name>/', views.users),
]
```

Проверим, корректно ли работает функция `path()` при запросе пользователя на получение информации о продукте. Для этого снова в окне терминала запустим локальный сервер разработки:

```
python manage.py runserver
```

и обратимся к странице `products` (рис. 4.13).

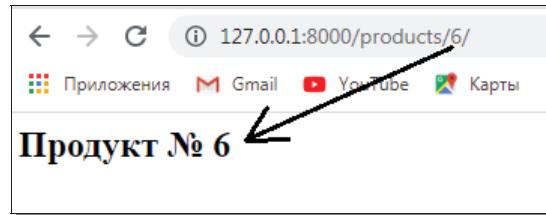


Рис. 4.13. Страница сайта, возвращенная пользователю, запросившему информацию о продукте, при использовании функции `path()` в файле `urls.py`

Как можно видеть, функция `path()` работает здесь аналогично функции `re_path`. То есть в зависимости от предпочтений программиста можно использовать любую из этих функций.

Параметры функции `path()` заключаются в угловые скобки в формате:

```
<спецификатор: название_параметра>
```

В нашем примере в маршруте обращения к странице с продуктами параметр `productid` имеет спецификатор `int` (целое число).

По умолчанию Django предоставляет следующие спецификаторы параметров функции:

- `str` — соответствует любой строке за исключением символа `(/)`. Если спецификатор не указан, то используется по умолчанию;
- `int` — соответствует любому положительному целому числу;
- `slug` — соответствует последовательности буквенных символов ASCII, цифр, дефиса и символа подчеркивания, например: `building-your-1st-django-site`;
- `uuid` — соответствует идентификатору UUID, например: `075194d3-6885-417e-a8a8-6c931e272f00`;
- `path` — соответствует любой строке, которая также может включать символ `(/)`, в отличие от спецификатора `str`.

4.5.3. Определение параметров по умолчанию в функции `path()`

Для примера зададим для функций в файле `views.py` значения параметров по умолчанию для тех страниц сайта, которые выдают информацию о продуктах и пользователях. Мы уже ранее задавали для продуктов значение по умолчанию: `productid=1`.

Теперь зададим для пользователя значения по умолчанию идентификатора пользователя (`id=1`) и имени пользователя (`name="Максим"`). После таких изменений файл `views.py` будет выглядеть следующим образом (листинг 4.12).

Листинг 4.12

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h2>Главная</h2>")

def about(request):
    return HttpResponse("<h2>О сайте</h2>")

def contact(request):
    return HttpResponse("<h2>Контакты</h2>")

def products(request, productid=1):
    output = "<h2>Продукт № {0}</h2>".format(productid)
    return HttpResponse(output)

def users(request, id=1, name="Максим"):
    output = "<h2>Пользователь</h2><h3>id: {0}<br/>Имя: {1}</h3>".format(id, name)
    return HttpResponse(output)
```

После этого для функций `products` и `users` в файле `urls.py` надо определить по два маршрута (листинг 4.13).

Листинг 4.13

```
path('products/', views.products), # маршрут по умолчанию
path('products/<int:productid>', views.products),
path('users/', views.users), # маршрут по умолчанию
path('users/<int:id>/<str:name>', views.users),
```

После этих изменений файл `urls.py` будет выглядеть следующим образом (листинг 4.14).

Листинг 4.14

```
from django.contrib import admin
from django.urls import path
from django.urls import re_path
from firstapp import views
```

```
urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
    path('products/', views.products), # маршрут по умолчанию
    path('products/<int:productid>/', views.products),
    path('users/', views.users), # маршрут по умолчанию
    path('users/<int:id>/<str:name>/', views.users),
]
```

Проверим, корректно ли работает функция `path()` при запросе пользователя на получение информации о продукте и пользователе в том случае, когда в запросе пользователя указаны не все параметры продукта и пользователя. Для этого снова в окне терминала запустим локальный сервер разработки:

```
python manage.py runserver
```

и обратимся к страницам `products` (рис. 4.14, *а*) и `users` (рис. 4.14, *б*).

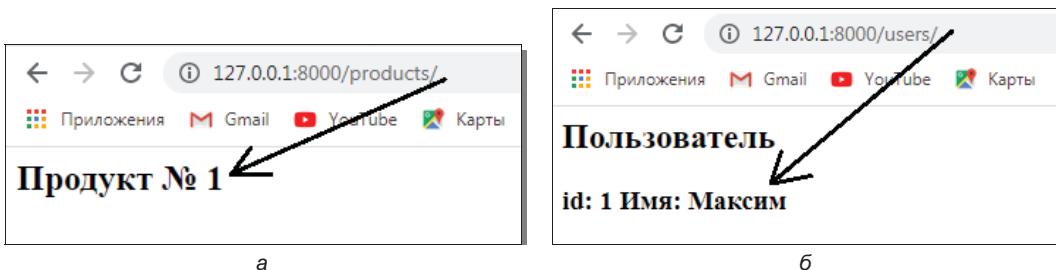


Рис. 4.14. Страницы сайта, возвращенные пользователю, запросившему информацию о продукте (*а*) и пользователе (*б*), с параметрами по умолчанию при использовании функции `path()` в файле `urls.py`

Как можно видеть, при отсутствии параметров в запросе пользователя функция `path()` не выдает здесь ошибку, а возвращает пользователю ответные страницы, используя параметры функций по умолчанию.

4.6. Параметры строки запроса пользователя

Следует четко различать параметры, которые передаются через *интернет-адрес (URL)*, и параметры, которые передаются через *строку запроса*. Например, в запросе:

http://localhost/index/3/Виктор/

два последних сегмента: **3/Виктор/** представляют собой параметры URL. А в запросе:

http://localhost/index?id=3&name= Виктор

те же самые значения **3** и **Виктор** представляют собой параметры строки запроса.

Параметры строки запроса указываются после символа вопросительного знака (?). Каждый такой параметр представляет собой пару «ключ-значение». Например, в параметре **id=3**: **id** — это ключ параметра, а **3** — его значение. Параметры в строке запроса отделяются друг от друга знаком амперсанда (&).

Для получения параметров из строки запроса применяется метод `request.GET.get()`.

Например, определим в файле `views.py` функции `products()` и `users()` следующим образом (листинг 4.15).

Листинг 4.15

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h2>Главная</h2>")

def about(request):
    return HttpResponse("<h2>О сайте</h2>")

def contact(request):
    return HttpResponse("<h2>Контакты</h2>")

def products(request, productid):
    category = request.GET.get("cat", "")
    output = "<h2>Продукт № {0} Категория: {1}</h2>"\
        .format(productid, category)
    return HttpResponse(output)

def users(request):
    id = request.GET.get("id", 1)
    name = request.GET.get("name", "Максим")
    output = "<h2>Пользователь</h2><h3>id: {0} Имя: {1}</h3 >"\
        .format(id, name)
    return HttpResponse(output)
```

Функция `products` принимает обычный параметр `productid` (идентификатор продукта), который будет передаваться через интернет-адрес (URL). И также из строки запроса извлекается значение параметра `cat` (категория продукта) — `request.GET.get("cat", "")`. Здесь первый аргумент функции — это название параметра строки запроса, значение которого надо извлечь, а второй аргумент — значение по умолчанию (на случай, если в строке запроса не оказалось подобного параметра).

В функции `users` из строки запроса извлекаются значения параметров `id` и `name`. При этом заданы следующие значения параметров по умолчанию: `id=1, name= "Максим"`.

Теперь в файле `urls.py` определим следующие маршруты (листинг 4.16).

Листинг 4.16

```
from django.contrib import admin
from django.urls import path
from django.urls import re_path
from firstapp import views

urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
    path('products/<int:productid>/', views.products),
    path('users/', views.users),
]
```

Снова в окне терминала запустим локальный сервер разработки:

```
python manage.py runserver
```

и обратимся к страницам products и users.

При обращении к приложению по адресу:

http://127.0.0.1:8000/products/3/?cat=Телефоны

число **3** будет представлять параметр URL, присваиваемый параметру `productid`, а значение **cat=Телефоны** — представлять параметр `cat` строки запроса. При этом пользователю будет возвращена страница, представленная на рис. 4.15.

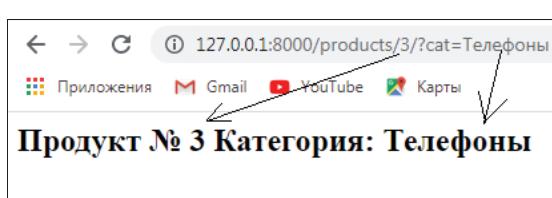


Рис. 4.15. Страница сайта, возвращенная пользователю, запросившему информацию о продукте № 3 категории Телефоны

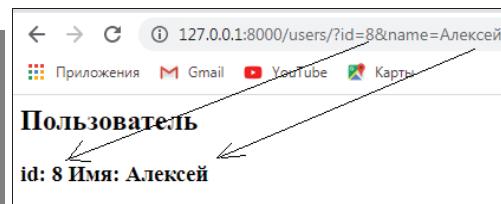


Рис. 4.16. Страница сайта, возвращенная пользователю, запросившему информацию о пользователе № 8 с именем Алексей

А при обращении по адресу:

http://127.0.0.1:8000/users/?id=8&name=Алексей

значения **8** и **Алексей** будут представлять соответственно параметры `id` и `name` (рис. 4.16).

Итак, мы рассмотрели очень важные моменты, связанные с адресацией запросов пользователя и формирования для него ответных страниц. Но часто возникает потребность в переадресации, поскольку после модификации сайта данные могут быть перемещены по другому адресу. Перейдем теперь к рассмотрению способов решения вопроса переадресации.

4.7. Переадресация и отправка пользователю статусных кодов

4.7.1. Переадресация

При перемещении документа с одного адреса на другой мы можем воспользоваться механизмом *переадресации*, чтобы указать пользователям и поисковику, что документ теперь доступен по новому адресу.

Переадресация бывает временная и постоянная. При *временной* переадресации мы указываем, что документ временно перемещен на новый адрес. В этом случае в ответ отправляется статусный код **302**. При постоянной переадресации мы уведомляем систему, что документ теперь постоянно будет доступен по новому адресу.

Для создания временной переадресации применяется класс:

HttpResponseRedirect

Для создания постоянной переадресации применяется класс:

HttpResponsePermanentRedirect

Оба класса расположены в пакете django.http.

Для рассмотрения переадресации в качестве примера напишем в файле views.py следующий код (листинг 4.17).

Листинг 4.17

```
from django.http import HttpResponseRedirect, HttpResponsePermanentRedirect

def index(request):
    return HttpResponse("Index")

def about(request):
    return HttpResponse("About")

def contact(request):
    return HttpResponseRedirect("/about")

def details(request):
    return HttpResponseRedirect("/") .
```

Что мы здесь сделали? При обращении к функции `contact` она станет перенаправлять пользователя по пути "about", который будет обрабатываться функцией `about`. А функция `details` станет использовать постоянную переадресацию и перенаправлять пользователя на «корень» (главную страницу) веб-приложения.

Для этого примера нам также необходимо в файле `urls.py` определить следующие маршруты (листинг 4.18).

Листинг 4.18

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

urlpatterns = [
    path('', views.index),
    path('about/', views.about),
    path('contact/', views.contact),
    path('details/', views.details),
]
```

Для проверки работы переадресации в окне терминала запустим локальный сервер разработки:

```
python manage.py runserver
```

По умолчанию будет загружена главная страница **Index** (рис. 4.17).

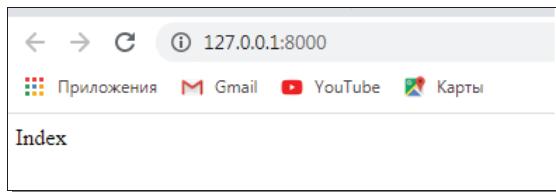


Рис. 4.17. Главная страница сайта, загруженная по умолчанию

Теперь в адресной строке браузера наберем адрес страницы **contact**:

http://127.0.0.1:8000/contact

При этом, поскольку мы задали переадресацию, вместо страницы **contact** будет загружена страница **About** (рис. 4.18).

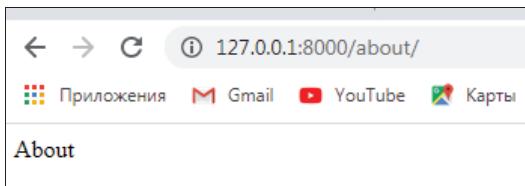


Рис. 4.18. Переадресация на страницу **About** при попытке вызова страницы **contact**

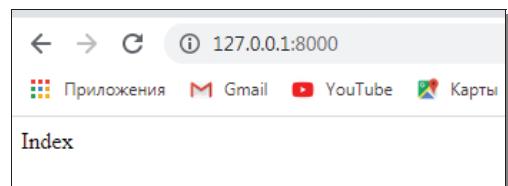


Рис. 4.19. Переадресация на страницу **Index** при попытке вызова страницы **details**

Теперь в адресной строке браузера наберем адрес страницы **details**:

http://127.0.0.1:8000/details

При этом, поскольку мы задали переадресацию, вместо страницы **details** будет загружена главная страница сайта **Index** (рис. 4.19).

4.7.2. Отправка пользователю статусных кодов

В пакете django.http есть ряд классов, которые позволяют отправлять пользователю определенный *статусный код* (табл. 4.2).

Таблица 4.2. Перечень статусных кодов

Статусный код	Класс
304 (Not Modified)	HttpResponseNotModified
400 (Bad Request)	HttpResponseBadRequest
403 (Forbidden)	HttpResponseForbidden
404 (Not Found)	HttpResponseNotFound
405 (Method Not Allowed)	HttpResponseNotAllowed
410 (Gone)	HttpResponseGone
500 (Internal Server Error)	HttpResponseServerError

Вот пример применения этих классов:

```
from django.http import *

def m304(request):
    return HttpResponseNotModified()

def m400(request):
    return HttpResponseBadRequest("<h2>Bad Request</h2>")

def m403(request):
    return HttpResponseForbidden("<h2>Forbidden</h2>")

def m404(request):
    return HttpResponseNotFound("<h2>Not Found</h2>")

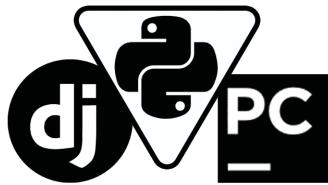
def m405(request):
    return HttpResponseNotAllowed("<h2>Method is not allowed</h2>")

def m410(request):
    return HttpResponseGone("<h2>Content is no longer here</h2>")

def m500(request):
    return HttpResponseServerError("<h2>Something is wrong</h2>")
```

4.8. Краткие итоги

В этой главе были представлены сведения о ключевых элементах Django: представлениях (views) и маршрутизации запросов пользователей. Ответы на поступившие от пользователей запросы возвращаются ему в виде HTML-страниц. В свою очередь, эти страницы формируются на основе шаблонов. Созданию и использованию шаблонов посвящена следующая глава.



ГЛАВА 5

Шаблоны

Отображение данных на сайте Django осуществляется с помощью шаблонов и встроенных в шаблоны тегов. Шаблоны представляют собой HTML-страницы. Дело в том, что на страницах HTML нельзя в прямую размещать код Python, поскольку браузеры его не воспринимают — они понимают только HTML. Мы также знаем, что страницы HTML по своей сути статичны, в то время как Python позволяет вносить в программы динамические изменения. Так вот, теги шаблонов Django позволяют вставлять результаты работы программы на Python в HTML-страницы, что дает нам возможность создавать динамические веб-сайты быстрее и проще. Из этой главы мы узнаем:

- что такое шаблоны и для чего они нужны;
- как использовать функцию `render()` и класс `TemplateResponse` для загрузки шаблонов;
- как передать в шаблоны простые и сложные данные;
- что такое статичные файлы, файлы CSS и как использовать статичные файлы в приложениях на Django;
- как можно изменить конфигурацию шаблонов HTML-страниц;
- как можно расширить шаблоны HTML-страниц на основе базового шаблона;
- как можно использовать специальные теги в шаблонах HTML-страниц.

5.1. Создание и использование шаблонов

Шаблоны (*templates*) отвечают за формирование внешнего вида приложения. Они предоставляют специальный синтаксис, который позволяет внедрять данные в код HTML.

В предыдущих главах мы создали и доработали проект `hello` с приложением `firstapp`. Теперь добавим в проект шаблоны. Для этого определим в корневой папке проекта новый каталог с именем `templates` (рис. 5.1). Вообще-то имя папки с шаблонами может быть любым, но, как правило, для лучшего восприятия структуры проекта этой папке все же лучше присвоить значащее имя `templates`.

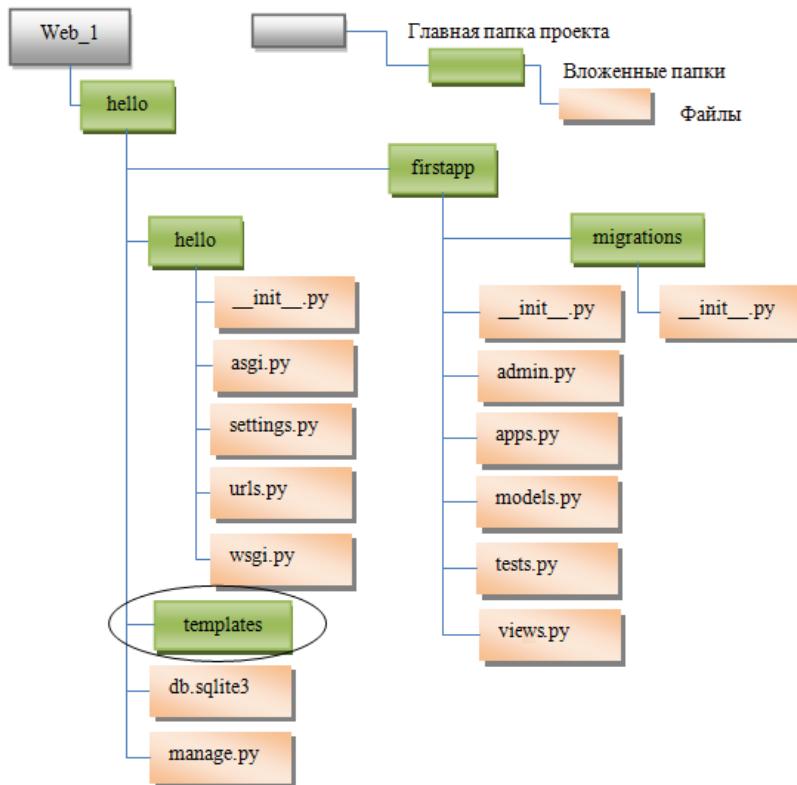


Рис. 5.1. Добавление папки templates в веб-приложение с проектом firstapp

Теперь нам надо указать, что этот каталог будет использоваться в качестве хранилища шаблонов. Для этого откроем файл `settings.py`. В этом файле настройка шаблонов производится с помощью переменной `TEMPLATES`:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.
                    django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Здесь параметр `DIRS` задает набор каталогов, которые хранят шаблоны. Но по умолчанию он пуст. Изменим этот фрагмент кода следующим образом (выделено серым фоном и полужирным шрифтом):

```
TEMPLATE_DIR = os.path.join(BASE_DIR, "templates")

TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [TEMPLATE_DIR],
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
},
]


```

Внесенные в проект изменения показаны на рис. 5.2.

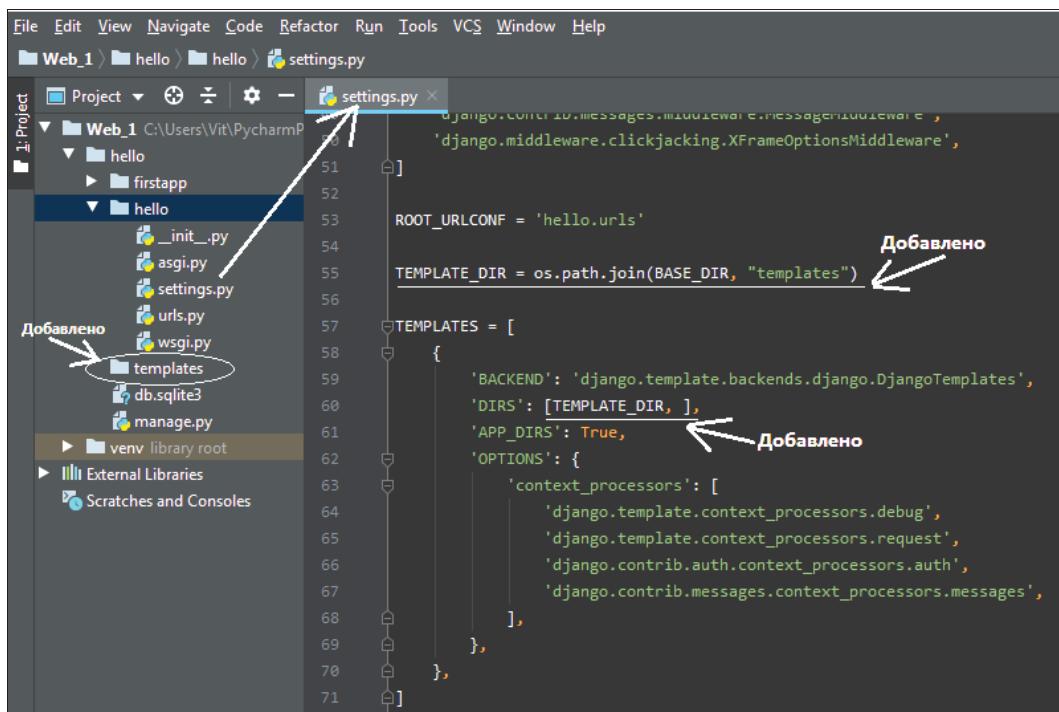


Рис. 5.2. Настройка параметров папки templates в файле settings.py

Теперь создадим в папке templates новый файл index.html. Для этого в окне программной оболочки PyCharm щелкните правой кнопкой мыши на имени проекта **Web_1** и из появившегося меню выполните команду: **template | File | HTML File** (рис. 5.3) — откроется окно **New HTML File** (рис. 5.4), в котором введите имя создаваемого файла: `index` и нажмите клавишу <Enter>.

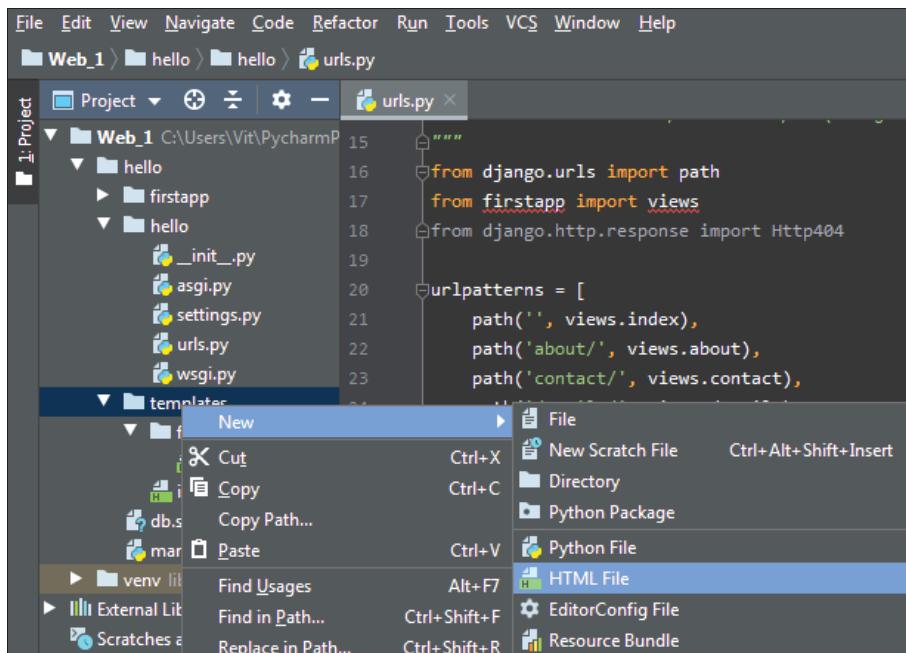


Рис. 5.3. Создание нового HTML-файла в PyCharm

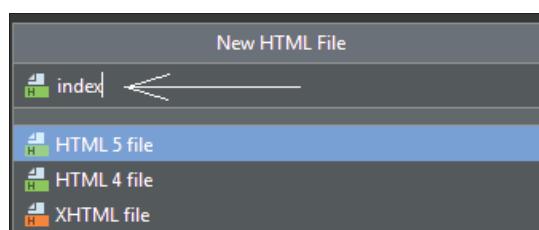


Рис. 5.4. Задание имени нового HTML-файла в PyCharm

В результате будет создан файл `index.html` (рис. 5.5), содержащий по умолчанию следующий программный код (листинг 5.1).

Листинг 5.1

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

</body>
</html>
```

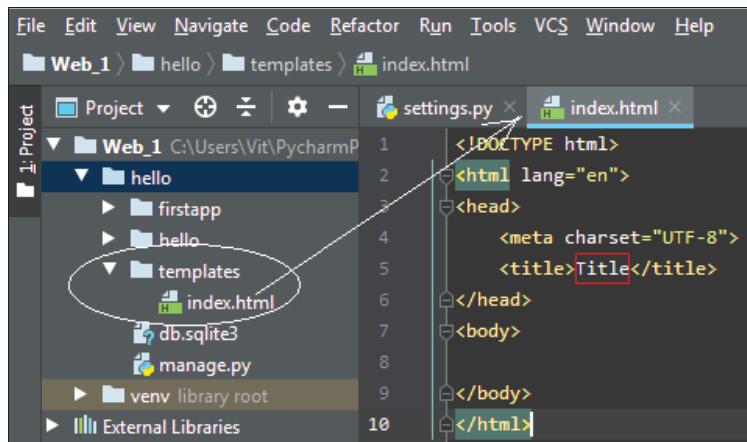


Рис. 5.5. Содержание файла index.html, созданного в PyCharm

Внесем в этот программный код следующие изменения (листинг 5.2).

Листинг 5.2

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Привет Django!</title>
    </head>
    <body>
        <h1>Вы на главной странице Django!</h1>
        <h2>templates/index.html</h2>
    </body>
</html>
```

По сути, это обычная веб-страница, содержащая код HTML. Теперь используем эту страницу для отправки ответа пользователю. Для этого перейдем в приложении firstapp к файлу `views.py`, который определяет функции для обработки запроса пользователя, и изменим этот файл следующим образом (листинг 5.3).

Листинг 5.3

```
from django.shortcuts import render

def index(request):
    return render(request, "index.html")
```

Что мы здесь сделали? Первым шагом мы из модуля `django.shortcuts` импортировали функцию `render` (предоставлять). Вторым шагом изменили функцию `def index(request)`. Теперь функция `index(request)` вызывает функцию `render`, которой передаются объект запроса пользователя `request` и файл шаблона `index.html`, который находится в папке `templates`. Эти изменения, сделанные в PyCharm, представлены на рис. 5.6.

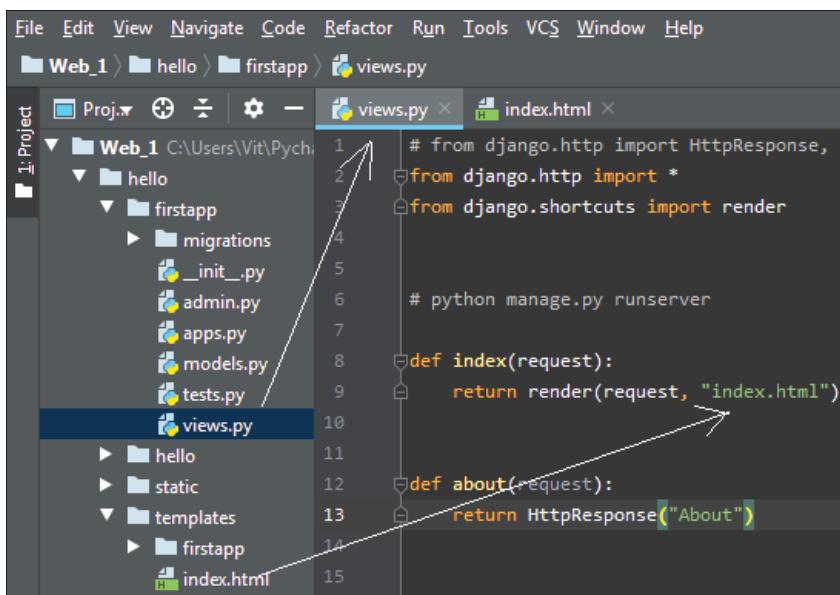


Рис. 5.6. Изменение содержания файла `views.py`, в программной оболочке PyCharm

Проверим, нужно ли вносить изменения в файл `urls.py` в главном проекте `hello`. Там должно быть прописано сопоставление функции `index` с запросом пользователя к «корню» веб-приложения (листинг 5.4).

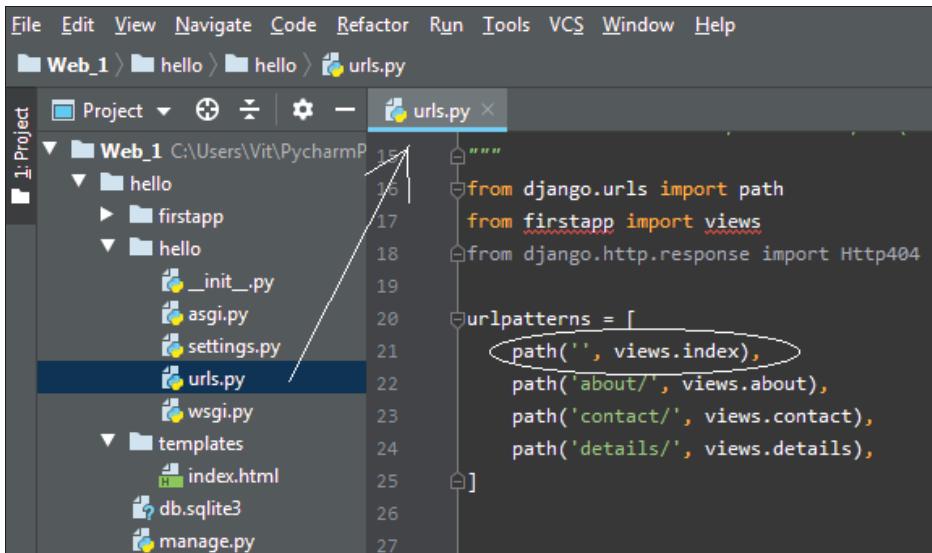
Листинг 5.4

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

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

Чтобы это проверить, откроем файл urls.py (рис. 5.7) и убедимся, что в нем для обращения пользователя к «корню» веб-приложения прописан следующий маршрут:

```
path('', views.index)
```



```
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Web_1 hello hello urls.py
Project urls.py
Web_1 C:\Users\Vit\PycharmP...
  hello
    firstapp
    hello
      __init__.py
      asgi.py
      settings.py
      urls.py
      wsgi.py
    templates
      index.html
      db.sqlite3
      manage.py
15 """
16 from django.urls import path
17 from firstapp import views
18 from django.http.response import Http404
19
20 urlpatterns = [
21     path('', views.index),
22     path('about/', views.about),
23     path('contact/', views.contact),
24     path('details/', views.details),
25 ]
26
27
```

Рис. 5.7. Маршрут перехода к функции index() при обращении пользователя к «корню» веб-приложения

После всех этих изменений запустим локальный сервер разработки командой:

```
python manage.py runserver
```

и перейдем в браузере по адресу: <http://127.0.0.1:8000/> — браузер нам отобразит главную страницу сайта (рис. 5.8).

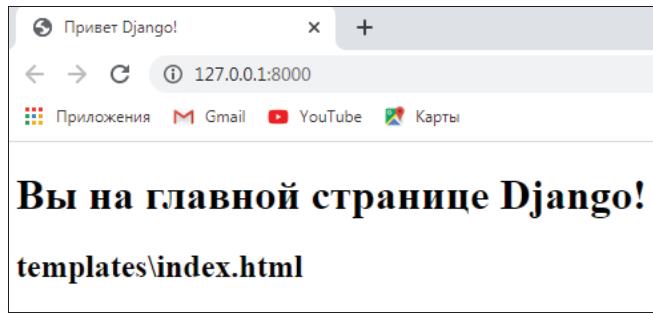


Рис. 5.8. Главная страница сайта, загруженная из шаблона templates\index.html

Нас можно поздравить — мы создали свой первый шаблон. Пока он пустой, но это не беда. Теперь мы можем вносить в этот шаблон изменения, менять его дизайн, вставлять в него данные и возвращать пользователю динамически генерируемые веб-страницы.

Однако в проекте Django нередко бывает несколько приложений. И каждое из этих приложений может иметь свой набор шаблонов. Чтобы разграничить шаблоны для отдельных проектов, можно создавать для шаблонов каждого приложения отдельный каталог. Например, в нашем случае у нас одно приложение — `firstapp`. Создадим для него в папке `templates` каталог `firstapp` (по имени приложения). И в этом каталоге определим файл `home.html` (рис. 5.9).

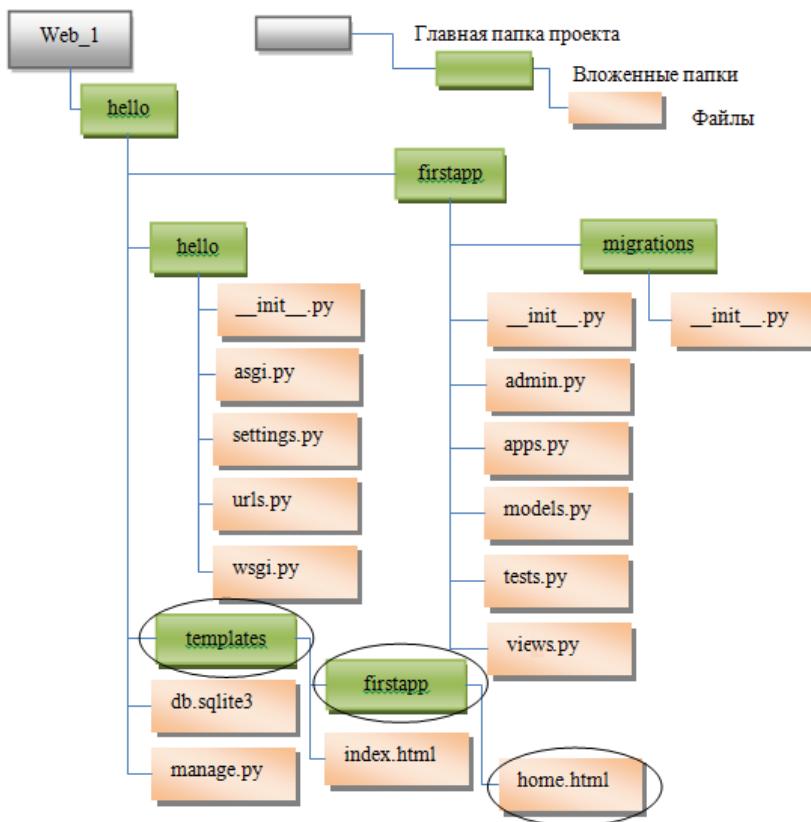


Рис. 5.9. Место файла `home.html` в структуре каталогов проекта `hello`

Внесем в содержание файла `home.html` следующие изменения (листинг 5.5), показанные на рис. 5.10.

Листинг 5.5

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
  
```

```
<body>
    <h1>Домашняя страница Django!</h1>
    <h2>templates/firstapp/home.html</h2>
</body>
</html>
```

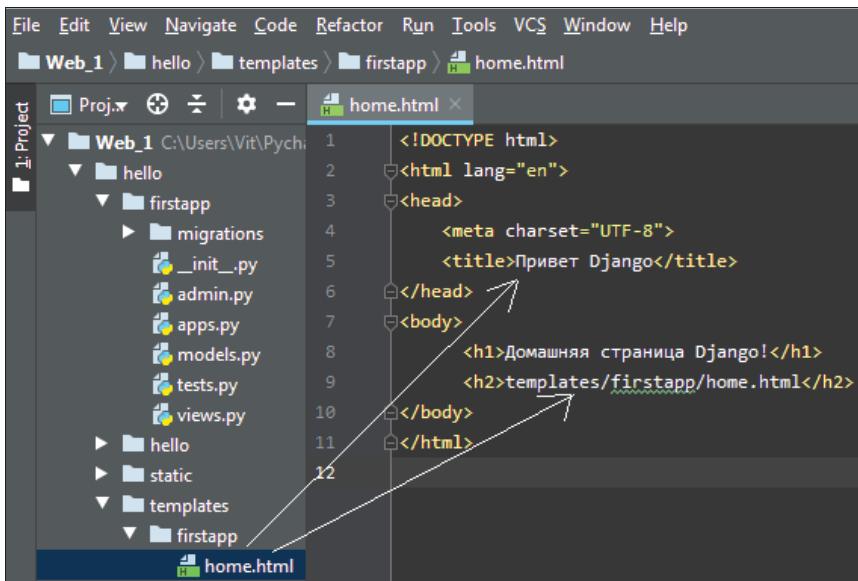


Рис. 5.10. Изменения, произведенные в файле home.html

Теперь изменим файл `views.py` нашего приложения, указав в нем новый путь к странице сайта, которую нужно выдать пользователю при его обращении к домашней странице сайта — `home.html` (рис. 5.11).

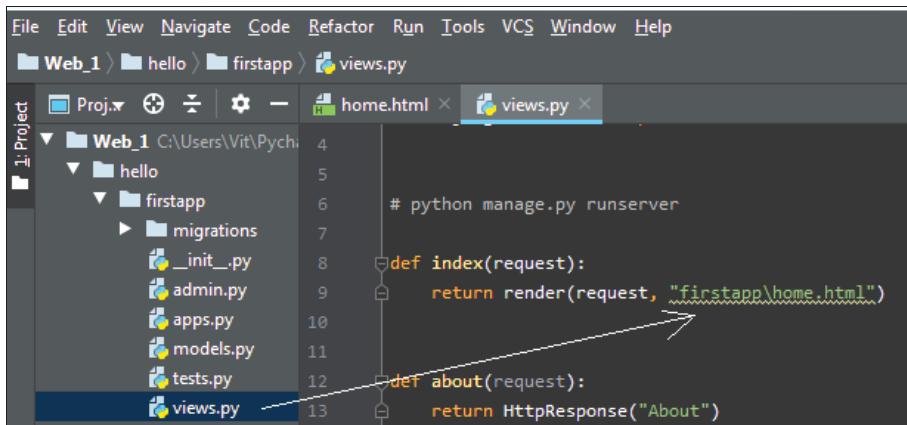


Рис. 5.11. Задание пути к странице `home.html`, которую нужно выдать пользователю при его обращении к домашней странице сайта

Теперь функция `index()` будет выглядеть следующим образом (листинг 5.6).

Листинг 5.6

```
def index(request):
    return render(request, "firstapp\home.html")
```

После этих изменений запустим локальный сервер разработки командой:

```
python manage.py runserver
```

и перейдем в браузере по адресу: <http://127.0.0.1:8000/> — браузер нам отобразит новую домашнюю страницу сайта `home.html` (рис. 5.12).

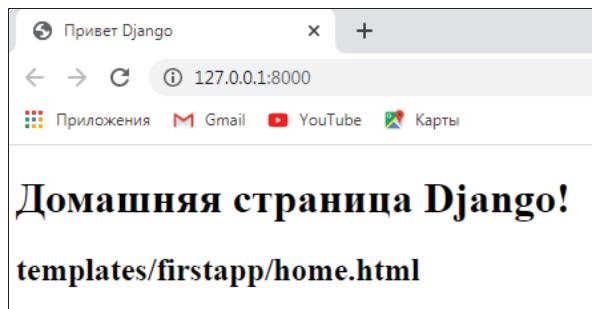


Рис. 5.12. Домашняя страница сайта `home.html`

Стоит отметить, что теперь в пути к шаблону страницы показывается и папка, в которой он находится: `firstapp/home.html`.

5.2. Класс `TemplateResponse`

В предыдущем разделе для загрузки (вызова) шаблона применялась функция `render()`, что является наиболее распространенным вариантом. Однако мы также можем использовать класс `TemplateResponse` (шаблонный ответ). Функция `def index(request)` при использовании класса `TemplateResponse` будет выглядеть следующим образом (листинг 5.7).

Листинг 5.7

```
from django.template.response import TemplateResponse

def index(request):
    return TemplateResponse(request, "firstapp\home.html")
```

Результат работы приложения с использованием класса `TemplateResponse` будет таким же, как и при использовании функции `render()` (листинг 5.8).

Листинг 5.8

```
from django.shortcuts import render

def index(request):
    return render(request, "firstapp/home.html")
```

Итак, мы научились создавать шаблоны HTML-страниц и возвращать их в виде ответов пользователю на их запросы. Но в приводимых примерах мы, по сути, возвращали пустые страницы, что конечному пользователю не совсем интересно. Он ожидает получить в ответ какую-то полезную информацию. То есть веб-приложение должно уметь заполнить шаблон страницы теми данными, которые запросил пользователь. Решению этой задачи и будет посвящен следующий раздел.

5.3. Передача данных в шаблоны

Одним из преимуществ шаблонов является то, что мы можем передать в них пользователю различные данные, которые будут динамически подгружены из базы данных через представления (views). Для вывода данных в шаблоне могут использоваться различные способы. Так, вывод самых простых данных может осуществляться с помощью двойной пары фигурных скобок:

```
{ {название_объекта} }
```

Вернемся к нашему проекту `hello`, содержащему приложение `firstapp`. Добавим в папку `templates\firstapp` еще один шаблон страницы — `index_app1.html` (рис. 5.13).

Внесем в код этой страницы (листинг 5.9) следующие изменения, показанные на рис. 5.14.

Листинг 5.9

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django!</title>
</head>
<body>
    <h1>{ {header} }</h1>
    <p>{ {message} }</p>

</body>
</html>
```

Как можно видеть, мы ввели здесь две новые переменные: `header` и `message`. Эти переменные и будут получать значения из представления (view).

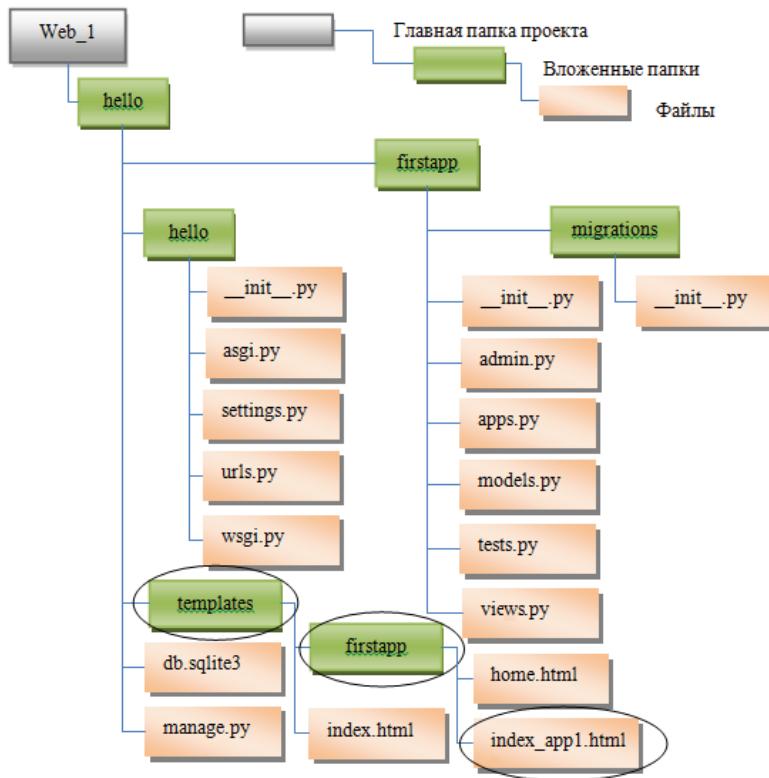


Рис. 5.13. В структуру приложения `firstapp` добавлен шаблон страницы `templates\firstapp\index_app1.html`

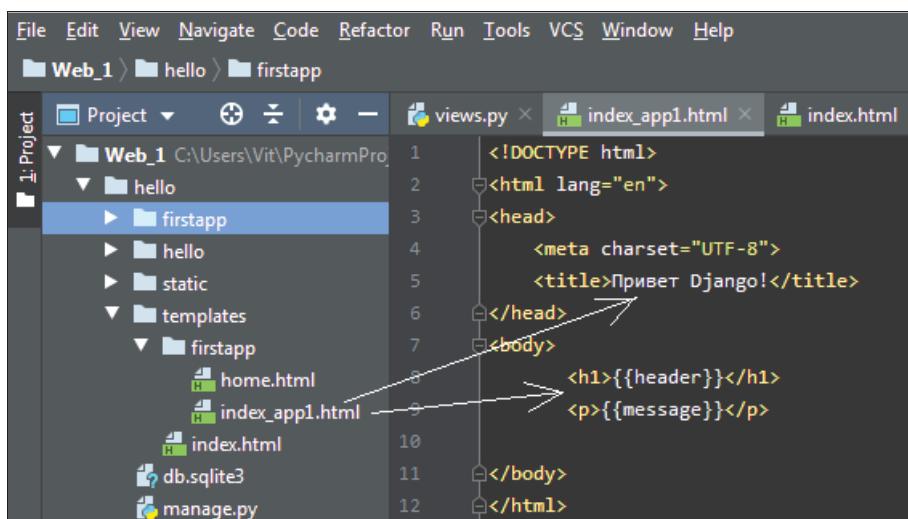


Рис. 5.14. Изменение кода страницы сайта `templates\firstapp\index_app1.html`

Чтобы из функции-представления передать данные в шаблон применяется еще один (третий) параметр в функции `render`, который называется *контекст* (`context`). В качестве примера изменим в файле `views.py` функцию `def index()` следующим образом (листинг 5.10). Эти изменения показаны на рис. 5.15.

Листинг 5.10

```
def index(request):
    # return render(request, "firstapp/home.html")
    data = {"header": "Передача параметров в шаблон Django",
            "message":
                "Загружен шаблон templates/firstapp/index_app1.html"}
    return render(request, "firstapp/index_app1.html", context=data)
```

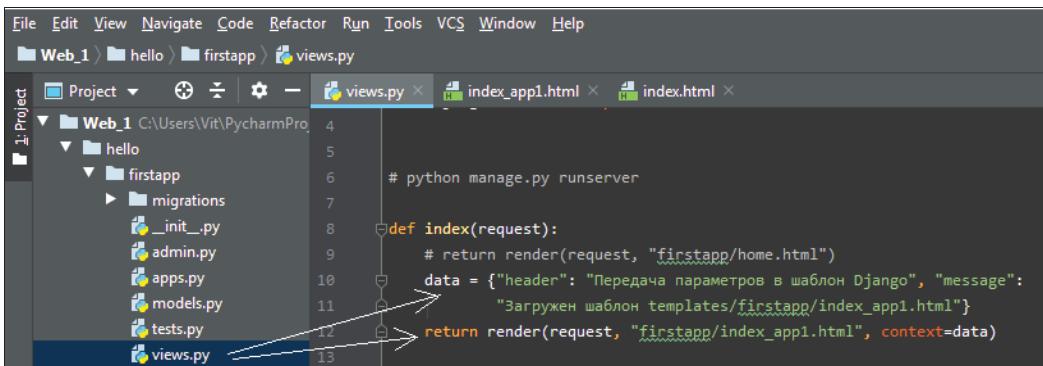


Рис. 5.15. Изменение кода страницы сайта в файле view.py (согласно листингу 5.10)

В нашем шаблоне мы использовали две переменные: `header` и `message`, соответственно, словарь `data`, который передается в функцию `render` через параметр `context`, теперь содержит два значения с ключами: `header` и `message`. Этим ключам мы присвоили следующие значения:

```
{"header": "Передача параметров в шаблон Django",
"message": "Загружен шаблон templates/firstapp/index_app1.html"}
```

В результате внесенных изменений при обращении к «корню» веб-приложения мы увидим в браузере следующую страницу (рис. 5.16).

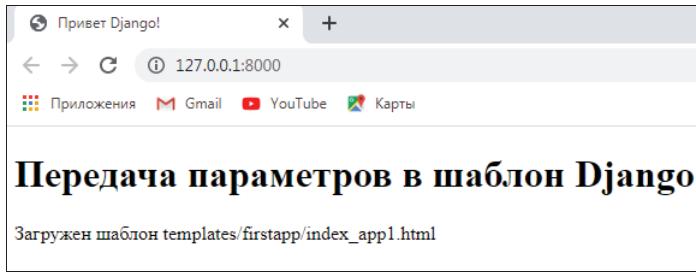


Рис. 5.16. Домашняя страница сайта с данными, подгруженными в шаблон index_app1.html

5.4. Передача в шаблон сложных данных

В предыдущем разделе мы встроили в шаблон простые текстовые данные. Рассмотрим теперь передачу пользователю через шаблон более сложных данных. Для этого изменим функцию `def index()` в представлении (в файле `views.py`) следующим образом (листинг 5.11). Эти изменения показаны на рис. 5.17.

Листинг 5.11

```
def index(request):
    header = "Персональные данные" # обычная переменная
    langs = ["Английский", "Немецкий", "Испанский"] # массив
    user = {"name": "Максим,", "age": 30} # словарь
    addr = ("Виноградная", 23, 45) # кортеж
    data = {"header": header, "langs": langs, "user": user, "address": addr}
    return render(request, "index.html", context=data)
```

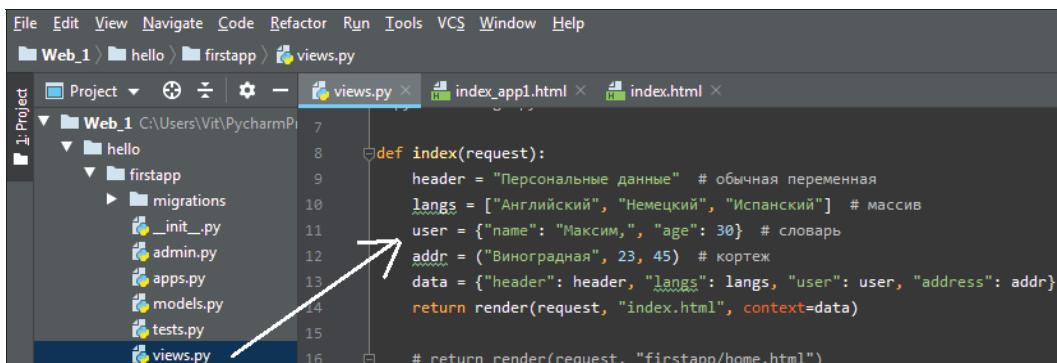


Рис. 5.17. Изменение кода страницы сайта в файле view.py (согласно листингу 5.11)

Здесь мы создали несколько переменных: `header`, `langs`, `user` и `addr` и все эти переменные обернули в словарь `data`. Затем в функции `render()` передали этот словарь третьему параметру — `context`.

Теперь нам нужно изменить сам шаблон `templates\index.html`, чтобы он смог принять новые данные (листинг 5.12). Эти изменения показаны на рис. 5.18.

Листинг 5.12

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Передача сложных данных</title>
</head>
<body>
    <h1>{ {header} }</h1>
```

```

<p>Имя: {{user.name}} Age: {{user.age}}</p>
<p>Адрес: ул. {{address.0}}, д. {{address.1}}, кв. {{address.2}}</p>
<p>Владеет языками: {{langs.0}}, {{langs.1}}</p>
</body>
</html>

```

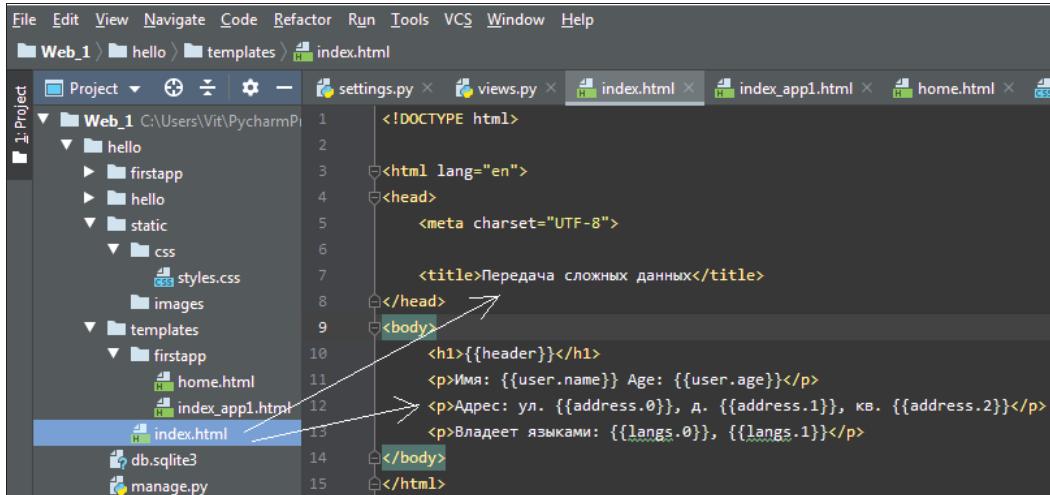


Рис. 5.18. Изменение кода страницы сайта templates\index.html

Поскольку объекты langs и address представляют, соответственно, массив и кортеж, то мы можем обратиться к их элементам через индексы, как мы это делали с ними в любом программном коде на Python. Например, первый элемент кортежа адреса (address) мы можем получить следующим образом: address.0. Соответственно, к элементам массива, содержащего языки (langs) можно обращать по номеру элемента в массиве: langs.0, langs.1.

Поскольку объект с информацией о пользователе (user) представляет словарь, то аналогичным образом мы можем обратиться к его элементам по ключам слова name и age следующим образом: user.name, user.age.

В итоге после этих преобразований мы получим в веб-браузере следующую страницу (рис. 5.19).

Если в функции def index() будет применяться класс TemplateResponse, то в его конструктор также через третий параметр можно передать данные для шаблона. В этом случае программный код для этой функции будет выглядеть следующим образом (листинг 5.13).

Листинг 5.13

```

from django.template.response import TemplateResponse

def index(request):
    header = "Персональные данные"                      # обычная переменная

```

```

langs = ["Английский", "Немецкий", "Испанский"] # массив
user = {"name": "Максим,", "age": 30}           # словарь
addr = ("Виноградная", 23, 45)                  # кортеж
data = {"header": header, "langs": langs, "user": user, "address": addr}
return TemplateResponse(request, "index.html", data)

```

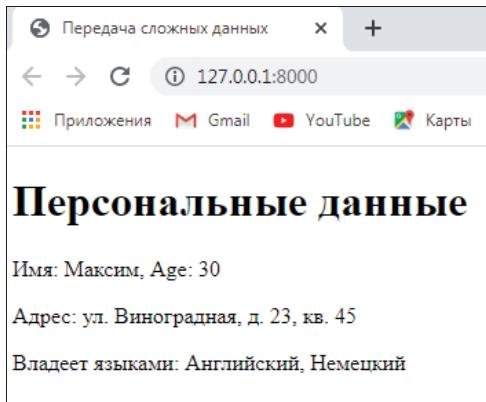


Рис. 5.19. Домашняя страница сайта с данными, подгруженными в шаблон index.html

5.5. Статичные файлы

Пришло время ближе познакомиться со статическими файлами. *Статическими файлами* называются все файлы каскадных таблиц стилей (Cascading Style Sheets, CSS), изображений, а также скриптов javascript, — т. е. файлы, которые не изменяются динамически и содержание которых не зависит от контекста запроса и однаково для всех пользователей. Можно воспринимать их как своего рода «макияж» для веб-страниц.

5.5.1. Основы каскадных таблиц стилей

Для придания привлекательности информации, выводимой на HTML-страницах, используются различные стили форматирования текстов, оформленные в виде *каскадных таблиц стилей* (CSS) с помощью их специального языка. Такой подход обеспечивает возможность прикреплять стиль (тип шрифта, его размер цвет и пр.) к структурированным документам. Обычно CSS-стили используются для создания и изменения стиля элементов веб-страниц и пользовательских интерфейсов, написанных на языках HTML, но также могут быть применены к любому виду XML-документа. Отделяя стиль представления документов от содержимого документов, CSS упрощает создание веб-страниц и обслуживание сайтов.

Объявление стиля состоит из двух частей: *селектора* и *объявления*. В HTML имена элементов нечувствительны к регистру, поэтому в селекторе значение `h1` работает так же, как и `H1`. Объявление состоит из двух частей: имени свойства (например,

color) и значения свойства (например, grey). Селектор сообщает браузеру, какой именно элемент форматировать, а в блоке объявления (код в фигурных скобках) указываются форматирующие команды — свойства и их значения (рис. 5.20).



Рис. 5.20. Задание стилей на HTML-страницах

Стили могут быть следующих видов:

- встроенные;
- внутренние;
- внешняя таблица стилей.

Внутренние стили имеют приоритет над внешними таблицами стилей, но уступают встроенным стилям, заданным через атрибут style.

При использовании *встроенных стилей* CSS-код располагается в HTML-файле, непосредственно внутри тега элемента с помощью атрибута style:

```
<p style="font-weight: bold; color: red;">Этот текст будет красного цвета!</p>
```

Рассмотрим использование стилей на нескольких примерах. Для этого вернемся к нашему проекту hello и перейдем к файлу hello\templates\firstapp\home.html. Этот файл содержит следующий текст (листинг 5.14).

Листинг 5.14

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <h1>Домашняя страница Django!</h1>
    <h2>templates/firstapp/home.html</h2>
</body>
</html>
```

Откроем файл views.py и изменим значение функции def index() следующим образом (листинг 5.15).

Листинг 5.15

```
def index(request):
    return render(request, "firstapp/home.html")
```

Теперь, если запустить локальный веб-сервер:

```
python manage.py runserver
```

и перейти на главную страницу: <http://127.0.0.1:8000/>, то мы увидим следующий результат (рис. 5.21).

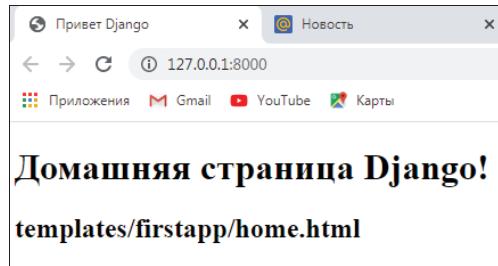


Рис. 5.21. Вывод страницы home.html
без использования стилей

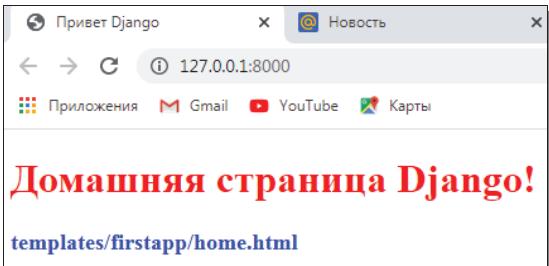


Рис. 5.22. Вывод страницы home.html
с использованием встроенного стиля

Изменим стиль выводимого текста: первую строку выведем красным цветом, а вторую — синим. Для этого изменим текст файла home.html следующим образом (листинг 5.16).

Листинг 5.16

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <!-- Это встроенный стиль -->
    <font style="color: red">
        <h1>Домашняя страница Django!</h1>
        <font style="color: blue; font-size: 12px">
            <h2>templates/firstapp/home.html</h2>
    </font>
</body>
```

Снова обратимся к странице home.html и получим теперь следующий результат (рис. 5.22).

Как можно здесь видеть (при выводе цветного изображения на экран компьютера), цвета выводимого текста поменялись: первая строка выведена красным цветом, а вторая — синим.

Использовать встроенные стили достаточно просто, но не совсем удобно. Если через какое-то время потребуется изменить цвет всех заголовков на всех страницах, то нужно будет менять код на каждой из страниц.

Внутренние стили отличаются от встроенных стилей тем, что они встраиваются в раздел `<head>...</head>` HTML-документа. Изменим текст файла `home.html` следующим образом (листинг 5.17).

Листинг 5.17

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- Это внутренний стиль -->

    <style>h1{color: green;}</style>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <!-- Это встроенный стиль -->

    <font style="color: red; font-size: 12px">
        <h1>Домашняя страница Django!</h1>
        <font style="color: blue; font-size: 12px">
            <h2>templates/firstapp/home.html</h2>
    </font>
</body>
</html>
```

Здесь мы в теге `<head>...</head>` для текста в теге `h1` задали зеленый цвет, но при этом в теге `<body>...</body>` оставили в стиле для этого текста красный цвет. Снова обратимся к странице `home.html` и теперь получим следующий результат (рис. 5.23).

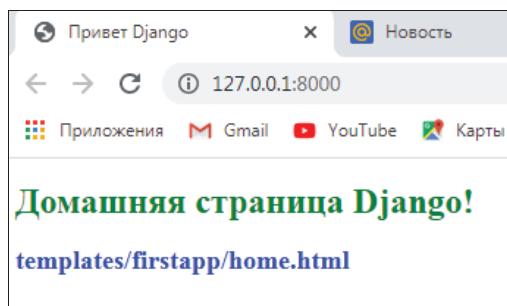


Рис. 5.23. Вывод страницы `home.html` с использованием внутреннего стиля

Несмотря на то что перед тегом `h1` во встроенным стиле задан красный цвет текста, выведен он в зеленом. То есть здесь задействован тот стиль (внутренний), который был указан в заголовке страницы.

Проверим теперь в работе приоритетность внутренних стилей над встроенными, для чего изменим текст файла `home.html` следующим образом (листинг 5.18).

Листинг 5.18

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- Это внутренний стиль -->
    <style>h1{color: green;}</style>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <!-- Это встроенный стиль -->
    <style>h1{color: red;}</style>
    <h1>Домашняя страница Django!</h1>
    <font style="color: blue; font-size: 12px">
        <h2>templates/firstapp/home.html</h2>
    </font>
</body>
</html>
```

Здесь в заголовке страницы для тега `h1` задан зеленый цвет текста (внутренний стиль), а в теле страницы для тега `h1` — красный (встроенный стиль). Поскольку встроенный стиль имеет приоритет над внутренним стилем, то текст заголовка должен быть выведен красным цветом. Результат работы этого программного кода показан на рис. 5.24.

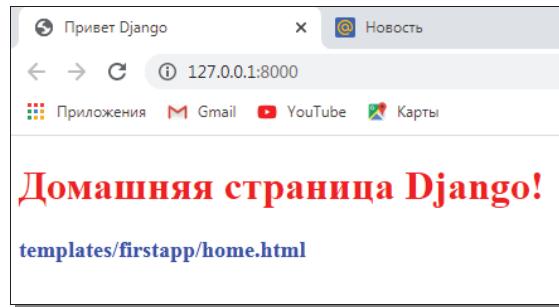


Рис. 5.24. Вывод страницы `home.html` для демонстрации приоритета встроенного стиля над внутренним

Как можно видеть (при выводе цветного изображения на экран компьютера), заголовок страницы имеет красный цвет, что подтверждает приоритет встроенного стиля над внутренним.

Наконец, перейдем к *внешней таблице стилей*, которая представляет собой текстовый файл с расширением `css`. В этом файле находится набор CSS-стилей для различных элементов HTML-страниц.

Файл каскадных страниц стилей создается в редакторе кода, так же как и HTML-страница. Однако внутри файла содержатся только стили, без HTML-разметки. Внешняя таблица стилей подключается к веб-странице с помощью тега `<link>`, расположенного внутри раздела `<head>...</head>`:

```
<head>
<link rel="stylesheet" href="css/style.css">
</head>
```

Такие стили работают для всех страниц сайта. Использованию внешних таблиц стилей и других статичных файлов посвящен следующий раздел.

5.5.2. Использование статичных файлов в приложениях на Django

В веб-приложениях, как правило, присутствуют различные статичные файлы — это изображения, файлы каскадных таблиц стилей (CSS), скрипты javascript и т. п. Рассмотрим, как мы можем использовать подобные файлы в веб-приложениях.

Обратимся к проекту из предыдущего раздела, где рассматривались вопросы создания шаблонов, и добавим в корневую папку проекта новую вложенную папку `static`. А чтобы не смешивать в одной папке различные типы статичных файлов, создадим для каждого типа файлов отдельную папку. В частности, создадим в папке `static` папку `images` — для изображений, и папку `css` — для таблиц стилей. Подобным образом можно создавать папки и для других типов файлов. Теперь структура нашего проекта будет выглядеть так, как показано на рис. 5.25.

Создадим в папке для таблиц стилей `css` файл `styles.css`. Для этого в окне программной оболочки PyCharm щелкните правой кнопкой мыши на папке `css` и из появившегося меню выполните команду **New | File** (рис. 5.26).

После нажатия на клавишу `<Enter>` откроется новое окно, в котором нужно набрать имя создаваемого файла: `styles.css` (рис. 5.27).

Внесите в файл `styles.css` следующий программный код (листинг 5.19), как показано на рис. 5.28.

Листинг 5.19

```
* static/css/styles.css */
body h1 {color: red;}
body h2 {color: green;}
```

Теперь используем этот файл в шаблоне. Для этого в начале файла шаблона необходимо определить инструкцию (для Django 3):

```
{% load static %}
```

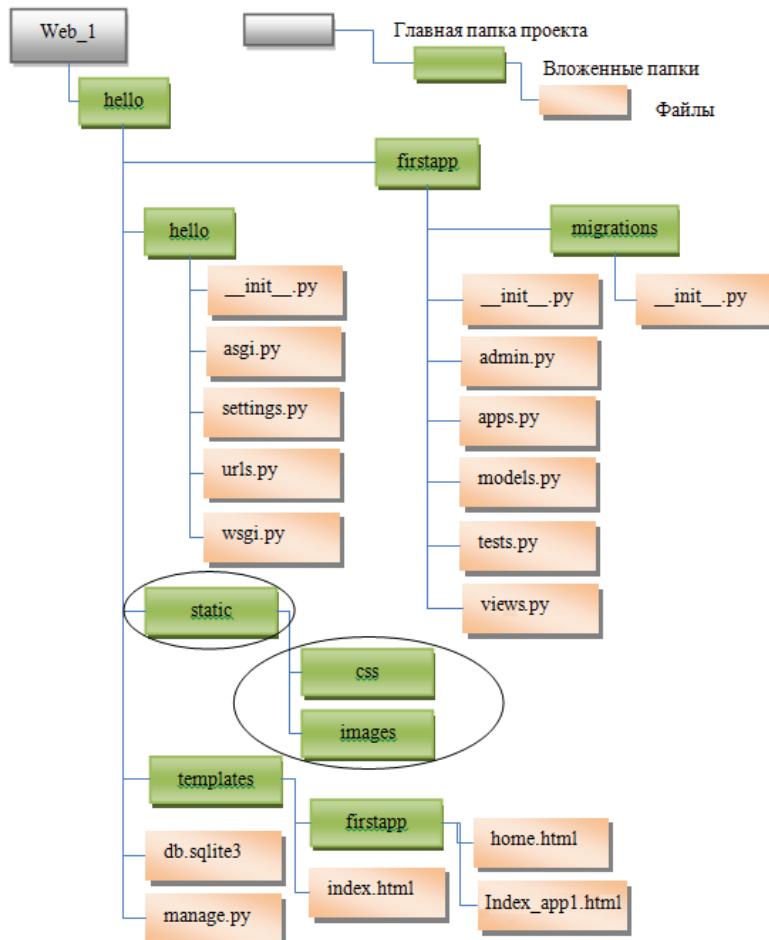


Рис. 5.25. Структура проекта `hello` после добавления в него папок для размещения статичных файлов

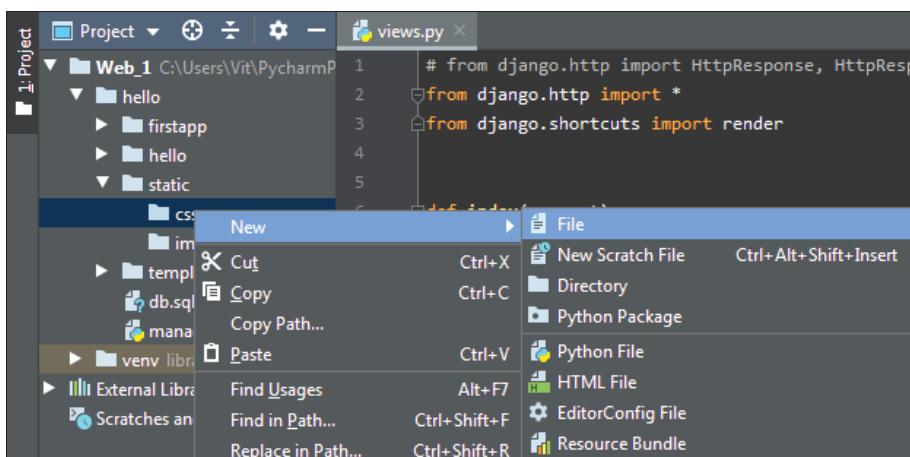


Рис. 5.26. Вход в режим создания нового файла в папке для таблиц стилей css

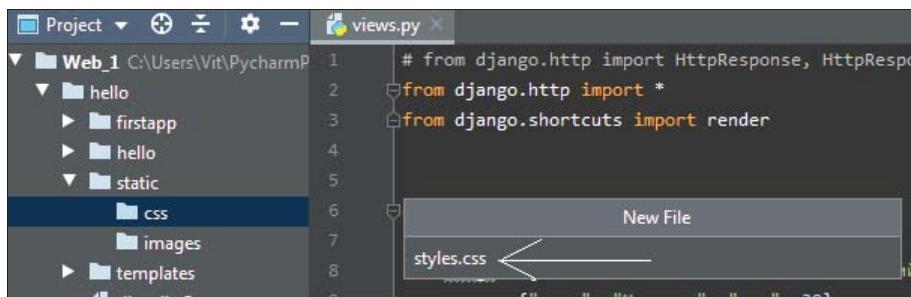


Рис. 5.27. Задание имени новому файлу в папке для таблиц стилей css

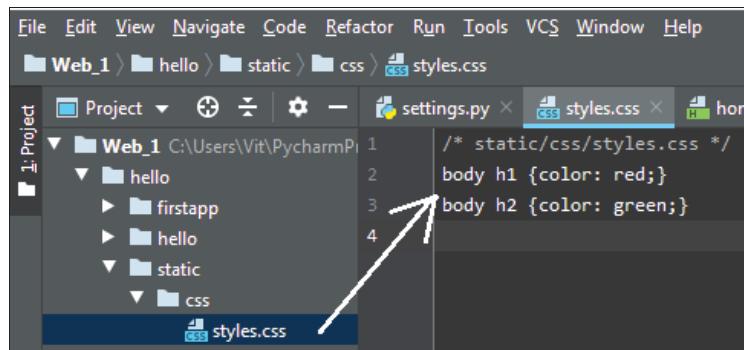


Рис. 5.28. Задание атрибутов в файле таблицы стилей styles.css

ПРИМЕЧАНИЕ

Если у вас Django 2 и ниже, инструкция должна быть следующей:

```
{% load staticfiles %}
```

Для определения пути к статическим файлам используются выражения такого типа:

```
{% static "путь к файлу внутри папки static" %}
```

Так что возьмем шаблон `home.html` из папки `templates\firstapp` и изменим его следующим образом (листинг 5.20), как показано на рис. 5.29.

Листинг 5.20

```

<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
    <link href="{% static 'css/styles.css' %}" rel="stylesheet">
</head>
<body>
    <h1>Домашняя страница Django!</h1>

```

```

<h2>templates/firstapp/home.html</h2>
</body>
</html>

```

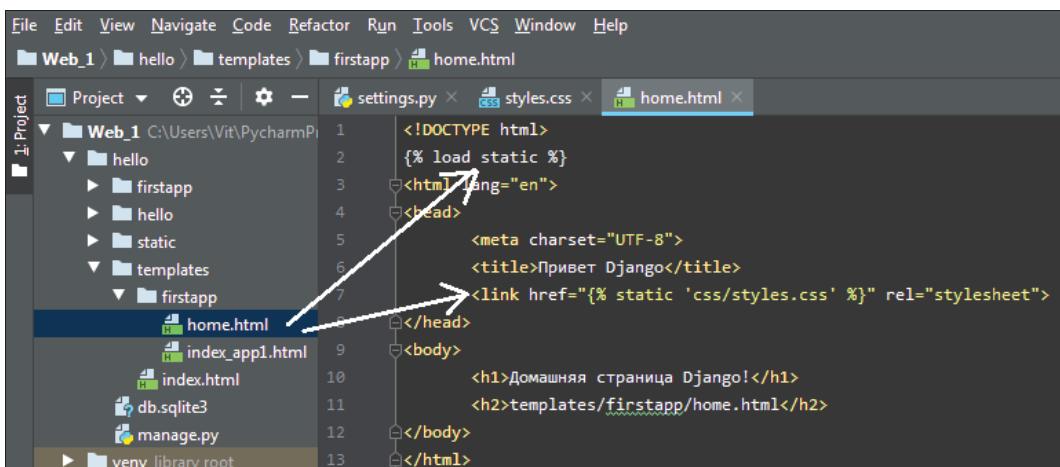


Рис. 5.29. Место инструкций `load static` и `link` в файле шаблона `home.html`

Чтобы файлы из папки static могли использоваться в приложениях, надо указать путь к этой папке в файле `settings.py`, добавив в конец файла следующий код (листинг 5.21), как показано на рис. 5.30.

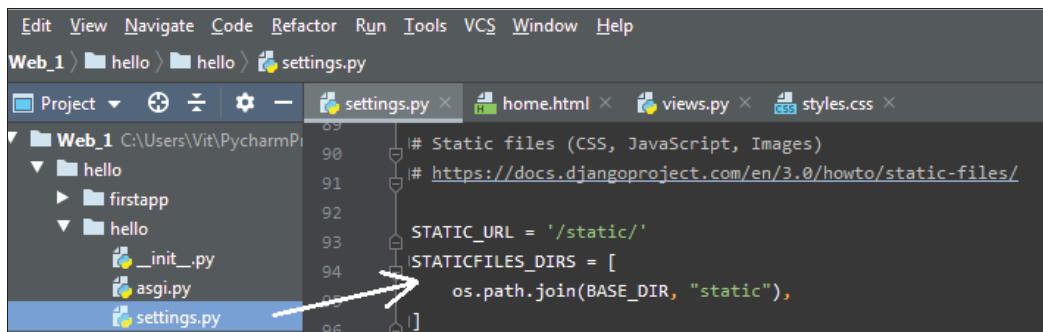


Рис. 5.30. Указание пути к папке static

Листинг 5.21

```

STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
]

```

После этих изменений обратимся к странице `home.html` и получим следующий результат (рис. 5.31).

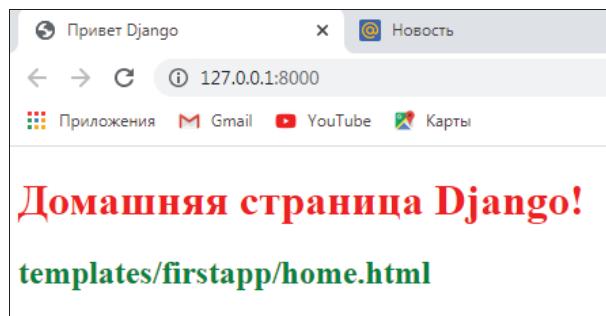


Рис. 5.31. Отображение текста на странице home.html в соответствии со стилями, взятыми из файла styles.css

Как можно здесь видеть (при выводе цветного изображения на экран компьютера), в соответствии со стилями, указанными в файле styles.css, текст тега `h1` выделен красным цветом, а текст тега `h2` — зеленым. При этом в самом HTML-файле стили для выделения текста каким-либо цветом не указаны, а лишь сделаны ссылки на файл со стилями. Такой подход очень удобен тем, что можно оперативно менять и настраивать стили на десятках страницах сайта, внося изменения в код всего одного файла.

Изображения тоже являются статическими файлами. Для хранения изображений мы ранее создали папку `images`. Разместим в этой папке файл с любым изображением и дадим ему имя `image1.jpg`. Теперь выведем его на нашей странице `home.html`, для чего модифицируем код страницы следующим образом (листинг 5.22).

Листинг 5.22

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
    <link href="{% static 'css/styles.css' %}" rel="stylesheet">
</head>
<body>
    <h1>Домашняя страница Django!</h1>
    <h2>templates/firstapp/home.html</h2>
    
</body>
</html>
```

Здесь мы добавили всего одну строку, которая обеспечивает вывод изображения:

```

```

Снова обратимся к странице `home.html` и получим следующий результат (рис. 5.32).

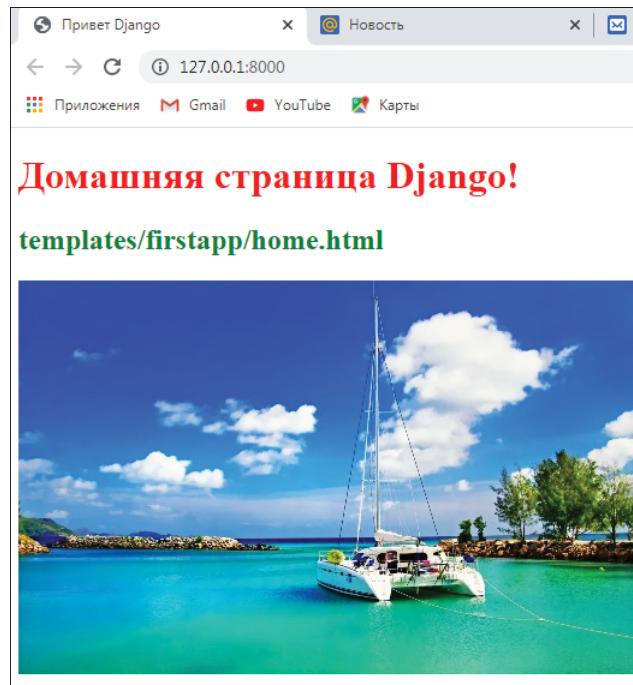


Рис. 5.32. Отображение на странице home.html текста в соответствии со стилями, взятыми из файла styles.css, и вывод рисунка из файла image1.jpg

Как можно видеть, изображение из папки static\images\image1.jpg успешно выведено на HTML-странице. Поскольку для изображений в файле styles.css не было указано стиля вывода, то оно отображается на странице в том виде, в каком было сохранено в файле image1.jpg. Однако для выводимых на HTML-страницах рисунков в файле styles.css тоже можно указывать стиль отображения. Внесем в файл styles.css следующие изменения (листинг 5.23).

Листинг 5.23

```
/* static/css/styles.css */
body h1 {color: red;}
body h2 {color: green;}
img{width:250px;}
```

Здесь мы указали, что ширина изображения должна быть 250 пикселов. После такого изменения страница home.html будет выглядеть, как показано на рис. 5.33.

Как можно видеть, то же изображение из файла image1.jpg на HTML-странице имеет другой размер, который соответствует стилю, указанному в файле styles.css.

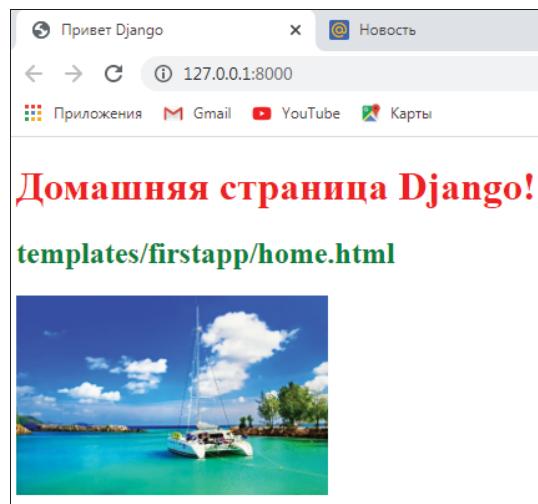


Рис. 5.33. Отображение на странице `home.html` текста в соответствии со стилями, взятыми из файла `styles.css`, и вывод рисунка из файла `image1.jpg` с использованием стиля для изображений

5.5.3. Использование класса `TemplateView` для вызова шаблонов HTML-страниц

В примерах предыдущих разделов, когда приходил запрос от браузера пользователя, система маршрутизации выбирала нужное представление (view), и уже оно вызывало шаблон для генерации ответа. Кроме того, при необходимости, представление могло обратиться к БД и вставить в шаблон некоторые данные из нее. Однако если нужно просто возвратить пользователю содержимое шаблона, то для этого необязательно определять функцию в представлении и обращаться к ней. Можно воспользоваться встроенным классом `TemplateView` и вызвать нужный шаблон, минуя обращение к представлению.

На рис. 5.34 представлен вариант приложения, в котором шаблон главной страницы сайта `home.html` вызывается функцией из представления `view`, а шаблоны страниц `about` и `contact` вызываются с использованием класса `TemplateView`. Проверим, как это работает на простых примерах, для чего определим несколько простейших шаблонов в папке `hello\templates\firstapp\`. Пусть это будет файл-шаблон `about.html` со следующим кодом (листинг 5.24).

Листинг 5.24

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
```

```
<body>
    <h1>Сведения о компании</h1>
    <h2>"Интеллектуальные программные системы"</h2>
    <h3>templates\firstapp\about.html</h3>
</body>
</html>
```

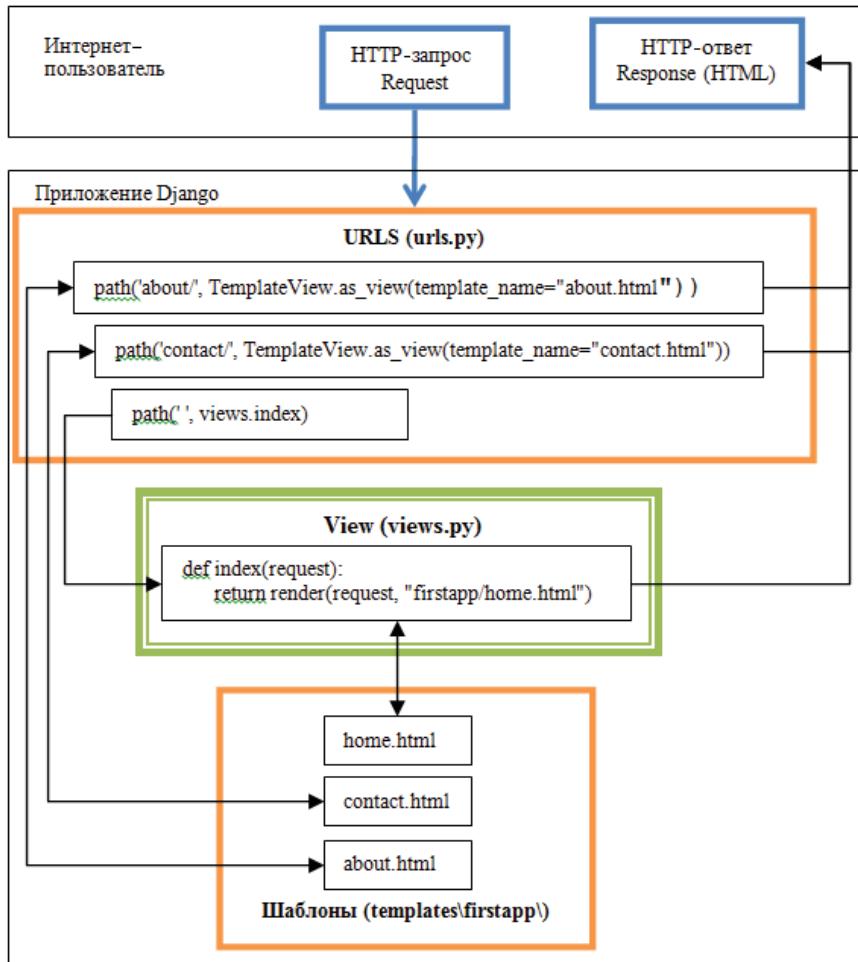


Рис. 5.34. Обращения к шаблонам HTML-страниц с использованием представления (view) и класса TemplateView

В этой же папке создадим файл-шаблон `contact.html` со следующим кодом (листинг 5.25).

Листинг 5.25

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <h1>Контактные данные компании</h1>
    <h2>"Интеллектуальные программные системы"</h2>
    <h3>Адрес: г. Москва, ул. Короленко, д. 24</h3>
    <h4>templates/firstapp/contact.html</h4>
</body>
</html>
```

Осталось внести в файл urls.py следующие изменения (листинг 5.26).

Листинг 5.26

```
from django.views.generic import TemplateView

urlpatterns = [
    path('', views.index),
    path('about/',TemplateView.as_view(template_name="firstapp/about.html")),
    path('contact/',TemplateView.as_view(template_name="firstapp/contact.html")),
]
```

В рассматриваемом программном коде сначала для вызова главной страницы сайта делается обращение к функции `index()` в представлении `view`, и уже функция `index()` вызывает главную страницу приложения `firstapp/home.html` (см. листинг 5.6):

```
def index(request):
    return render(request, "firstapp/home.html")
```

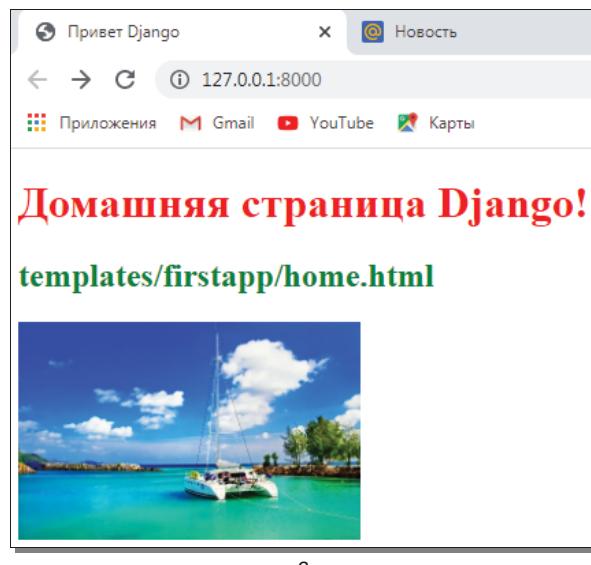
Затем в двух строках программного кода, приведенных в листинге 5.26, для вызова страниц `firstapp\about.html` и `firstapp\contact.html` используется класс `TemplateView`:

```
path('about/',TemplateView.as_view(template_name="firstapp/about.html")),
path('contact/',TemplateView.as_view(template_name="firstapp/contact.html")),
```

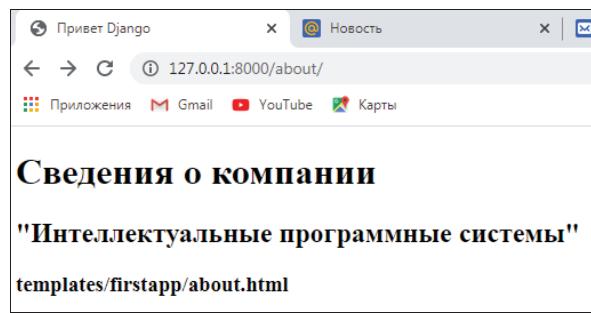
По своей сути класс `TemplateView` предоставляет функциональность представления. С помощью метода `as_view()` через параметр `template_name` задается путь к шаблону, который и будет использоваться в качестве ответа.

Запустим локальный веб-сервер и посмотрим содержимое всех трех страниц. Если все было сделано правильно, то мы получим следующий результат (рис. 5.35).

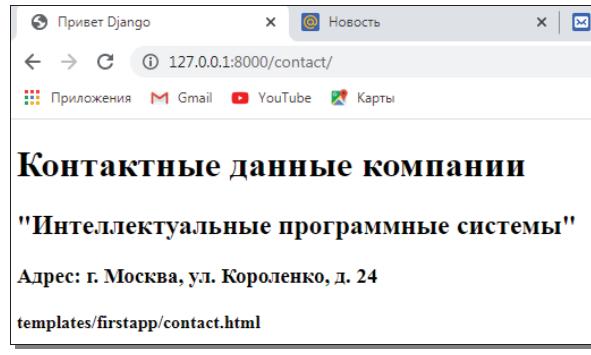
С помощью параметра `extra_context` в метод `as_view` можно передать данные для их отображения в шаблоне. Данные должны представлять собой словарь. В качестве примера передадим в шаблон `contact.html` виды работ, которые выполняет компания. Для этого используем переменную `work` и следующим образом изменим код файла `urls.py` (листинг 5.27).



a



б



в

Рис. 5.35. Примеры обращения к шаблонам HTML-страниц с использованием представления view (а) и класса TemplateView (б и в)

Листинг 5.27

```
urlpatterns = [
    path('', views.index),
    path('about/', TemplateView.as_view(template_name=
        "firstapp/about.html")),
    path('contact/', TemplateView.as_view(template_name=
        "firstapp/contact.html",
        extra_context={"work":
            "Разработка программных продуктов"})),
]
```

Здесь в шаблон contact.html передается объект work, который представляет строку со следующей информацией: "Разработка программных продуктов". И теперь мы можем использовать этот объект в шаблоне contact.html (листинг 5.28).

Листинг 5.28

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <h1>Контактные данные компании</h1>
    <h2>"Интеллектуальные программные системы"</h2>
    <h3>{{ work }}</h3>
    <h3>Адрес: г. Москва, ул. Короленко, д. 24</h3>
    <h4>templates/firstapp/contact.html</h4>
</body>
</html>
```

В итоге получим следующий результат (рис. 5.36).

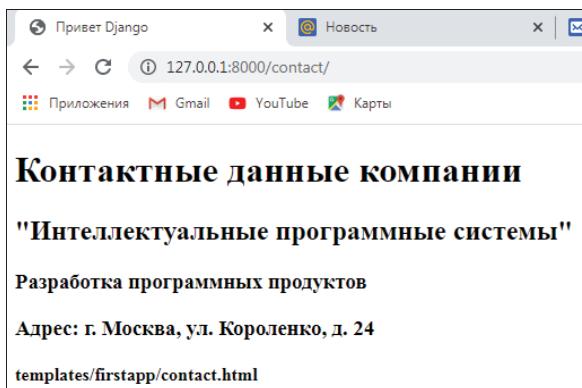


Рис. 5.36. Пример использования объекта work в шаблоне contact.html

5.5.4. Конфигурация шаблонов HTML-страниц

За конфигурацию шаблонов проекта отвечает переменная `TEMPLATES`, расположенная в файле `settings.py`:

```
TEMPLATE_DIR = os.path.join(BASE_DIR, "templates")

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR, ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Здесь используются следующие переменные:

- `BACKEND` — указывает, что надо использовать шаблоны Django;
- `DIRS` — указывает на каталоги (папки) в проекте, которые будут содержать шаблоны;
- `APP_DIRS` — при значении `True` указывает, что поиск шаблонов будет производиться не только в каталогах (папках), указанных в параметре `DIRS`, но и в их подкаталогах (подпапках). Если такое поведение недопустимо, то можно установить значение `False`;
- `OPTIONS` — указывает, какие обработчики (процессоры) будут использоваться при обработке шаблонов.

Как правило, шаблоны помещаются в общую папку в проекте либо в ее подпапки. Например, вы можете определить в проекте общую папку `templates`. Однако если в проекте имеется несколько приложений, которые должны использовать какие-то свои шаблоны, то, чтобы избежать проблем с именованием, можно создать для каждого приложения свою подпапку, в которую помещаются шаблоны для конкретного приложения. Например, в проекте `hello` у нас создано приложение `firstapp`. В этом случае в папке `templates` можно создать подпапку `firstapp` и уже в ней хранить все шаблоны, которые относятся к приложению `firstapp` (рис. 5.37).

А в файле `setting.py` мы можем определить путь к шаблонам следующим образом (листинг 5.29).

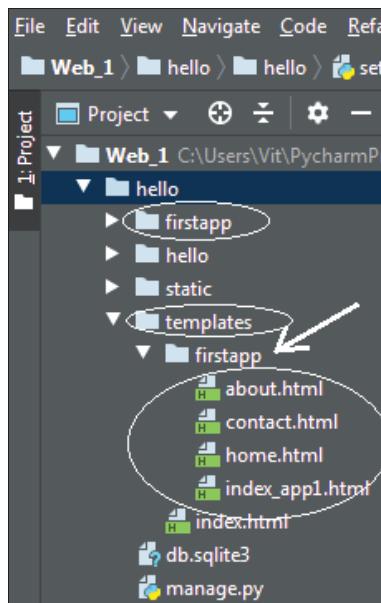


Рис. 5.37. В папке templates создана подпапка для шаблонов приложения firstapp

Листинг 5.29

```
TEMPLATE_DIR = os.path.join(BASE_DIR, "templates")

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR, ],
```

В этом случае берется определенная в самом начале файла `settings.py` переменная `BASE_DIR` (листинг 5.30),

Листинг 5.30

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

которая представляет путь к проекту, и к этому пути добавляется папка с шаблонами — `templates`.

Есть еще один способ указания пути к расположению шаблонов, который предполагает наличие в каждом приложении своей папки для шаблонов (листинг 5.31).

Листинг 5.31

```
TEMPLATE_DIR = os.path.join(os.path.dirname(
    (os.path.abspath(__file__))), "templates")
```

```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [TEMPLATE_DIR, ],
    . . . . .
}
```

5.5.5. Расширение шаблонов HTML-страниц на основе базового шаблона

Хорошим тоном программирования считается формирование единообразного стиля сайта, когда веб-страницы имеют одни и те же структурные элементы: меню, шапку сайта (`header`), подвал (`footer`), боковую панель (`sidebar`) и т. д.

Конечно, можно сконфигурировать каждый шаблон по отдельности. Однако если возникнет необходимость изменить какой-то блок — например, добавить в общее меню еще один пункт, то придется менять все шаблоны, которых может быть довольно много. И в этом случае более рационально многократно использовать один базовый шаблон, который определяет все основные блоки страниц сайта. Попросту говоря, нужно создать некий базовый шаблон. Тогда все остальные шаблоны будут иметь одинаковую базовую структуру и одни и те же блоки, при этом для отдельных блоков можно формировать различное содержимое.

В качестве примера создадим базовый шаблон, который назовем `base.html` (листинг 5.32).

Листинг 5.32

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}Заголовок{% endblock title %}</title>
</head>
<body>
    <h1>{% block header %}{% endblock header %}</h1>
    <div>{% block content %}{% endblock content %}</div>
    <div>Подвал страницы</div>
</body>
</html>
```

Здесь с помощью элементов:

```
{% block название_блока %}{% endblock название_блока %}
```

определяются отдельные блоки шаблонов. При этом для каждого блока определяется открывающий элемент:

```
{% block название_блока %}
```

и закрывающий элемент:

```
{% endblock название_блока %}
```

Например, блок `title` имеет такую структуру:

```
{% block title %}Заголовок{% endblock title %}
```

Когда другие шаблоны будут применять этот базовый шаблон, то они могут определить для блока `title` какое-то свое содержимое. Подобным же образом здесь определены блоки `header` и `content`.

Для каждого блока можно определить содержимое по умолчанию. Так, для блока `title` это строка `Заголовок`. И если другие шаблоны, которые станут использовать этот шаблон, не определят содержимое для блока `title`, то этот блок будет использовать строку `Заголовок`. Впрочем, содержимое по умолчанию для блоков определять не обязательно. Самых же блоков при необходимости можно определить сколько угодно.

Как можно видеть, в базовом шаблоне также определен подвал страницы (`footer`). Поскольку мы хотим сделать подвал общим для всех страниц, то мы его не определяем как отдельный блок, а задаем значение этой части страницы в виде некой константы.

Теперь применим наш базовый шаблон. Создадим в каталоге `templates\firstapp` новый файл-шаблон главной страницы сайта с именем `index.html` и напишем в нем следующий код (листинг 5.33).

Листинг 5.33

```
{% extends "firstapp/base.html" %}  
{% block title %}Index{% endblock title %}  
{% block header %}Главная страница{% endblock header %}  
{% block content%}Связка шаблонов index.html и base.html{% endblock content %}
```

Здесь с помощью выражения `{% extends "firstapp/base.html" %}` мы определили, что эта страница будет формироваться (расширяться) на основе базового шаблона с именем `base.html`, который находится в каталоге шаблонов приложения `firstapp` (по пути: `firstapp\base.html`). Затем задали содержимое для блоков `title`, `header` и `content`. Эти значения и будут переданы в базовый шаблон (`base.html`). Стоит отметить, что не обязательно указывать содержимое для всех блоков базового шаблона.

Изменим следующим образом код функции `def index ()` в представлении `view` (листинг 5.34).

Листинг 5.34

```
def index(request):  
    return render(request, "firstapp/index.html")
```

То есть в этой функции мы в качестве домашней страницы сайта указали файл index.html.

В итоге по связке шаблонов index.html и base.html в браузере будет выдан следующий результат (рис. 5.38).

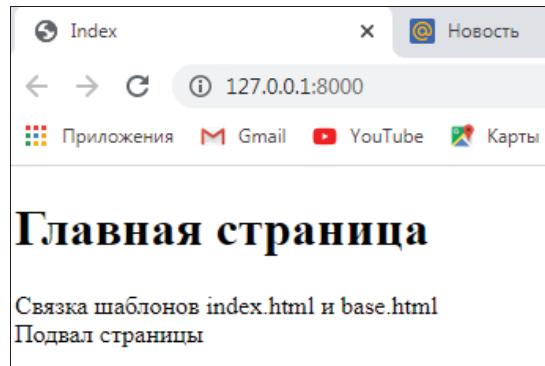


Рис. 5.38. Пример главной страницы сайта index.html, созданной на основе базового шаблона base.html

Теперь используем тот же базовый шаблон base.html для формирования другой страницы сайта — about.html. В предыдущем разделе в эту страницу был внесен следующий программный код (см. листинг 5.24):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <h1>Сведения о компании</h1>
    <h2>"Интеллектуальные программные системы"</h2>
    <h3>templates/firstapp/about.html</h3>
</body>
</html>
```

Удалим этот код и заменим его следующим (листинг 5.35).

Листинг 5.35

```
{% extends "firstapp/base.html" %}
{% block title %}about{% endblock title %}
{% block header %}Сведения о компании{% endblock header %}
{% block content %}
<p>Интеллектуальные программные системы</p>
<p>Связка шаблонов about.html и base.html</p>
{% endblock content %}
```

Здесь с помощью выражения `{% extends "firstapp/base.html" %}` мы определили, что эта страница будет формироваться (расширяться) на основе базового шаблона с именем `base.html`, который находится в каталоге шаблонов приложения `firstapp` (по пути: `firstapp\base.html`). Затем задали содержимое для блоков `title`, `header` и `content`. Эти значения и будут переданы в базовый шаблон (`base.html`).

В результате по связке шаблонов `about.html` и `base.html` в браузере будет выдан следующий результат (рис. 5.39).

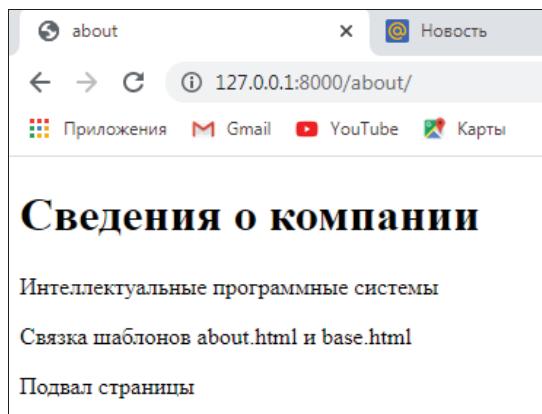


Рис. 5.39. Пример страницы сайта `about.html`, созданной на основе базового шаблона `base.html`

Подводя итого этого раздела, отметим, что на основе одного базового шаблона можно создавать разные страницы сайта, которые имеют одну форму, но будут содержать совершенно разную информацию.

5.6. Использование специальных тегов в шаблонах HTML-страниц

В Django предоставлена возможность использовать в шаблонах ряд специальных тегов, которые упрощают вывод некоторых данных. Рассмотрим некоторые наиболее часто используемые теги.

5.6.1. Тег для вывода текущих даты и времени

Для вывода дат в Django используется следующий тег:

```
{% now "формат_данных" %}
```

Тег `now` позволяет вывести системное время. В качестве параметра тегу `now` передается формат данных, который указывает, как форматировать время и дату. Для форматирования времени и дат используются следующие символы (табл. 5.1).

Все возможные форматы для вывода даты и времени можно посмотреть в оригинальной документации на Django.

Таблица 5.1. Символы для форматирования даты и времени

№ п/п	Символ	Значение даты и времени
1	Y	Год в виде четырех цифр (2021)
2	y	Год в виде последних двух цифр (21)
3	F	Полное название месяца (Июль)
4	M	Сокращенное название месяца — 3 символа (Июл)
5	m	Номер месяца — две цифры (07)
6	N	Аббревиатура месяца в стиле Ассошиэйтед Пресс
7	n	Номер месяца — одна цифра (7)
8	j	День месяца (1–31)
9	l	День недели — текст (среда)
10	h	Часы (0–12) — 9:15
11	H	Часы (0–24) — 21:15
12	i	Минуты (0–59)
13	s	Секунды (0–59)

Изменим следующим образом код шаблона страницы `about.html`, которая была создана в предыдущем разделе (листинг 5.36).

Листинг 5.36

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

{% block title %}about{% endblock title %}

{% block header %}Сведения о компании{% endblock header %}

{% block content %}

<p>Интеллектуальные программные системы</p>
<p>Примеры вывода даты и времени</p>
<p>Год в виде четырех цифр – {% now "Y" %}</p>
<p>Год в виде последних двух цифр – {% now "y" %}</p>
<p>Полное название месяца – {% now "F" %}</p>
<p>Сокращенное название месяца – {% now "M" %}</p>
<p>Номер месяца, две цифры – {% now "m" %}</p>
<p>Аббревиатура месяца (Ассошиэйтед Пресс) – {% now "N" %}</p>
<p>Номер месяца, одна цифра – {% now "n" %}</p>
<p>День месяца – {% now "j" %}</p>
<p> День недели – текст – {% now "l" %}</p>
<p>Часы (0–12) – {% now "h" %}</p>
<p>Часы (0–24) – {% now "H" %}</p>
<p>Минуты (0–59) – {% now "i" %}</p>
<p>Секунды (0–59) – {% now "s" %}</p>
<p>Дата (день/месяц/год) – {% now "j/m/Y" %}</p>
```

```
<p>Время (час:мин:сек) - { % now "H:i:s" %}</p>
{ % endblock content %}
```

Если обратиться к странице about.html после этих изменений, то мы получим следующий результат (рис. 5.40).

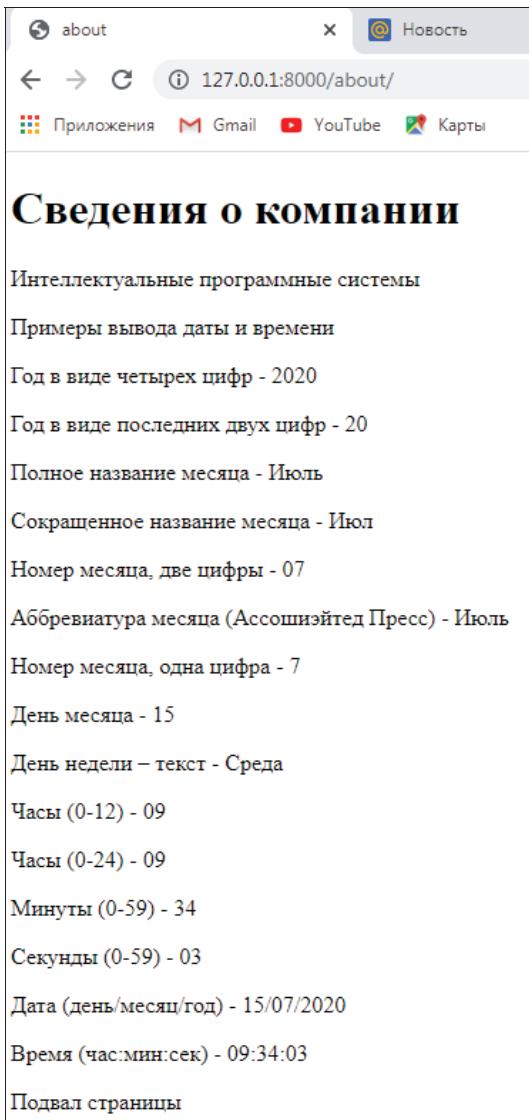


Рис. 5.40. Пример страницы сайта about.html (варианты вывода текущих даты и времени)

5.6.2. Тег для вывода информации по условию

Тег для вывода информации в зависимости от соблюдения какого-либо условия выглядит следующим образом:

```
{% if %} {% endif %}
```

В качестве параметра тегу `if` передается выражение, которое должно возвращать `True` или `False`.

Предположим, что из представления `view` в шаблон передаются некоторые значения — например, возраст клиента (`age`). Изменим следующим образом текст функции `def index()` в файле `view.py` (листинг 5.37).

Листинг 5.37

```
from django.http import *
from django.shortcuts import render

def index(request):
    data = {"age" : 50}
    return render(request, "firstapp/index.html", context=data)
```

В шаблоне страницы `firstapp\index.html` в зависимости от значения переменной `age` мы можем выводить разную информацию. Изменим следующим образом текст в файле `firstapp\index.html` (листинг 5.38).

Листинг 5.38

```
{% extends "firstapp/base.html" %}
{% block title %}Index{% endblock title %}
{% block header %}Главная страница{% endblock header %}
{% block content%}
<p>Анализ возраста клиента</p>
{% if age > 65 %}
    <p>Клиент достиг пенсионного возраста</p>
{% else %}
    <p>Клиент не является пенсионером</p>
{% endif %}
{% endblock content %}
```

На этой странице мы проверяем условие — является ли клиент пенсионером. Если возраст клиента больше 65 лет, то на странице будет выдано сообщение **Клиент достиг пенсионного возраста**, в противном случае будет выведено сообщение **Клиент не является пенсионером**. Проверим работу этого тега при значении возраста: `{"age" : 50}` (рис. 5.41).

Теперь изменим следующим образом текст функции `def index()` в файле `view.py` — укажем возраст клиента 66 лет:

```
data = {"age" : 66}
```

В этом случае на главной странице `firstapp\index.html` мы получим другое сообщение (рис. 5.42).

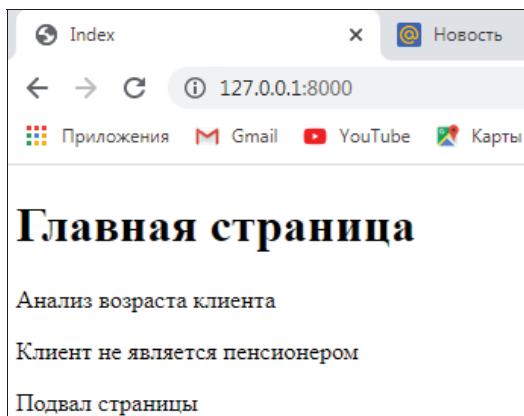


Рис. 5.41. Пример работы тега `if` на странице сайта `index.html` (при возрасте клиента 50 лет)

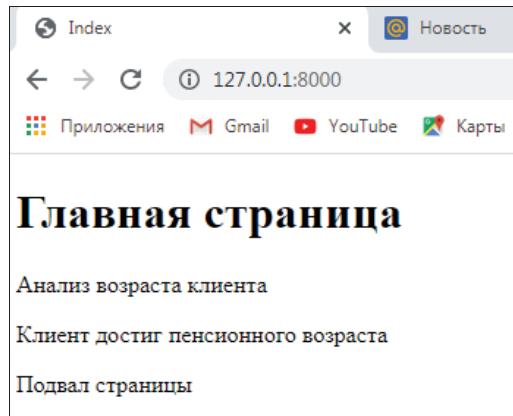


Рис. 5.42. Пример работы тега `if` на странице сайта `index.html` (при возрасте клиента 66 лет)

5.6.3. Тег для вывода информации в цикле

Тег `for` позволяет создавать циклы. Этот тег принимает в качестве параметра некоторую коллекцию и пробегается по этой коллекции, обрабатывая каждый ее элемент. Тег имеет следующую структуру:

```
{% for "Индекс элемента" in "Коллекция элементов" %}  
{%- endfor %}
```

Предположим, что из представления `view` в шаблон передается массив значений — например, категории товаров (`cat`). Изменим следующим образом текст функции `def index()` в файле `view.py` (листинг 5.39).

Листинг 5.39

```
from django.http import *  
from django.shortcuts import render  
  
def index(request):  
    cat = ["Ноутбуки", "Принтеры", "Сканеры", "Диски", "Шнуры"]  
    return render(request, "firstapp/index.html",  
                  context={"cat": cat})
```

Чтобы в шаблоне страницы `firstapp\index.html` в цикле выводить информацию из массива `cat`, изменим следующим образом код в файле `firstapp\index.html` (листинг 5.40).

Листинг 5.40

```
{% extends "firstapp/base.html" %}  
{% block title %}Index{% endblock title %}
```

```
{% block header %}Главная страница{% endblock header %}
{% block content%}
<p>Категории продуктов</p>
{% for i in cat %}
    <li>{{ i }}</li>
    {% endfor %}
{% endblock content %}
```

Результат работы тега `for`, обеспечивающего вывод значений массива в цикле, представлен на рис. 5.43.

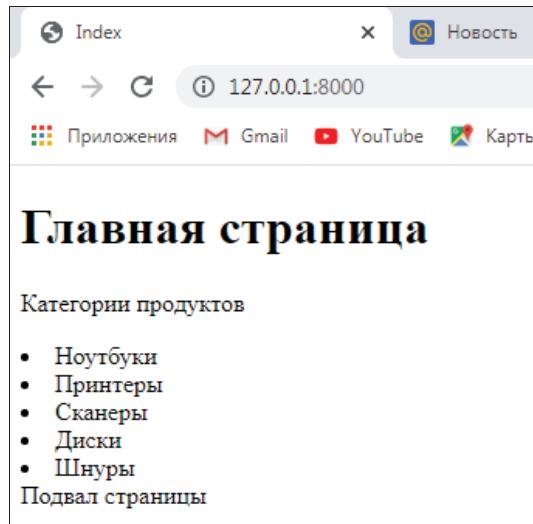


Рис. 5.43. Пример работы тега `for` на странице сайта `index.html`

Вполне возможно, что массив, переданный из представления `view` в шаблон, окажется пустым. На этот случай мы можем использовать дополнительный тег:

```
{% empty %}.
```

Для этого блок `content` на странице `firstapp\index.html` можно изменить следующим образом (листинг 5.41).

Листинг 5.41

```
{% block content%}
<p>Категории продуктов</p>
{% for i in cat %}
    <li>{{ i }}</li>
    {% empty %}
        <li>Категории продуктов отсутствуют</li>
    {% endfor %}
{% endblock content %}
```

Если мы теперь в файле view.py обнулим массив cat, т. е. сделаем его «пустым»:

```
cat = []
```

то тег `for` сработает следующим образом (рис. 5.44).

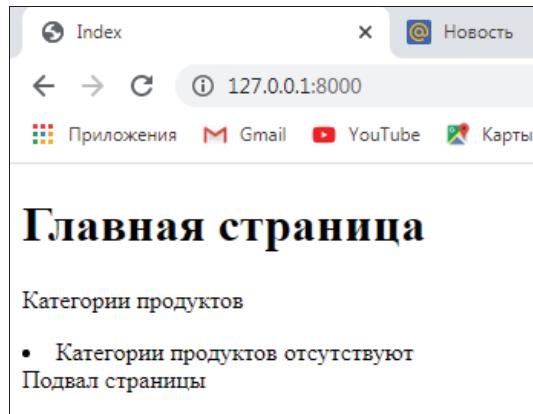


Рис. 5.44. Пример работы тега `for` на странице сайта index.html при «пустом» массиве категорий продуктов

5.6.4. Тег для задания значений переменным

Если требуется определить переменную и использовать ее внутри шаблона, то для этого можно использовать тег: `{% with %}`.

Изменим следующим образом код в файле firstapp\index.html (листинг 5.42).

Листинг 5.42

```
{% extends "firstapp/base.html" %}  
{% block title %}Index{% endblock title %}  
{% block header %}Главная страница{% endblock header %}  
{% block content%}  
<p>Категории продуктов</p>  
{% for i in cat %}  
    <li>{{ i }}</li>  
    {% empty %}  
        <li>Категории продуктов отсутствуют</li>  
    {% endfor %}  
  
{% with name="ЧАЙХАНА" nom=1 %}  
    <div>  
        <p>При магазине работает {{ name }}</p>  
        <p>Номер № {{ nom }}</p>  
    </div>  
    {% endwith %}  
  
{% endblock content %}
```

Здесь внутри шаблона мы определили и вывели значения двух переменных: `name` и `nom`. После этих изменений страница `firstapp\index.html` будет иметь следующий вид (рис. 5.45).

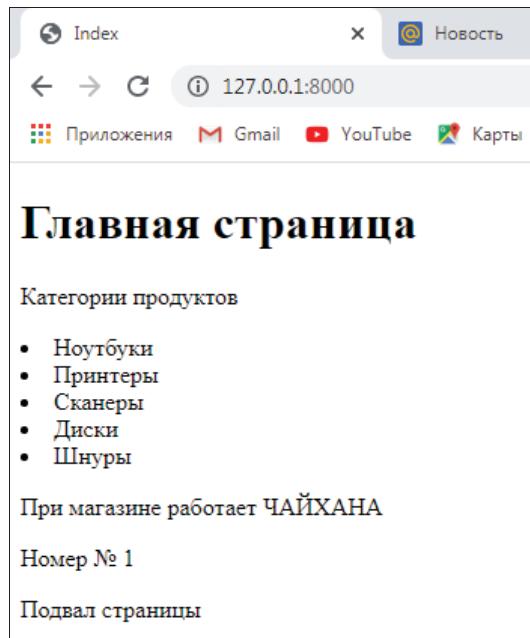
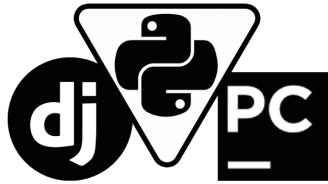


Рис. 5.45. Пример работы тега `with` на странице сайта `index.html`

5.7. Краткие итоги

В этой главе были приведены элементарные теоретические сведения о шаблонах Django и показано, как можно создать шаблоны и передать в шаблоны различные данные из программы на Python. Мы узнали, что через шаблоны можно сформировать и выдать пользователю ответ на его запрос. Однако динамичные веб-сайты должны выполнять и обратную функцию, а именно — принять от пользователя некоторые данные и сохранить их в БД. Для этого в Django используются *формы*. Так что пора перейти к следующей главе, в которой рассматриваются возможности взаимодействия пользователей с удаленной базой данных сайта через формы Django.



ГЛАВА 6

Формы

Форма HTML — это группа из одного или нескольких полей (виджетов) на веб-странице, которая используется для получения информации от пользователей для последующей отправки на сервер и представляет собой таким образом гибкий механизм сбора пользовательских данных. Формы несут целый набор виджетов для ввода различных типов данных: текстовые поля, флагшки, переключатели, установщики дат и пр. Формы являются относительно безопасным способом взаимодействия пользователя и клиента и сервера, поскольку позволяют отправлять данные в POST-запросах, применяя защиту от межсайтовой подделки запроса (Cross Site Request Forgery, CSRF). Из материалов этой главы вы узнаете:

- что такое формы Django и для чего они нужны;
- как применять в формах POST-запросы;
- какие поля можно использовать в формах;
- как можно выполнить настройку формы и ее полей;
- как можно изменить внешний вид формы;
- как осуществляется проверка (валидация) данных в формах;
- как можно добавить стили к полям формы.

6.1. Определение форм

Если вы планируете создавать сайты и приложения, который принимают и сохраняют данные от пользователей, вам необходимо использовать формы. Django предоставляет широкий набор инструментов для этого. Формы в HTML позволяют пользователю вводить текст, выбирать опции, изменять различные объекты, загружать рисунки на страницы и т. п., а потом отправлять эту информацию на сервер.

Django предоставляет различные возможности по работе с формами. Можно определить функциональность форм в одном месте и затем использовать их многократно в разных местах. При этом упрощается проверка корректности данных, связывание форм с моделями данных и многое другое.

Каждая форма определяется в виде отдельного класса, который расширяет класс `forms.Form`. Классы размещаются внутри проекта, где они используются. Нередко они помещаются в отдельный файл, который называется, к примеру, `forms.py`. Однако формы могут размещаться и внутри уже имеющихся в приложении файлов — например, в `views.py` или `models.py`.

Создадим в приложении `firstapp` проекта `hello` новый файл `forms.py` и поместим в него следующий код (листинг 6.1), как показано на рис. 6.1.

Листинг 6.1

```
from django import forms

class UserForm(forms.Form):
    name = forms.CharField()
    age = forms.IntegerField()
```

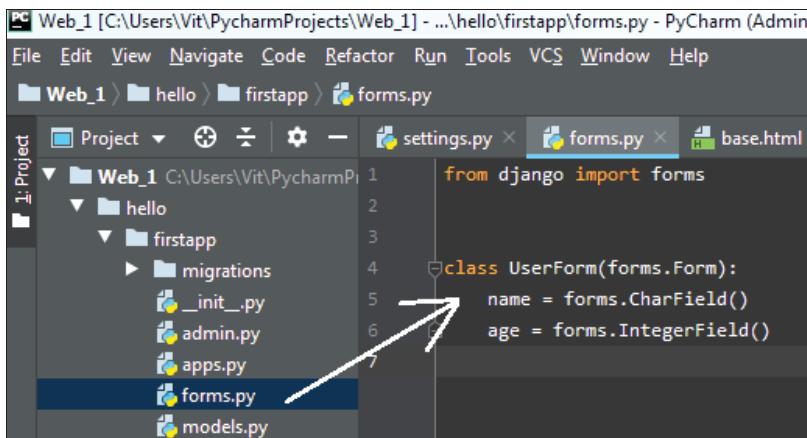


Рис. 6.1. Добавление файла `forms.py` в приложение `firstapp`

Здесь класс формы называется `UserForm`. В нем определены два поля. Поле `name` (имя) имеет тип `forms.CharField` и будет генерировать текстовое поле: `input type="text"`. Поле `age` (возраст) имеет тип `forms.IntegerField` и будет генерировать числовое поле: `input type="number"`. То есть первое поле предназначено для ввода текста, а второе — для ввода чисел.

Далее в файле `views.py` определим следующий код для функции `index()` (листинг 6.2).

Листинг 6.2

```
from django.shortcuts import render
from .forms import UserForm
```

```
def index(request):
    userform = UserForm()
    return render(request, "firsatapp/index.html",
                  {"form": userform})
```

Здесь созданный нами объект формы передается в шаблон index.html в виде переменной `form`.

Теперь изменим шаблон index.html следующим образом (листинг 6.3).

Листинг 6.3

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
    <table>
        {{ form }}
    </table>
</body>
</html>
```

В веб-браузере эта страница будет выглядеть так (рис. 6.2).

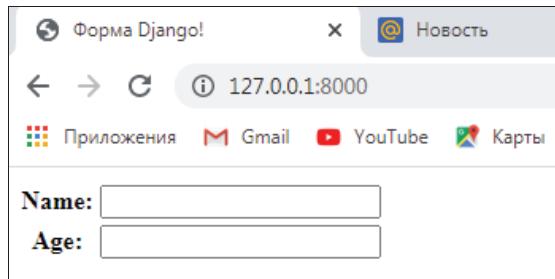


Рис. 6.2. Вид формы forms.py приложения firstapp в веб-браузере

В обрамленные рамками поля пользователь может вводить свои данные. Следует заметить, что в модуле forms.py мы задали только идентификаторы полей `name` и `age` и не задавали метки полей (`label`), которые будут выводиться на экран. В Django эти метки генерируются автоматически. В частности, как можно видеть, для наших двух полей были сгенерированы метки: **Name** и **Age**. По умолчанию в Django в качестве метки берется имя поля, и его первый символ меняется на заглавную букву. С одной стороны, такой прием избавляет программиста от дополнительного задания значения для метки поля, но это не всегда удобно. Часто требуется задавать

более осмысленные и достаточно длинные значения для меток полей и на языке, отличном от английского.

Однако если мы в PyCharm попытаемся задать имя поля кириллицей на русском языке, то получим предупреждение об ошибке (рис. 6.3). Впрочем, несмотря на это предупреждение, программа будет выполняться, поскольку Django допускает использование в именах полей символов, отличных от ASCII (рис. 6.4).

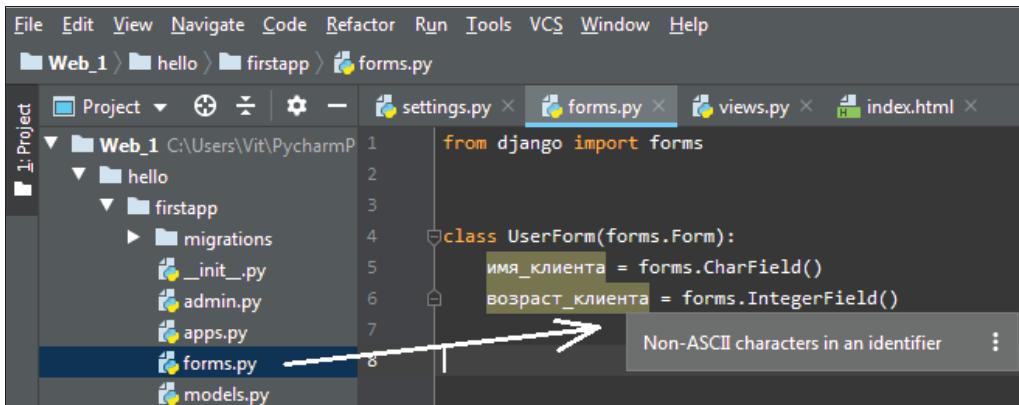


Рис. 6.3. Предупреждение об ошибке при использовании кириллицы при задании имени поля

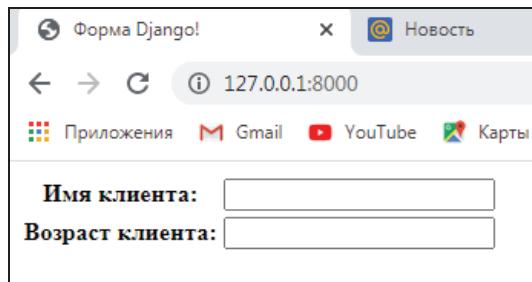


Рис. 6.4. Демонстрация возможности использования кириллицы при задании имени поля

Обратите внимание: если имя поля в Django состоит из нескольких слов, то их нельзя разделять пробелами, а нужно использовать символ нижнего подчеркивания — например, `имя_клиента`, `возраст_клиента` и т. п. И, как можно видеть на рис. 6.4, Django не только меняет в кириллической записи первый символ на заглавную букву, но и еще автоматически заменяет символ нижнего подчеркивания на пробел. Кстати, предупреждение об ошибке в PyCharm можно отключить. Для этого нужно выполнить следующую команду меню: **Settings | Editor | Inspections | Internationalization | Non-ASCII characters** и снять флажок с опции **Non-ASCII characters** (рис. 6.5).

Однако желательно избегать использования кириллицы и других символов, отличных от ASCII, в определении имен полей. Это впоследствии может привести к некорректному отображению информации, когда приложение будет развернуто на внешнем веб-ресурсе.

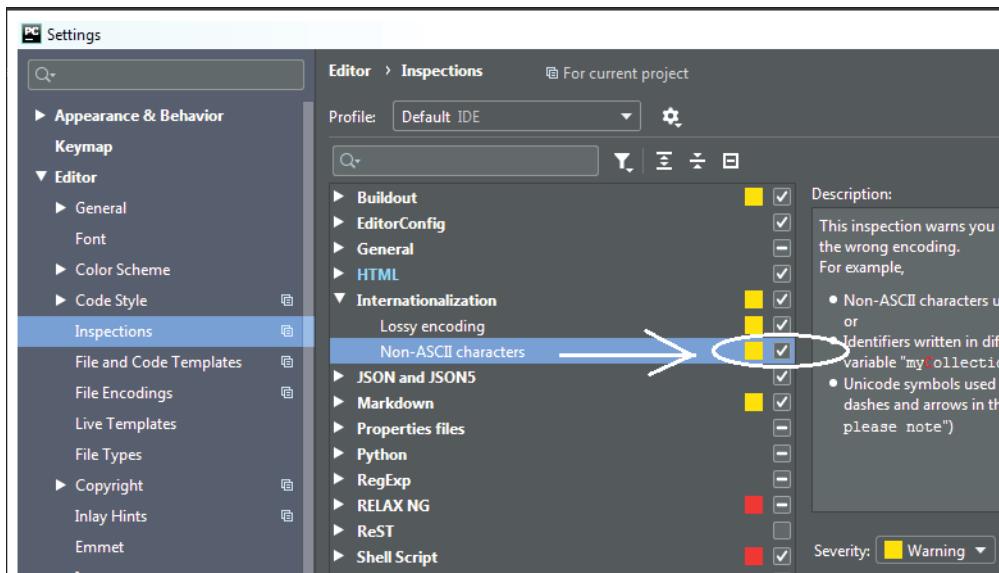


Рис. 6.5. Отключение предупреждения об ошибке при использовании кириллицы в именах полей

В Django имеется другая возможность задания метки для поля — с помощью параметра `label`. Немного изменим содержимое файла `forms.py` и поместим в него следующий код (листинг 6.4), как показано на рис. 6.6. Здесь мы задали имя поля латиницей, а метку поля определили кириллицей. В результате на экран будет выдана страница, показанная на рис. 6.7.

Листинг 6.4

```
from django import forms

class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента")
    age = forms.IntegerField(label="Возраст клиента")
```

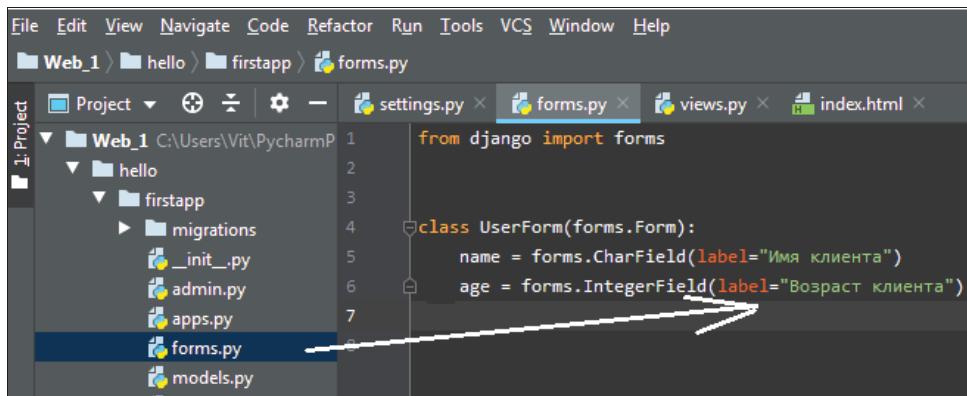


Рис. 6.6. Задание метки для поля формы

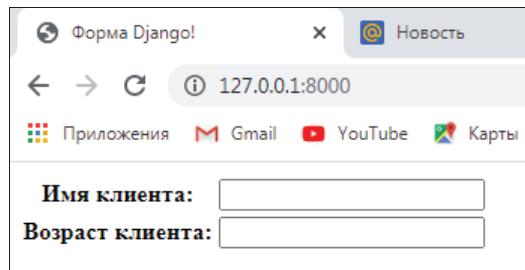


Рис. 6.7. Вид формы forms.py приложения firstapp в веб-браузере при явном определении меток полей

Кроме параметра `label` поля форм Django имеют еще ряд параметров, которые разработчик может определять программным способом. На них мы остановимся более подробно в следующих разделах.

6.2. Использование в формах POST-запросов

В форму, которую мы создали в предыдущем разделе, пользователь может только вводить свои данные. Давайте с помощью Django создадим полнофункциональную форму, в которую можно не только вводить данные, но и отправлять их на сервер. Для этого сначала изменим шаблон `index.html` (листинг 6.5).

Листинг 6.5

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
        <table>
            {{ form }}
        </table>
        <input type="submit" value="Отправить" >
    </form>
</body>
</html>
```

Для создания формы здесь использован стандартный элемент HTML `<form>`. В начале формы помещен встроенный тег Django `{% csrf_token %}`, который позволяет защитить приложение от CSRF-атак, добавляя в форму csrf-токен в виде скрытого

поля. В нижней части формы помещена кнопка для отправки содержимого этой формы на сервер.

Пояснение

CSRF (Cross-Site Request Forgery, также XSRF) — опаснейшая атака, которая приводит к тому, что хакер может выполнить на неподготовленном сайте массу различных действий от имени других зарегистрированных посетителей.

Далее в представлении (в файле `views.py`) определим следующий код для функции `index()` (листинг 6.6).

Листинг 6.6

```
from django.http import *
from .forms import UserForm
from django.shortcuts import render

def index(request):
    if request.method == "POST":
        name = request.POST.get("name") # получить значение поля Имя
        age = request.POST.get("age") # получить значение поля Возраст
        output = "<h2>Пользователь</h2><h3>Имя - {0},
                  Возраст - {1}</h3>".format(name, age)
        return HttpResponse(output)
    else:
        userform = UserForm()
        return render(request, "firstapp/index.html", {"form": userform})
```

Разберемся, какие действия запрограммированы в этом коде. Представление обрабатывает сразу два типа запросов: GET и POST. Для определения типа запроса используется проверка значения `request.method` в структуре `if...else`.

Если запрос представляет тип GET (ветка `else`), то мы просто формируем пользовательскую форму `userform` и отправляем ее для ввода данных в шаблон `index.html`. Таким образом, при первом обращении к приложению мы вначале увидим сформированную нами форму ввода. Введем в нее некоторые данные (рис. 6.8).

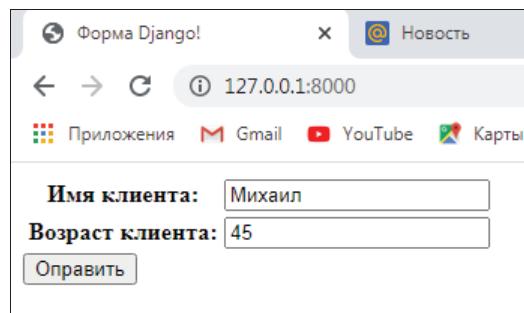


Рис. 6.8. Ввод данных в форму при первой загрузке страницы `index.html`

Если запрос будет иметь типа POST (`request.method == "POST"`), то это данные формы, отправляемые по нажатию кнопки **Отправить**. В этом случае отправляемые из формы данные присваиваются переменным `name` и `age`, а их значения — переменной `output`. После этого значения из `output` отправляются через объект `HttpResponse`. В нашем случае ответ отправляется пользователю на ту же HTML-страницу (рис. 6.9).

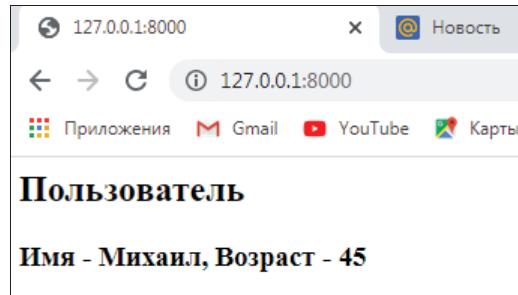


Рис. 6.9. Ответ из формы по нажатию кнопки **Отправить** на странице index.html

Конечно, мы здесь реализовали простейший вариант, когда все действия совершаются в пределах одной страницы. В реально работающих приложениях делается либо переадресация (данные отправляются на другую страницу), или они записываются в базу данных, что мы и сделаем в последующих разделах.

6.3. Использование полей в формах Django

6.3.1. Настройка среды для изучения полей разных типов

В формах Django используются собственные классы для создания полей, через которые пользователь может вводить в приложение различные признаки и данные. Для демонстрации примеров применения различных полей мы воспользуемся файлом-шаблоном для главной страницы `firstapp\index.html`. В этот файл в предыдущем разделе мы внесли следующий код (листинг 6.7), который здесь оставим без изменения.

Листинг 6.7

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
```

```
<table>
    {{ form }}
</table>
<input type="submit" value="Отправить" >
</form>
</body>
</html>
```

В наших примерах мы задействуем файл представления `views.py`, в котором сформируем следующий код для функции `index()`(листинг 6.8).

Листинг 6.8

```
from .forms import UserForm
from django.shortcuts import render

def index(request):
    userform = UserForm()
    return render(request, "firstapp/index.html", {"form": userform})
```

В этой функции мы активируем нашу форму `userform` и вызываем главную страницу приложения `firstapp\index.html`, в которой будет отображаться эта форма.

И наконец, все изучаемые нами поля мы будем размещать в коде файла `forms.py`. Из примера предыдущего раздела в этом файле должны сохраниться следующие строки программного кода (листинг 6.9).

Листинг 6.9

```
from django import forms

class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента")
    age = forms.IntegerField(label="Возраст клиента")
```

Итак, у нас все готово, и можно приступить к рассмотрению примеров использования различных типов полей для ввода данных.

6.3.2. Типы полей в формах Django и их общие параметры

Для начала мы познакомимся со списком наиболее употребляемых полей, которые используются в формах Django (табл. 6.1), и рядом свойств, которые являются общими для всех типов полей. А после этого сделем более детальный разбор примеров использования полей.

Таблица 6.1. Список основных полей, используемых в формах Django

№ п/п	Тип поля	Описание типа поля
1	forms.BooleanField	Создает поле checkbox
2	forms.NullBooleanField	Создает поле выбора (не выбрано, да, нет)
3	forms.CharField	Предназначено для ввода текста
4	forms.EmailField	Предназначено для ввода адреса электронной почты
5	forms.GenericIPAddressField	Предназначено для ввода IP-адреса
6	forms.RegexField (regex="reg_выр")	Предназначено для ввода текста, который должен соответствовать определенному регулярному выражению
7	forms.SlugField()	Предназначено для ввода текста, который условно называется «slug» — это последовательность символов в нижнем регистре, чисел, дефисов и знаков подчеркивания
8	forms.URLField()	Предназначено для ввода ссылок
9	forms.UUIDField()	Предназначено для ввода UUID (универсального уникального идентификатора)
10	forms.ComboField (fields=[field1, field2,...])	Аналогично обычному текстовому полю, однако требует, чтобы вводимый текст соответствовал требованиям тех полей, которые передаются через параметр fields
11	forms.MultiValueField (fields=[field1, field2,...])	Предназначено для создания сложных компоновок, состоящих из нескольких полей
12	forms.FilePathField (path="каталог файлов")	Создает список select, который содержит все папки и файлы в определенном каталоге
13	forms.FileField()	Предназначено для выбора файла
14	forms.ImageField()	Предназначено для выбора файла, но при этом добавляет ряд дополнительных возможностей
15	forms.DateField()	Предназначено для установки даты. В создаваемое поле вводится текст, который может быть сконвертирован в дату — например: 2021-12-25 или 11/25/21
16	forms.TimeField()	Предназначено для ввода времени — например: 14:30:59 или 14:30
17	forms.DateTimeField()	Предназначено для ввода даты и времени, например, 2021-12-25 14:30:59 или 11/25/21 14:30
18	forms.DurationField()	Предназначено для ввода временного промежутка. Вводимый текст должен соответствовать формату "DD HH:MM:SS", например, 2 1:10:20 (2 дня 1 час 10 минут 20 секунд)
19	forms.SplitDateTimeField()	Создает два текстовых поля для ввода соответственно даты и времени

Таблица 6.1 (окончание)

№ п/п	Тип поля	Описание типа поля
20	forms.IntegerField()	Предназначено для ввода целых чисел
21	forms.DecimalField()	Предназначено для ввода чисел с дробной частью
22	forms.FloatField()	Предназначено для ввода чисел с плавающей точкой
23	forms.ChoiceField (choises=кортеж_кортежей)	Генерирует список select, каждый из его элементов формируется на основе отдельного кортежа. Например, =(1, "English"), (2, "German"), (3, "French")
24	forms.TypeChoiceField (choises=кортеж_кортежей, coerce=функция_преобразования, empty_value=None)	Также генерирует список select на основе кортежа. Однако дополнительно принимает функцию преобразования, которая преобразует каждый элемент. И также принимает параметр empty_value, который указывает на значение по умолчанию
25	forms.MultipleChoiceField (choises=кортеж_кортежей)	Также генерирует список select на основе кортежа, как и forms.ChoiceField, добавляя к создаваемому полю атрибут multiple="multiple". То есть список поддерживает множественный выбор
26	forms.TypedMultipleChoiceField (choises=кортеж_кортежей, coerce=функция_преобразования, empty_value=None)	Аналог forms.TypeChoiceField для списка с множественным выбором

Каждое поле, когда оно выводится на HTML-страницу, имеет типичный для него внешний вид. Например, поле для ввода текста имеет обрамление в виде рамки. За внешний вид полей отвечает *виджет* — представление элемента ввода на HTML-странице. Виджеты не следуют путать с полями формы. Поля формы имеют дело с логикой проверки ввода и используются непосредственно в шаблонах. Виджеты же имеют дело с отображением элементов ввода HTML-формы на веб-странице и извлечением необработанных отправленных данных. Однако виджеты должны быть назначены полям формы.

Всякий раз, когда вы указываете поле в форме, Django задействует виджет по умолчанию, соответствующий тому типу данных, которые должны отображаться. Ранее рассмотренные поля при генерации разметки использовали определенные виджеты из пакета `forms.widgets`. Например, класс `CharField` — виджет `forms.widgets.TextInput`, а класс `ChoiceField` — виджет `forms.widgets.Select`. Однако есть ряд виджетов, которые по умолчанию не используются полями форм, но тем не менее мы можем их применять:

- `PasswordInput` — генерирует поле для ввода пароля: `<input type="password" >`;
- `HiddenInput` — генерирует скрытое поле: `<input type="hidden" >`;
- `MultipleHiddenInput` — генерирует набор скрытых полей;

- TextArea — генерирует многострочное текстовое поле: <textarea></textarea>;
- RadioSelect — генерирует список переключателей (радиокнопок): <input type="radio" >;
- CheckboxSelectMultiple — генерирует список флажков: <input type="checkbox" >;
- TimeInput — генерирует поле для ввода времени (например, 12:41 или 12:41:32);
- SelectDateWidget — генерирует три поля select для выбора дня, месяца и года;
- SplitHiddenDateTimeWidget — использует скрытое поле для хранения даты и времени;
- FileInput — генерирует поле для выбора файла.

Каждое поле, которое размещено на форме, ориентируется на собственную логику проверки введенных данных, а также принимает несколько других аргументов, которые можно назначить при инициализации поля.

Каждый экземпляр класса Field имеет метод clean(), который принимает единственный аргумент и вызывает исключение django.forms.ValidationError в случае ошибки или возвращает чистое значение:

```
Field.clean(value)
```

Вот пример использования этого метода:

```
from django import forms
f = forms.EmailField()
f.clean('info@example.com')
f.clean('Ошибка в написании email адреса')
```

Некоторые классы Field принимают дополнительные аргументы. Приведенные далее аргументы принимаются всеми полями:

- required — указывает на необходимость обязательного заполнения поля (по умолчанию: required=True). Для того чтобы сделать поле необязательным для заполнения, нужно указать: required=False;
- label — позволяет определить видимую пользователем метку для поля (например, если имя поля name и на экране нужно показать **Ваше имя**, то это можно сделать следующим образом: name = forms.CharField(label='Ваше имя');
- label_suffix — позволяет переопределить атрибут формы label_suffix для каждого поля. Например:

```
class ContactForm(forms.Form):
    age = forms.IntegerField()
    nationality = forms.CharField()
    captcha_answer = forms.IntegerField(label='2 + 2',
                                         label_suffix=' =')
f = ContactForm(label_suffix='?')
```

- initial — позволяет определять начальное значение для поля при его отображении на незаполненной форме. Например:

```
from django import forms
class CommentForm(forms.Form):
    name = forms.CharField(initial='Ваше имя')
    url = forms.URLField(initial='http://')
    comment = forms.CharField()
f = CommentForm(auto_id=False)
```

- `widget` — позволяет указать класс `Widget`, который следует использовать при отображении поля вместо того, который задан по умолчанию;
- `help_text` — позволяет указать описание данных, которые нужно внести в поле. Если вы укажете `help_text`, он будет показан около поля при отображении формы с помощью вспомогательных методов. Например:

```
from django import forms
class HelpTextContactForm(forms.Form):
    subject = forms.CharField(max_length=100,
                              help_text='Не более 100 символов')
    message = forms.CharField()
    sender = forms.EmailField(help_text='email адрес')
    cc_myself = forms.BooleanField(required=False)
f = HelpTextContactForm(auto_id=False)
```

Здесь представлен пример формы, в которой `help_text` определен у двух полей. Мы используем `auto_id=False` для упрощения вывода;

- `error_messages` — позволяет изменить стандартные сообщения об ошибках, которые выдает поле. Создайте словарь с ключами тех сообщений, которые вы желаете изменить. Если не задать значение этому аргументу:

```
from django import forms
generic = forms.CharField()
generic.clean('')
```

то будет выдано стандартное сообщение об ошибке: **This field is required.**

Если этот аргумент задать явно, например:

```
name = forms.CharField(error_messages={'required':
                                         'Ошибка, не введено имя'})
name.clean('')
```

то будет выдано заданное сообщение об ошибке: **Ошибка, не введено имя;**

- `validators` — позволяет указать список функций, осуществляющих проверку поля;
- `localize` — включает локализацию для данных формы как на входе, так и на выходе;
- `disabled` — этот аргумент принимает булево значение. При значении `True` — выключает поля формы, используя HTML-атрибут `disabled`. Даже если пользователь отправит данные этого поля, они будут проигнорированы и использовано значение из начальных данных, которые были переданы в форму.

6.3.3. Поле *BooleanField* для выбора решения: да\нет

Метод `forms.BooleanField` создает поле `<input type="checkbox">`. Возвращает значение `Boolean`: `True` — если флажок отмечен и `False` — если флажок не отмечен.

При инициализации этого поля по умолчанию устанавливается виджет `CheckboxInput` и возвращаемое значение `False`, при этом флажок в `checkbox` отсутствует. Пример создания такого поля приведен в листинге 6.10.

Листинг 6.10

```
from django import forms
class UserForm(forms.Form):
    basket = forms.BooleanField(label="Положить товар в корзину")
```

Если мы внесем этот код в модуль `forms.py` и запустим наше приложение, то получим следующее отображение этого поля на форме (рис. 6.10).

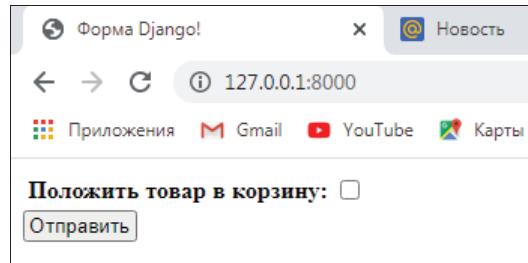


Рис. 6.10. Поле `BooleanField` на странице `index.html`

Если клиент не захочет положить товар в корзину, оставив это поле пустым, и нажмет кнопку **Отправить**, то в ответ получит следующее сообщение (рис. 6.11).

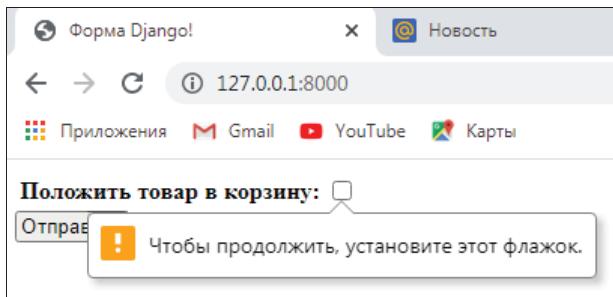


Рис. 6.11. Ответ из формы после нажатия кнопки **Отправить** на странице `index.html`

Похоже, логика работы приложения нарушена: клиент не хочет приобретать некий товар, но приложение вынуждает его установить флажок в поле выбора. Это происходит потому, что для всех полей по умолчанию установлен признак обязательности внесения значений в поле (`required=True`), а при инициализации поля оно не

заполнено (в нем нет флашка). Чтобы пользователь имел возможность оставить это поле пустым, нужно при инициализации поля отключить требование обязательности его заполнения, т. е. свойству `required` присвоить значение `False`. Иными словами, при создании в приложении этого поля надо использовать следующий код (листинг 6.11).

Листинг 6.11

```
from django import forms
class UserForm(forms.Form):
    basket = forms.BooleanField(label="Положить товар в корзину", required=False)
```

В этом случае пользователь может либо поставить флашок в поле **Положить товар в корзину**, либо оставить его пустым, и приложение будет работать корректно.

6.3.4. Поле `NullBooleanField` для выбора решения: да\нет

Метод `forms.NullBooleanField` создает на HTML-странице следующую разметку:

```
<select>
<option value="1" selected="selected">Неизвестно</option>
<option value="2">Да</option>
<option value="3">Нет</option>
</select>
```

Внесем изменение в модуль `forms.py` и создадим там это поле с помощью следующего кода (листинг 6.12).

Листинг 6.12

```
from django import forms
class UserForm(forms.Form):
    vyb = forms.NullBooleanField(label="Вы поедете в Сочи этим летом?")
```

Если запустить наше приложение, то мы получим такое отображение этого поля на форме (рис. 6.12).

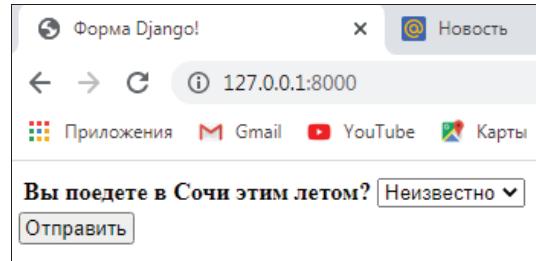


Рис. 6.12. Поле `NullBooleanField` на странице `index.html`

Поле `NullBooleanField` по умолчанию использует виджет `NullBooleanSelect` и имеет пустое значение `None`. В зависимости от действия пользователя оно может вернуть три значения: `None` (Неизвестно), `True` (Да) или `False` (Нет). При этом оно никогда и ничего не проверяет (т. е. не вызывает метод `ValidationError`). Если пользователь нажмет на стрелочку в этом поле, то ему будут предложены три варианта выбора (рис. 6.13).

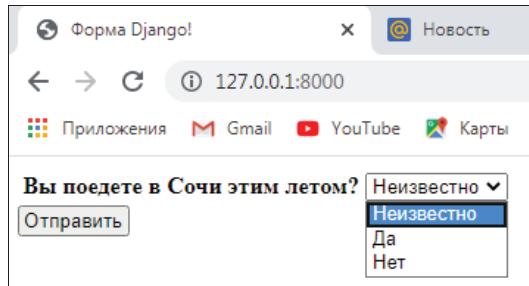


Рис. 6.13. Варианты выбора в поле `NullBooleanField` на странице `index.html`

6.3.5. Поле `CharField` для ввода текста

Метод `forms.CharField` создает на HTML-странице следующую разметку:

```
<input type="text">
```

Это поле служит для ввода текста. По умолчанию здесь используется виджет `TextInput`, начальное значение которого — пустая строка (значение `None`) или текст, который был задан в свойстве поля `empty_value`.

Внесем изменение в модуль `forms.py` и создадим там это поле с помощью следующего кода (листинг 6.13).

Листинг 6.13

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента")
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.14).

Не забываем, что в таком варианте по умолчанию это поле будет обязательным для заполнения. Если пользователь оставит его пустым, то при нажатии на кнопку **Отправить** ему будет выдано напоминание, и программа не даст продвинуться вперед до тех пор, пока поле не будет заполнено (рис. 6.15).

Если логикой программы допускается оставить это поле пустым, то его нужно создавать с помощью следующего программного кода:

```
name = forms.CharField(label="Имя клиента", required=False)
```

Форма Django!

127.0.0.1:8000

Сервисы Gmail YouTube Карты

Имя клиента:

Отправить

Рис. 6.14. Поле CharField на странице index.html

Форма Django!

127.0.0.1:8000

Сервисы Gmail YouTube Карты

Имя клиента:

Отправить

Заполните это поле.

Рис. 6.15. Предупреждение о необходимости заполнить поле CharField

Количество символов, которые пользователь может ввести в поле CharField, можно задать с помощью следующих аргументов:

- max_length — максимальное количество символов в поле;
- min_length — минимальное количество символов в поле.

Например, мы можем создать это поле с помощью следующего кода:

```
name = forms.CharField(label="Имя клиента", max_length=15,
                      help_text="ФИО не более 15 символов")
```

В этом случае пользователь не сможет внести в поле более чем 15 символов, и рядом с полем появится дополнительная подсказка (рис. 6.16).

Форма Django!

127.0.0.1:8000

Сервисы Gmail YouTube Карты

Имя клиента:

ФИО не более 15 символов

Отправить

Рис. 6.16. Подсказка об ограничении количества символов в поле CharField

6.3.6. Поле *EmailField* для ввода электронного адреса

Метод forms.EmailField создает на HTML-странице следующую разметку:

```
<input type="email">
```

Это поле служит для ввода электронного адреса.

Внесем изменение в модуль forms.py и создадим там такое поле с помощью следующего кода (листинг 6.14).

Листинг 6.14

```
from django import forms
class UserForm(forms.Form):
    email = forms.EmailField(label="Электронный адрес",
                           help_text="Обязательный символ - @")
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.17).

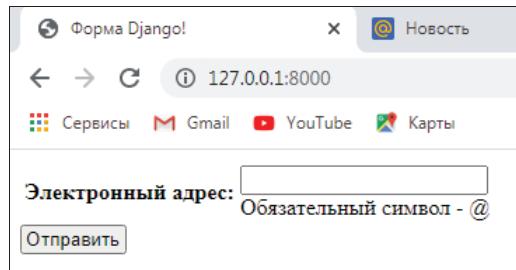


Рис. 6.17. Поле EmailField на странице index.html

Поле `EmailField` по умолчанию использует виджет `EmailInput` с пустым значением `None` и метод `EmailValidator` с умеренно сложным регулярным выражением, проверяющий, что введенное в поле значение является действительным адресом электронной почты.

Поле принимает два необязательных аргумента для проверки количества введенных символов: `max_length` — максимальная длина строки и `min_length` — минимальная длина строки. Они гарантируют, что длина введенной строки не превышает или равна заданной в этих аргументах длине строки.

Если пользователь ошибся и ввел некорректный электронный адрес, то при нажатии на кнопку **Отправить** ему будет выдано напоминание, и программа не даст продвинуться вперед до тех пор, пока поле не будет заполнено правильно (рис. 6.18).

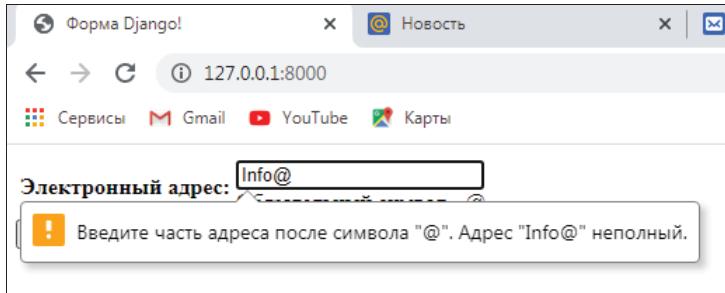


Рис. 6.18. Подсказка об ошибке при вводе электронного адреса в поле EmailField

6.3.7. Поле *GenericIPAddressField* для ввода IP-адреса

Метод `forms.GenericIPAddressField` создает на HTML-странице следующую разметку:

```
<input type="text">
```

Это поле служит для ввода IP-адреса.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.15).

Листинг 6.15

```
from django import forms
class UserForm(forms.Form):
    ip_adres = forms.GenericIPAddressField(label="IP адрес",
                                             help_text="Пример формата 192.0.2.0") .
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.19).

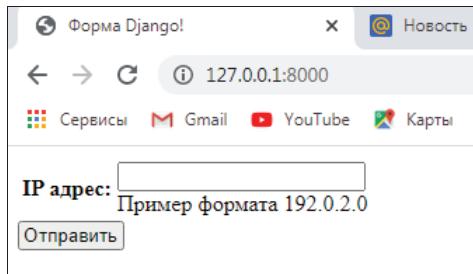


Рис. 6.19. Поле `GenericIPAddressField` на странице `index.html`

Поле `GenericIPAddressField` по умолчанию использует виджет `TextInput` с пустым значением `None`, при этом проверяется, что введенное значение является действительным IP-адресом.

6.3.8. Поле *RegexField* для ввода текста

Метод `forms.RegexField` создает на HTML-странице следующую разметку:

```
<input type="text">
```

Это поле предназначено для ввода текста, который должен соответствовать определенному регулярному выражению.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.16).

Листинг 6.16

```
from django import forms
class UserForm(forms.Form):
    reg_text = forms.RegexField(label="Текст", regex="^0-9[A-F]0-9$")
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.20).

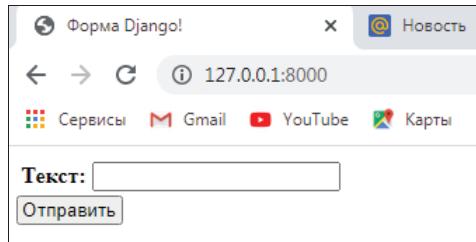


Рис. 6.20. Поле RegexField на странице index.html

Поле RegexField по умолчанию использует виджет TextInput с пустым значением None и метод RegexValidator, проверяющий, что введенное в поле значение соответствует определенному регулярному выражению, которое указывается в виде строки или скомпилированного объекта регулярного выражения.

Поле также принимает параметры: max_length — максимальная длина строки и min_length — минимальная длина строки, которые работают так же, как и для поля CharField.

6.3.9. Поле *SlugField* для ввода текста

Метод forms.SlugField создает на HTML-странице следующую разметку:

```
<input type="text">
```

Это поле предназначено для ввода текста, который условно называется «slug» и представляет собой последовательность символов в нижнем регистре, чисел, дефисов и знаков подчеркивания.

Внесем изменение в модуль forms.py и создадим там такое поле с помощью следующего кода (листинг 6.17).

Листинг 6.17

```
from django import forms
class UserForm(forms.Form):
    slug_text = forms.SlugField(label="Введите текст")
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.21).

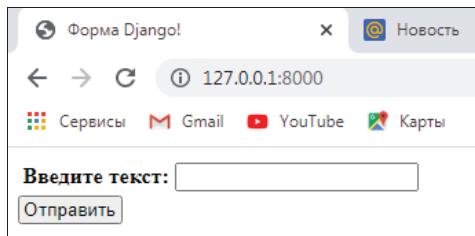


Рис. 6.21. Поле SlugField на странице index.html

Поле SlugField по умолчанию использует виджет TextInput с пустым значением None и методы validate_slug или validate_unicode_slug, проверяющие, что введенное значение содержит только буквы, цифры, подчеркивания и дефисы.

Поле также принимает необязательный параметр allow_unicode — логическое указание для поля принимать символы Unicode в дополнение к символам ASCII. Значение по умолчанию — False.

6.3.10. Поле *URLField* для ввода универсального указателя ресурса (URL)

Метод forms.URLField создает на HTML-странице следующую разметку:

```
<input type="url">
```

Это поле предназначено для ввода универсального указателя ресурса (URL) — например, такого: <http://www.google.com>.

Внесем изменение в модуль forms.py и создадим там такое поле с помощью следующего кода (листинг 6.18).

Листинг 6.18

```
from django import forms
class UserForm(forms.Form):
    url_text = forms.URLField(label="Введите URL",
                               help_text="Например http://www.google.com")
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.22).

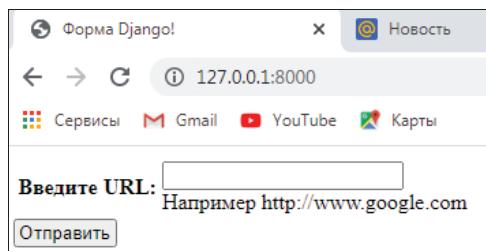


Рис. 6.22. Поле URLField на странице index.html

Поле `URLField` по умолчанию использует виджет `URLInput` с пустым значением `None` и метод `URLValidator`, проверяющий, что введенное значение является действительным URL.

Поле также принимает параметры: `max_length` — максимальная длина строки и `min_length` — минимальная длина строки, которые работают так же, как и для поля `CharField`.

Если пользователь при вводе URL допустит ошибку в формате этого типа данных, ему будет выдано соответствующее предупреждение (рис. 6.23).

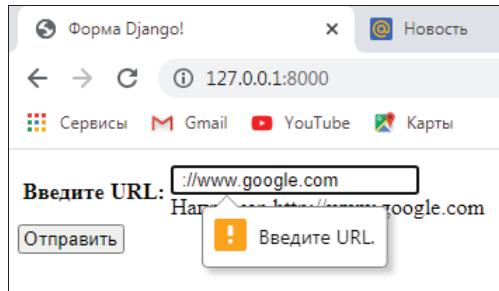


Рис. 6.23. Предупреждение об ошибке в поле `URLField`

6.3.11. Поле `UUIDField` для ввода универсального уникального идентификатора UUID

Метод `forms.UUIDField` создает на HTML-странице следующую разметку:

```
<input type="text">
```

Это поле предназначено для ввода универсального уникального идентификатора UUID (например: 123e4567-e89b-12d3-a456-426655440000).

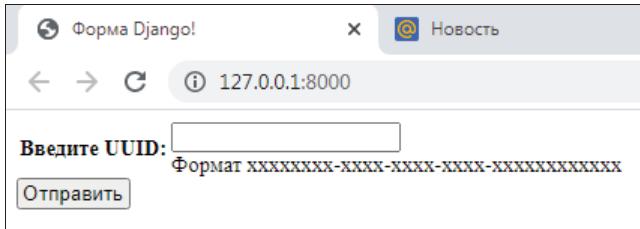
Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.19).

Листинг 6.19

```
from django import forms
class UserForm(forms.Form):
    uuid_text = forms.UUIDField(label="Введите UUID",
                                help_text="Формат xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx") .
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.24).

Поле `UUIDField` по умолчанию использует виджет `TextInput` с пустым значением `None`. Поле будет принимать любой формат строки, принятый в качестве HEX-аргумента для `UUID`-конструктора.



The screenshot shows a browser window titled 'Форма Django!' with the URL '127.0.0.1:8000'. The page contains a single input field labeled 'Введите UUID:' with a placeholder 'Форматxxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'. Below the input field is a button labeled 'Отправить'.

Рис. 6.24. Поле `UUIDField` на странице `index.html`

6.3.12. Поле `ComboField` для ввода текста с проверкой соответствия заданным форматам

Метод `forms.ComboField` создает на HTML-странице следующую разметку:

```
<input type="text">
```

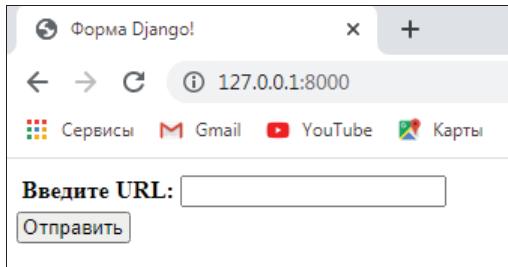
Это поле имеет структуру `forms.ComboField(fields=[field1, field2,...])` и представляет собой аналог обычного текстового поля за тем исключением, что требует, чтобы вводимый текст соответствовал требованиям тех полей, которые передаются через параметр `fields`.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.20).

Листинг 6.20

```
from django import forms
class UserForm(forms.Form):
    combo_text = forms.ComboField(label="Введите URL",
                                   fields=[forms.URLField(),
                                           forms.CharField(max_length=20)])
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.25).



The screenshot shows a browser window titled 'Форма Django!' with the URL '127.0.0.1:8000'. The page contains a single input field labeled 'Введите URL:' with a placeholder. Below the input field is a button labeled 'Отправить'. At the top of the page, there is a navigation bar with links to 'Сервисы', 'Gmail', 'YouTube', and 'Карты'.

Рис. 6.25. Поле `ComboField` на странице `index.html`

Для поля `ComboBoxField` делается проверка введенного значения на соответствие каждому из полей, указанных в качестве аргумента при его создании. Оно принимает один дополнительный обязательный аргумент: `fields`. Это список полей, которые следует использовать для проверки значения, введенного в поле (в порядке, в котором они предоставляются). В нашем примере вводимые пользователем данные будут проверяться на соответствие формату поля для URL, чтобы количество символов не превысило заданной величины (в нашем случае — 20 символов).

6.3.13. Поле `FilePathField` для создания списка файлов

Метод `forms.FilePathField` создает на HTML-странице следующую разметку:

```
<select>
    <option value="folder/file1">folder/file1</option>
    <option value="folder/file2">folder/file2</option>
    <option value="folder/file3">folder/file3</option>
    //.....
</select>
```

Поле `FilePathField` по умолчанию использует виджет `Select` с пустым значением `None` и проверяет, существует ли выбранный вариант в списке вариантов. Ключи сообщений об ошибках: `required`, `invalid_choice`. Поле позволяет делать выбор из файлов внутри определенного каталога. При этом ему требуются пять дополнительных аргументов:

- `path` — абсолютный путь к каталогу (папке), содержимое которого вы хотите отобразить в списке. Это обязательный параметр, и такой каталог должен существовать;
- `recursive` — разрешает или запрещает доступ к подкаталогам. Если этот параметр имеет значение `False` (по умолчанию), то будут предложены к выбору файлы только в указанном каталоге. Если параметр будет иметь значение `True`, то к выбору будут предложены все вложенные каталоги;
- `match` — этот параметр представляет шаблон регулярного выражения (будут отображаться только файлы с именами, совпадающими с этим шаблоном). Регулярное выражение применяется к названию файла, а не к полному пути. Например, выражение `"foo.*\.txt$"` покажет файл `foo23.txt`, но отфильтрует файлы `bar.txt` или `foo23.gif`;
- `allow_files` — указывает, следует ли включать файлы в указанном каталоге (по умолчанию `True`), при этом параметр `allow_folders` должен иметь значение `True`;
- `allow_folders` — указывает, следует ли включать подкаталоги в указанном каталоге (по умолчанию `False`), при этом параметр `allow_files` должен иметь значение `True`.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.21).

Листинг 6.21

```
from django import forms
class UserForm(forms.Form):
    file_path = forms.FilePathField(label="Выберите файл",
                                    path="C:/Python37/")
```

Здесь мы задали возможность выбора файла из папки C:\Python37\.

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.26).

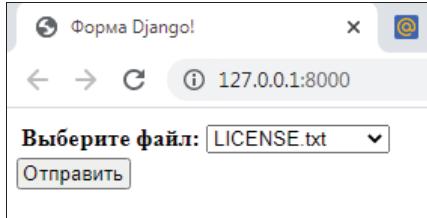


Рис. 6.26. Поле FilePathField на странице index.html

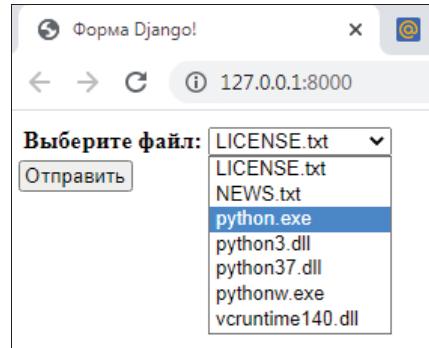


Рис. 6.27. В поле FilePathField предоставлена возможность выбора файла из списка доступных

Если мы теперь щелкнем мышью на этом поле, то откроется список файлов, которые содержатся в указанном каталоге. Пользователь будет иметь возможность выбора любого файла из предложенного списка (рис. 6.27).

Внесем еще одно изменение в модуль forms.py, с помощью которого реализуем возможность отображать в поле FilePathField не только файлы, но и вложенные каталоги (листинг 6.22).

Листинг 6.22

```
from django import forms
class UserForm(forms.Form):
    file_path = forms.FilePathField(label="Выберите файл",
                                    path="C:/Python37/",
                                    allow_files="True",
                                    allow_folders="True")
```

Если теперь запустить наше приложение, то мы увидим, что кроме файлов у нас появилась возможность выбора и вложенных каталогов (рис. 6.28).

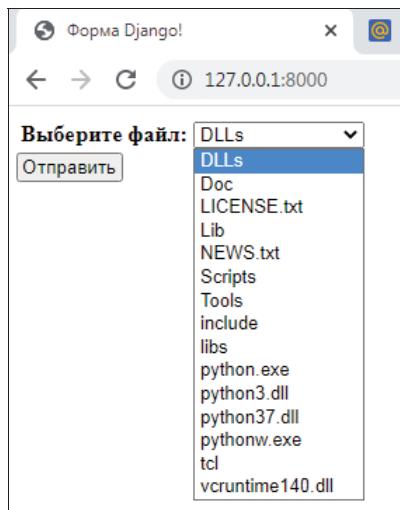


Рис. 6.28. В поле `FilePathField` предоставлена возможность выбора файла и вложенных каталогов из списка доступных

6.3.14. Поле `FileField` для выбора файлов

Метод `forms.FileField` создает на HTML-странице следующую разметку:

```
<input type="file">
```

Поле `FileField` предназначено для выбора и загрузки файлов и по умолчанию использует виджет `ClearableFileInput` с пустым значением `None`. Поле формирует объект `UploadedFile`, который упаковывает содержимое файла и имя файла в один объект. Поле принимает два необязательных аргумента для проверки длины вводимой строки: `max_length` (гарантирует, что имя файла не превысит максимальную заданную длину) и `allow_empty_file` (гарантирует, что проверка пройдет успешно, даже если содержимое файла пустое).

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.23).

Листинг 6.23

```
from django import forms
class UserForm(forms.Form):
    file = forms.FileField(label="Файл")
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.29).

Как можно видеть, поле `FileField` представлено в виде кнопки с надписью **Выберите файл** и сообщением **Файл не выбран**. Если теперь нажать на кнопку **Выберите файл**, то откроется новое окно, в котором можно перемещаться по любым папкам компьютера и просматривать и выбирать любые файлы (рис. 6.30).

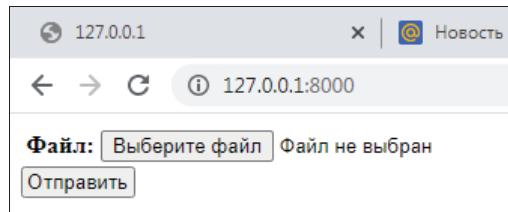


Рис. 6.29. Поле FileField на странице index.html

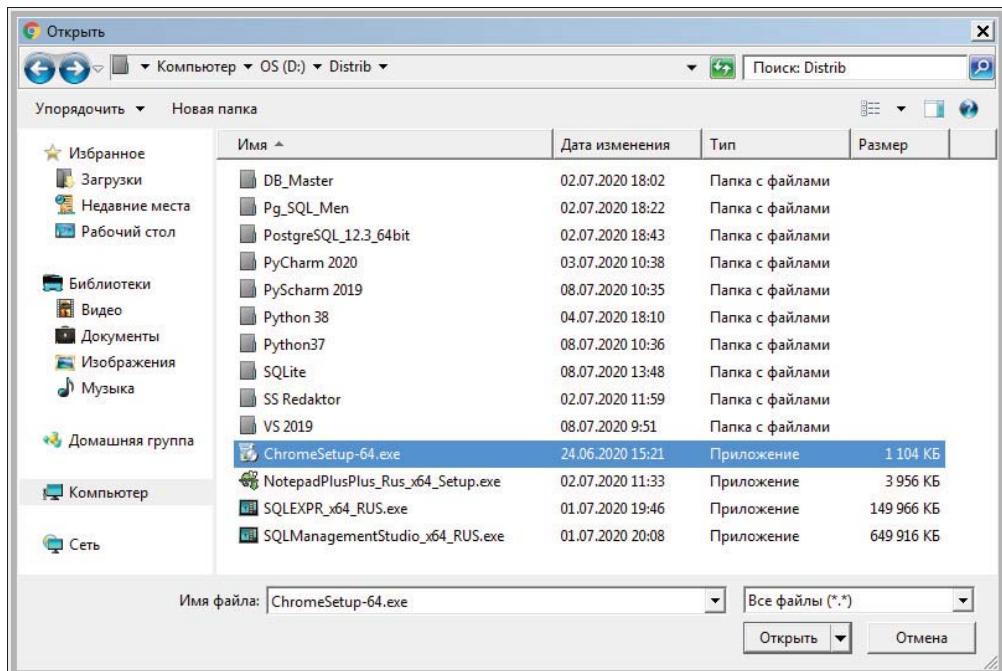


Рис. 6.30. Возможность выбора файла из окна ОС Windows при активации поля FileField

Предположим, что был выбран файл `ChromeSetup-64.exe` и нажата кнопка **Открыть**, — окно Windows будет закрыто, а имя выбранного файла показано на веб-странице (рис. 6.31).

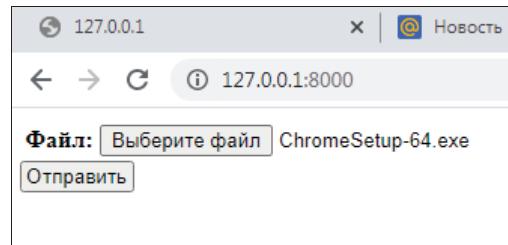


Рис. 6.31. Поле FileField на странице index.html после того, как пользователь выбрал файл

6.3.15. Поле *ImageField* для выбора файлов изображений

Метод `forms.ImageField` создает на HTML-странице следующую разметку:

```
<input type="file">
```

Поле `ImageField` предназначено для выбора и загрузки файлов, представляющих собой изображения, и по умолчанию использует виджет `ClearableFileInput` с пустым значением `None`. Поле формирует объект `UploadedFile`, который упаковывает содержимое файла и имя файла в один объект.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.24).

Листинг 6.24

```
from django import forms
class UserForm(forms.Form):
    file = forms.ImageField(label="Изображение") .
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.32).

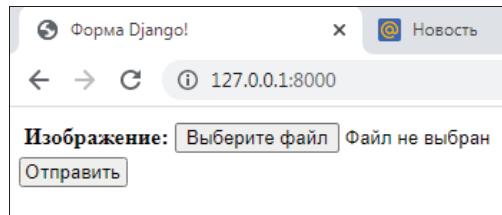


Рис. 6.32. Поле `ImageField` на странице `index.html`

Как можно видеть, поле `ImageField` представлено в виде кнопки с надписью **Выберите файл** и сообщением **Файл не выбран**. Если теперь нажать на кнопку **Выберите файл**, то откроется новое окно, в котором можно перемещаться по любым папкам компьютера и просматривать и выбирать любые файлы (рис. 6.33).

Предположим, что был выбран файл `image1.jpg` и нажата кнопка **Открыть**, — окно Windows будет закрыто, а имя выбранного файла показано на веб-странице (рис. 6.34).

6.3.16. Поле *DateField* для ввода даты

Метод `forms.DateField` создает на HTML-странице следующую разметку:

```
<input type="text">
```

Поле `DateField` служит для ввода дат (например, `2021-12-25` или `25/12/2021`) и по умолчанию использует виджет `DateInput` с пустым значением `None`. При этом дела-

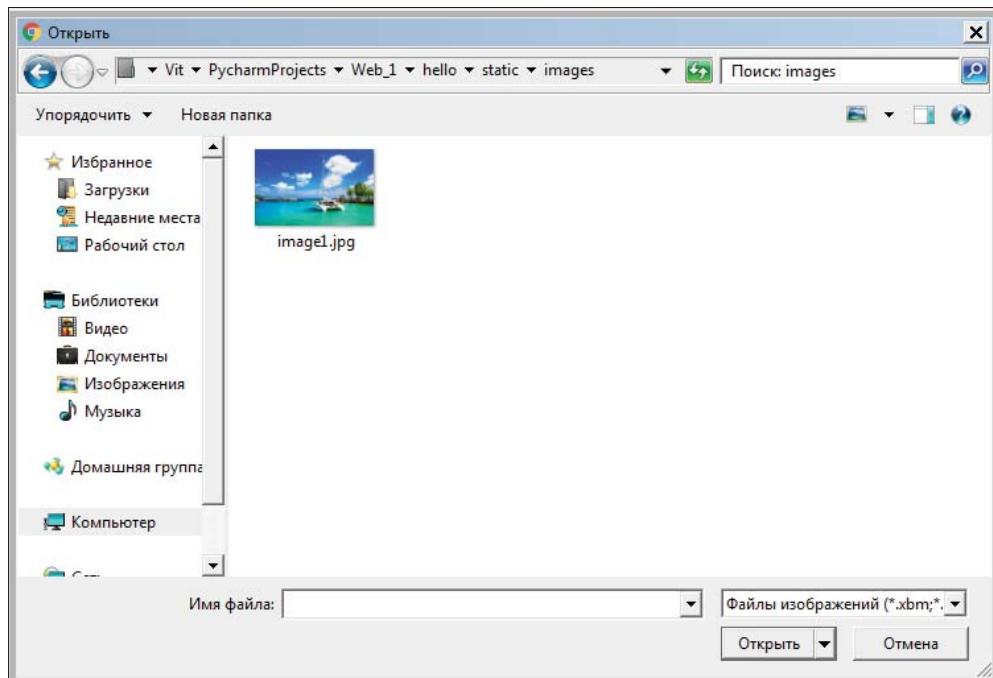


Рис. 6.33. Возможность выбора файла изображения из окна ОС Windows при активации поля ImageField

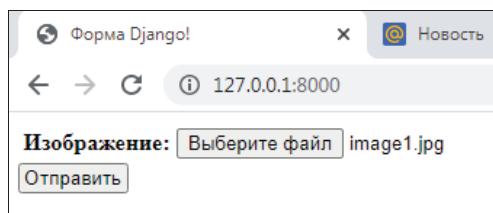


Рис. 6.34. Поле ImageField на странице index.html после того, как пользователь выбрал файл с изображением

ется проверка, является ли введенное значение либо строкой `datetime.date`, либо форматированной в определенном формате датой. Поле принимает один необязательный аргумент `input_formats`.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.25).

Листинг 6.25

```
from django import forms
class UserForm(forms.Form):
    date = forms.DateField(label="Введите дату") .
```

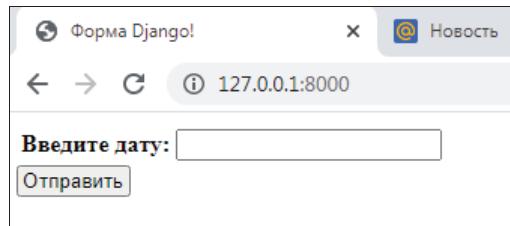


Рис. 6.35. Поле DateField на странице index.html

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.35).

6.3.17. Поле *TimeField* для ввода времени

Метод `forms.TimeField` создает на HTML-странице следующую разметку:

```
<input type="text">
```

Поле `TimeField` служит для ввода значений времени (например, 14:30:59 или 14:30) и по умолчанию использует виджет `TimeInput` с пустым значением `None`. При этом делается проверка, является ли введенное значение либо строкой `datetime.time`, либо форматированным в определенном формате значением времени. Поле принимает один необязательный аргумент `input_formats`.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.26).

Листинг 6.26

```
from django import forms
class UserForm(forms.Form):
    time = forms.DateTimeField(label="Введите время") .
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.36).

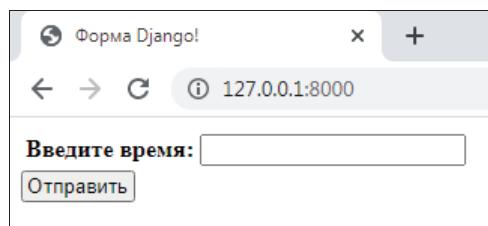


Рис. 6.36. Поле TimeField на странице index.html

6.3.18. Поле *DateTimeField* для ввода даты и времени

Метод `forms.DateTimeField` создает на HTML-странице следующую разметку:

```
<input type="text">
```

Поле `DateTimeField` служит для ввода даты и времени (например, `2021-12-25 14:30:59` или `25/12/2021 14:30`) и по умолчанию использует виджет `DateTimeInput` с пустым значением `None`. При этом делается проверка, является ли введенное значение либо строкой `datetime.datetime`, `datetime.date`, либо форматированными в определенном формате значениями даты и времени. Поле принимает один необязательный аргумент `input_formats`.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.27).

Листинг 6.27

```
from django import forms
class UserForm(forms.Form):
    date_time = forms.DateTimeField(label="Введите дату и время") .
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.37).

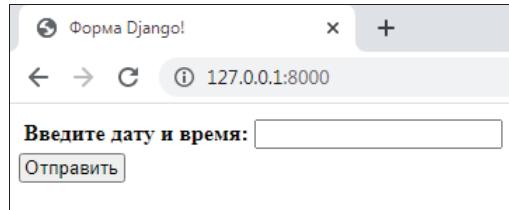


Рис. 6.37. Поле `DateTimeField` на странице `index.html`

6.3.19. Поле *DurationField* для ввода промежутка времени

Метод `forms.DurationField` создает на HTML-странице следующую разметку:

```
<input type="text">
```

Поле `DurationField` предназначено для ввода временного промежутка. Вводимый текст должен соответствовать формату "`DD HH:MM:SS`" — например, `2 1:10:20` (2 дня 1 час 10 минут 20 секунд). По умолчанию поле использует виджет `TextInput` с пустым значением `None`. При этом делается проверка, является ли введенное значение строкой, которую можно преобразовать в `timedelta`. Поле принимает два аргумента для проверки длины вводимой строки: `datetime.timedelta.min` и `datetime.timedelta.max`. Значение должно быть заключено между этими величинами.

Внесем изменение в модуль forms.py и создадим там такое поле с помощью следующего кода (листинг 6.28).

Листинг 6.28

```
from django import forms
class UserForm(forms.Form):
    time_delta = forms.DurationField
        (label="Введите промежуток времени") .
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.38).

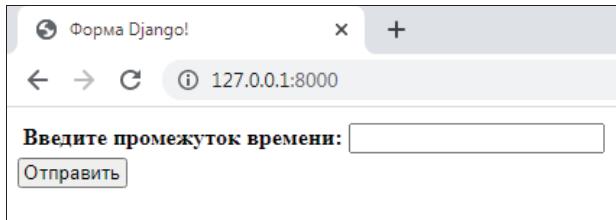


Рис. 6.38. Поле DurationField на странице index.html

6.3.20. Поле *SplitDateTimeField* для раздельного ввода даты и времени

Метод forms.SplitDateTimeField создает на HTML-странице следующую разметку:

```
<input type="text" name="_0" >
<input type="text" name="_1" >
```

Поле SplitDateTimeField предназначено для ввода даты и времени в два раздельных текстовых поля и по умолчанию использует виджет SplitDateTimeWidget с пустым значением None. В первое поле вводится дата, во второе поле — время. При этом делается проверка, является ли введенное значение либо строкой datetime.datetime, либо форматированными в определенном формате значениями даты и времени.

Поле принимает два необязательных аргумента:

- ❑ input_date_formats — список форматов, используемых для попытки преобразования строки в допустимый объект datetime.date. Если аргумент input_date_formats не указан, используются форматы ввода по умолчанию для поля DateField;
- ❑ input_time_formats — список форматов, используемых для попытки преобразования строки в допустимый объект datetime.time. Если аргумент input_time_formats не указан, используются форматы ввода по умолчанию для поля TimeField.

Внесем изменение в модуль forms.py и создадим там такое поле с помощью следующего кода (листинг 6.29).

Листинг 6.29

```
from django import forms
class UserForm(forms.Form):
    date_time = forms.SplitDateTimeField
        (label="Введите дату и время")
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.39).

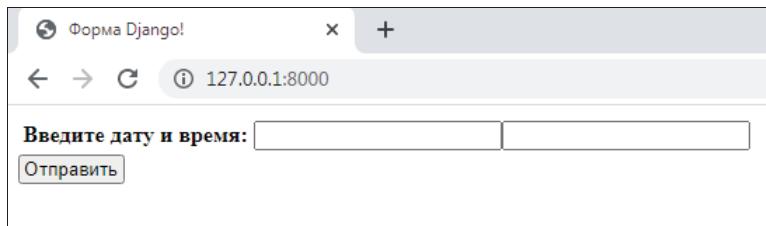


Рис. 6.39. Поле SplitDateTimeField на странице index.html

6.3.21. Поле IntegerField для ввода целых чисел

Метод forms.IntegerField создает на HTML-странице следующую разметку:

```
<input type="number">
```

Поле IntegerField служит для ввода целых чисел и по умолчанию использует виджет NumberInput с пустым значением None. При этом делается проверка, является ли вводимое число целым. Поле принимает два необязательных аргумента для проверки максимального (`max_value`) и минимального (`min_value`) значения вводимого числа и использует методы `.MaxValueValidator` и `.MinValueValidator`, если эти ограничения заданы.

Внесем изменение в модуль forms.py и создадим там такое поле с помощью следующего кода (листинг 6.30).

Листинг 6.30

```
from django import forms
class UserForm(forms.Form):
    num = forms.IntegerField(label="Введите целое число")
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.40).

Форма Django!

← → C 127.0.0.1:8000

Введите целое число:

Отправить

Рис. 6.40. Поле IntegerField на странице index.html

Если в поле IntegerField попытаться ввести число с дробной частью, то будет выдано предупреждение об ошибке ввода (рис. 6.41).

Форма Django!

← → C 127.0.0.1:8000

Введите целое число:

! Введите допустимое значение. Ближайшие допустимые значения: 12 и 13.

Рис. 6.41. Предупреждение об ошибке при вводе в поле IntegerField числа с дробной частью

6.3.22. Поле *DecimalField* для ввода десятичных чисел

Метод forms.DecimalField создает на HTML-странице следующую разметку:

```
<input type="number">
```

Поле служит DecimalField для ввода десятичных чисел и по умолчанию использует виджет NumberInput с пустым значением None. При этом делается проверка, является ли вводимое число десятичным.

Поле принимает четыре необязательных аргумента:

- max_value — максимальное значение вводимого числа;
- min_value — минимальное значение вводимого числа;
- max_whole_digits — максимальное количество цифр (до десятичной запятой плюс после десятичной запятой, с разделенными начальными нулями), допустимое в значении;
- decimal_places — максимально допустимое количество знаков после запятой.

Сообщения об ошибках, выдаваемые при выходе вводимого значения за пределы max_value и min_value, могут содержать тег %(limit_value)s, замещаемый соответствующим пределом. Аналогичным образом сообщения об ошибках при выходе вводимых значений за пределы, определяемые параметрами max_digits, max_decimal_places и max_whole_digits, могут содержать тег %(max)s.

Внесем изменение в модуль forms.py и создадим там такое поле с помощью следующего кода (листинг 6.31).

Листинг 6.31

```
from django import forms
class UserForm(forms.Form):
    num = forms.DecimalField(label="Введите десятичное число") .
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.42).

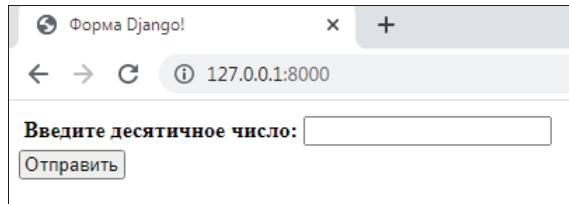


Рис. 6.42. Поле DecimalField на странице index.html

В поле DecimalField невозможен ввод никаких символов, кроме чисел и разделителя дробной части, а также значений, выходящих за пределы заданных необязательных аргументов. Чтобы показать это, изменим программный код и укажем, что дробная часть числа ограничена двумя знаками (листинг 6.32).

Листинг 6.32

```
from django import forms
class UserForm(forms.Form):
    num = forms.DecimalField(label="Введите десятичное число",
                            decimal_places=2)
```

Если теперь в поле DecimalField попытаться ввести число с более длинной дробной частью, то будет выдано предупреждение об ошибке ввода (рис. 6.43).

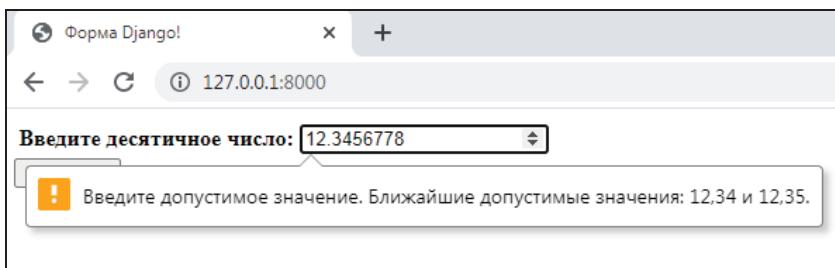


Рис. 6.43. Предупреждение об ошибке при вводе в поле DecimalField числа с дробной частью, превышающей заданное ограничение

6.3.23. Поле *FloatField* для ввода чисел с плавающей точкой

Метод `forms.FloatField` создает на HTML-странице следующую разметку:

```
<input type="number">
```

Поле `FloatField` служит для ввода чисел с плавающей точкой и по умолчанию использует виджет `NumberInput` с пустым значением `None`. При этом делается проверка, является ли вводимое число числом с плавающей точкой. Поле принимает два необязательных аргумента для проверки максимального (`max_value`) и минимального (`min_value`) значения вводимого числа, контролирующие диапазон значений, разрешенных в поле.

внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.33).

Листинг 6.33

```
from django import forms
class UserForm(forms.Form):
    num = forms.FloatField(label="Введите число")
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.44).



Рис. 6.44. Поле `FloatField` на странице `index.html`

6.3.24. Поле *ChoiceField* для выбора данных из списка

Метод `forms.ChoiceField` создает на HTML-странице следующую разметку:

```
<select>
    <option value="1">Date 1</option>
    <option value="2"> Date 2</option>
    <option value="3"> Date 3</option>
</select>
```

Поле `ChoiceField` служит для выбора данных из списка и по умолчанию использует виджет `Select` с пустым значением `None`.

Поле имеет структуру `forms.ChoiceField(choices=кортеж_кортежей)` и генерирует список `select`, каждый из элементов которого формируется на основе отдельного кортежа. Например, следующее поле:

```
forms.ChoiceField(choices=((1, "Английский"),
                           (2, "Немецкий"),
                           (3, "Французский")))
```

сформирует список для выбора из трех элементов.

Поле `ChoiceField` принимает один аргумент — `choices`. Это либо итерация из двух кортежей, используемая в качестве элемента выбора для этого поля, либо вызываемая функция, которая возвращает такую итерацию.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.34).

Листинг 6.34

```
from django import forms
class UserForm(forms.Form):
    language = forms.ChoiceField(label="Выберите язык",
                                 choices=((1, "Английский"),
                                         (2, "Немецкий"),
                                         (3, "Французский")))
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.45).

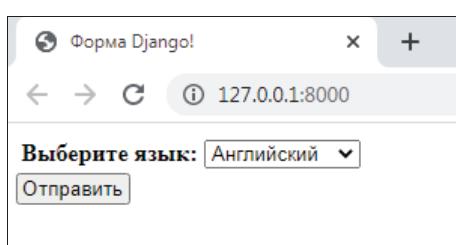


Рис. 6.45. Поле `ChoiceField` на странице `index.html`

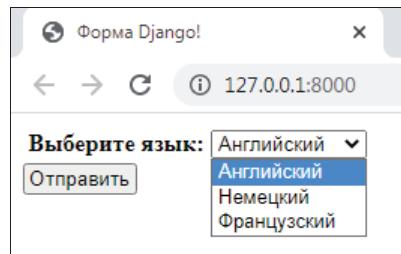


Рис. 6.46. Раскрывающийся список для выбора данных в поле `ChoiceField`

Если пользователь щелкнет мышью в этом поле, то откроется список выбора (рис. 6.46).

6.3.25. Поле `TypedChoiceField` для выбора данных из списка

Метод `forms.TypedChoiceField` создает на HTML-странице следующую разметку:

```
<select>
<option value="1">Date 1</option>
```

```
<option value="2"> Date 2</option>
<option value="3"> Date 3</option>
</select>
```

Поле `TypedChoiceField` служит для выбора данных из списка. Оно аналогично полю `ChoiceField` и по умолчанию также использует виджет `Select` с пустым значением `None`.

Поле имеет следующую структуру:

```
forms.TypedChoiceField(choises=кортеж_кортежей,
                      coerce=функция_преобразования,
                      empty_value=None)
```

Поле генерирует список `select` на основе кортежа, однако дополнительно принимает еще два аргумента:

- `coerce` — функцию преобразования, которая принимает один аргумент и возвращает его приведенное значение;
- `empty_value` — значение, используемое для представления «пусто». По умолчанию используется пустая строка.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.19).

Листинг 6.35

```
from django import forms
class UserForm(forms.Form):
    city = forms.TypedChoiceField(label="Выберите город",
                                  empty_value=None,
                                  choices=((1, "Москва"),
                                           (2, "Воронеж"),
                                           (3, "Курск")))
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.47).

Если пользователь щелкнет мышью в этом поле, то откроется список выбора (рис. 6.48).

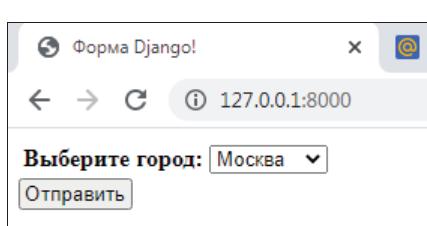


Рис. 6.47. Поле `TypedChoiceField` на странице `index.html`

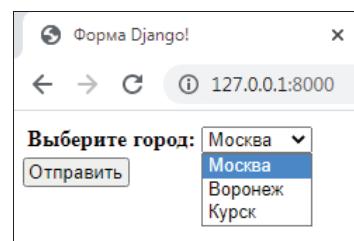


Рис. 6.48. Раскрывающийся список для выбора данных в поле `TypedChoiceField`

6.3.26. Поле *MultipleChoiceField* для выбора данных из списка

Поле *MultipleChoiceField* имеет следующую структуру:

```
forms.MultipleChoiceField(choises=кортеж_кортежей)
```

Поле генерирует список *select* на основе кортежа, как и поле *forms.ChoiceField*, добавляя к создаваемому полю атрибут *multiple="multiple"* и обеспечивая тем самым поддержку множественного выбора.

По умолчанию поле *MultipleChoiceField* использует виджет *SelectMultiple* с пустым значением [] (пустой список), а также — как и поле *ChoiceField* — принимает один дополнительный обязательный аргумент *choices*.

Внесем изменение в модуль *forms.py* и создадим там такое поле с помощью следующего кода (листинг 6.36).

Листинг 6.36

```
from django import forms
class UserForm(forms.Form):
    country = forms.MultipleChoiceField(label="Выберите страны",
                                         choices=((1, "Англия"),
                                                   (2, "Германия"),
                                                   (3, "Испания"),
                                                   (4, "Россия")))
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.49).

Как можно видеть, в поле одновременно выводится весь список, и пользователь имеет возможность выбрать из него несколько элементов. Для этого необходимо поочередно щелкнуть мышью на нужных элементах списка, удерживая при этом нажатой клавишу <Ctrl>, — выбранные пользователем элементы будут выделены цветом (рис. 6.50). В рассматриваемом случае выбраны две страны: Германия и Россия.

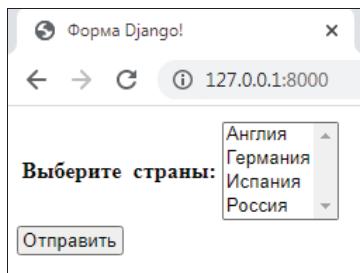


Рис. 6.49. Поле *MultipleChoiceField* на странице index.html

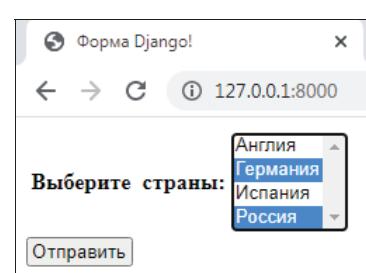


Рис. 6.50. В поле *MultipleChoiceField* предоставлена возможность выбора из списка нескольких элементов

6.3.27. Поле *TypedMultipleChoiceField* для выбора данных из списка

Поле *TypedMultipleChoiceField* имеет следующую структуру:

```
forms.TypedMultipleChoiceField(choises=кортеж_кортежей,
                               coerce=функция_преобразования,
                               empty_value=None)
```

Поле генерирует список `select` на основе кортежа, как и поле `forms.TypedChoiceField`, добавляя к создаваемому полю атрибут `multiple="multiple"` и обеспечивая тем самым поддержку множественного выбора.

По умолчанию поле *TypedMultipleChoiceField* использует виджет `SelectMultiple` с пустым значением `[]` (пустой список), а также принимает два дополнительных аргумента:

- `coerce` — функцию преобразования, которая принимает один аргумент и возвращает его приведенное значение;
- `empty_value` — значение, используемое для представления «пусто». По умолчанию используется пустая строка.

Внесем изменение в модуль `forms.py` и создадим там такое поле с помощью следующего кода (листинг 6.37).

Листинг 6.37

```
from django import forms
class UserForm(forms.Form):
    city = forms.TypedMultipleChoiceField(label="Выберите город",
                                           empty_value=None,
                                           choices=((1, "Москва"),
                                                     (2, "Воронеж"),
                                                     (3, "Курск"),
                                                     (4, "Томск")))
```

Если запустить наше приложение, то мы получим следующее отображение этого поля на форме (рис. 6.51).

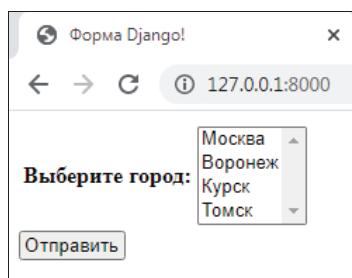


Рис. 6.51. Поле *TypedMultipleChoiceField* на странице index.html

Как можно видеть, в поле одновременно выводится весь список, и пользователь имеет возможность выбрать из него несколько элементов. Для этого необходимо поочередно щелкнуть мышью на нужных элементах списка, удерживая при этом нажатой клавишу <Ctrl>, — выбранные пользователем элементы будут выделены цветом (рис. 6.52). В рассматриваемом случае выбраны два города: **Воронеж** и **Томск**.

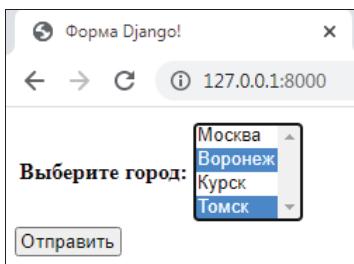


Рис. 6.52. В поле TypedMultipleChoiceField предоставлена возможность выбора из списка нескольких элементов

6.4. Настройка формы и ее полей

В предыдущем разделе мы познакомились с различными типами полей для ввода данных и основными их свойствами — такими как, например, метка поля (`label`), подсказки (`help_text`) и т. п. В этом разделе мы познакомимся с тем, как можно менять некоторые другие свойства полей и их положение на веб-странице.

6.4.1. Изменение внешнего вида поля с помощью параметра `widget`

Параметр `widget` позволяет задать объект, который будет использоваться для генерации HTML-разметки и тем самым определять внешний вид поля на веб-странице. Если пользователь явно не указывает виджет, то Django применит тот виджет, который задан по умолчанию. Однако пользователь может задать для поля свой виджет, заменив им тот, который используется по умолчанию. Рассмотрим это на примере.

Внесем изменение в модуль `forms.py` и создадим там три поля с помощью следующего кода (листинг 6.38).

Листинг 6.38

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя")
    age = forms.IntegerField(label="Возраст")
    comment = forms.CharField(label="Комментарий")
```

The screenshot shows a browser window with the title 'Форма Django!'. The address bar displays '127.0.0.1:8000'. The form contains three text input fields: 'Имя' (Name), 'Возраст' (Age), and 'Комментарий' (Comment). Below the comment field is a large text area. At the bottom is a blue 'Отправить' (Send) button.

Рис. 6.53. Три текстовых поля на странице index.html с виджетами TextInput по умолчанию

Если запустить наше приложение, то мы получим следующее отображение этих полей на форме (рис. 6.53).

Поскольку мы явно не указали виджет для этих полей, то по умолчанию для них использовался виджет `TextInput`. Для ввода имени и возраста этот виджет вполне подходит, а вот для комментария его использовать не совсем удобно, поскольку желательно иметь для ввода текста более широкое поле. Назначим для поля ввода комментариев другой виджет — `Textarea`. Для этого следующим образом изменим код в модуле `forms.py` (листинг 6.39).

Листинг 6.39

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя")
    age = forms.IntegerField(label="Возраст")
    comment = forms.CharField(label="Комментарий",
                              widget=forms.Textarea)
```

После этого изменения форма будет выглядеть следующим образом (рис. 6.54).

The screenshot shows a browser window with the title 'Форма Django!'. The address bar displays '127.0.0.1:8000'. The form contains three text input fields: 'Имя' (Name), 'Возраст' (Age), and 'Комментарий' (Comment). The 'Комментарий' field is represented by a large text area. At the bottom is a blue 'Отправить' (Send) button.

Рис. 6.54. Три текстовых поля на странице index.html с виджетом `Textarea` для поля с комментариями

6.4.2. Задание начальных значений полей с помощью свойства *initial*

Когда в программе создается новое поле, то оно, как правило, имеет пустое значение. Однако его можно сделать не пустым, воспользовавшись свойством *initial*.

Внесем изменение в модуль *forms.py*, задав полям для ввода имени и возраста начальные значения с помощью следующего кода (листинг 6.40).

Листинг 6.40

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя", initial="Введите ФИО")
    age = forms.IntegerField(label="Возраст", initial=18)
    comment = forms.CharField(label="Комментарий",
                               widget=forms.Textarea)
```

После этих изменений при запуске нашего приложения созданные поля не будут пустыми (рис. 6.55).

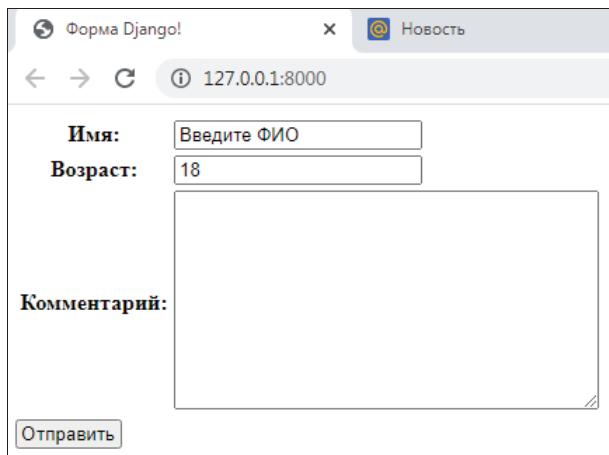


Рис. 6.55. Текстовые поля на странице index.html с заданными начальными значениями

6.4.3. Задание порядка следования полей на форме

По умолчанию все поля на форме идут в той последовательности, в которой они описаны в модуле инициализации формы. Однако в процессе работы над проектом разработчик может многократно добавлять или удалять поля. Чтобы иметь возможность гибко менять порядок следования полей на форме, можно использовать свойство формы *field_order*. Это свойство позволяет переопределить порядок следования полей как в классе формы, так и при определении объекта формы в представлении.

В классе формы это можно сделать, например, следующим образом (листинг 6.41).

Листинг 6.41

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя", initial="Введите ФИО")
    age = forms.IntegerField(label="Возраст", initial=18)
    field_order = ["age", "name"]
```

Здесь мы указали, что поле `age` будет отображаться на форме раньше, чем поле `name`, несмотря на то что поле `name` было создано первым. При запуске приложения поля будут идти в той последовательности, как указано в свойстве `field_order` (рис. 6.56).

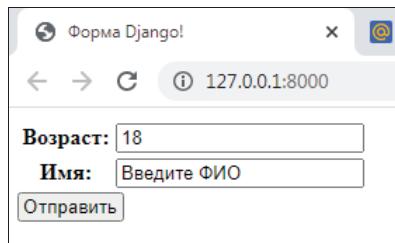


Рис. 6.56. Изменение порядка следования полей на форме с использованием свойства `field_order` в описании класса формы

Теперь изменим порядок следования полей при определении объекта формы в представлении. Для этого внесем изменение в программный код представления `views.py` (листинг 6.42).

Листинг 6.42

```
from django.http import *
from .forms import UserForm
from django.shortcuts import render

def index(request):
    userform = UserForm(field_order=["age", "name"])
    return render(request, "firstapp/index.html", {"form": userform})
```

А также изменим программный код в модуле `forms.py` (листинг 6.43).

Листинг 6.43

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя", initial="Введите ФИО")
    age = forms.IntegerField(label="Возраст", initial=18)
```

Здесь при описании класса формы поле `name` стоит на первом месте, а поле `age` — на втором. Однако в представлении `views.py` был задан обратный порядок следования этих полей с помощью свойства `field_order`:

```
userform = UserForm(field_order=["age", "name"])
```

В результате поля на форме будут выведены именно в этой последовательности — сначала `age`, а затем `name`.

6.4.4. Задание подсказок к полям формы

Для того чтобы предоставить пользователю дополнительную информацию о том, какие данные следует вводить в то или иное поле, можно использовать свойство поля `help_text`.

Восстановим естественный порядок следования полей при определении объекта формы в представлении. Для этого внесем изменение в программный код представления `views.py` (листинг 6.44).

Листинг 6.44

```
from django.http import *
from .forms import UserForm
from django.shortcuts import render

def index(request):
    userform = UserForm()
    return render(request, "firstapp/index.html", {"form": userform})
```

А в программный код в модуле `forms.py` добавим подсказки пользователю с использованием свойства `help_text` (листинг 6.45).

Листинг 6.45

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя", help_text ="Введите ФИО")
    age = forms.IntegerField(label="Возраст", help_text ="Введите возраст")
```

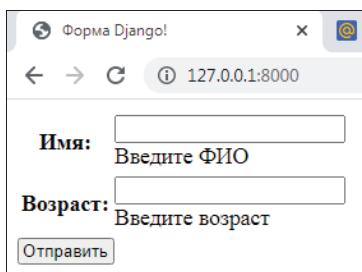


Рис. 6.57. Подсказки для полей формы с использованием свойства `help_text`

После этих изменений при запуске нашего приложения рядом с соответствующими полями появятся подсказки (рис. 6.57).

6.4.5. Настройки вида формы

Общее отображение полей на форме можно настроить с помощью специальных методов:

- `as_table()` — отображение полей в виде таблицы;
- `as_ul()` — отображение полей в виде списка;
- `as_p()` — отображение каждого поля формы в отдельном параграфе (абзаце).

Посмотрим, как работают эти методы на примере, для чего внесем следующие изменения в шаблон для главной страницы `firstapp\index.html` (листинг 6.46).

Листинг 6.46

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
<h2>Форма как таблица</h2>
<form method="POST">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" value="Отправить" >
</form>

<h2>Форма как список</h2>
<form method="POST">
    {% csrf_token %}
    <ul>
        {{ form.as_ul }}
    </ul>
    <input type="submit" value="Отправить" >
</form>

<h2>Форма как параграф</h2>
<form method="POST">
    {% csrf_token %}
    <div>
        {{ form.as_p }}
    </div>
```

```
<input type="submit" value="Отправить" >  
</form>  
  
</body>  
</html>
```

После этих изменений при запуске нашего приложения созданные поля на форме будут представлены в трех видах (рис. 6.58).

The screenshot shows a web browser window titled 'Форма Django!' with the URL '127.0.0.1:8000'. The page contains three sections demonstrating different form display styles:

- Форма как таблица**: Shows two text input fields side-by-side, each preceded by its label ('Имя:' and 'Возраст:'). Below them is a 'Отправить' button.
- Форма как список**: Shows two text input fields, each preceded by a bullet point ('• Имя:' and '• Возраст:'). Below them is a 'Отправить' button.
- Форма как параграф**: Shows two text input fields stacked vertically, each preceded by its label ('Имя:' and 'Возраст:'). Below them is a 'Отправить' button.

Рис. 6.58. Текстовые поля на странице index.html представлены в трех разных видах

В первом случае поля формы представлены в виде *таблицы*. При этом идентификаторы полей выровнены по центру, а правая и левая граница каждого поля — между собой. Во втором случае поля формы представлены в виде *списка*. При этом слева от идентификатора поля присутствует признак элемента списка (маркер). В третьем случае поля формы представлены в виде *параграфа* (абзаца). При этом идентификаторы полей и сами поля выровнены по левому краю.

6.4.6. Проверка (валидация) данных

Теоретически пользователь может ввести в форму и отправить какие угодно данные. Однако не все данные бывают уместными или корректными. Например, в поле для возраста пользователь может ввести отрицательное или четырехзначное число, что вряд ли может считаться корректным возрастом. В Django для проверки корректности вводимых данных используется *механизм валидации*.

Основными элементами валидации являются правила, которые определяют параметры корректности вводимых данных. Например, для всех полей по умолчанию устанавливается обязательность ввода значения, а при генерации HTML-кода для поля ввода задается атрибут `required` (обязательное). И когда мы попробуем отправить форму, если в какое-либо из ее полей не введено никакого значения, то получим соответствующее предупреждение. Проверим это на примере. И для начала восстановим код в шаблоне главной страницы `firstapp\index.html`, оставив там отображение полей формы в виде таблицы (листинг 6.47).

Листинг 6.47

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
<h2>Валидация данных</h2>
<form method="POST">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" value="Отправить" >
</form>
</body>
</html>
```

А в программном коде модуля `forms.py` сформируем четыре поля: Имя, Возраст, Электронный адрес и Согласны получать рекламу (листинг 6.48).

Листинг 6.48

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя")
    age = forms.IntegerField(label="Возраст")
    email = forms.EmailField(label="Электронный адрес")
    reklama = forms.BooleanField(label="Согласны получать рекламу")
```

Если одно из полей в форме окажется незаполненным, то при нажатии на кнопку **Отправить** пользователь получит предупреждение об ошибке (рис. 6.59).

Таким образом, чтобы отправить данные, пользователь должен будет обязательно ввести какое-либо значение в незаполненное поле. Однако это не всегда нужно, поскольку одни поля всегда должны быть заполнены, а другие — нет. Предположим,

что в нашем примере пользователь не желает получать рекламу. Однако он не сможет отправить данные до тех пор, пока не поставит флажок в поле **Согласны получать рекламу** (рис. 6.60).

Рис. 6.59. Предупреждение об ошибке (не заполнено текстовое поле)

Рис. 6.60. Невозможность отправки данных при незаполненном поле

Для того чтобы дать пользователю возможность оставить какое-либо поле пустым, необходимо явно отключить для него атрибут `required`. В нашем случае для этого поле `reklama` нужно инициировать с помощью следующего программного кода:

```
reklama = forms.BooleanField(label="Согласны получать рекламу",
                             required=False)
```

Для полей, которые требуют ввода текста, — например, `CharField`, `EmailField` и подобных, иногда требуется ограничивать количество вводимых символов. Это можно сделать с помощью параметров: `max_length` — максимальное количество символов и `min_length` — минимальное количество символов. Так, в поле, созданное с помощью следующего кода:

```
name = forms.CharField(min_length=2, max_length=20)
```

можно будет внести не менее двух и не более двадцати символов.

По аналогии для числовых полей `IntegerField`, `DecimalField` и `FloatField` можно устанавливать параметры: `max_value` и `min_value`, которые задают соответственно максимальное допустимое и минимально допустимое значения. Для поля `DecimalField` дополнительно можно задать еще один параметр — `decimal_places`, который определяет максимальное количество знаков после запятой. Так, в поле, созданное с помощью следующего кода:

```
age = forms.IntegerField(min_value=1, max_value=120)
weight = forms.DecimalField(min_value=3, max_value=200,
                           decimal_places=2)
```

значение возраста допускается вводить в пределах от 1 до 120 лет, а значение веса — в пределах от 3 до 200 кг. При этом количество знаков после десятичной точки может быть не более двух.

Рассмотренные здесь атрибуты позволяют делать проверку введенных значений на стороне клиента. Однако возможны варианты, когда пользователи все же смогут отправить форму с заведомо некорректными данными — например, воспользовавшись инструментами для разработчиков. Чтобы предупредить такое развитие событий, можно подправить исходный код формы, добавив к ней атрибут `novalidate`, который отключает в веб-браузере проверку данных на стороне клиента, и предусмотреть проверку корректности данных на стороне сервера. Для этого у формы вызывается метод `is_valid()`, который возвращает `True`, если данные корректны, и `False` — если данные некорректны. Чтобы использовать этот метод, надо создать объект формы и передать ей пришедшие из запроса данные.

Рассмотрим пример проверки корректности данных на стороне сервера, для чего внесем следующие изменения в программный код модуля `views.py` (листинг 6.49).

Листинг 6.49

```
from django.http import *
from .forms import UserForm
from django.shortcuts import render

def index(request):
    if request.method == "POST":
        userform = UserForm(request.POST)
        if userform.is_valid():
            name = userform.cleaned_data["name"]
            return HttpResponseRedirect("<h2>Имя введено корректно –
{0}</h2>".format(name))
        else:
            return HttpResponseRedirect("Ошибка ввода данных")
    else:
        userform = UserForm()
        return render(request, "firstapp/index.html",
                     {"form": userform})
```

Что делает этот программный код? Если приходит POST-запрос, то вначале происходит заполнение формы пришедшими данными:

```
userform = UserForm(request.POST)
```

Потом с помощью метода `is_valid()` делается проверка их корректности:

```
if userform.is_valid():
```

Если данные введены корректно, то через объект `cleaned_data` в переменную `name` заносим введенное пользователем значение и формируем ответную страницу с сообщением, что данные корректны:

```
name = userform.cleaned_data["name"]
return HttpResponse("<h2>Имя введено корректно –
{0}</h2>".format(name))
```

Если данные введены не корректно, то формируем ответную страницу с сообщением об ошибке:

```
return HttpResponse("Ошибка ввода данных")
```

Если POST-запрос отсутствует, то просто происходит вызов пользовательской формы:

```
userform = UserForm()
return render(request, "firstapp/index.html",
{"form": userform})
```

Для тестирования нашей формы нужно отключить проверку корректности данных на стороне клиента. Для этого откроем шаблон главной страницы `firstapp\index.html` и установим в нем атрибут `novalidate` (листинг 6.50).

Листинг 6.50

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
<h2>Валидация данных</h2>
<form method="POST" novalidate>
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" value="Отправить" >
</form>
</body>
</html>
```

Внесем в программный код последние корректизы — оставим в модуле forms.py всего одно поле с именем клиента (листинг 6.51).

Листинг 6.51

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя")
```

Итак, все готово. Теперь при запуске приложения мы получим нашу форму с одним полем для ввода данных (рис. 6.61).

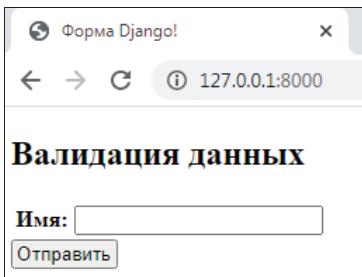


Рис. 6.61. Вид формы при незаполненном поле с именем клиента

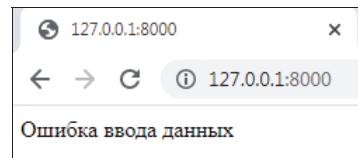


Рис. 6.62. Сообщение об ошибке пользователя со стороны сервера

Оставим поле с именем клиента пустым и нажмем кнопку **Отправить**. Поскольку мы с помощью атрибута novalidate искусственно отключили возможность проверки данных на стороне клиента, то пустое поле с именем отправится на сторону сервера, где оно будет обработано с помощью метода userform.is_valid(). Поскольку поле оказалось незаполненным, то сервер вернет пользователю страницу с сообщением об ошибке (рис. 6.62).

Теперь заполним поле с именем клиента и нажмем кнопку **Отправить**. Поскольку данные были введены корректно, то метод userform.is_valid() вернет пользователю сообщение о корректном вводе данных (рис. 6.63).

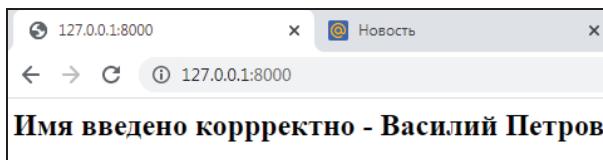


Рис. 6.63. Сообщение о корректности данных, полученное со стороны сервера

6.4.7. Детальная настройка полей формы

Формы и поля допускают установку ряда параметров, которые позволяют частично настраивать отображение полей и форм. Тем не менее этого нередко бывает недос-

таточно. Например, необходимо применить стили или добавить рядом с полем ввода какой-нибудь специальный текст. В Django имеется возможность коренным образом изменить всю композицию создаваемых полей.

В частности, в шаблоне компонента мы можем обратиться к каждому отдельному полю формы через название формы: `form.название_поля`. По названию поля мы можем непосредственно получить генерируемый им HTML-элемент без внешних надписей и какого-то дополнительного кода. Кроме того, каждое поле имеет ряд ассоциированных с ним значений:

- `form.название_поля.name` — возвращает название поля;
- `form.название_поля.value` — возвращает значение поля, которое ему было передано по умолчанию;
- `form.название_поля.label` — возвращает текст метки, которая генерируется рядом с полем;
- `form.название_поля.id_for_label` — возвращает `id` для поля, которое по умолчанию создается по схеме `id_имя_поля`;
- `form.название_поля.auto_id` — возвращает `id` для поля, которое по умолчанию создается по схеме `id_имя_поля`;
- `form.название_поля.label_tag` — возвращает элемент `label`, который представляет метку рядом с полем;
- `form.название_поля.help_text` — возвращает текст подсказки, ассоциированной с полем;
- `form.название_поля.errors` — возвращает ошибки валидации, связанные с полем;
- `form.название_поля.css_classes` — возвращает CSS-классы поля;
- `form.название_поля.as_hidden` — генерирует для поля разметку в виде скрытого поля `<input type="hidden">`;
- `form.название_поля.is_hidden` — возвращает `True` или `False` в зависимости от того, является ли поле скрытым;
- `form.название_поля.as_text` — генерирует для поля разметку в виде текстового поля `<input type="text">`;
- `form.название_поля.as_textarea` — генерирует для поля разметку в виде `<textarea></textarea>`;
- `form.название_поля.as_widget` — возвращает виджет Django, который ассоциирован с полем.

Так, чтобы, например, получить текст метки поля, которое называется `age`, нам надо использовать выражение `form.age.label`.

Рассмотрим возможности применения этих значений полей на простом примере, для чего в модуле `forms.py` создадим два поля с именем и возрастом клиента (листинг 6.52).

Листинг 6.52

```
from django import forms

class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента")
    age = forms.IntegerField(label="Возраст клиента")
```

В представлении `views.py` передадим эту форму в шаблон с использованием следующего кода (листинг 6.53).

Листинг 6.53

```
from django.http import *
from .forms import UserForm
from django.shortcuts import render

def index(request):
    userform = UserForm()
    if request.method == "POST":
        userform = UserForm(request.POST)
        if userform.is_valid():
            name = userform.cleaned_data["name"]
            return HttpResponseRedirect("<h2>Имя введено корректно –
                                         {0}</h2>".format(name))
    return render(request, "firstapp/index.html",
                  {"form": userform})
```

И в шаблоне главной страницы `firstapp\index.html` пропишем следующий код использования полей формы (листинг 6.54).

Листинг 6.54

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Форма Django!</title>
</head>
<body>
<h2>Валидация группы полей</h2>
<form method="POST" novalidate>
    {% csrf_token %}
    <div class="form-group">
        {% for field in form %}
            <div>{{field.label_tag}}</div>
            <div>{{field}}</div>
```

```

<div class="error">{{field.errors}}</div>
{%
    endfor %
}
</div>
<input type="submit" value="Отправить" >
</form>
</body>
</html>

```

В этом шаблоне форма представляет собой набор полей, расположенных внутри цикла `for`. С помощью выражения `{% for field in form %}` в цикле делается проход по всем полям формы, при этом есть возможность управлять отображением как самих полей, так и связанных с ними атрибутов: ошибок, текста подсказки, меток и т. д.

Здесь мы сгруппировали отображение следующих значений полей формы:

- `field.label_tag` — элемент `label`, который представляет метку рядом с полем;
- `field` — само поле;
- `field.errors` — ошибки валидации, связанные с полем.

Если мы теперь запустим наше приложение, то получим следующий вид формы (рис. 6.64).

Рис. 6.64. Вид формы при незаполненных полях с данными о клиенте

Рис. 6.65. Сообщение об ошибке при незаполненных полях с данными о клиенте

Если мы теперь при незаполненных полях нажмем кнопку **Отправить**, то возле каждого поля получим сообщение о наличии ошибки (рис. 6.65).

Если мы оставим незаполненными только часть полей и нажмем кнопку **Отправить**, то сообщение о наличии ошибки будет показано только рядом с незаполненным полем (рис. 6.66).

Рис. 6.66. Сообщение об ошибке только для незаполненного поля с данными о клиенте

Одно поле может содержать несколько ошибок. В этом случае можно использовать тег `for` для их последовательного вывода:

```
{% for error in field.errors %}
    <div class="alert alert-danger">{{error}}</div>
{% endfor %}
```

6.4.8. Присвоение стилей полям формы

Поля формы имеют определенные стили по умолчанию. Если же мы хотим применить к ним какие-то собственные стили и классы, то нам надо использовать ряд заложенных в **Django** механизмов.

Прежде всего, имеется возможность вручную выводить каждое поле и определять правила присвоения стилей ему или окружающим его блокам. В качестве примера в модуле `forms.py` создадим простейшую форму с двумя полями, имеющими некоторые ограничения (листинг 6.55).

Листинг 6.55

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента", min_length=3)
    age = forms.IntegerField(label="Возраст клиента", min_value=1, max_value=100)
```

А в шаблоне главной страницы `firstapp\index.html` пропишем применение к ним собственных стилей (листинг 6.56).

Листинг 6.56

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="utf-8">
    <title>Форма Django!</title>
    <style>
        .alert{color:red}
        .form-group{margin: 10px 0;}
        .form-group input{width:250px;height: 25px;border-radius:10px;}
    </style>
</head>

<body>
<h2>Присвоение стилей полям</h2>
<form method="POST" novalidate>
    {% csrf_token %}
    <div class="form-group">
        {% for field in form %}
            <div>{{field.label_tag}}</div>
            <div>{{field}}</div>
            {% if field.errors%}
                {% for error in field.errors %}
                    <div class="alert alert-danger">
                        {{error}}
                    </div>
                {% endfor %}
            {% endif %}
        {% endfor %}
    </div>
    <input type="submit" value="Отправить"
    </form>
</body>
</html>
```

Здесь в заголовке head задан стиль alert{color:red} (красный цвет для отображения ошибок) и стили вывода полей, которые объединены в группы (отступы, ширина и высота рамки, радиус округления углов).

Если после этих изменений запустить наше приложение, то будет видно, что рамки имеют округлую форму, а сообщение об ошибках выдаются красным цветом (рис. 6.67).

В Django имеется еще один механизм для присвоения стилей формам — через использование свойств формы: required_css_class и error_css_class. Эти свойства применяют CSS-класс к метке, создаваемой для поля формы, и к блоку ассоциированных с ним ошибок. Рассмотрим применение этого механизма на следующем примере. Для этого в модуль forms.py внесем следующие изменения — добавим два класса (листинг 6.57).

Форма Django! 127.0.0.1:8000

Присвоение стилей полям

Имя клиента:

Обязательное поле.

Возраст клиента:

Обязательное поле.

Отправить

Рис. 6.67. Сообщение об ошибках в форме с данными о клиенте, полям которой присвоены собственные стили

Листинг 6.57

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента", min_length=3)
    age = forms.IntegerField(label="Возраст клиента", min_value=1, max_value=100)
    required_css_class = "field"
    error_css_class = "error"
```

В этом случае в шаблоне главной страницы `firstapp\index.html` у нас должны быть определены классы `field` и `error` (выделены серым фоном и полужирным шрифтом), как показано в листинге 6.58.

Листинг 6.58

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Форма Django!</title>
    <style>
        .field{font-weight:bold;}
        .error{color:red;}
    </style>
</head>

<body class="container">
    <h2>Стилизация полей</h2>
    <form method="POST" novalidate>
```

```

{< csrf_token %}
<table>
{&form%}
</table>
<input type="submit" value="Оправить" >
</form>
</body>
</html>

```

Если после этих изменений запустить наше приложение, то будет видно, что метки полей выводятся полужирным шрифтом черного цвета (рис. 6.68).

Рис. 6.68. Вид формы с присвоенными стилями при незаполненных полях с данными о клиенте

Если теперь нажать кнопку **Отправить**, оставив поля незаполненными, то сообщения о наличии ошибки будут показаны красным цветом (рис. 6.69).

Если же мы оставим незаполненными только часть полей и нажмем кнопку **Отправить**, то сообщение о наличии ошибки будет показано лишь рядом с незаполненным полем (рис. 6.70).

Рис. 6.69. Сообщение об ошибках с данными о клиенте в форме с присвоенными стилями

Рис. 6.70. Сообщение об ошибке с частично заполненными данными о клиенте в форме с присвоенными стилями

В Django имеется и третий механизм присвоения стилей формам — через установку стилей в виджетах. Рассмотрим это на примере. Для этого в создающие поля кодовые строки модуля `forms.py` внесем следующие изменения (листиng 6.59).

Листинг 6.59

```
from django import forms

class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента",
                           widget=forms.TextInput(attrs={"class": "myfield"}))
    age = forms.IntegerField(label="Возраст клиента",
                            widget=forms.NumberInput(attrs={"class": "myfield"}))
```

В этом коде через параметр виджетов `attrs` задаются атрибуты того HTML-элемента, который будет генерироваться. В частности, здесь для обоих полей устанавливается атрибут `class`, который представляет класс `myfield`.

Теперь нужно в шаблоне главной страницы `firstapp\index.html` определить класс `myfield` (выделено серым фоном и полужирным шрифтом), как показано в листинге 6.60.

Листинг 6.60

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Форма Django!</title>
    <style>
        .myfield{
            border:2px solid #ccc;
            border-radius:5px;
            height:25px;
            width:200px;
            margin: 10px 10px 10px 0;
        }
    </style>
</head>

<body class="container">
<h2>Присвоение стилей полям</h2>
<form method="POST">
    {% csrf_token %}
    <div>
        {% for field in form %}
```

```
<div class="row">
    {{field.label_tag}}
    <div class="col-md-10">{{field}}</div>
</div>
{%
endfor %}
</div>
<input type="submit" value="Отправить" >
</form>
</body>
</html>
```

Если после этих изменений запустить наше приложение, то форма будет выведена в следующем виде (рис. 6.71).

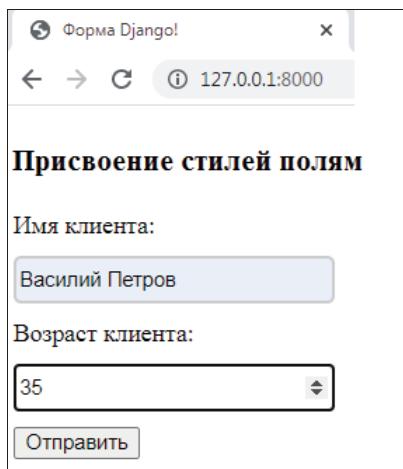


Рис. 6.71. Вид формы со стилями, присвоенными через параметры виджетов

6.5. Краткие итоги

В этой главе были представлены сведения о формах. Мы узнали, как создать форму, как определить и настроить в форме поля, как проверить корректность введенных в форму данных. Теперь нам нужно разобраться с тем, как Django работает с самими данными. То есть понять, как можно сохранить в БД данные, которые пользователь ввел в форме, или как получить из БД данные, запрошенные пользователем через свой веб-браузер. В Django все действия с информацией, которая хранится в БД, осуществляются через так называемые *модели данных*, или просто *модели*. Изучению моделей данных посвящена следующая глава. В ней рассматриваются методы работы с данными через модели, не прибегая к SQL-запросам — традиционному способу взаимодействия с СУБД.

ГЛАВА 7



Модели данных Django

Практически любое веб-приложение, так или иначе, работает с базой данных. При этом с системой управления базами данных (СУБД) приложение общается каким-нибудь универсальным способом — например, посредством языка SQL. Однако программисту чаще всего хочется иметь некую абстракцию, позволяющую большую часть времени работать с привычными сущностями языка. Такой абстракцией является Object-Relational Mapping (ORM) — отображение сущностей предметной области и их взаимосвязей в объекты, удобные для использования программистом.

Имеются разные подходы к тому, как нужно изолировать пользователя от конкретного хранилища данных. Есть такие, которые полностью скрывают всю работу с БД — вы пользуетесь объектами, изменяете их состояние, а ORM неявно синхронизирует состояние объектов и сущностей в хранилище. Другие ORM всего лишь оборачивают сущности БД в структуры языка, но все запросы нужно писать вручную. Это два разных полюса, каждый со своими плюсами и минусами. Авторы Django решили остаться где-то посередине.

Используя Django ORM, вы работаете с объектами и выполняете вручную их загрузку и сохранение, однако используете для этого привычные средства языка — вызовы методов и свойств. При этом Django же берет на себя обеспечение правильной работы вашего приложения с конкретными хранилищами данных. Эта изоляция от конкретного хранилища позволяет использовать разные БД в разных условиях: при разработке и тестировании задействовать легковесные СУБД, а при развертывании сайтов в реальных условиях применять более мощные и производительные базы данных.

Термин «модель» часто используется в качестве замены словосочетания «Django ORM», поэтому мы будем им пользоваться и в рамках этой книги. «Модель» говорит нам о том, что наша предметная область смоделирована с помощью средств фреймворка. При этом отдельные сущности тоже называются *моделями* — модели поменьше собираются в большую «Модель» всей предметной области.

Связь между моделями и таблицами в БД максимально прямая: одна таблица — одна модель. В этом плане Django ORM не отходит далеко от схемы БД — вы все-

где имеете представление о том, как фактически описаны ваши данные с точки зрения СУБД. Что очень полезно, когда нужно что-либо где-то оптимизировать!

Из материалов этой главы вы узнаете:

- какие типы полей можно использовать в модели данных Django;
- какие существуют методы для работы с данными в Django (добавление, чтение, обновление, удаление данных из БД);
- как можно манипулировать с данными на конкретных примерах работы с объектами модели данных;
- как организовать различные типы связей между таблицами в модели данных.

7.1. Создание моделей и миграции базы данных

Модели в Django описывают структуру используемых в программе данных. Эти данные хранятся в базах данных, и с помощью моделей как раз осуществляется взаимодействие с такими базами.

По умолчанию Django в качестве базы данных задействует SQLite. Она очень проста в использовании и не требует запущенного сервера. Все файлы базы данных могут легко переноситься с одного компьютера на другой. Однако при необходимости мы можем использовать в Django большинство других распространенных СУБД.

Для работы с базами данных в проекте Django в файле `settings.py` определен параметр `DATABASES`, который по умолчанию выглядит следующим образом:

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

Конфигурация используемой базы данных в таком случае складывается из двух параметров. Параметр `ENGINE` указывает на применяемый для доступа к БД движок. В нашем случае это встроенный пакет `django.db.backends.sqlite3`. Второй параметр — `NAME` указывает на имя и путь к базе данных. После первого запуска проекта в нем по умолчанию будет создан файл `db.sqlite3`, который, собственно, и будет служить в качестве базы данных.

Чтобы использовать другие системы управления базами данных, необходимо установить соответствующий пакет (табл. 7.1).

При создании приложения по умолчанию в его каталог добавляется файл `models.py`, который применяется для определения и описания моделей. Модель представляет собой класс, унаследованный от `django.db.models.Model`.

Таблица 7.1. Дополнительные пакеты для работы *Django* с различными СУБД

СУБД	Пакет	Команда установки
PostgreSQL	psycopg2	pip install psycopg2
MySQL	mysql-python	pip install mysql-python
Oracle	cx_Oracle	pip install cx_Oracle

Рассмотрим процесс создания модели на простом примере и создадим модель, описывающую клиента, имеющего две характеристики: имя (`name`) и возраст (`age`). Изменим файл `models.py` следующим образом (листинг 7.19), как показано на рис. 7.1.

Листинг 7.1

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()
```

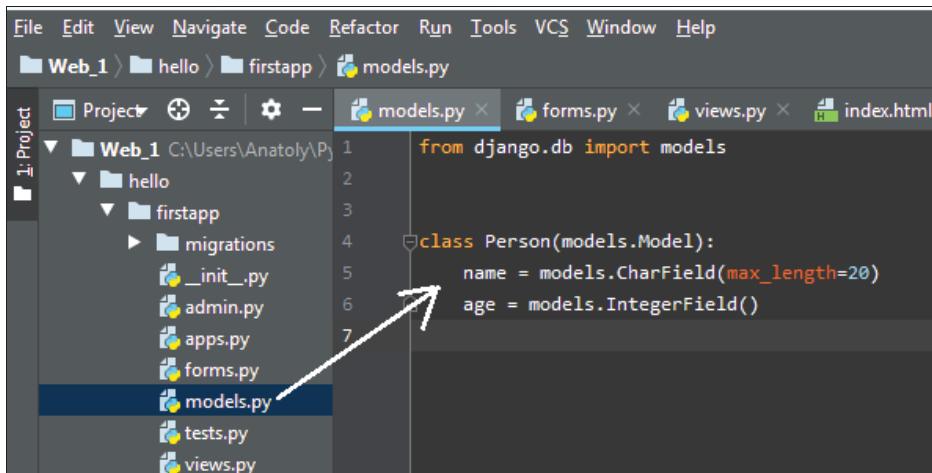


Рис. 7.1. Добавление кода в файл models.py приложения firstapp

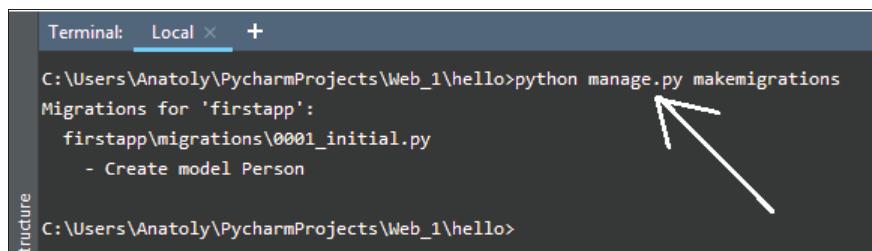
Здесь в программном коде определена простейшая модель с именем `Person`, представляющая характеристики человека (клиента системы). В модели определены два поля для хранения имени и возраста. Поле `name` представляет тип `CharField` — текстовое поле, которое хранит последовательность символов. В нашем случае — имя клиента. Для `CharField` указан параметр `max_length`, задающий максимальную длину хранящейся строки. Поле `age` представляет тип `IntegerField` — числовое поле, которое хранит целые числа. Оно предназначено для хранения возраста человека.

Каждая модель сопоставляется с определенной таблицей в базе данных. Однако пока у нас нет в БД ни одной таблицы, которая бы хранила объекты модели Person. На следующем шаге нам надо создать такую таблицу. В Django это делается с помощью *миграции* — процесса внесения изменений в базу данных в соответствии с определениями в модели.

Миграция формируется в два шага. На первом шаге необходимо создать миграцию (файл с параметрами миграции) с помощью команды:

```
python manage.py makemigrations
```

Откроем окно терминала PyCharm и выполним эту команду (рис. 7.2).



```
Terminal: Local +  
C:\Users\Anatoly\PycharmProjects\Web_1\hello>python manage.py makemigrations  
Migrations for 'firstapp':  
    firstapp\migrations\0001_initial.py  
        - Create model Person  
C:\Users\Anatoly\PycharmProjects\Web_1\hello>
```

Рис. 7.2. Создание миграции модели данных в окне терминала PyCharm

Как можно видеть, после выполнения этой команды создан файл с миграцией firstapp\migrations\0001_initial.py и получено сообщение: **Create model Person** (создана модель Person).

Новый файл 0001_initial.py, расположенный в папке migrations нашего приложения firstapp, будет иметь примерно следующее содержимое (рис. 7.3):

```
from django.db import migrations, models  
  
class Migration(migrations.Migration):  
  
    initial = True  
  
    dependencies = [  
    ]  
  
    operations = [  
        migrations.CreateModel(  
            name='Person',  
            fields=[  
                ('id', models.AutoField(auto_created=True,  
                                         primary_key=True, serialize=False,  
                                         verbose_name='ID')),  
                ('name', models.CharField(max_length=20)),  
                ('age', models.IntegerField()),  
            ],  
        ),  
    ]
```

```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
Web_1 > hello > firstapp > migrations > 0001_initial.py
Project hello firstapp migrations 0001_initial.py models.py forms.py views.py index
1 # Generated by Django 3.0.8 on 2020-07-28 10:51
2
3 from django.db import migrations, models
4
5
6 class Migration(migrations.Migration):
7     initial = True
8
9     dependencies = [
10 ]
11
12     operations = [
13         migrations.CreateModel(
14             name='Person',
15             fields=[
16                 ('id', models.AutoField(auto_created=True)),
17                 ('name', models.CharField(max_length=20)),
18                 ('age', models.IntegerField()),
19             ],
20         ),
21     ],
22

```

Рис. 7.3. Окно PyCharm: описание полей модели данных в файле миграции 0001_initial.py

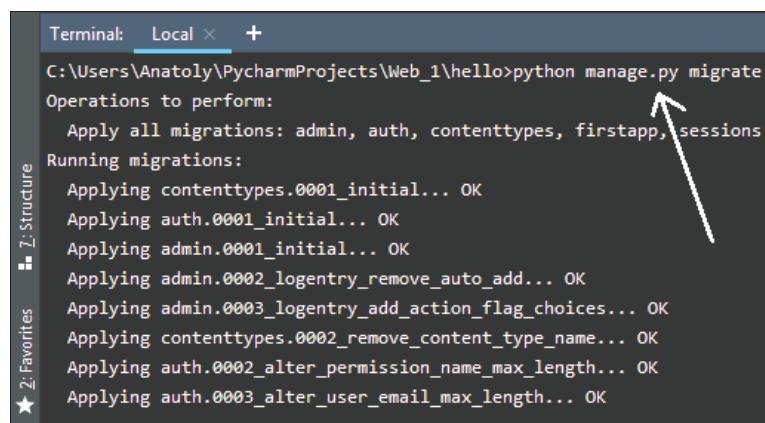
Это и есть результаты первого шага миграции. Здесь можно заметить, что в процессе миграции создано не два поля, как было описано в модели, а три — появилось дополнительное поле `id`, которое будет представлять первичный ключ в таблице базы данных. Такое поле добавляется автоматически по умолчанию. Поэтому в Django в самой модели не требуется для таблиц базы данных явным образом определять ключевые поля — они будут создаваться автоматически. На первом шаге было создано только описание полей для таблицы `Person`, в которой будут храниться данные о клиентах, при этом сама таблица в БД еще отсутствует.

На втором шаге миграции необходимо на основе описания полей в модели данных создать соответствующую таблицу в БД. Это делается с помощью команды:

```
python manage.py migrate
```

Откроем окно терминала PyCharm и выполним эту команду (рис. 7.4).

Теперь, если мы откроем присутствующую в проекте базу данных `db.sqlite3` с помощью программы для просмотра БД (например, SQLiteStudio), то увидим, что она содержит ряд таблиц (рис. 7.5). В основном это все служебные таблицы. Но нас интересует наша таблица `Person`, в которой будут храниться данные о клиентах. В Django имена таблиц формируются автоматически, при этом имя таблицы состоит из двух частей: из имени приложения и имени модели. В нашем случае таблица будет иметь имя `firstapp_person`.



```
Terminal: Local × +  
C:\Users\Anatoly\PycharmProjects\Web_1\hello>python manage.py migrate  
Operations to perform:  
  Apply all migrations: admin, auth, contenttypes, firstapp, 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
```

Рис. 7.4. Окно терминала PyCharm: создание таблиц в БД при миграции модели данных

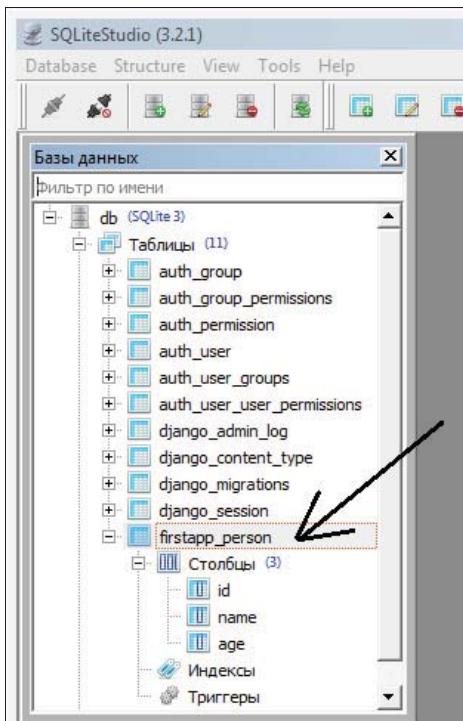


Рис. 7.5. Окно приложения
SQLiteStudio:
таблица firstapp_person,
созданная в БД при миграции
модели данных

7.2. Типы полей в модели данных Django

В Django для определения моделей данных можно использовать следующие типы полей:

- `BinaryField()` — хранит бинарные данные;
- `BooleanField()` — хранит значение `True` или `False` (0 или 1);

- NullBooleanField() — хранит значение True, False или Null;
- DateField() — хранит дату;
- TimeField() — хранит время;
- DateTimeField() — хранит дату и время;
- DurationField() — хранит период времени (интервал времени);
- AutoField() — хранит целочисленное значение, которое автоматически инкрементируется (увеличивается на 1). Обычно применяется для первичных ключей;
- BigIntegerField() — представляет целое число (значение типа Number), которое укладывается в диапазон от -9223372036854775808 до 9223372036854775807 (в зависимости от выбранной СУБД диапазон может немного отличаться);
- DecimalField(decimal_places=X, max_digits=Y) — представляет значение числа с дробной частью (типа Number), которое имеет максимум X разрядов и Y знаков после запятой;
- FloatField() — хранит значение типа Number, которое представляет число с плавающей точкой;
- IntegerField() — хранит значение типа Number, которое представляет целочисленное значение;
- PositiveIntegerField() — хранит значение типа Number, которое представляет положительное целочисленное значение (от 0 до 2147483647);
- PositiveSmallIntegerField() — хранит значение типа Number, которое представляет небольшое положительное целочисленное значение (от 0 до 32767);
- SmallIntegerField() — хранит значение типа Number, которое представляет небольшое целочисленное значение (от -32768 до 32767);
- CharField(max_length=N) — хранит строку длиной не более N символов;
- TextField() — хранит строку неопределенной длины;
- EmailField() — хранит строку, которая представляет email-адрес (значение автоматически проверяется встроенным валидатором EmailValidator);
- FileField() — хранит строку, которая представляет имя файла;
- FilePathField() — хранит строку, которая представляет путь к файлу длиной в 100 символов;
- ImageField() — хранит строку, которая представляет данные об изображении;
- GenericIPAddressField() — хранит строку, которая представляет IP-адрес в формате IP4v или IP6v;
- SlugField() — хранит строку, которая может содержать только буквы в нижнем регистре, цифры, дефис и знак подчеркивания;
- URLField() — хранит строку, которая представляет валидный URL-адрес;
- UUIDField() — хранит строку, которая представляет UUID-идентификатор.

Сопоставление типов полей в моделях данных Django с типами полей в различных СУБД приведено в табл. 7.2.

Таблица 7.2. Сопоставление типов полей в моделях данных Django с типами полей в различных СУБД

Тип	SQLite	MySQL	PostgreSQL	Oracle
BinaryField()	BLOB NOT NULL	longblob NOT NULL	bytea NOT NULL	BLOB NULL
BooleanField()	bool NOT NULL	bool NOT NULL	boolean NOT NULL	NUMBER(1) NOT NULL CHECK("Значение" IN(0,1))
NullBooleanField()	bool NULL	bool NULL	boolean NULL	NUMBER(1) NOT NULL CHECK("Значение" IN(0,1)) OR ("Значение" IS NULL))
DateField()	date NULL	date NULL	date NULL	DATE NOT NULL
TimeField()	time NULL	time NULL	time NULL	TIMESTAMP NOT NULL
DateTimeField()	datetime NULL	datetime NULL	timestamp NULL	TIMESTAMP NOT NULL
DurationField()	bigrnt NOT NULL	bigrnt NOT NULL	interval NOT NULL	INTERVAL DAY(9) TO SECOND(6) NOT NULL
AutoField()	integer NOT NULL AUTOINCREMENT NOT NULL	integer AUTO_INCREMENT NOT NULL	serial NOT NULL	NUMBER(11) NOT NULL
BigIntegerField	bigrnt NOT NULL	bigrnt NOT NULL	bigrnt NOT NULL	NUMBER(19) NOT NULL
DecimalField(decimal_places=X, max_digits=Y)	decimal NOT NULL	numeric(X, Y) NOT NULL	numeric(X, Y) NOT NULL	NUMBER(10, 3) NOT NULL
FloatField	real NOT NULL	double precision NOT NULL	double precision NOT NULL	DOUBLE PRECISION NOT NULL
IntegerField	integer NOT NULL	integer NOT NULL	integer NOT NULL	NUMBER(11) NOT NULL
PositiveIntegerField	integer unsigned NOT NULL	integer UNSIGNED NOT NULL	integer NOT NULL CHECK ("Значение" > 0)	NUMBER NOT NULL CHECK ("Значение" > 0)
PositiveSmallIntegerField	smallint unsigned NOT NULL	smallint UNSIGNED NOT NULL	smallint NOT NULL CHECK ("Значение" > 0)	NUMBER(11) NOT NULL CHECK ("Значение" > 0)
SmallIntegerField	smallint NOT NULL	smallint NOT NULL	smallint NOT NULL	NUMBER(11) NOT NULL
CharField(max_length=N)	varchar(N) NOT NULL	varchar(N) NOT NULL	varchar(N) NOT NULL	NVARCHAR2(N) NULL
TextField()	text NOT NULL	longtext NOT NULL	text NOT NULL	NCLOB NULL

Таблица 7.2 (окончание)

Тип	SQLite	MySQL	PostgreSQL	Oracle
EmailField()	varchar(254) NOT NULL	varchar(254) NOT NULL	varchar(254) NOT NULL	NVARCHAR2(254) NULL
FileField()	varchar(100) NOT NULL	varchar(100) NOT NULL	varchar(100) NOT NULL	NVARCHAR2(100) NULL
FilePathField()	varchar(100) NOT NULL	varchar(100) NOT NULL	varchar(100) NOT NULL	NVARCHAR2(100) NULL
ImageField()	varchar(100) NOT NULL	varchar(100) NOT NULL	varchar(100) NOT NULL	NVARCHAR2(100) NULL
GenericIPAddressField()	char(39) NOT NULL	char(39) NOT NULL	inet NOT NULL	VARCHAR2(39) NULL
SlugField()	varchar(50) NOT NULL	varchar(50) NOT NULL	varchar(50) NOT NULL	NVARCHAR2(50) NULL
URLField()	varchar(200) NOT NULL	varchar(200) NOT NULL	varchar(200) NOT NULL	NVARCHAR2(200) NULL
UUIDField()	char(32) NOT NULL	char(32) NOT NULL	uuid NOT NULL	VARCHAR2(32) NULL

7.3. Манипуляция с данными в Django на основе CRUD

Аббревиатура CRUD обозначает четыре основные операции, которые используются при работе с базами данных. Этот термин представляет собой сочетание первых букв английских слов: создание (Create), чтение (Read), модификация (Update) и удаление (Delete). Это, по своей сути, стандартная классификация функций для манипуляций с данными. В SQL этим функциям соответствуют операторы Insert (создание записей), Select (чтение записей), Update (редактирование записей) и Delete (удаление записей).

В Django при создании моделей данных они наследуют свое поведение от класса `django.db.models.Model`, который предоставляет базовые операции с данными (добавление, чтение, обновление и удаление).

Рассмотрим выполнение этих операций с данными на примере модели `Person`, которая была создана в предыдущих разделах:

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()
```

7.3.1. Добавление данных в БД

В Django для добавления данных в БД можно использовать два метода: `create()` (создать) и `save()` (сохранить).

Для добавления данных методом `create()` для нашей модели можно использовать следующий код:

```
igor = Person.objects.create(name="Игорь", age=23)
```

Если добавление пройдет успешно, то объект будет иметь `id`, который можно получить через `igor.id`.

Однако, в своей сути, метод `create()` использует другой метод — `save()`, и его мы также можем использовать самостоятельно для добавления объекта в БД:

```
igor = Person(name="Игорь", age=23)  
igor.save()
```

После успешного добавления данных в БД можно аналогично получить идентификатор добавленной записи — через `igor.id` и сами добавленные значения — через `igor.name` и `igor.age`.

7.3.2. Чтение данных из БД

В Django получить значение данных из БД можно несколькими методами:

- `get()` — для одного объекта;
- `get_or_create()` — для одного объекта с добавлением его в БД;
- `all()` — для всех объектов;
- `filter()` — для группы объектов по фильтру;
- `exclude()` — для группы объектов с исключением некоторых;
- `in_bulk()` — для группы объектов в виде словаря.

Методы `all()`, `filter()` и `exclude()` возвращают объект `QuerySet`. Это, по сути, некое промежуточное хранилище, в котором содержится информация, полученная из БД. Объект `QuerySet` может быть создан, отфильтрован и затем использован фактически без выполнения запросов к базе данных. База данных не будет затронута, пока вы не инициируете новое выполнение `QuerySet`. Рассмотрим последовательно все указанные методы.

Метод `get()`

Метод `get()` возвращает один объект — т. е. одну запись из БД по определенному условию, которое передается в качестве параметра. Пусть, например, в программе имеется следующий код:

```
klient1 = Person.objects.get(name="Виктор")  
klient2 = Person.objects.get(age=25)  
klient3 = Person.objects.get(name="Василий", age=23)
```

В этом случае в переменную `klient1` будет считан объект, у которого поле `name="Виктор"`, в переменную `klient2` — объект, у которого поле `age=25`. Соответственно, в переменную `klient3` будет считан объект, у которого поле `name="Василий"` и поле `age=23`.

При использовании этого метода надо учитывать, что он предназначен для выборки таких объектов, которые имеются в базе данных в единственном числе. Если в таблице не окажется подобного объекта, то будет выдана ошибка `имя_модели.DoesNotExist`. Если же в таблице присутствуют несколько объектов, которые соответствуют указанному условию, то будет сгенерировано исключение `MultipleObjectsReturned`. Поэтому следует применять этот метод с достаточной осторожностью.

Метод `get_or_create()`

Метод `get_or_create()` получает объект из БД, а если его там нет, то он будет добавлен в БД как новый объект. Рассмотрим следующий код:

```
bob, created = Person.objects.get_or_create(name="Bob", age=24)
print(bob.name)
print(bob.age)
```

Этот код вернет добавленный в БД объект (в нашем случае — переменную `bob`) и булево значение (`created`), которое будет иметь значение `True`, если добавление прошло успешно.

Метод `all()`

Если необходимо получить все имеющиеся объекты из базы данных, то применяется метод `all()` — например:

```
people = Person.objects.all()
```

Метод `filter()`

Если требуется получить все объекты, которые соответствуют определенному критерию, то применяется метод `filter()`, который в качестве параметра принимает критерий выборки. Например:

```
people = Person.objects.filter(age=23)
people2 = Person.objects.filter(name="Tom", age=23)
```

Здесь в `people` будут помещены все объекты из БД, для которых `age=23`, а в `people2` — все объекты с параметрами `name="Tom"` и `age=23`.

Метод `exclude()`

Метод `exclude()` позволяет выбрать из БД все записи, за исключением тех, которые соответствуют переданному в качестве параметра критерию:

```
people = Person.objects.exclude(age=23)
```

Здесь в `people` будут помещены все объекты из БД, за исключением тех, у которых `age=23`.

Можно комбинировать оба эти метода:

```
people = Person.objects.filter(name="Tom").exclude(age=23)
```

Здесь в `people` будут помещены все объекты из БД с именем `name="Tom"`, за исключением тех, у которых `age=23`.

Метод `in_bulk()`

Метод `in_bulk()` является наиболее эффективным способом для чтения большого количества записей. Он возвращает словарь, т. е. объект `dict`, тогда как методы `all()`, `filter()` и `exclude()` возвращают объект `QuerySet`.

Все объекты из БД можно получить с помощью следующего кода:

```
people = Person.objects.in_bulk()
```

Для получения доступа к одной из выбранных из БД записей нужно использовать идентификатор записи в словаре:

```
for id in people:  
    print(people[id].name)  
    print(people[id].age)
```

С помощью метода `in_bulk()` можно получить и часть объектов из БД. Например, в следующем программном коде из БД будут получены только те объекты, у которых ключевые значения полей равны 1 и 3:

```
people2 = Person.objects.in_bulk([1,3])  
for id in people2:  
    print(people2[id].name)  
    print(people2[id].age)
```

Здесь метод `in_bulk` возвращает словарь, где ключи представляют `id` объектов, а значения по этим ключам — собственно эти объекты, т. е., в нашем случае, объекты `Person`.

7.3.3. Обновление данных в БД

Для обновления объекта в БД применяется метод `save()`. При этом Django полностью обновляет объект и все его свойства, даже если мы их не изменили. Например:

```
nic = Person.objects.get(id=2)  
nic.name = "Николай Петров"  
nic.save()
```

Здесь мы сначала в переменную `nic` прочитали из БД все данные о человеке, информация о котором хранится в БД в записи с `id=2`. Затем во второй строке изменили содержимое поля `name`. И наконец, в третьей строке вернули (записали) всю информацию об этом человеке в БД.

Когда нужно обновить только определенные поля, следует использовать параметр `update_fields`. Так, если, например, в приведенном примере требуется изменить только одно поле `name`, это можно сделать с помощью следующего кода:

```
nic = Person.objects.get(id=2)
nic.name = "Николай Петров"
nic.save(update_fields=["name"])
```

Такой подход позволяет повысить скорость работы приложения, особенно в тех случаях, когда требуется обновить большой массив информации.

Другой способ обновления объектов в БД предоставляет метод `update()` в сочетании с методом `filter()`, которые вместе выполняют один запрос к базе данных. Предположим, что нам нужно обновить имя клиента в записи таблицы БД с `id=2`. Это можно сделать с помощью следующего кода:

```
Person.objects.filter(id=2).update(name="Михаил")
```

При таком подходе не нужно предварительно получать из БД обновляемый объект, что обеспечивает увеличение скорости взаимодействия приложения с БД.

Иногда возникает необходимость изменить значение столбца в БД на основании уже имеющегося значения. В этом случае мы можем использовать функцию `F()`:

```
from django.db.models import F
Person.objects.all(id=2).update(age = F("age") + 1)
```

Здесь полю `age` присваивается уже имеющееся значение, увеличенное на единицу. При этом важно учитывать, что метод `update` обновляет все записи в таблице, которые соответствуют условию (в приведенном примере таким условием является `id=2`).

Когда необходимо обновить все записи в столбце таблицы БД вне зависимости от условия, то надо комбинировать метод `update()` с методом `all()` без параметров:

```
from django.db.models import F
Person.objects.all().update(name="Михаил")
Person.objects.all().update(age = F("age") + 1)
```

Здесь всем клиентам будет присвоено значение Михаил, и возраст всех клиентов будет увеличен на единицу.

Для обновления записей в БД есть еще один метод — `update_or_create`. Если запись существует, то этот метод ее обновит, а если записи нет, то добавит ее в таблицу:

```
values_for_update={"name": "Михаил", "age": 31}
bob, created = Person.objects.update_or_create(id=2,
                                                defaults = values_for_update)
```

Метод `update_or_create()` здесь принимает два параметра. Первый параметр предоставляет критерий выборки объектов, которые должны обновляться. Второй параметр предоставляет объект со значениями, которые будут переданы записям, соответствующим критерию из первого параметра. Если соответствующих критерию записей обнаружено не будет, в таблицу добавится новый объект, а переменной

created присвоено значение True. В приведенном примере критерием для обновления записи является идентификатор записи id=2. А поля, которые будут обновлены: "name": "Михаил" и "age": 31.

7.3.4. Удаление данных из БД

Для удаления информации из БД используется метод `delete()`. Удаление единственной записи из таблицы можно выполнить с помощью ее `id`. Например:

```
person = Person.objects.get(id=2)
person.delete()
```

Здесь в первой строке в элемент `person` загружена информация из строки таблицы базы данных с `id=2`, а во второй строке вызван метод, который удалил из таблицы БД эту строку.

Если не требуется получение отдельного объекта из базы данных, тогда можно удалить объект одной строкой программного кода с помощью комбинации методов `filter()` и `delete()`:

```
Person.objects.filter(id=4).delete()
```

Этой командой будет удалена строка с `id=4` непосредственно в базе данных, без предварительной загрузки ее содержимого в приложение.

7.3.5. Просмотр строки SQL-запроса к базе данных

Рассмотренные здесь методы при обращении к базе данных фактически используют SQL-запросы, хотя это и скрыто от разработчика. В Django с помощью свойства `query` можно получить и посмотреть текст выполняемого SQL-запроса. Например, при выполнении кода:

```
people = Person.objects.filter(name="Tom").exclude(age=34)
print(people.query)
```

на консоли отобразится следующий SQL-запрос:

```
SELECT "firstapp_person"."id", "firstapp_person"."name",
       "firstapp_person"."age"
  FROM "firstapp_person"
 WHERE ("firstapp_person"."name" = Tom
       AND NOT ("firstapp_person"."age" = 34))
```

7.4. Пример работы с объектами модели данных (чтение и запись информации в БД)

Рассмотрим работу с объектами модели данных на простом примере. Для этого воспользуемся моделью данных `Person`, которую мы создали в предыдущих разделах. Эта модель данных хранится в файле `models.py` (листинг 7.2).

Листинг 7.2

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()
    objects = models.Manager()
```

Согласно приведенной модели в БД предусмотрено два поля для хранения сведений о клиенте: имя клиента (`name`) и возраст клиента (`age`). Кроме того, при выполнении миграции в базе данных (в файле `db.sqlite3`) была создана таблица для хранения сведений о клиентах с именем `firstapp_person` и тремя полями: `id` — идентификатор записи, `name` — имя клиента и `age` — возраст клиента.

В Django за взаимодействие с БД отвечает представление (view). Так что на следующем шаге мы в файле `views.py` реализуем две функции: для получения сведений из БД и для сохранения данных, введенных пользователем (листинг 7.3).

Листинг 7.3

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from .models import Person

# получение данных из БД и загрузка index.html
def index(request):
    people = Person.objects.all()
    return render(request, "index.html", {"people": people})

# сохранение данных в БД
def create(request):
    if request.method == "POST":
        klient = Person()
        klient.name = request.POST.get("name")
        klient.age = request.POST.get("age")
        klient.save()
    return HttpResponseRedirect("/")
```

Здесь нами созданы две функции: `index()` и `create()`. В функции `index()` мы получаем все данные в объект `people` с помощью метода `Person.objects.all()` и затем передаем их в шаблон `index.html`. В функции `create()` мы получаем из запроса типа POST данные, которые пользователь ввел в форму, затем сохраняем их в БД с помощью метода `save()` и выполняем через функцию `index()` их переадресацию на главную страницу веб-сайта — т. е. на `index.html`.

Теперь в папке `templates` определим шаблон главной страницы `index.html`, в которой будут отображаться введенные пользователем данные (листинг 7.4).

Листинг 7.4

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Модели в Django</title>
</head>
<body class="container">
    <form method="POST" action="create/">
        {% csrf_token %}
        <p>
            <label>Введите имя</label><br>
            <input type="text" name="name" />
        </p>
        <p>
            <label>Введите возраст</label><br>
            <input type="number" name="age" />
        </p>
        <input type="submit" value="Сохранить" />
    </form>
    {% if people.count > 0 %}
        <h2>Список пользователей</h2>
        <table>
            <tr><th>Id</th><th>Имя</th><th>Возраст</th></tr>
            {% for person in people %}
                <tr><td>{{ person.id }}</td>
                    <td>{{ person.name }}</td>
                    <td>{{ person.age }}</td>
                </tr>
            {% endfor %}
        </table>
    {% endif %}
</body>
</html>
```

И наконец, в файле urls.py свяжем маршруты с представлениями (листинг 7.5).

Листинг 7.5

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

urlpatterns = [
    path('', views.index),
    path('create/', views.create),
]
```

Что ж, у нас все готово, и можно запустить приложение. При первом запуске мы получим страницу, которая будет иметь вид, представленный на рис. 7.6.

Когда же пользователь заведет данные нескольких клиентов (пользователей), информация о них будет считана из БД и отображена на главной странице (рис. 7.7).

Модели в Django
127.0.0.1:8000

Введите имя

Введите возраст

Рис. 7.6. Начальный вид страницы index.html

Модели в Django
Новость
← → С 127.0.0.1:8000

Введите имя

Введите возраст

Список пользователей

Id	Имя	Возраст
1	Климов Игорь	55
2	Житомирский Владимир	55
3	Шестов Алексей	50
4	Чурилко Николай	60

Рис. 7.7. Вид страницы index.html после ввода данных о нескольких пользователях

Если мы теперь с использованием менеджера SQLiteStudio откроем содержимое таблицы firstapp_person, то увидим в ней данные о клиентах, которые пользователь занес в БД с помощью созданного нами приложения (рис. 7.8).

SQLiteStudio (3.2.1)

Database Structure View Tools Help

Базы данных

Фильтр по имени

db (SQLite3)

- Таблицы (11)
 - auth_group
 - auth_group_permissions
 - auth_permission
 - auth_user
 - auth_user_groups
 - auth_user_user_permissions
 - django_admin_log
 - django_content_type
 - django_migrations
 - django_session
 - firstapp_person
- Столбцы (3)

firstapp_person (db)

Структура Данные Ограничения Индексы

Табличный вид | Форма |

id	name	age
1	Климов Игорь	55
2	Житомирский Владимир	55
3	Шестов Алексей	50
4	Чурилко Николай	60

Рис. 7.8. Информация о пользователях в базе данных SQLite

Как можно видеть, мы добились своей цели, т. е. занесли информацию в БД через интерфейс HTML-страницы. При этом мы использовали два метода: `Person.objects.all()` (получение информации из БД) и `klient.save()` (запись информации о клиентах в БД).

7.5. Пример работы с объектами модели данных: редактирование и удаление информации из БД

Рассмотрим пример с редактированием и удалением объектов модели. Для этого продолжим работу с проектом из предыдущего раздела. Для начала добавим в файл `views.py` функции, которые будут выполнять редактирование и удаление информации из БД (листинг 7.6).

Листинг 7.6

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from .models import Person

# получение данных из БД и загрузка index.html
def index(request):
    people = Person.objects.all()
    return render(request, "index.html", {"people": people})

# сохранение данных в БД
def create(request):
    if request.method == "POST":
        klient = Person()
        klient.name = request.POST.get("name")
        klient.age = request.POST.get("age")
        klient.save()
    return HttpResponseRedirect("/")

# изменение данных в БД
def edit(request, id):
    try:
        person = Person.objects.get(id=id)

        if request.method == "POST":
            person.name = request.POST.get("name")
            person.age = request.POST.get("age")
            person.save()
        return HttpResponseRedirect("/")
    
```

```

else:
    return render(request, "edit.html", {"person": person})
except Person.DoesNotExist:
    return HttpResponseRedirect("<h2>Клиент не найден</h2>")

# удаление данных из БД
def delete(request, id):
    try:
        person = Person.objects.get(id=id)
        person.delete()
        return HttpResponseRedirect("/")
    except Person.DoesNotExist:
        return HttpResponseRedirect("<h2>Клиент не найден</h2>") .

```

Первые две функции: `index()` и `create()` — здесь остаются без изменений.

Функция `edit()` выполняет редактирование объекта. Она в качестве параметра принимает идентификатор объекта `id` из базы данных. В начале по этому идентификатору мы пытаемся найти объект в БД с помощью метода `Person.objects.get(id=id)`. Поскольку в случае отсутствия объекта мы можем столкнуться с исключением `Person.DoesNotExist` (объект не найден), то нужно обработать это исключение (если по каким-либо причинам мы получим несуществующий идентификатор). И если объект найден не будет, пользователю вернется сообщение об ошибке 404 — через вызов метода `return HttpResponseRedirect()`.

Когда объект найден, обработка будет делиться на две ветви. Если поступит запрос POST, т. е. пользователь отправил новые (измененные) данные для объекта, мы сохраняем эти данные в БД и выполняем переадресацию на главную страницу сайта. Если поступит запрос GET, отображаем пользователю страницу `edit.html` с формой для редактирования объекта.

Функция `delete()` аналогичным образом находит объект и выполняет его удаление.

В непрофессиональной версии PyCharm в программном коде может появиться подсветка, предупреждающая о наличии ошибки (рис. 7.9). Такое предупреждение не повлияет на работу программы. Однако его можно убрать, добавив одну строчку в код модуля `models.py` (в листинге 7.7 она выделена серым фоном и полужирным шрифтом).

Листинг 7.7

```

from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()
    objects = models.Manager()
DoesNotExist = models.Manager

```

```
index.html x models.py x views.py x urls.py x
30         person.age = request.POST.get("age")
31         person.save()
32         return HttpResponseRedirect("/")
33     else:
34         return render(request, "edit.html", {"person": person})
35     except Person.DoesNotExist: ←
36         return HttpResponseRedirect("<h2>Person not found</h2>")
37
38
39 # удаление данных из бд
40 def delete(request, id):
41     try:
42         person = Person.objects.get(id=id)
43         person.delete()
44         return HttpResponseRedirect("/")
45     except Person.DoesNotExist: ←
46         return HttpResponseRedirect("<h2>Person not found</h2>")
```

Рис. 7.9. Предупреждение о наличии возможной ошибки

Теперь нужно создать HTML-страницу, в которой пользователь может редактировать данные. Добавим в папку templates файл edit.html со следующим содержимым (листинг 7.8).

Листинг 7.8

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Модели в Django</title>
</head>
<body class="container">
<form method="POST">
    {% csrf_token %}
    <p>
        <label>Введите имя</label><br>
        <input type="text" name="name" value="{{person.name}}" />
    </p>
    <p>
        <label>Введите возраст</label><br>
        <input type="number" name="age" value="{{person.age}}" />
    </p>
    <input type="submit" value="Сохранить" >
</form>
</body>
</html>
```

Чтобы обеспечить редактирование и удаление объектов непосредственно на главной странице, изменим шаблон index.html, где выводится список объектов (клиентов), добавив в него ссылки на страницы редактирования и удаления объектов из БД (листинг 7.9).

Листинг 7.9

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Модели в Django</title>
</head>
<body class="container">
    <form method="POST" action="create/">
        {% csrf_token %}
        <p>
            <label>Введите имя</label><br>
            <input type="text" name="name" />
        </p>
        <p>
            <label>Введите возраст</label><br>
            <input type="number" name="age" />
        </p>
        <input type="submit" value="Сохранить" />
    </form>
    {% if people.count > 0 %}
        <h2>Список пользователей</h2>
        <table>
            <thead>
                <th>Id</th><th>Имя</th><th>Возраст</th><th></th>
            </thead>
            {% for person in people %}
            <tr>
                <td>{{ person.id }}</td>
                <td>{{ person.name }}</td>
                <td>{{ person.age }}</td>
                <td><a href="edit/{{person.id}}">Изменить</a> |<a href="delete/{{person.id}}">Удалить</a></td>
            </tr>
            {% endfor %}
        </table>
        {% endif %}
    </body>
</html>
```

Затем в файле urls.py сопоставим функции edit и delete с маршрутами (листинг 7.10).

Листинг 7.10

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

urlpatterns = [
    path('', views.index),
    path('create/', views.create),
    path('edit/<int:id>/', views.edit),
    path('delete/<int:id>/', views.delete),
]
```

Запустим проект. Теперь на главной странице рядом с каждым объектом (клиентом) появятся две ссылки: **Изменить** и **Удалить** (рис. 7.10).

Id	Имя	Возраст	
1	Климов Игорь	55	Изменить Удалить
2	Житомирский Владимир	55	Изменить Удалить
3	Шестов Алексей	50	Изменить Удалить
4	Чурилко Николай	60	Изменить Удалить

Рис. 7.10. Главная страница с возможностью редактирования сведений о клиентах

Ведите имя	<input type="text" value="Климов Игорь"/>
Ведите возраст	<input type="text" value="60"/>
<input type="button" value="Сохранить"/>	

Рис. 7.11. Страница для изменения сведений о клиенте

По нажатию на ссылку **Изменить** откроется страница для редактирования сведений о клиенте. Изменим возраст одного из них с 55 лет на 60 (рис. 7.11).

После того, как будет нажата кнопка **Сохранить**, мы вернемся на главную страницу, где будет видно, что параметр клиента **Возраст** действительно изменился (рис. 7.12).

Осталось проверить, действительно ли в базу данных были внесены эти изменения. Откроем содержимое таблицы `firstapp_person` с помощью SQLiteStudio. Действительно, в БД произошло изменение содержимого поля `age` (рис. 7.13).

Теперь нажмем на главной странице на ссылку **Удалить** для клиента **Игорь Климов** с `id = 1` — сведения о нем исчезнут с главной страницы (рис. 7.14).

Модели в Django

Новость

127.0.0.1:8000

Введите имя
Введите возраст
Сохранить

Список пользователей

Id	Имя	Возраст	Изменить	Удалить
1	Климов Игорь	60	Изменить	Удалить
2	Житомирский Владимир	55	Изменить	Удалить
3	Шестов Алексей	50	Изменить	Удалить
4	Чурилко Николай	60	Изменить	Удалить

Рис. 7.12. Главная страница после редактирования возраста клиента

SQLiteStudio (3.2.1)

Database Structure View Tools Help

Базы данных

Фильтр по имени

db (SQLite 3)

Таблицы (11)

- auth_group
- auth_group_permissions
- auth_permission
- auth_user
- auth_user_groups
- auth_user_user_permissions
- django_admin_log
- django_content_type
- django_migrations
- django_session
- firstapp_person

firstapp_person (db)

Структура Данные Ограничения Индексы

Табличный вид | Форма |

id	name	age
1	Климов Игорь	60
2	Житомирский Владимир	55
3	Шестов Алексей	50
4	Чурилко Николай	60

Рис. 7.13. Новое значение возраста клиента в БД после его редактирования на HTML-странице

Модели в Django

Новость

127.0.0.1:8000

Введите имя
Введите возраст
Сохранить

Список пользователей

Id	Имя	Возраст	Изменить	Удалить
2	Житомирский Владимир	55	Изменить	Удалить
3	Шестов Алексей	50	Изменить	Удалить
4	Чурилко Николай	60	Изменить	Удалить

Рис. 7.14. Главная страница после удаления клиента с идентификатором id = 1

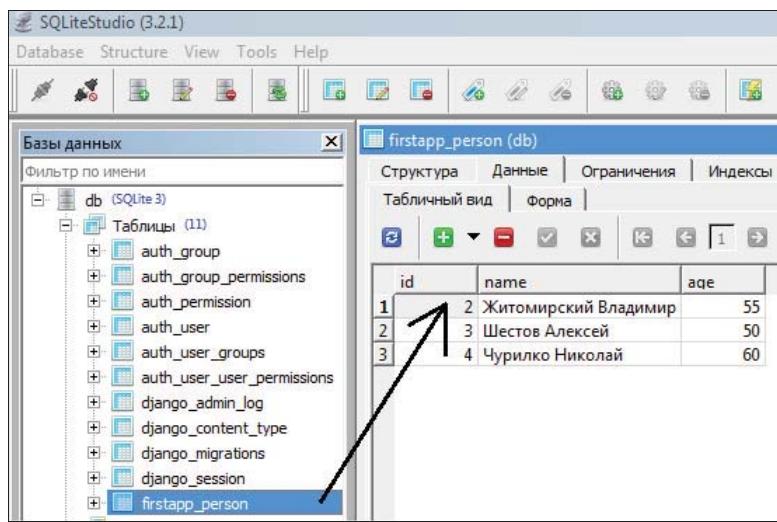


Рис. 7.15. Клиент с `id=1` после его удаления через HTML-страницу в БД больше нет

В том, что клиент с `id=1` действительно удален из БД, можно убедиться, если открыть таблицу `forstapp_person` с помощью SQLiteStudio (рис. 7.15).

7.6. Организация связей между таблицами в модели данных

Большинство таблиц в базе данных имеют связи между собой. Django позволяет определить три наиболее употребительных типа отношений: «один-ко-многим», «многие-ко-многим» и «один-к-одному».

7.6.1. Организация связей между таблицами «один-ко-многим»

Рассмотрим организацию связи между таблицами БД «один-ко-многим», при которой одна главная сущность может быть связаны с несколькими зависимыми сущностями. Приведем несколько примеров таких связей:

- одна компания, которая выпускает множество видов товаров;
- один автомобиль, который состоит из множества составных частей;
- одна гостиница, которая имеет множество комнат с разными характеристиками;
- одна книга, у которой несколько авторов;
- один город, который имеет множество улиц;
- одна улица, которая имеет множество домов, и т. п.

Покажем, как можно связать две таблицы в БД через связанные модели на примере «одна компания — множество товаров»:

```
from django.db import models

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

class Product(models.Model):
    company = models.ForeignKey(Company, on_delete = models.CASCADE)
    name = models.CharField(max_length=30)
    price = models.IntegerField()
```

В этом примере модель `Company` представляет собой производителя продукции и является *главной моделью* (главной таблицей в БД), а модель `Product` представляет собой различные товары, производимые этой компанией, и является *зависимой моделью* (зависимой таблицей в БД).

Конструктор типа `models.ForeignKey` в классе `Product` настраивает связь с главной сущностью. Здесь первый параметр указывает, с какой моделью будет создаваться связь, — в нашем случае это модель `Company`. Второй параметр (`on_delete`) задает опцию удаления объекта текущей модели при удалении связанного объекта главной модели. В частности, в приведенном коде задано *каскадное удаление* (`models.CASCADE`). То есть если из БД будет удалена компания, то автоматически из БД будет удалена и вся продукция, которая выпускается этой компанией.

Всего для параметра `on_delete` могут использовать следующие значения:

- `models.CASCADE` — автоматически удаляет строку (строки) из зависимой таблицы, если удаляется связанная строка из главной таблицы;
- `models.PROTECT` — блокирует удаление строки из главной таблицы, если с ней связаны какие-либо строки в зависимой таблице;
- `models.SET_NULL` — устанавливает значение `NULL` при удалении связанной строки из главной таблицы;
- `models.SET_DEFAULT` — устанавливает значение по умолчанию для внешнего ключа в зависимой таблице (в таком случае для этого столбца должно быть задано значение по умолчанию);
- `models.DO_NOTHING` — при удалении связанной строки из главной таблицы не производится никаких действий в зависимой таблице.

Итак, внесем соответствующие изменения в файл `models.py` (листинг 7.11).

Листинг 7.11

```
from django.db import models

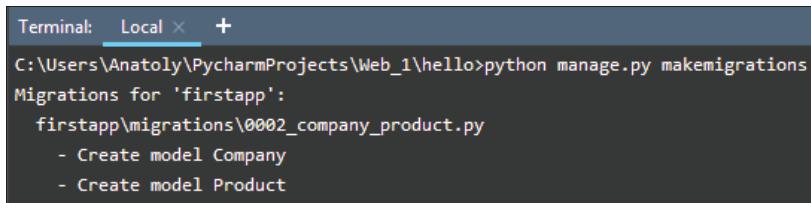
class Person(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()
    objects = models.Manager()
    DoesNotExist = models.Manager()
```

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

class Product(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    name = models.CharField(max_length=30)
    price = models.IntegerField()
```

И выполним миграцию с использованием команды, которую запустим в окне терминала PyCharm (рис. 7.16):

```
python manage.py makemigrations
```



Terminal: Local +
C:\Users\Anatoly\PycharmProjects\Web_1\hello>python manage.py makemigrations
Migrations for 'firstapp':
 firstapp\migrations\0002_company_product.py
 - Create model Company
 - Create model Product

Рис. 7.16. Создание миграции для моделей данных Company и Product

В результате выполнения этой команды на основе моделей Company и Product в каталоге migrations автоматически будет создан новый файл 0002_company_product.py со следующим содержанием:

```
from django.db import migrations, models
import django.db.models.deletion

class Migration(migrations.Migration):

    dependencies = [
        ('firstapp', '0001_initial'),
    ]

    operations = [
        migrations.CreateModel(
            name='Company',
            fields=[
                ('id', models.AutoField(auto_created=True,
                                      primary_key=True, serialize=False,
                                      verbose_name='ID')),
                ('name', models.CharField(max_length=30)),
            ],
        ),
        migrations.CreateModel(
            name='Product',
```

```

fields=[

    ('id', models.AutoField(auto_created=True,
                           primary_key=True, serialize=False,
                           verbose_name='ID')),
    ('name', models.CharField(max_length=30)),
    ('price', models.IntegerField()),
    ('company', models.ForeignKey(
        on_delete=django.db.models.deletion.CASCADE,
        to='firstapp.Company')),

],
),
]

```

Разберемся, какие действия прописаны в этом файле. Во-первых, его номер 0002 говорит о том, что это уже вторая миграция. Во-вторых — что эта миграция является зависимой от файла с первичной миграцией:

```
dependencies = [('firstapp', '0001_initial'),]
```

А далее следует код для создания двух моделей данных:

- модели таблицы данных о компании — с именем name='Company' и двумя полями: 'id' и 'name';
- модели таблицы данных о продукции компании — с именем name='Product' и четырьмя полями: 'id', 'name', 'price' и 'company'.

При этом указано, что в таблице со сведениями о продуктах компании поле company является связующим с главной таблицей, и предусмотрено каскадное удаление всех записей для случая, если компания будет удалена из главной таблицы.

Теперь внесем эти изменения в саму базу данных, для чего в окне терминала PyCharm (рис. 7.17) выполним команду:

```
python manage.py migrate
```

```

Terminal: Local × +
C:\Users\Anatoly\PycharmProjects\Web_1\hello>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, firstapp, sessions
Running migrations:
  Applying firstapp.0002_company_product... OK

C:\Users\Anatoly\PycharmProjects\Web_1\hello>

```

Рис. 7.17. Создание таблиц в БД на основе миграции моделей данных Company и Product

В результате миграции на основе моделей Company и Product в базе данных SQLite автоматически будут созданы таблицы: firstapp_company и firstapp_product (рис. 7.18).

В сформированных таблицах автоматически созданы ключевые поля для идентификации записей (`id`), а также в таблице firstapp_product создано поле `company_id` для связи этой дочерней таблицы с родительской таблицей Company.

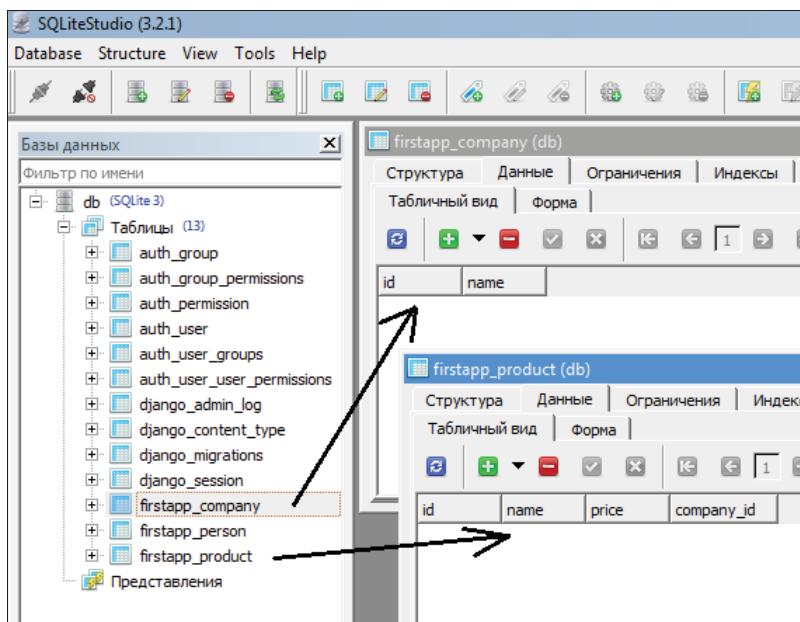


Рис. 7.18. Таблицы, созданные в БД на основе миграции моделей данных Company и Product

На уровне таблиц в БД мы видим, что таблица `Product` связана с таблицей `Company` через столбец "company_id". Однако в самом определении модели `Product` нет поля `company_id`, а есть только поле `company`, и именно через него в программном коде можно получать связанные данные. Например, если требуется получить идентификатор компании, которая производит этот продукт, нужно воспользоваться командой:

```
id_company = Product.objects.get(id=1).company.id
```

Если требуется получить название компании, которая производит этот продукт, нужно воспользоваться командой:

```
name_company = Product.objects.get(id=1).company.name
```

Так, если требуется получить перечень товаров, которые производятся компанией «Мираторг», следует воспользоваться командой:

```
Product.objects.filter(company__name="Мираторг")
```

Здесь нужно обратить особое внимание на выражение `company__name`. С помощью выражения `model__ свойство` (обратите внимание: два подчеркивания!) можно использовать свойство главной модели для фильтрации объектов (записей в таблице БД) зависимой модели.

С точки зрения модели `Company` (родительская таблица в БД) она не имеет никаких свойств, которые бы связывали ее с моделью `Product` (дочерняя таблица в БД). Но с помощью команды, имеющей следующий синтаксис:

```
"главная_модель"."зависимая_модель"_set
```

можно изменить направление связи. То есть на основании записи из главной модели вы сможете получать связанные записи из зависимой (подчиненной) модели. Рассмотрим следующий программный код:

```
from .models import Company, Product
firma = Company.objects.get(name="Электрон")
    # получение всех товаров фирмы "Электрон"
tovar = firma.product_set.all()
    # получение количества товаров фирмы "Электрон"
kol_tovar = firma.product_set.count()
    # получение товаров, название которых начинается на "Ноутбук"
Tovar = firma.product_set.filter(name__startswith="Ноутбук")
```

Здесь в переменную `firma` из объекта `Company` (а фактически из таблицы БД `Company`) мы получаем сведения о компании с именем «Электрон». Затем в переменную `tovar` считываем сведения обо всех продуктах фирмы «Электрон» (по сути, из таблицы БД `firstapp_product`).

Причем с помощью выражения `_set` можно выполнять операции добавления, изменения, удаления объектов зависимой модели из главной модели. Рассмотрим следующий программный код (листинг 7.12).

Листинг 7.12

```
# создаем объект Company с именем Электрон
firma = Company.objects.create(name="Электрон")

# создание товара компании
firma.product_set.create(name="Samsung S20", price=42000)

# отдельное создание объекта с последующим добавлением в БД
ipad = Product(name="iPad", price=34200)
# при добавлении необходимо указать параметр bulk =False
firma.product_set.add(ipad, bulk =False)

# исключает из компании все товары,
# при этом товары остаются в БД и не привязаны к компании
# работает, если в зависимой модели ForeignKey(Company, null = True)
# firma.product_set.clear()

# то же самое, только в отношении одного объекта
# ipad = Product.objects.get(name="iPad")
# firma.product_set.remove(ipad)
```

Стоит отметить три метода, которые могут быть использованы в сочетании с выражением `_set`:

- `add()` — добавляет как саму запись в дочернюю таблицу, так и связь между объектом зависимой модели и объектом главной модели. По своей сути метод `add()`

вызывает для модели еще и метод `update()` для добавления связи. Однако это требует, чтобы обе модели уже были в базе данных. И здесь применяется параметр `bulk=False` — чтобы объект зависимой модели сразу был добавлен в БД и для него была установлена связь;

- `clear()` — удаляет связь между всеми объектами зависимой модели и объектом главной модели. При этом сами объекты зависимой модели (дочерней таблицы) остаются в базе данных, и для их внешнего ключа устанавливается значение `NULL`. Поэтому метод `clear()` будет работать, если в самой зависимой модели при установке связи использовался параметр `null=True`, т. е. `ForeignKey(Company, null = True)`;
- `remove()` — так же, как и `clear()`, удаляет связь, только между одним объектом зависимой модели и объектом главной модели. При этом также все объекты дочерней таблицы остаются в БД. И также в самой зависимой модели при установке связи должен использоваться параметр `null=True`.

7.6.2. Организация связей между таблицами «многие-ко-многим»

Связь «многие-ко-многим» — это связь, при которой множественным записям из одной таблицы (A) могут соответствовать множественные записи из другой таблицы (B). Примером такой связи может служить учебное заведение, где преподаватели обучают учащихся. В большинстве учебных заведений (школа, университет) каждый преподаватель обучает многих учащихся, а каждый учащийся может обучаться у нескольких преподавателей. Еще один пример — это книги и авторы книг. У одной книги может быть несколько авторов, в то же время у одного автора может быть несколько книг.

Связь «многие-ко-многим» создается с помощью трех таблиц: две из них (A и B) — «источники» и одна таблица — соединительная. Первичный ключ соединительной таблицы (A–B) — составной. Она состоит из двух полей: двух внешних ключей, которые ссылаются на первичные ключи таблиц A и B. Все первичные ключи должны быть уникальными. Это подразумевает и то, что комбинация полей A и B должна быть уникальной в таблице A–B.

Еще одним примером такой связи являются номера гостиницы и ее гости. Между таблицами «Гости» и «Комнаты» (или номера) существует связь «многие-ко-многим»: одна комната может быть заказана многими гостями в течение определенного времени, и в течение этого же промежутка времени гость может заказать в гостинице разные комнаты. Однако соединительная таблица в таком случае может не являться классической соединительной таблицей, состоящей только из двух внешних ключей. Она может быть отдельной сущностью с дополнительными полями, имеющей связи с двумя другими сущностями (при этом уникальность ключей должна соблюдаться).

Вследствие природы отношения «многие-ко-многим» совершенно неважно, какая из таблиц является родительской, а какая — дочерней, потому что по своей сути такой тип отношений является симметричным.

Для представления отношения «многие-ко-многим» Django самостоятельно создает промежуточную связывающую таблицу. По умолчанию имя этой таблицы образуется из имен двух соединяемых таблиц.

Рассмотрим, как можно связать две таблицы в БД через связанные модели на примере: «много учебных курсов — много студентов». Для создания отношения «многие-ко-многим» применяется тип связи `ManyToManyField`. Итак, добавим в файл `models.py` следующий код (листинг 7.13).

Листинг 7.13

```
from django.db import models

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

class Student(models.Model):
    name = models.CharField(max_length=30)
    courses = models.ManyToManyField(Course)
```

В этом примере модель `Course` представляет собой учебные курсы, а модель `Student` — студентов. Здесь нет родительской и дочерней таблицы, они равнозначны.

Новая сущность `courses`, устанавливающая отношение «многие-ко-многим», создается с использованием конструктора `models.ManyToManyField`. В результате генерируется промежуточная таблица, через которую, собственно, и будет осуществляться связь.

Теперь запустим в окне терминала PyCharm (рис. 7.19) миграцию с использованием команды:

```
python manage.py makemigrations
```

```
Terminal: Local × +
C:\Users\Anatoly\PycharmProjects\Web_1\hello>python manage.py makemigrations
Migrations for 'firstapp':
  firstapp\migrations\0003_course_student.py
    - Create model Course
    - Create model Student
```

Рис. 7.19. Создание миграции для моделей данных `Course` и `Student`

В результате выполнения этой команды на основе моделей `Course` и `Student` в каталоге `migrations` автоматически будет создан новый файл `0003_course_student.py`, имеющий следующее содержание:

```
operations = [
    migrations.CreateModel(
        name='Course',
```

```
fields=[('id', models.AutoField(auto_created=True,
                                primary_key=True, serialize=False,
                                verbose_name='ID')),
        ('name', models.CharField(max_length=30)),
    ],
),
migrations.CreateModel(
    name='Student',
    fields=[('id', models.AutoField(auto_created=True,
                                primary_key=True, serialize=False,
                                verbose_name='ID')),
        ('name', models.CharField(max_length=30)),
        ('courses',
         models.ManyToManyField
         (to='firstapp.Course')),
    ],
),
]
]
```

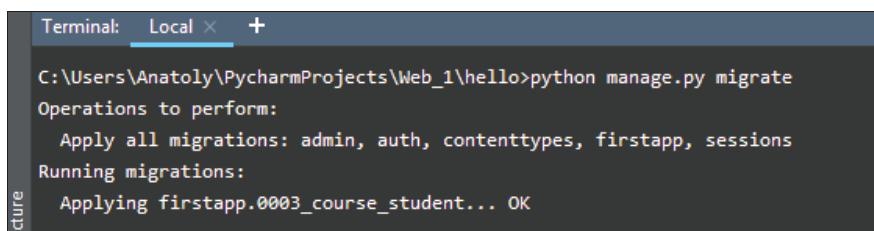
Здесь прописан код для создания двух моделей данных:

- модели таблицы данных о курсах — с именем `name='Course'` и двумя полями: `'id'` и `'name'`;
- модели таблицы данных о студентах — с именем `name='Student'` и двумя полями: `'id'` и `'name'`.

При этом указано, что между таблицами имеется связующая таблица, которая обеспечивает связь «многие-ко-многим», и к этим связям можно обращаться через свойство `courses`.

Теперь внесем эти изменения в саму базу данных, для чего в окне терминала PyCharm (рис. 7.20) выполним команду:

```
python manage.py migrate
```



The screenshot shows a PyCharm terminal window titled "Terminal: Local". The command `python manage.py migrate` is entered and executed. The output shows the following steps:
Operations to perform:
Apply all migrations: admin, auth, contenttypes, firstapp, sessions
Running migrations:
Applying firstapp.0003_course_student... OK

Рис. 7.20. Создание таблиц в БД на основе миграции моделей данных Course и Student

В результате миграции на основе моделей `Course` и `Student` в базе данных SQLite автоматически будут созданы уже не две, а три таблицы: `firstapp_course`, `firstapp_student` и `firstapp_course_student` (рис. 7.21).

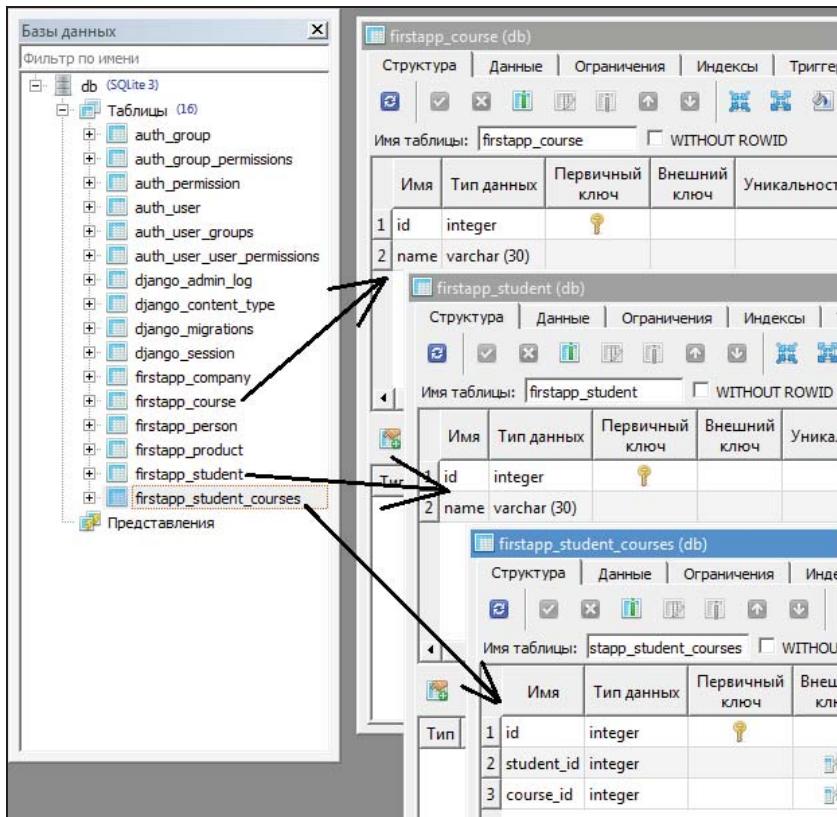


Рис. 7.21. Три таблицы, созданные в БД на основе миграции моделей данных с множественными связями Course и Student

В нашем случае таблица с именем `firstapp_student_courses` выступает в качестве связующей таблицы. Ей автоматически присвоено имя по шаблону:

`имя_приложения+имя_таблицы+имя_связующего_поля_из_таблицы`

А в самой связующей таблице имеются всего два поля с ключами из двух связанных таблиц: `student_id` и `course_id`.

Через свойство `courses` в модели `Student` мы можем получать связанные со студентом курсы и управлять ими. Рассмотрим следующий код:

```
# создадим студента с именем Виктор
stud_viktor = Student.objects.create(name="Виктор")

# создадим один курс и добавим в него Виктора
stud_viktor.courses.create(name="Математика")

# получим все курсы студента Виктора
all_courses = Student.objects.get(name="Виктор").courses.all()

# получаем всех студентов, которые посещают курс Математика
all_student = Student.objects.filter(courses__name="Математика")
```

Стоит обратить внимание на последнюю строку кода, где производится выборка студентов по посещаемому курсу. Для передачи в метод `filter()` имени курса используется параметр, название которого начинается с названия свойства, через которое идет связь со второй моделью (`courses`). И далее через два знака подчеркивания указывается имя свойства второй модели — например, `courses__name` или `courses__id`. Иными словами, мы можем получить информацию о курсах студента через свойство `courses`, которое определено в модели `Student`.

Однако имеется возможность получать информацию и о студентах, которые изучают определенные курсы. В этом случае надо использовать синтаксис `_set`. Рассмотрим следующий программный код:

```
# создадим курс программирования на Python
kurs_python = Course.objects.create(name="Python")

# создаем студента и добавляем его на курс
kurs_python.student_set.create(name="Виктор")

# отдельно создаем студента и добавляем его на курс
alex = Student(name="Александр")
alex.save()
kurs_python.student_set.add(alex)

# получим всех студентов курса
students = kurs_python.student_set.all()

# получим количество студентов по курсу
number = kurs_python.student_set.count()

# удаляем с курса одного студента
kurs_python.student_set.remove(alex)

# удаляем всех студентов с курса
kurs_python.student_set.clear()
```

Стоит учитывать, что не всегда такая организация связи «многие-ко-многим» может подойти. Например, в нашем случае создается промежуточная таблица, которая хранит только `id` студента и `id` курса. Если нам надо в промежуточной таблице хранить еще какие-либо данные — например, дату зачисления студента на курс, его оценки и т. д., то такая конфигурация не подойдет. И тогда правильнее будет создать промежуточную сущность вручную (например, запрос или хранимую процедуру), которая связана отношением «один-ко-многим» с обеими моделями.

7.6.3. Организация связей между таблицами «один-к-одному»

При организации связи «один-к-одному» каждая запись из таблицы А может быть ассоциирована только с одной записью таблицы В. Связь «один-к-одному» легко

моделируется в одной таблице. Записи такой таблицы содержат данные, которые находятся в связи «один-к-одному» с первичным ключом.

В редких случаях связь «один-к-одному» моделируется с использованием двух таблиц. Такой вариант иногда необходим, чтобы преодолеть ограничения СУБД, или с целью увеличения производительности (производится, например, вынесение ключевого поля в отдельную таблицу для ускорения поиска по другой таблице). Или вы сами захотите разнести две сущности, имеющие связь «один-к-одному», по разным таблицам. Например, всю базовую информацию о пользователе (имя, возраст, электронный адрес и пр.) выделить в одну модель, а его учетные данные (логин, пароль, время последнего входа в систему, количество неудачных входов и т. п.) — в другую. Но обычно наличие двух таблиц в связи «один-к-одному» считается плохой практикой.

Рассмотрим, как можно связать две таблицы в БД через связанные модели на примере: «пользователь системы — учетные данные пользователя». Для создания отношения «один-к-одному» применяется тип связи `models.OneToOneField()`. Добавим в файл `models.py` следующий код (листинг 7.14).

Листинг 7.14

```
from django.db import models

class User(models.Model):
    name = models.CharField(max_length=20)

class Account(models.Model):
    login = models.CharField(max_length=20)
    password = models.CharField(max_length=20)
    user = models.OneToOneField(User,
                                on_delete = models.CASCADE,
                                primary_key = True)
```

Здесь мы создали модель пользователя (`User`) с одним полем `name` и модель учетных данных пользователя (`Account`) с двумя полями: `login` и `password`. Для создания отношения «один-к-одному» был применен конструктор типа `models.OneToOneField()`. Его первый параметр указывает, с какой моделью будет ассоциирована эта сущность (в нашем случае ассоциация с моделью `User`). Второй его параметр (`on_delete=models.CASCADE`) говорит, что данные текущей модели (`Account`) будут удаляться в случае удаления связанного объекта главной модели (`User`). Третий параметр (`primary_key=True`) указывает, что внешний ключ (через который идет связь с главной моделью) одновременно будет выступать и в роли первичного ключа. И соответственно, создавать отдельное поле для первичного ключа.

Теперь внесем эти изменения в саму базу данных, для чего выполним в окне терминала PyCharm последовательно две команды:

```
python manage.py makemigrations
python manage.py migrate
```

В результате миграции в базе данных SQLite будут созданы следующие таблицы: таблица `firstapp_user` с полями `id` и `name` и таблица `firstapp_account` с полями `login`, `password` и `user_id` (рис. 7.22).

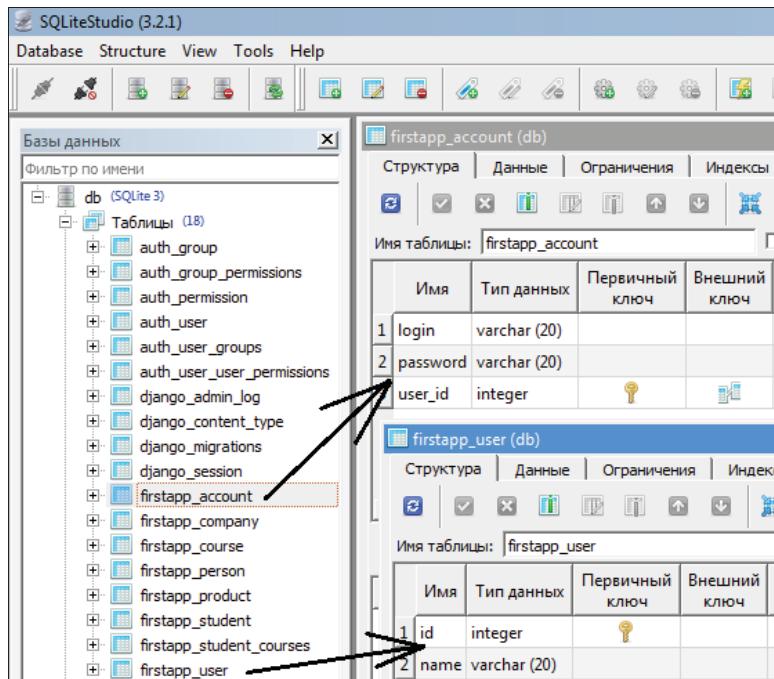


Рис. 7.22. Таблицы, созданные в БД на основе миграции моделей данных со связью «один-к-одному» на основе моделей User И Account

Как можно видеть, в таблице `firstapp_account` нет собственного первичного ключа (`id`) — его роль выполняет ключ `user_id`, который одновременно служит для связи с таблицей `firstapp_user`.

С помощью свойства `users` в модели `Account` мы можем манипулировать связанным объектом модели `User`:

```
# создадим пользователя Александр
alex = User.objects.create(name="Александр")

# создадим аккаунт пользователя Александр
acc = Account.objects.create(login = "1234",
                             password="6565", user= alex)

# изменяем имя пользователя
acc.user.name = "Саша"
# сохраняем изменения в БД
acc.user.save()
```

При этом через модель `User` мы также можем оказывать влияние на связанный объект `Account`. Несмотря на то что явным образом в модели `User` определено только

одно свойство — `name`, при связи «один-к-одному» неявно создается еще одно свойство, которое называется по имени зависимой модели и указывает на связанный объект этой модели. То есть в нашем случае это свойство будет называться `account`:

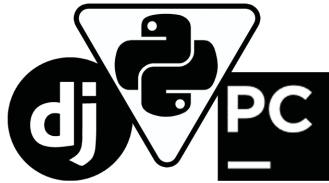
```
# создадим пользователя Александр
alex = User.objects.create(name="Александр")

# создадим аккаунт пользователя
acc = Account(login = "1234", password="6565")
alex.account = acc
alex.account.save()

# обновляем данные
alex.account.login = "qwerty"
alex.account.password = "123456"
alex.account.save()
```

7.7. Краткие итоги

В этой главе были представлены сведения о моделях и о том, как на основе моделей создать таблицы в БД. Мы узнали, как с использованием моделей создавать, читать, редактировать и удалять записи из БД. Выяснили, что через модели можно устанавливать связи между таблицами. То есть сейчас у нас есть все необходимые знания для того, чтобы создать свой собственный сайт. Это позволяет нам перейти к следующей главе, в которой на небольшом и достаточно понятном примере расписаны все шаги по созданию сайта. При этом мы не станем концентрироваться на дизайне веб-страниц, а больше внимания уделим технологическим вопросам и использованию различных возможностей Django.



ГЛАВА 8

Пример создания веб-сайта на Django

В этой главе и последующих, опираясь на теоретические положения и практические примеры, с которыми мы познакомились в предыдущих главах книги, мы попытаемся создать хоть и учебный, но вполне работающий сайт. При этом будут рассмотрены следующие вопросы:

- создание прототипа сайта при помощи Django;
- запуск и остановка сервера для разработки приложения;
- создание модели данных для веб-приложения;
- использование административной панели Django для управления сайтом.

ПРИМЕЧАНИЕ

В ходе реализации учебного примера этой главы нужно достаточно внимательно отслеживать, в какой папке (каталоге) приложения требуется создавать те или иные файлы, а также в каком каталоге терминала вы находитесь. Если вы по ошибке создали файл не в том каталоге, который рекомендован, Django не найдет его в нужном месте, и приложение будет выдавать ошибки.

8.1. Создание структуры сайта при помощи Django

Создадим сайт, которому дадим условное название «Мир книг». Это может быть интернет-ресурс, который дает доступ к скачиванию электронных копий книг, или книжный интернет-магазин, или электронная библиотека, в которую можно записаться и подбирать и заказывать там себе книги в режиме on-line. Как можно догадаться, цель создания такого достаточно простого сайта заключается в том, чтобы представить онлайн-каталог книг, откуда пользователи смогут загружать доступные книги и где у них будет возможность управлять своими профилями.

Разработанное нами приложение впоследствии можно будет расширять, улучшать его дизайн, создавать на его основе аналогичные сайты. Но, что более важно, на примере его разработки вы разберетесь с порядком действий при использовании фреймворка Django для создания веб-приложений.

На первом шаге мы создадим каталог книг, в котором пользователи смогут только просматривать доступные книги. Это позволит нам изучить операции, которые присутствуют при создании практически любого сайта, где осуществляется взаимодействие с базами данных, — чтение и отображение информации из БД.

По мере развития проекта на сайте будут задействованы более сложные функции и возможности Django. Например, мы сможем расширить функционал сайта, позволив пользователям резервировать книги, покажем, как использовать формы для ввода информации в БД, как реализовать авторизацию пользователей и т. п.

Итак, приступим к созданию «скелета» нашего сайта. Запустите приложение PyCharm и создайте новый проект с именем `World_books`. Для этого в главном меню приложения выполните команду **File | New Project**, после чего откроется окно, в котором введите имя нашего проекта: `World_books` (рис. 8.1).

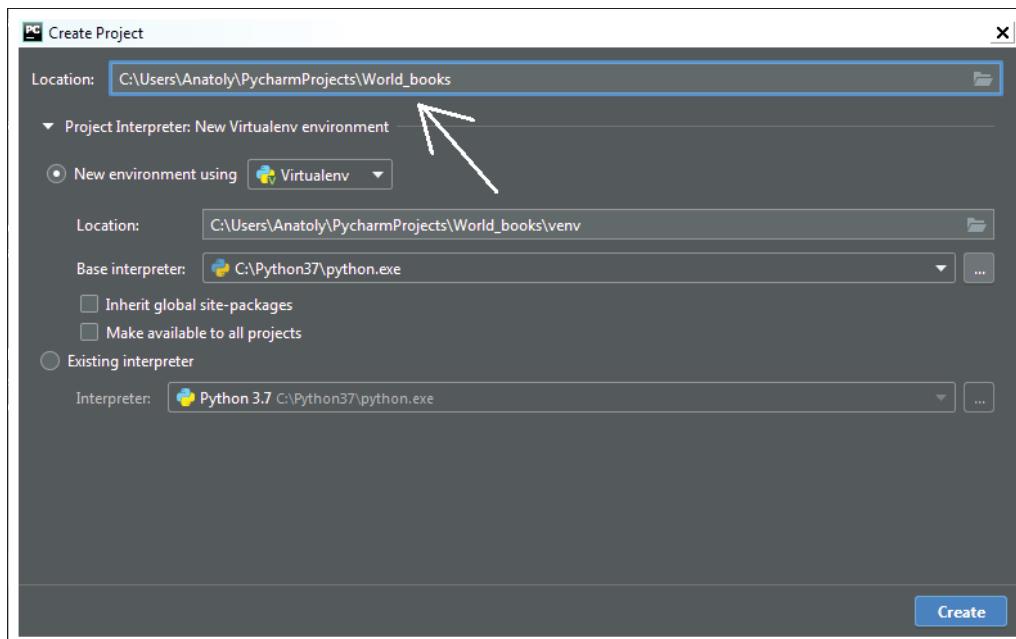


Рис. 8.1. Создание нового проекта `World_books` в окне приложения PyCharm

Теперь загрузите веб-фреймворк Django, для чего в главном меню PyCharm выполните команду **File | Settings | Project: World_books | Project Interpreter** и в открывшемся окне щелкните мышью на значке  (рис. 8.2).

Здесь можно либо пролистать весь список и найти библиотеку **Django**, либо набрать наименование этой библиотеки в верхней строке поиска, и она будет найдена в раскрывшемся списке (рис. 8.3).

Нажмите на кнопку **Install Package**, и выбранная библиотека **Django** будет добавлена в ваш проект (рис. 8.4). Для завершения этого процесса нажмите кнопку **OK** в окне **Settings**.

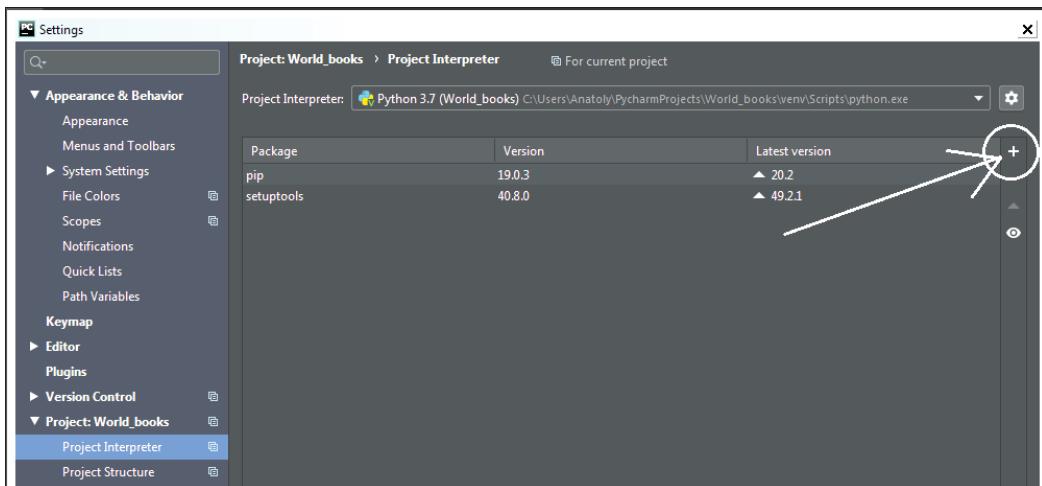


Рис. 8.2. Окно PyCharm для загрузки дополнительных модулей и библиотек

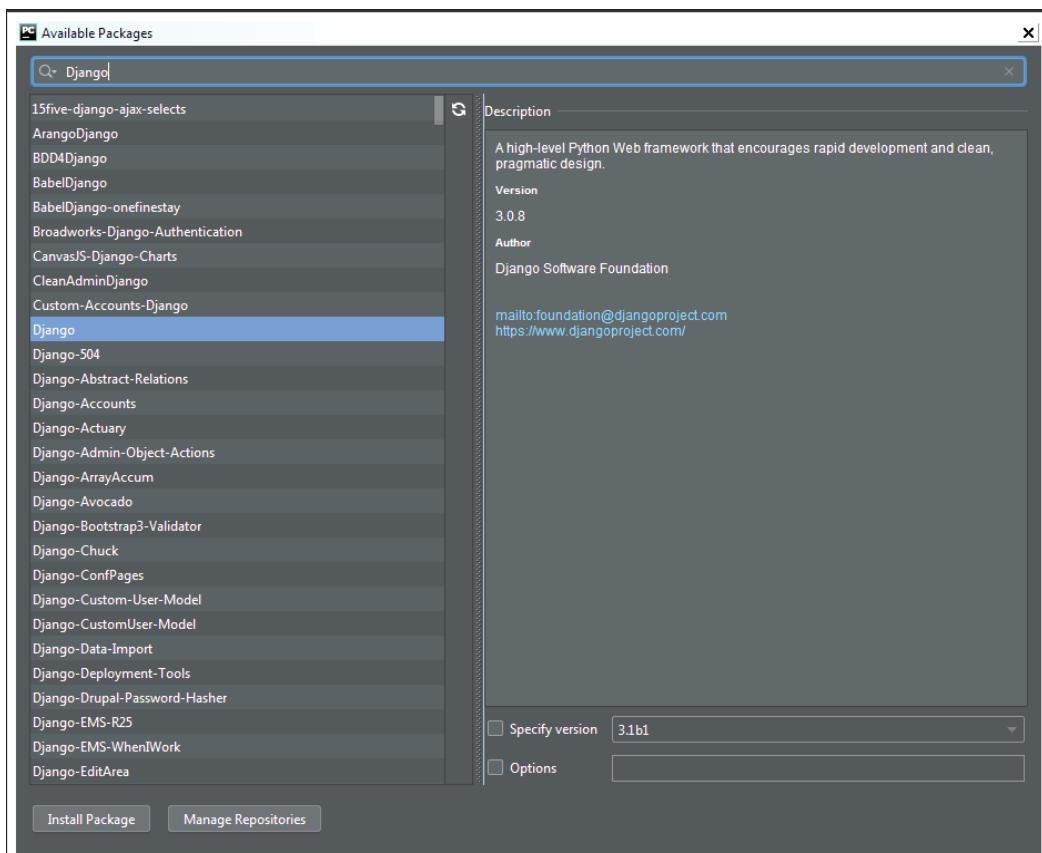


Рис. 8.3. Поиск библиотеки Django в списке доступных библиотек

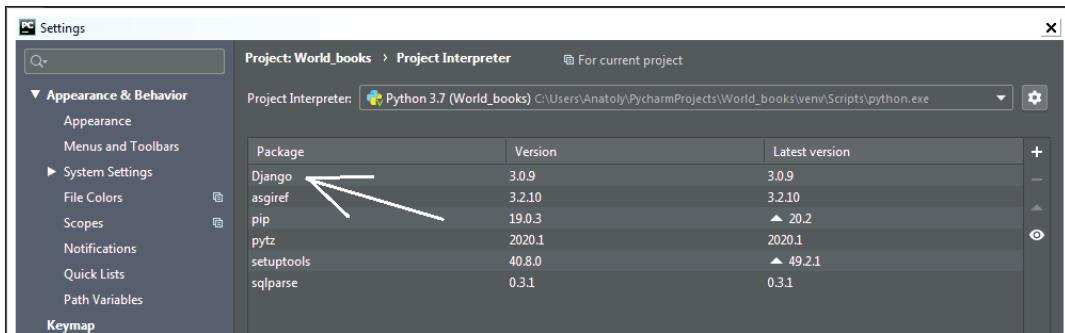


Рис. 8.4. Библиотека Django в списке доступных библиотек проекта World_books

Итак, мы создали проект World_books, но это пока только проект приложения PyCharm, который еще не является проектом Django. Поэтому войдите в окно терминала PyCharm и создайте новый проект Django с именем WebBooks. Для этого в окне терминала нужно выполнить следующую команду (рис. 8.5):

```
django-admin startproject WebBooks
```

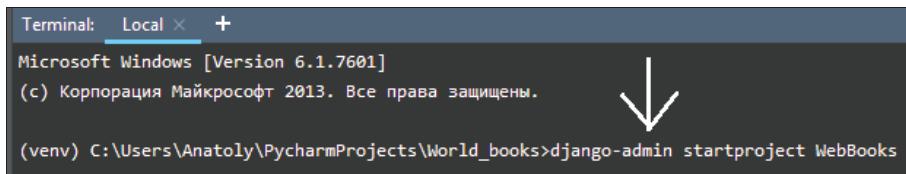


Рис. 8.5. Создание нового проекта Django с именем WebBooks в окне терминала PyCharm

В результате выполнения этой команды в папке World_books проекта PyCharm будет создана новая папка WebBooks проекта Django (рис. 8.6).

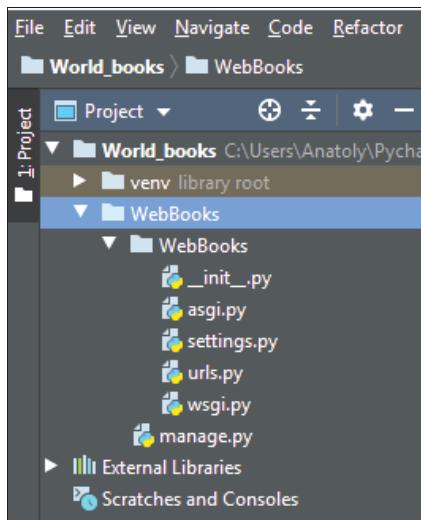


Рис. 8.6. Структура файлов проекта Django с именем WebBooks

Как можно видеть, папка проекта WebBooks включает одну вложенную папку WebBooks — это ключевой каталог нашего проекта, в котором находятся следующие файлы:

- `settings.py` — содержит все настройки проекта. Здесь мы регистрируем приложения, задаем размещение статичных файлов, делаем настройки базы данных и т. п.);
- `urls.py` — задает ассоциации интернет-адресов (URL) с представлениями. Этот файл может содержать все настройки URL, но обычно его делят на части — по одной на каждое приложение, как будет показано далее;
- файлы `asgi.py` и `wsgi.py` используются для организации связи между Django приложением и внешним веб-сервером. Мы задействуем эти файлы позже, когда будем рассматривать развертывание сайта в сети Интернет.

Кроме вложенной папки WebBooks, в головной папке проекта WebBooks имеется также файл `manage.py`. Это скрипт, который обеспечивает создание приложений, работу с базами данных, запуск отладочного сервера. Именно этот скрипт мы и будем использовать на следующем шаге для создания приложения.

Итак, создадим приложение с именем `catalog`. Для этого сначала необходимо из приложения PyCharm (внешняя папка `World_books`) перейти в приложение Django — войти во вложенную папку `WebBooks`, в которой находится скрипт `manage.py`. Для этого в окне терминала PyCharm выполните следующую команду (рис. 8.7):

```
cd WebBooks
```

```
Terminal: Local × +  
(venv) C:\Users\Anatoly\PycharmProjects\World_books>cd WebBooks  
(venv) C:\Users\Anatoly\PycharmProjects\World_books\WebBooks>
```

Рис. 8.7. Переход в папку с именем `WebBooks` проекта Django

Вот теперь, находясь в папке проекта Django с именем `WebBooks` (той, где находится скрипт `manage.py`), создадим приложение `catalog`. Для этого выполним следующую команду (рис. 8.8):

```
python manage.py startapp catalog
```

```
Terminal: Local × +  
(venv) C:\Users\Anatoly\PycharmProjects\World_books>django-admin startproject WebBooks  
(venv) C:\Users\Anatoly\PycharmProjects\World_books>cd WebBooks  
(venv) C:\Users\Anatoly\PycharmProjects\World_books\WebBooks>python manage.py startapp catalog
```

Рис. 8.8. Создание в проекте Django с именем `WebBooks` приложения `catalog`

В результате в папке верхнего уровня WebBooks появится новая папка с именем catalog (рис. 8.9) — в ней и будут содержаться файлы нашего приложения catalog (рис. 8.10).

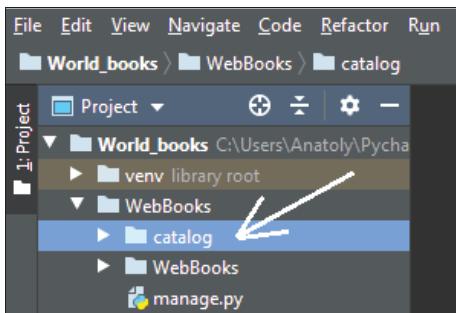


Рис. 8.9. Папка catalog в проекте Django с именем WebBooks

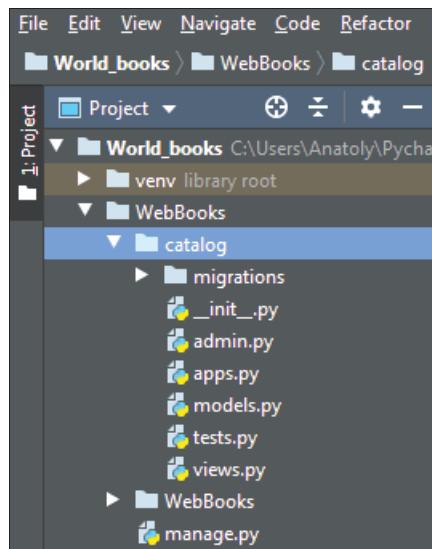


Рис. 8.10. Содержимое папки catalog приложения catalog

Эти файлы предназначены для выполнения следующих функций:

- папка migrations — хранит миграции (файлы, которые позволяют автоматически обновлять базу данных по мере изменения моделей данных);
- «пустой» файл __init__.py — его присутствие позволяет Django и Python распознавать папку catalog как папку с модулями Python, что дает возможность использовать объекты этих модулей внутри других частей проекта;
- файл admin.py — содержит информацию с настройками административной части приложения;
- файл apps.py — предназначен для регистрации приложений;
- файл models.py — содержит описание моделей данных, с которыми будет работать приложение;
- файл tests.py — содержит тесты, которые можно будет использовать для тестирования приложения;
- файл views.py — содержит контроллеры или представления (views), которые обеспечивают взаимодействие приложения с БД (выборка, добавление, удаление записей) и с запросами пользователя.

Большинство этих файлов уже содержат некоторый шаблонный программный код для работы с указанными объектами. Однако следует отметить, что для нашего проекта ряда необходимых файлов не хватает — в частности, не были созданы пап-

ки для файлов шаблонов и статичных файлов. Далее мы создадим эти папки и файлы (они не обязательны для каждого сайта, но будут нужны в нашем примере).

Созданное приложение нужно зарегистрировать в проекте Django. Это необходимо для того, чтобы различные утилиты распознавали его при выполнении некоторых действий (например, при добавлении моделей в базу данных). Приложения регистрируются добавлением их названий в список `INSTALLED_APPS` в файле настроек проекта `settings.py`.

Откройте файл `WebBooks\WebBooks\settings.py`, найдите в нем список зарегистрированных приложений `INSTALLED_APPS` и добавьте наше приложение `catalog` в конец списка (выделено серым фоном и полужирным шрифтом), как показано на рис. 8.11:

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

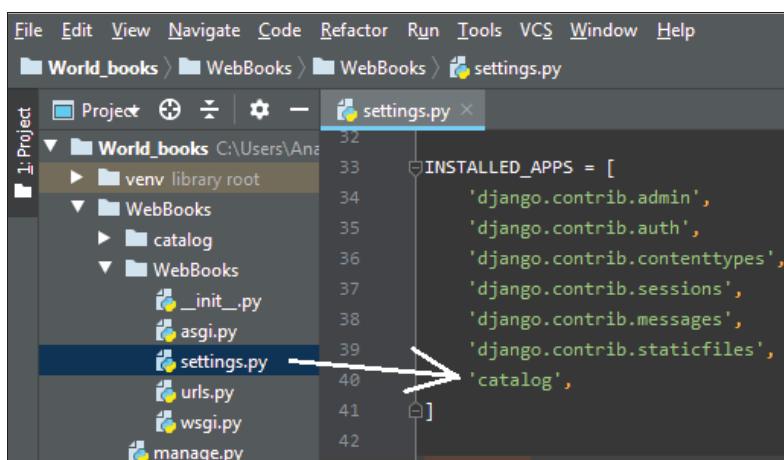


Рис. 8.11. Регистрация приложения `catalog` в проекте Django

Теперь следует указать базу данных для нашего проекта. По возможности имеет смысл использовать одну и ту же базу данных как в процессе разработки проекта, так и при размещении его в сети Интернет. Это позволит исключить возможные различия в поведении проекта после его публикации.

Однако для нашего проекта мы воспользуемся базой данных SQLite, потому что по умолчанию приложение уже настроено для работы с ней. В этом можно убедиться, открыв соответствующий раздел в файле `settings.py` (рис. 8.12).

```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
World_books > WebBooks > WebBooks > settings.py
Project + - settings.py
World_books C:\Users\Ana
  venv library root
    WebBooks
      catalog
      WebBooks
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
        manage.py
1: Project
74 # Database
75 # https://docs.djangoproject.com/en/3.0/ref/settings/#...
76
77 DATABASES = [
78     'default': {
79         'ENGINE': 'django.db.backends.sqlite3',
80         'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
81     }
82 ]
83
84 # Password validation

```

Рис. 8.12. Настройки проекта для работы с базой данных SQLite

Как можно видеть, к проекту подключен движок БД SQLite, а файл, который содержит сами данные, носит имя db.sqlite3.

Файл `settings.py` также служит и для выполнения некоторых других настроек (значения их по умолчанию представлены на рис. 8.13):

- `LANGUAGE_CODE` — строка, представляющая код языка. Значение по умолчанию: `en-us` — американская версия английского. Код русского языка для России — `ru-ru`;
- `.TIME_ZONE` — строка, представляющая часовой пояс. Значение по умолчанию: `UTC` (`America/Chicago`). Значение для России: `Europe/Moscow`;
- `USE_I18N` — логическое значение, которое указывает, должна ли быть включена система перевода Django. По умолчанию: `True`. Если значение `USE_I18N` устано-

```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
World_books > WebBooks > WebBooks > settings.py
Project + - settings.py
World_books C:\Users\Ana
  venv library root
    WebBooks
      catalog
      WebBooks
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
        manage.py
1: Project
100 # Internationalization
101 # https://docs.djangoproject.com/en/3.0/ref/settings/#...
102
103 LANGUAGE_CODE = 'en-us'
104
105 TIME_ZONE = 'UTC'
106
107 USE_I18N = True
108
109 USE_L10N = True
110
111 USE_TZ = True

```

Рис. 8.13. Настройки проекта в файле `settings.py` по умолчанию

вить в `False`, то Django не будет загружать механизм перевода — это один из способов повышения производительности приложения;

- ❑ `USE_I18N` — логическое значение, которое указывает, будет ли включено локализованное форматирование данных. По умолчанию: `True` (при этом Django будет отображать числа и даты в формате текущей локализации);
- ❑ `USE_TZ` — логическое значение, которое указывает, будет ли `datetime` привязан к часовому поясу. По умолчанию: `True` (при этом Django будет использовать внутренние часы с учетом часового пояса, в противном случае будут задействованы нативные даты по местному времени).

В нашем случае имеет смысл поменять параметры `LANGUAGE_CODE` и `TIME_ZONE`. В привязке этих параметров к русскому языку и Московскому региону эти настройки будут выглядеть следующим образом:

```
LANGUAGE_CODE = 'ru-ru'
TIME_ZONE = 'Europe/Moscow'
USE_I18N = True
USE_L10N = True
USE_TZ = True
```

В проекте может быть несколько приложений, но мы ограничимся одним. И после всех выполненных нами действий полная структура нашего веб-приложения будет выглядеть так, как представлено на рис. 8.14.

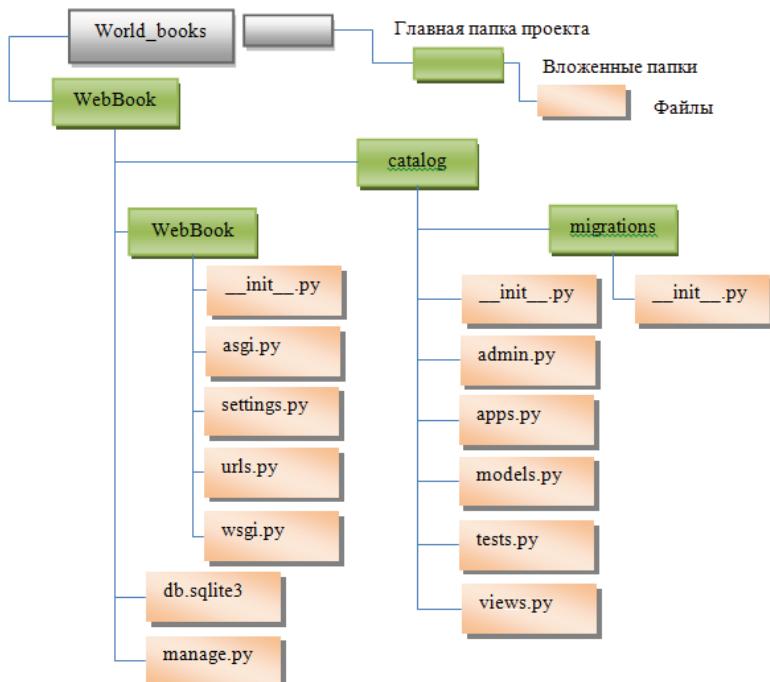


Рис. 8.14. Полная структура файлов и папок сайта с проектом `WebBook` и приложением `catalog`

Теперь определим какие-нибудь простейшие действия, которые будет выполнять это приложение, — например, отправлять в ответ пользователю строку **Главная страница сайта Мир книг**.

Для этого перейдем в проекте приложения catalog к файлу views.py, который по умолчанию должен выглядеть так:

```
from django.shortcuts import render
# Create your views here.
```

и изменим этот код следующим образом (листинг 8.1).

Листинг 8.1

```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.

def index(request):
    return HttpResponse("Главная страница сайта Мир книг!")
```

Здесь мы импортируем класс `HttpResponse()` из стандартного пакета `django.http`. Затем определяем функцию `index()`, которая в качестве параметра получает объект запроса пользователя `request`. Класс `HttpResponse()` предназначен для создания ответа, который отправляется пользователю. Так что с помощью выражения `return HttpResponse()` мы как раз и отправляем пользователю строку "Главная страница сайта Мир книг!".

Теперь в проекте Django откроем файл `urls.py`, обеспечивающий сопоставление маршрутов с представлениями (`views`), которые будут обрабатывать запросы по этим маршрутам. По умолчанию этот файл выглядит так:

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

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

В первой строке из модуля `django.contrib` импортируется класс `Admin`, который предоставляет возможности работы с интерфейсом администратора сайта. Во второй строке из модуля `django.urls` импортируется функция `path`. Она задает возможность сопоставлять запросы пользователя (определенные маршруты) с функциями их обработки. Так, в нашем случае маршрут `'admin/'` будет обрабатываться методом `admin.site.urls`. То есть если пользователь запросит показать страницу администратора сайта (`'admin/'`), то будет вызван метод `admin.site.urls`, который вернет пользователю HTML-страницу администратора.

Но ранее мы определили функцию `index()` в файле `views.py`, который возвращает пользователю текстовую строку. Поэтому изменим сейчас файл `urls.py` следующим образом (листинг 8.2).

Листинг 8.2

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

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

Чтобы использовать функцию `views.index`, мы здесь сначала импортируем модуль `views`. Затем сопоставляем маршрут `('')` — это, по сути, запрос пользователя к корневой странице сайта, с функцией `views.index`. То есть если пользователь запросит показать главную (корневую) страницу сайта `('')`, то будет вызвана функция `index` из файла `views.py`. А в этой функции мы указали, что пользователю нужно вернуть HTML-страницу с единственным сообщением: **Главная страница сайта Мир книг!**. Соответственно, и маршрут с именем `'home'` также будет сопоставляться с запросом к «корню» приложения.

Теперь запустим приложение командой:

```
python manage.py runserver
```

и перейдем в браузере по адресу `http://127.0.0.1:8000/` — браузер отобразит нам сообщение: **Главная страница сайта Мир книг!** (рис. 8.15).

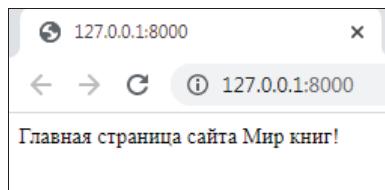


Рис. 8.15. Первый запуск проекта WebBook с приложением catalog

Вспомним, что в файле `urls.py` (см. листинг 8.2) второй строкой прописана возможность запуска встроенного в Django приложения «Администратор сайта» — через маршрут `'admin/'`. Вызовем этот модуль, указав в браузере интернет-адрес (URL): `http://127.0.0.1:8000/admin/`, и получим следующий ответ (рис. 8.16).

Как можно видеть, интерфейс приложения администрирования сайта выведен на русском языке. Это результат того, что в файле `settings.py` мы изменили параметры локализации и установили русский язык (параметр `ru-ru`). К приложению «Администратор сайта» мы вернемся немного позже. А пока можно сделать вывод, что

мы создали полноценный «скелет» веб-приложения, которое теперь можно расширять, добавляя новые страницы, представления (views) и модели (models) и устанавливая URL-соответствия между представлениями и моделями данных. Самое время перейти к следующему разделу и начать создавать базу данных, а также писать код, который научит сайт делать то, что он должен делать.

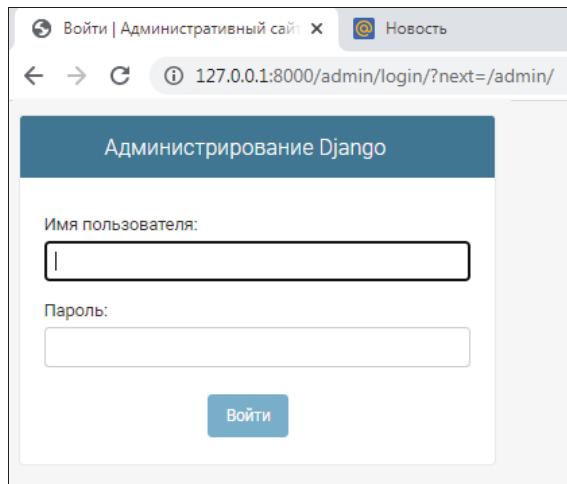


Рис. 8.16. Вызов административной страницы сайта проекта WebBook

8.2. Разработка структуры моделей данных сайта «Мир книг»

В этом разделе мы узнаем, как строить модели для сайта «Мир книг», и рассмотрим некоторые из основных типов полей таблиц, в которых будет храниться информация сайта. Здесь также будет показано несколько основных способов доступа к данным модели.

Веб-приложения Django получают доступ и управляют данными через объекты Python, называемые *моделями*. Модели определяют структуру хранимых данных, включая типы полей, их максимальный размер, значения по умолчанию, параметры выбора, текст меток для форм и пр. Структура и описание модели не зависят от используемой базы данных — она может быть выбрана позднее путем изменения всего нескольких настроек проекта. После выбора какой-либо базы данных разработчику не придется напрямую взаимодействовать с ней (создавать таблицы и связи между ними) — это сделает Django, используя уже разработанную структуру модели.

Перед тем как перейти к созданию моделей данных, нам необходимо определить, какую информацию мы будем хранить в БД, сформировать перечень таблиц БД и организовать связи между ними.

Итак, на нашем сайте мы будем хранить данные о книгах. К этим данным относятся: название книги, ее автор, краткое резюме, язык, на котором написана книга, жанр книги, ее ISBN (International Standard Book Number, международный стандартный книжный номер). Кроме того, необходимо хранить сведения о количестве доступных экземпляров книги, о статусе доступности каждого экземпляра и т. п. Нам также понадобится иметь более подробную информацию об авторе книги, чем просто его имя. Да еще у книги может быть несколько авторов с одинаковыми или похожими именами. Конечно, потребуется возможность сортировки информации на основе названия книги, автора, языка, жанра и других признаков.

При проектировании моделей данных имеет смысл создавать самостоятельные модели для каждого объекта. Под *объектом* мы будем понимать группу связанный информации. В нашем случае очевидными объектами являются: жанры книг, сами книги, их авторы, экземпляры книг, их язык.

Следует также определить, какие данные можно и желательно выдавать в виде списка выбора. Это рекомендуется в тех случаях, когда все возможные варианты списка заранее неизвестны или могут впоследствии измениться. Можно выделить следующие данные, которые можно выдавать в виде списков: жанр книги (Фантастика, Поэзия), язык (русский, английский, французский, японский).

Как только будет определен перечень моделей данных (таблицы БД и поля таблиц), нам нужно подумать об отношениях между моделями (между таблицами в БД). Как было показано в предыдущей главе, Django позволяет определять отношения «один-к-одному» (конструктор `OneToOneField`), «один-ко-многим» (конструктор `ForeignKey`) и «многие-ко-многим» (конструктор `ManyToManyField`).

Для нашего сайта мы предусмотрим в БД пять таблиц для хранения информации, и в соответствии с этим нам потребуется сформировать структуру и описание пяти моделей данных. Вот их перечень:

- книги (общие сведения о книгах);
- экземпляры книг (статус конкретных физических книг, доступных в системе);
- автор книги;
- язык, на котором написана книга;
- жанры книг.

Структура этих моделей данных показана на рис. 8.17. Как можно видеть, в ее состав входят перечень таблиц, которые будут созданы в БД, перечень и типы полей, которые будут в каждой таблице, определены также связи между таблицами.

Итак, мы определились с перечнем хранимых данных. На основе этой информации можно приступить к созданию моделей данных. Однако прежде познакомимся с основными элементами, которые нужно будет создавать при формировании моделей данных. Этому и посвящен следующий раздел.

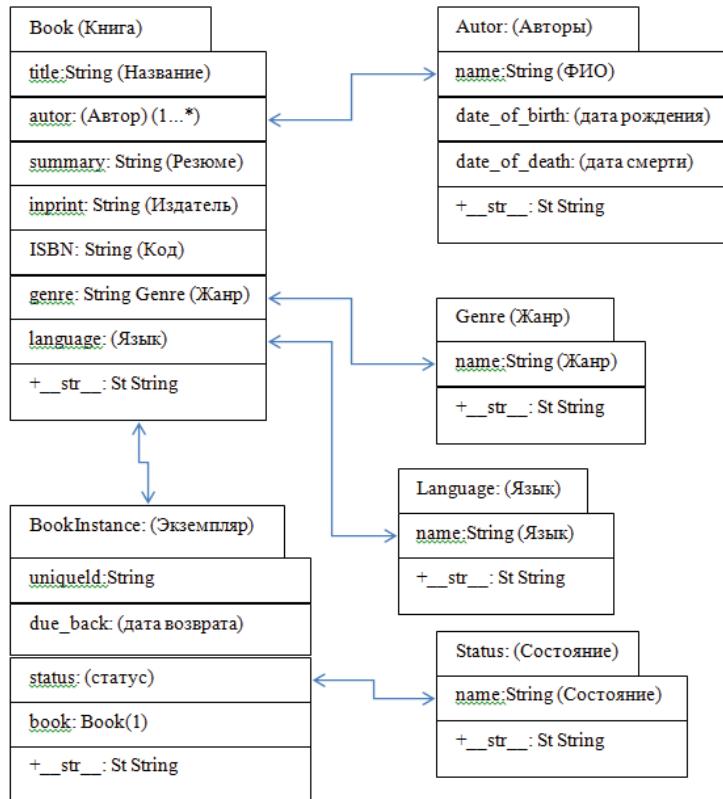


Рис. 8.17. Структура моделей данных сайта «Мир книг»

8.3. Основные элементы моделей данных в Django

В этом разделе мы вкратце разберемся с тем, как формируются модели данных, рассмотрим некоторые из наиболее важных полей и их аргументы, узнаем, что такое метаданные и как с помощью методов можно управлять данными в моделях.

Модели в приложении обычно описываются в файле `models.py`. Они реализуются как подклассы или объекты класса `django.db.models.Model` и могут включать поля, метаданные и методы. В приведенном далее фрагменте кода показан пример типичной модели, имеющей условное название `MyModelName`:

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

```
# Типичный класс, определяющий модель, производный от класса Model
class MyModelName(models.Model):
```

```
# Метаданные
class Meta:
    ordering = ["-my_field_name"]

# Методы
def get_absolute_url(self):
    # Возвращает url-адрес для доступа к экземпляру MyModelName
    return reverse('model-detail-view', args=[str(self.id)])

def __str__(self):
    # Стока для представления объекта MyModelName в Admin site
    return self.field_name
```

В этом программном коде определена модель данных (будущая таблица в БД) с именем `MyModelName`. Таблица БД будет иметь поле `my_field_name` для хранения данных (с количеством символов в имени не более 20). Предусмотрена сортировка данных по полю `my_field_name` и в виде функций заданы два метода. В следующих разделах мы более подробно рассмотрим каждый из элементов внутри модели.

8.3.1. Поля и их аргументы в моделях данных

Модель может иметь произвольное количество полей любого типа. Каждое поле представляет столбец данных, который будет сохраняться в одной из таблиц базы данных. Каждая запись (строка в таблице базы данных) будет состоять из значений тех полей, которые описаны в модели. Давайте рассмотрим описание поля из приведенного ранее примера:

```
my_field_name = models.CharField(max_length=20,
                                  help_text="Не более 20 символов")
```

В соответствии с этим описанием таблица БД будет иметь одно поле с именем `my_field_name`, тип поля — `CharField`. Этот тип поля позволяет хранить в БД строки, состоящие из буквенных и цифровых символов. Система предусматривает проверку содержания этого поля при вводе в него пользователем каких-либо значений в формате HTML. То есть Django не позволит ввести в него недопустимые символы.

Поля могут принимать некоторые аргументы, которые дополнительно определяют, как поле будет хранить данные или как оно может применяться пользователем. В нашем примере полю присваивается два аргумента:

- `max_length=20` — указывает, что максимальная длина этого поля составляет 20 символов;
- `help_text="Не более 20 символов"` — предоставляет дополнительную текстовую подсказку, чтобы помочь пользователям узнать, какое ограничение наложено на длину этого поля.

Метка поля и подсказки будут показаны пользователю через HTML-форму.

Имя поля используется для обращения к нему в запросах и шаблонах. У каждого поля также есть *метка*, значение которой можно задать через аргумент `verbose_`

`name`. Метка по умолчанию задается автоматически путем изменения первой буквы имени поля на заглавную букву и замены любых символов подчеркивания пробелом. Например, поле с именем `my_field_name` будет иметь метку по умолчанию `My field name`.

Поля по умолчанию отображаются в форме (например, на сайте администратора) в том же порядке, в котором они объявляются, хотя этот порядок может быть переопределён в процессе разработки проекта.

Поля, описываемые в моделях данных и имеющие разные типы, могут иметь достаточно большое количество *общих аргументов*:

- ❑ `help_text` — предоставляет текстовую метку для HTML-форм (например, на сайте администратора или на форме пользователя);
- ❑ `verbose_name` — удобно читаемое имя для поля, используемое в качестве его метки. Если этот аргумент не указан, то Django автоматически сформирует его по умолчанию от имени поля;
- ❑ `default` — значение по умолчанию для поля. Это может быть значение или вызываемый объект, и в этом случае объект будет вызываться каждый раз, когда создается новая запись;
- ❑ `null` — если аргумент имеет значение `True`, то Django будет хранить пустые значения в базе данных для полей как `NULL`, где это уместно (поле типа `CharField` вместо этого сохранит пустую строку). По умолчанию принимается значение `False`;
- ❑ `blank` — если аргумент имеет значение `True`, поле в ваших формах может быть пустым. По умолчанию принимается значение `False`, означающее, что проверка формы Django заставит вас ввести в это поле некоторое значение. Этот аргумент часто используется с аргументом `null = True` — если вы разрешаете ввод пустых значений, вы также должны обеспечить, чтобы база данных могла принять эти пустые значения;
- ❑ `choices` — предоставление выбора из вариантов значений для этого поля. Если параметр `choices` задан, то соответствующий виджет формы будет являться полем с вариантами выбора из нескольких значений (вместо стандартного текстового поля);
- ❑ `primary_key` — если аргумент имеет значение `True`, то текущее поле будет выступать в качестве *первичного ключа* для модели (первичный ключ — это специальный столбец базы данных, предназначенный для однозначной идентификации всех разных записей таблицы). Если в качестве первичного ключа не указано ни одно поле, то Django автоматически добавит ключевое поле (`id`) в таблицу данных.

Указанные общие аргументы могут использоваться при объявлении следующих типов полей:

- ❑ `CharField` — служит для определения строк фиксированной длины: от короткой до средней. Следует указывать максимальную длину строки `max_length` для хранения данных;

- `TextField` — используется для больших строк произвольной длины. Вы можете указать для поля `max_length`, но это значение будет задействовано только тогда, когда поле отображается в формах (оно не применяется на уровне базы данных);
- `IntegerField` — это поле для хранения значений (целого числа) и для проверки введенных в форму значений в виде целых чисел;
- `DateField` и `DateTimeField` — служат для хранения дат и информации о дате или времени (как Python `datetime.date` и `datetime.datetime`, соответственно). Эти поля могут дополнительно объявлять параметры: `auto_now=True` (для установки поля на текущую дату каждый раз, когда модель сохраняется), `auto_now_add` (только для установки даты, когда модель была впервые создана) и по умолчанию (чтобы установить дату по умолчанию, которую пользователь может переустановить);
- `EmailField` — служит для хранения и проверки адресов электронной почты;
- `FileField` и `ImageField` — служат для загрузки файлов и изображений (`ImageField` просто добавляет дополнительную проверку, является ли загруженный файл изображением). Они имеют параметры для определения того, как и где хранятся загруженные файлы;
- `AutoField` — это особый тип `IntegerField`, который автоматически увеличивается. Первичный ключ (`id`) этого типа автоматически добавляется в вашу модель, если вы явно не укажете его;
- `ForeignKey` — служит для указания отношения «один-ко-многим» к другой модели базы данных (например, автомобиль определенной модели имеет одного производителя, но производитель может делать много автомобилей разных моделей). Сторона отношения «один» — это модель, содержащая ключ;
- `ManyToManyField` — служит для определения отношения «многие-ко-многим» (например, книга может иметь несколько жанров, и каждый жанр может содержать несколько книг). В нашем приложении для библиотек мы будем использовать этот тип поля аналогично типу `ForeignKey`, но их можно использовать более сложными способами для описания отношений между группами. Эти типы полей имеют параметр `on_delete`, определяющий, что происходит, когда связанная запись удаляется (например, значение `models.SET_NULL` просто установило бы значение `NULL`).

Существует много других типов полей, включая поля для разных типов чисел (большие целые числа, малые целые числа, дробные), логические значения, URL-адреса, символы «slugs», уникальные идентификаторы и другие, в том числе «связанные со временем» сведения (продолжительность, время и т. д.). Эти поля мы достаточно подробно описали, когда рассматривали формы Django.

8.3.2. Метаданные в моделях Django

Метаданные — это информация о другой информации или данные, относящиеся к дополнительной информации о содержимом или объекте. Метаданные раскрыва-

ют сведения о признаках и свойствах, характеризующих какие-либо сущности, позволяющие автоматически искать и управлять ими в больших информационных потоках. К метаданным относятся многие параметры модели, заданные по умолчанию, которые, кстати, можно переопределить. Это, например, имена таблиц, порядок сортировки данных, список индексов, имя поля и т. п.

В Django имеется возможность объявить метаданные о своей модели на уровне самой модели. Для этого нужно объявив класс `Meta`, как показано в следующем коде:

```
class Meta:
    ordering = ["-my_field_name"]
```

В этом коде показана одна из наиболее полезных функций метаданных — управление сортировкой записей. Порядок сортировки можно задать, указав название поля (или полей). Сам порядок сортировки зависит от типа поля. Например, для текстового поля сортировка будет выполняться в алфавитном порядке, а поля даты будут отсортированы в хронологическом порядке. Прямой порядок сортировки можно заменить на обратный, для этого используется символ минус (-). Согласно порядку, показанному в примере кода, данные текстового поля `my_field_name` (имя клиента) будут отсортированы в алфавитном порядке: от буквы Я до буквы А, поскольку символ (-) изменяет порядок сортировки с прямого на обратный.

Если, например, мы решили по умолчанию сортировать книги по двум полям и написали следующий код:

```
class Meta:
    ordering = ["title", "-pubdate"]
```

то книги будут отсортированы по названию согласно алфавиту от А до Я, а затем по дате публикации внутри каждого названия — от самого нового (последнего) издания до самого старого (первого).

Другим распространенным атрибутом является `verbose_name`, или подробное (многословное) имя для класса. Например:

```
from django.db import models

class MyModelName(models.Model):

    my_field_name = models.CharField(max_length=20,
                                    help_text="Не более 20 символов")
    class Meta:
        verbose_name = "Название книги"
```

ПРИМЕЧАНИЕ

Полный список метаданных доступен в документации по Django.

8.3.3. Методы в моделях Django

Модель также может иметь *методы*, т. е. различные способы манипулирования данными (чтение, обновление, удаление). Минимально в каждой модели должен быть определен стандартный метод класса для Python: `__str__()`. Это необходи-

мо, чтобы вернуть удобно читаемую строку для каждого объекта. Эта строка используется для представления отдельных записей в модуле администрирования сайта (и в любом другом месте, где вам нужно обратиться к экземпляру модели). Часто этот метод возвращает наименование поля из модели, например:

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

Другим распространенным методом, который включается в модели Django, является метод `get_absolute_url()`, который возвращает URL-адрес для отображения отдельных записей модели на веб-сайте. Если этот метод определен, то Django автоматически добавит кнопку **Просмотр на сайте** на экранах редактирования записей модели (на сайте администратора). Вот типичный шаблон для `get_absolute_url()`:

```
def get_absolute_url(self):  
    return reverse('model-detail-view', args=[str(self.id)])
```

Предположим, что требуется использовать URL-адрес, например:

/myapplication/mymodelname/2

для отображения отдельной записи модели (где **2** — идентификатор для определенной записи). Чтобы выполнить работу, необходимую для отображения записи, нужно создать URL-карту. Функция `reverse()` может преобразовать ваш URL-адрес (в приведенном примере с именем `model-detail-view`), в URL-адрес правильного формата. Конечно, для выполнения этой работы все равно придется прописать соответствие URL-адреса с представлением (`view`) и шаблоном.

8.3.4. Методы работы с данными в моделях Django

Когда классы моделей данных определены, их можно использовать для создания, обновления или удаления записей, для выполнения запросов, а также получения всех записей или отдельных подмножеств записей из таблиц БД.

Чтобы создать запись в таблице БД, нужно определить (создать) экземпляр модели, а затем вызвать метод `save()`:

```
# Создать новую запись с помощью конструктора модели  
a_record = MyModelName(my_field_name="Книга о вкусной еде")  
  
# Сохраните запись в базе данных.  
a_record.save()
```

Если поле `MyModelName` предназначено для хранения названий книг, то в соответствующей таблице БД появится запись с названием этой книги: Книга о вкусной еде.

Можно получать доступ к полям в записи БД и изменять их значения. После внесение изменений данных необходимо вызвать метод `save()`, чтобы сохранить изменившиеся значения в базе данных:

```
# Доступ к значениям полей модели  
print(a_record.id) # получить идентификатор записи  
print(a_record.my_field_name) # получить название книги
```

```
# Изменить название книги
a_record.my_field_name="Книга о вкусной и здоровой пище"
a_record.save()
```

Вы можете искать в БД записи, которые соответствуют определенным критериям, используя атрибуты объектов модели. Для поиска и выборки данных из БД в Django служит объект `QuerySet`. Это интегрирующий объект — он содержит несколько объектов, которые можно перебирать (прокручивать). Чтобы извлечь все записи из таблицы базы данных, используйте метод `objects.all()`. Например, чтобы получить названия всех книг из БД, нужно применить следующий код:

```
all_books = Book.objects.all()
```

Метод `filter()` позволяет отфильтровать данные для `QuerySet` в соответствии с указанным полем и критерием фильтрации. Например, чтобы отфильтровать книги, содержащие в названии слово «дикые» (дикие животные, дикие растения и т. п.), а затем подсчитать их, мы могли бы воспользоваться следующим кодом:

```
wild_books = Book.objects.filter(title__contains='дикие')
n_w_books = Book.objects.filter(title__contains='дикие').count()
```

В этих фильтрах использован формат: `field_name__match_type`, где `field_name` — имя поля, по которому фильтруются данные, а `match_type` — тип соответствия поля определенному критерию (обратите внимание, что между ними стоит двойное подчеркивание). В приведенном коде полем, по которому фильтруются данные, является название книги (`title`) и задан тип соответствия `contains` (с учетом регистра).

В Django можно использовать в фильтрах различные типы соответствия (совпадения). Основные типы соответствия приведены в табл. 8.1.

Таблица 8.1. Основные типы соответствия (совпадения), используемые в Django

№ п/п	Вызов типа соответствия	Назначение типа соответствия	Аналог на языке SQL
1	<code>exact</code>	Точное совпадение с учетом регистра	<code>SELECT ... WHERE id = 14</code>
2	<code>iexact</code>	Точное совпадение без учета регистра	<code>SELECT ... WHERE name ILIKE 'beatles blog'</code>
3	<code>contains</code>	С учетом регистра	<code>SELECT ... WHERE headline LIKE '%Lennon%'</code>
4	<code>icontains</code>	Без учета регистра	<code>SELECT ... WHERE headline ILIKE '%Lennon%'</code>
5	<code>in</code>	Равно одному из элементов списка или кортежа	<code>SELECT ... WHERE id IN (1, 3, 4)</code>
6	<code>gt</code>	Больше чем	<code>SELECT ... WHERE id > 4</code>
7	<code>gte</code>	Больше или равно	<code>SELECT ... WHERE id >= 4</code>
8	<code>lt</code>	Меньше чем	<code>SELECT ... WHERE id < 4</code>
9	<code>lte</code>	Меньше или равно	<code>SELECT ... WHERE id <= 4</code>
10	<code>range</code>	Внутри диапазона	<code>SELECT ... WHERE id > 4 AND id < 8</code>

Полный список типов соответствий, которые используются в фильтрах, можно посмотреть в оригинальной документации на Django.

В некоторых случаях возникает необходимость фильтровать поле, которое определяет отношение «один-ко-многим» к другой модели. В этом случае вы можете «индексировать» поля в связанной модели с дополнительными двойными подчеркиваниями. Так, например, чтобы выбрать по фильтру книги с определенным жанром, нужно указать имя в поле жанра:

```
books_containing_genre =  
    Book.objects.filter(genre__name__icontains='Фантастика')
```

При использовании этого кода из БД будут выбраны все книги из жанров: фантастика, научная фантастика.

8.4. Формирование моделей данных для сайта «Мир книг»

Итак, мы подошли к тому моменту, когда можно приступить к формированию моделей для сайта «Мир книг». Фактически мы проектируем структуру базы данных: определяем перечень таблиц, создаем в них поля, формируем связи между таблицами. Для этого откроем PyCharm, загрузим ранее созданный проект `World_books` и откроем файл `models.py`, который находится в приложении `catalog` (путь к этому файлу: `World_books\WebBooks\catalog\models.py`). В шаблоне для размещения кода в верхней части страницы присутствует строка, которая импортирует модуль для проектирования моделей (рис. 8.18).

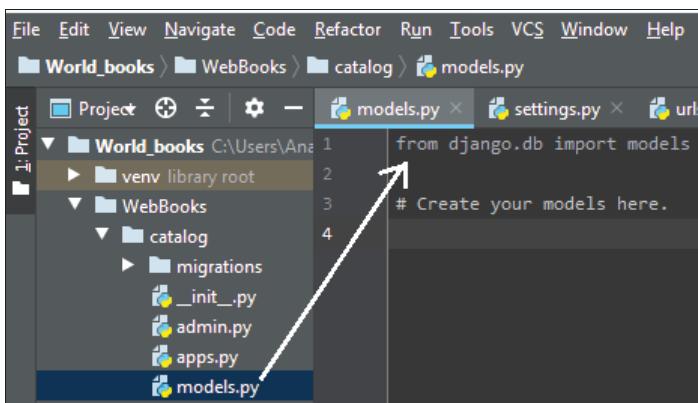


Рис. 8.18. Файл-шаблон `models.py` для проектирования моделей данных сайта «Мир книг»

Таким образом, в этом файле уже импортирован базовый класс `models.Model`, от которого будут наследоваться все наши модели данных. А также закомментирована строка, которую можно перевести как **Создавайте свои модели здесь.**

Добавим в этот файл строку импорта класса `reverse` (выделена серым фоном и полужирным шрифтом), который обеспечит получение абсолютных URL-адресов (эта

его возможность будет в дальнейшем использована для получения ссылки на страницы с деталями списков из самих списков данных):

```
from django.db import models
from django.urls import reverse
```

А теперь приступим к созданию моделей для нашего сайта.

8.4.1. Модель для хранения жанров книг

Начнем мы с формирования справочника жанров книг. Для хранения информации о жанрах книг спроектируем модель (таблицу) с именем `Genre` (листинг 8.3). Эта модель будет содержать всего одно поле — `name`, в котором мы станем хранить наименования жанров книг.

Листинг 8.3

```
class Genre(models.Model):
    name = models.CharField(max_length=200,
                           help_text=" Введите жанр книги",
                           verbose_name="Жанр книги")

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

Как можно видеть, модель `Genre` содержит одно текстовое поле (типа `CharField`), которое называется `name`. Это поле будет использоваться для описания жанра книги — оно ограничено 200 символами и имеет текст подсказки: `help_text`. В качестве удобно читаемого имени (`verbose_name`) указано значение "Жанр книги", поэтому именно оно будет выводиться на экран в формах рядом с полем `name`. В конце модели объявлен метод `__str__()`, который просто возвращает имя жанра, которое будет внесено в конкретную запись.

8.4.2. Модель для хранения языков книг

Продолжим формирование справочников и создадим справочник языков, на которых написаны книги. Для хранения информации о языках спроектируем модель (таблицу) с именем `Language` (листинг 8.4). Эта модель будет содержать всего одно поле — `lang`, в котором мы будем хранить наименования языков.

Листинг 8.4

```
class Language(models.Model):
    name = models.CharField(max_length=20,
                           help_text=" Введите язык книги",
                           verbose_name="Язык книги")
```

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

Как можно видеть, модель `Language` содержит одно текстовое поле (типа `CharField`), которое называется `name`. Это поле будет использоваться для хранения языка книги — оно ограничено 20 символами и имеет текст подсказки: `help_text`. В качестве удобно читаемого имени (`verbose_name`) указано значение "Язык книги", поэтому именно оно будет выводиться на экран в формах рядом с полем `name`. В конце модели объявлен метод `__str__()`, который просто возвращает наименование языка книги, которое будет внесено в конкретную запись.

8.4.3. Модель для хранения авторов книг

Теперь создадим справочник, в котором будем хранить сведения об авторах книг. Для хранения информации об авторах спроектируем модель (таблицу) с именем `Author` (листинг 8.5). Эта модель будет содержать несколько полей: `first_name` — имя автора, `last_name` — фамилия автора, `date_of_birth` — дата его рождения, `date_of_death` — дата смерти.

Листинг 8.5

```
class Author(models.Model):
    first_name = models.CharField(max_length=100,
                                   help_text="Введите имя автора",
                                   verbose_name="Имя автора")
    last_name = models.CharField(max_length=100,
                                 help_text="Введите фамилию автора",
                                 verbose_name="Фамилия автора")
    date_of_birth = models.DateField(
        help_text="Введите дату рождения",
        verbose_name="Дата рождения",
        null=True, blank=True)
    date_of_death = models.DateField(
        help_text="Введите дату смерти",
        verbose_name="Дата смерти",
        null=True, blank=True)

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

Как можно видеть, модель `Author` содержит текстовые поля (типа `CharField`) для хранения имен и фамилий авторов. Эти поля ограничены 100 символами, имеют текст подсказки: `help_text` и удобно читаемые имена (`verbose_name`). Два поля для ввода дат (типа `DateField`) предназначены для хранения даты рождения и даты смерти автора книги. Они могут содержать пустые значения (`null=True`), т. е. не обязательны для заполнения.

Метод `__str__()` возвращает фамилию автора.

8.4.4. Модель для хранения книг

Прежде чем проектировать модель для хранения данных о книгах, рассмотрим, какие связи нужно предусмотреть для тех справочников, которые мы уже создали: жанр, язык и авторы.

Здесь мы исходим из того, что каждая книга будет относиться только к одному жанру. При этом один жанр может соответствовать множеству книг. Значит, здесь присутствует связь «один-ко-многим» (рис. 8.19).

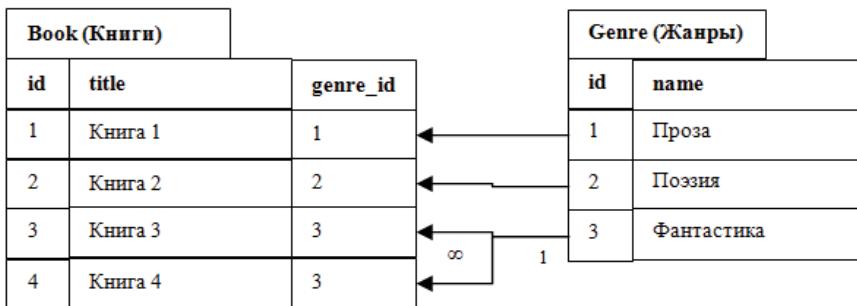


Рис. 8.19. Связь «один-ко-многим» между жанрами книг и книгами

Записи в таблице `Genre` не повторяются — они уникальны и существуют в единственном числе (в начале линии, показывающей связь, стоит **1**). А вот в таблице `Book` может быть множество книг, которые относятся к одному и тому же жанру (на конце линии связи стоит знак бесконечности). В таком случае `Genre` является *главной* (или родительской) таблицей, а `Book` — *связанной* (или дочерней) таблицей.

Что касается языка, на котором написана книга, то мы исходим из того, что каждая книга написана на одном языке. При этом один язык может соответствовать множеству книг. Значит, здесь тоже присутствует связь «один-ко-многим» (рис. 8.20).

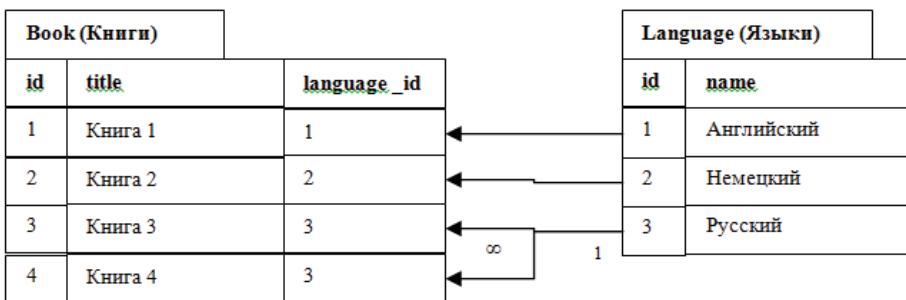


Рис. 8.20. Связь «один-ко-многим» между языками и книгами

Записи в таблице `Language` не повторяются — они уникальны и существуют в единственном числе (в начале линии, показывающей связь, стоит **1**). А вот в таблице `Book` может быть множество книг, которые написаны на одном языке (на конце линии связи стоит знак бесконечности).

ний связи стоит знак бесконечности). В таком случае `Language` является главной (или родительской) таблицей, а `Book` — связанной (или дочерней) таблицей.

Несколько сложнее обстоит дело с книгами и их авторами. Каждая книга может быть написана несколькими авторами, но и автор может написать несколько книг (как самостоятельно, так и в соавторстве). Значит, здесь присутствует связь «многие-ко-многим», а для реализации такой связи необходим новый объект в модели, или новая связующая таблица в базе данных (рис. 8.21).

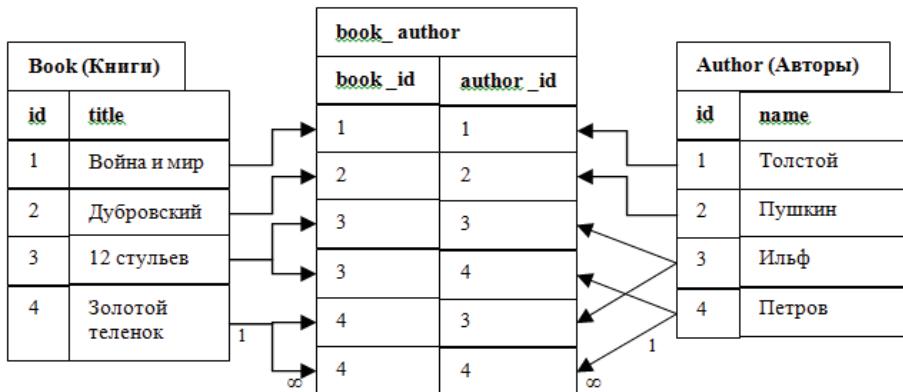


Рис. 8.21. Связь «многие-ко-многим» между авторами и книгами

Как можно видеть, книги «Война и мир» и «Дубровский» имеют по одному автору (Толстой и Пушкин). А вот следующие две книги «12 стульев» и «Золотой теленок» — имеют двух авторов (Ильф и Петров). Записи в таблицах `Book` и `Author` не повторяются — они уникальны и существуют в единственном числе (в начале линии, показывающей связь, стоит 1). А вот в связывающей их таблице `book_author` может быть как множество книг, так и множество авторов (на конце линии связи стоит знак бесконечности). В таком случае таблицы `Book` и `Author` являются главными и равнозначными, а `book_author` — связующей таблицей, которая содержит только индексы связанных записей из главных таблиц.

Вот теперь, с учетом рассмотренных связей, можно приступить к созданию таблицы для самих книг. В ней мы будем хранить всю информацию о собственно книгах (название книги, жанр, язык, автор и т. п.), но не о конкретных (физических) экземплярах книг. Каждый экземпляр книги может находиться в различных состояниях, и для этого будет создана отдельная модель (таблица в БД).

Информация о книгах будет содержаться в следующих полях:

- `title` — название книги;
- `genre` — жанр книги;
- `language` — язык;
- `author` — автор;

- summary — аннотация книги;
- isbn — уникальный международный номер книги.

Описание модели данных о книгах должно выглядеть следующим образом (листинг 8.6).

Листинг 8.6

```
class Book(models.Model):  
    title = models.CharField(max_length=200,  
                            help_text="Введите название книги",  
                            verbose_name="Название книги")  
    genre = models.ForeignKey('Genre', on_delete=models.CASCADE,  
                            help_text=" Выберите жанр для книги",  
                            verbose_name="Жанр книги", null=True)  
    language = models.ForeignKey('Language',  
                                on_delete=models.CASCADE,  
                                help_text="Выберите язык книги",  
                                verbose_name="Язык книги", null=True)  
    author = models.ManyToManyField('Author',  
                                help_text="Выберите автора книги",  
                                verbose_name="Автор книги", null=True)  
    summary = models.TextField(max_length=1000,  
                            help_text="Введите краткое описание книги",  
                            verbose_name="Аннотация книги")  
    isbn = models.CharField(max_length=13,  
                            help_text="Должно содержать 13 символов",  
                            verbose_name="ISBN книги")  
  
    def __str__(self):  
        return self.title  
  
    def get_absolute_url(self):  
        # Возвращает URL-адрес для доступа к  
        # определенному экземпляру книги.  
        return reverse('book-detail', args=[str(self.id)])
```

В приведенной модели все поля имеют поясняющий текст (`help_text`) и удобно читаемое имя (`verbose_name`).

Для названия книги создано поле `title` — это текстовое поле типа `CharField`. На название книги наложено ограничение 200 символов.

Для хранения информации о жанре книги создано поле `genre`, которое по первичному ключу связано с моделью `Genre`. Это поле допускает наличие пустого значения.

Для хранения информации о языке, на котором написана книга, создано поле `language`, которое по первичному ключу связано с моделью `Language`. Это поле допускает наличие пустого значения.

Для хранения информации об авторе (авторах) книги создано поле `author`, которое по первичному ключу связано с моделью `Author`. Это поле допускает наличие пустого значения.

Для хранения аннотации книги создано поле `summary` — это текстовое поле типа `CharField`. На аннотацию книги наложено ограничение 1000 символов.

Для хранения уникального международного номера книги создано поле `isbn` — это текстовое поле типа `CharField`. На уникальный номер книги наложено ограничение 13 символов.

Метод `__str__()` возвращает наименование книги.

В конце этого кода определен метод `get_absolute_url()`, который возвращает URL-адрес, — его можно использовать для доступа к детальным записям для этой модели. Например, на странице пользователю выдана строка с названием книги, которое подсвечивается как ссылка на другую страницу. Если щелкнуть мышью на название книги, то будет загружена страница, например, с детальной информацией об этой книге. То есть наличие такой функции позволит сопоставить URL-адреса на страницы, в которых содержится подробная информация о книге, с соответствующим представлением и шаблоном HTML-страницы.

8.4.5. Модель для хранения отдельных экземпляров книг и их статуса

Вполне возможно, что одна и та же книга будет иметь несколько экземпляров, и каждый экземпляр может иметь некое состояние. Например, на книгу в книжном интернет-магазине сделан заказ — она находится на складе, но еще не доставлена покупателю (стоит в резерве). При этом другие книги тоже находятся на складе, но еще не нашли своего покупателя (выставлены на продажу). Если это библиотека, то конкретный экземпляр книги может находиться в хранилище или в читальном зале, а может быть выдан читателю на дом. Если библиотечная книга выдана читателю, то резонно хранить дату ее возврата. Таким образом, необходимо создать еще одну модель (таблицу в БД), которая будет предназначена для хранения сведений о каждом конкретном экземпляре книги. Назовем эту модель `BookInstance` (экземпляр книги).

Модель `BookInstance` будет иметь связи сразу с двумя таблицами: модель данных (таблица) `Book`, где хранятся сведения о книгах, и модель данных (таблица) `Status`, где содержится информация о статусе (состоянии) конкретного экземпляра книги (рис. 8.22).

Как здесь показано, в базе данных интернет-магазина имеется информация о трех экземплярах книги «Война и мир» и трех экземплярах книги «Дубровский». При этом два экземпляра книги «Война и мир» находятся на складе, а один экземпляр заказан покупателем, но еще не продан. Все три экземпляра книги «Дубровский» проданы.

Таблицы `Book` и `Status` содержат всего по одной записи об объекте, а в таблице `BookInstance` имеется как множество записей о самих объектах (книгах), так и мно-

жество записей об их состоянии (статусе). То есть мы имеем дело со связями «один-ко-многим». Модель для книг Book у нас уже создана, значит, нам нужно создать еще две модели: Status и BookInstance. Начнем с модели Status (листинг 8.7).

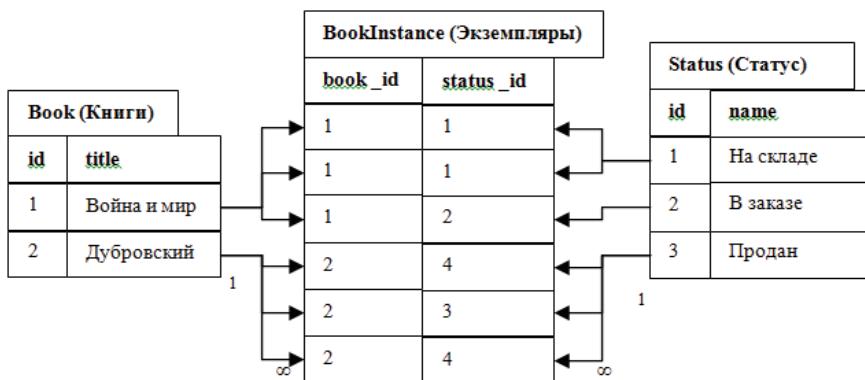


Рис. 8.22. Связи между книгами, экземплярами книг и статусом каждого экземпляра

Листинг 8.7

```
class Status(models.Model):
    name = models.CharField(max_length=20,
                           help_text="Введите статус экземпляра книги",
                           verbose_name="Статус экземпляра книги")

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

Как можно видеть, модель `Status` содержит одно текстовое поле (типа `CharField`), которое называется `name`. Это поле будет использоваться для описания статуса книги — оно ограничено 20 символами и имеет текст подсказки: `help_text`. В качестве удобно читаемого имени (`verbose_name`) указано значение "Статус экземпляра книги", поэтому именно оно будет выводиться на экран в формах рядом с полем `name`. В конце модели объявлен метод `_str_()`, который просто возвращает значение статуса экземпляра книги, которое будет внесено в конкретную запись.

Теперь можно перейти к созданию модели BookInstance (листинг 8.8).

Листинг 8.8

```
imprint = models.CharField(max_length=200,
                           help_text="Введите издательство и год выпуска",
                           verbose_name="Издательство")
status = models.ForeignKey('Status', on_delete=models.CASCADE,
                           null=True,
                           help_text='Изменить состояние экземпляра',
                           verbose_name="Статус экземпляра книги")
due_back = models.DateField(null=True, blank=True,
                           help_text="Введите конец срока статуса",
                           verbose_name="Дата окончания статуса")

def __str__(self):
    return '%s %s %s' % (self.inv_nom, self.book, self.status)
```

Для названия книги создано поле `book`, значение для этого поля будет подгружено из модели `Book`, с которой имеется связь по первичному ключу.

Для хранения информации об издательстве (наименование издательства, год выпуска книги и т. п.) создано поле `imprint` — это текстовое поле типа `CharField`, ограниченное 200 символами.

Для задания статуса экземпляра книги (на складе, в заказе, продана и пр.) используется поле `status`, значение для этого поля будет подгружено из модели `Status`, с которой имеется связь по первичному ключу.

Для задания даты окончания действия статуса для экземпляра книги (например, даты окончания действия заказа) создано поле `due_back` — это поле для ввода дат типа `DateField`.

Для сортировки экземпляров книг по умолчанию создана модель `Meta`, при этом все экземпляры будут отсортированы по дате окончания действия статуса.

Модель `__str__()` представляет объект `BookInstance`, используя название книги, ее инвентарный номер и статус.

Будем считать, что весь приведенный здесь код внесен в файл `models.py`, расположенный по пути: `World_books\WebBooks\catalog\models.py`.

Таким образом, на этом этапе все модели созданы, и можно приступить к миграции данных. Как было отмечено в предыдущей главе, миграция выполняется в два этапа. На первом шаге с помощью менеджера Django формируется программный код, который будет создавать таблицы в БД. Для этого в окне терминала PyCharm выполните команду:

```
python manage.py makemigrations
```

В результате будет создан файл миграции: `World_books\WebBooks\catalog\migrations\0001_initial.py` и получено следующее сообщение (рис. 8.23).

В этом сообщении говорится, что созданы 6 моделей, добавлены 2 поля и выдано предупреждение, что отношение «многие-ко-многим» не допускает наличия `null=True`. Исправим это в следующем коде модели `Book`:

The screenshot shows a PyCharm interface with a terminal window titled 'Terminal: Local'. The command entered is '(venv) C:\Users\Anatoly\PycharmProjects\World_books\WebBooks>python manage.py makemigrations'. The output shows:

```
(venv) C:\Users\Anatoly\PycharmProjects\World_books\WebBooks>python manage.py makemigrations
System check identified some issues:

WARNINGS:
catalog.Book.author: (fields.W340) null has no effect on ManyToManyField.

Migrations for 'catalog':
  catalog\migrations\0001_initial.py
    - Create model Author
    - Create model Book
    - Create model Genre
    - Create model Language
    - Create model Status
    - Create model BookInstance
    - Add field genre to book
    - Add field language to book
```

The left sidebar shows project structure and favorites.

Рис. 8.23. Создание миграции для сайта «Мир книг»

```
author = models.ManyToManyField('Author',
                               help_text="Выберите автора книги",
                               verbose_name="автор книги",
                               null=True)
```

то есть уберем из него параметр `null=True`. Теперь фрагмент этого кода должен выглядеть следующим образом:

```
author = models.ManyToManyField('Author',
                               help_text="Выберите автора книги",
                               verbose_name="автор книги")
```

Снова запустим миграцию и получим сообщение о создании дополнительного файла миграции и об успешном изменении поля `author` в модели `Book` (рис. 8.24).

The screenshot shows a PyCharm interface with a terminal window titled 'Terminal: Local'. The command entered is '(venv) C:\Users\Anatoly\PycharmProjects\World_books\WebBooks>python manage.py makemigrations'. The output shows:

```
(venv) C:\Users\Anatoly\PycharmProjects\World_books\WebBooks>python manage.py makemigrations
Migrations for 'catalog':
  catalog\migrations\0002_auto_20200807_1424.py
    - Alter field author on book
```

Рис. 8.24. Создание дополнительной миграции для сайта «Мир книг»

В файле миграции `0001_initial.py` содержится следующий программный код, обеспечивающий создание таблиц в БД:

```
from django.db import migrations, models
import django.db.models.deletion

class Migration(migrations.Migration):

    initial = True
```

```
dependencies = [
]

operations = [
    migrations.CreateModel(
        name='Author',
        fields=[
            ('id', models.AutoField(auto_created=True,
                                   primary_key=True, serialize=False,
                                   verbose_name='ID')),
            ('first_name', models.CharField(
                help_text='Введите имя автора',
                max_length=100,
                verbose_name='Имя автора')),
            ('last_name', models.CharField(
                help_text='Введите фамилию автора',
                max_length=100,
                verbose_name='Фамилия автора')),
            ('date_of_birth', models.DateField(blank=True,
                                              help_text='Введите дату рождения',
                                              null=True,
                                              verbose_name='Дата рождения')),
            ('date_of_death', models.DateField(blank=True,
                                              help_text='Введите дату смерти',
                                              null=True, verbose_name='Дата смерти')),
        ],
    ),
    migrations.CreateModel(
        name='Book',
        fields=[
            ('id', models.AutoField(auto_created=True,
                                   primary_key=True, serialize=False,
                                   verbose_name='ID')),
            ('title', models.CharField(help_text='Введите название книги',
                                      max_length=200,
                                      verbose_name='Название книги')),
            ('summary', models.TextField(
                help_text='Введите краткое описание книги',
                max_length=1000,
                verbose_name='Аннотация книги')),
            ('isbn', models.CharField(help_text='Должно содержать 13 символов',
                                     max_length=13,
                                     verbose_name='ISBN книги')),
            ('author', models.ManyToManyField(
                help_text='Выберите автора книги',
                verbose_name='Авторы книги'))
        ],
    )
]
```

```
        null=True, to='catalog.Author',
        verbose_name='автор книги')),
    ],
),
migrations.CreateModel(
    name='Genre',
    fields=[
        ('id', models.AutoField(auto_created=True,
            primary_key=True,
            serialize=False, verbose_name='ID')),
        ('name', models.CharField(
            help_text='Введите жанр книги',
            max_length=200,
            verbose_name='Жанр книги')),
    ],
),
migrations.CreateModel(
    name='Language',
    fields=[
        ('id', models.AutoField(auto_created=True,
            primary_key=True, serialize=False,
            verbose_name='ID')),
        ('lang', models.CharField(
            help_text=' Введите язык книги',
            max_length=20,
            verbose_name='Язык книги')),
    ],
),
migrations.CreateModel(
    name='Status',
    fields=[
        ('id', models.AutoField(auto_created=True,
            primary_key=True, serialize=False,
            verbose_name='ID')),
        ('name', models.CharField(
            help_text='Введите статус экземпляра книги',
            max_length=20,
            verbose_name='Статус экземпляра книги')),
    ],
),
migrations.CreateModel(
    name='BookInstance',
    fields=[
        ('id', models.AutoField(auto_created=True,
            primary_key=True, serialize=False,
            verbose_name='ID')),
```

```
('imprint', models.CharField(
    help_text='Введите издательство и год выпуска',
    max_length=200,
    verbose_name='Издательство')),
('due_back', models.DateField(blank=True,
    help_text='Введите конец срока статуса',
    null=True,
    verbose_name='Дата окончания статуса')),
('book', models.ForeignKey(null=True,
    on_delete=django.db.models.deletion.CASCADE,
    to='catalog.Book')),
('status', models.ForeignKey(
    help_text='Изменить состояние экземпляра',
    null=True,
    on_delete=django.db.models.deletion.CASCADE,
    to='catalog.Status',
    verbose_name='Статус экземпляра книги')),
],
),
migrations.AddField(
    model_name='book',
    name='genre', field=models.ForeignKey(
        help_text='Выберите жанр для книги',
        null=True,
        on_delete=django.db.models.deletion.CASCADE,
        to='catalog.Genre',
        verbose_name='Жанр книги'),
),
migrations.AddField(
    model_name='book',
    name='language', field=models.ForeignKey(
        help_text='Выберите язык книги',
        null=True,
        on_delete=django.db.models.deletion.CASCADE,
        to='catalog.Language',
        verbose_name='Язык книги'),
),
]
```

Как можно видеть, Django автоматически сгенерировал программный код, с помощью которого будут созданы соответствующие таблицы в БД. В этом коде содержатся наименования таблиц, наименования полей и их типы, допускаемые значения, метки полей и т. п. Кроме того, описаны дополнительные ключевые поля, о которых не упоминалось в моделях, но которые необходимы для корректной работы системы.

Теперь в окне терминала PyCharm выполним команду, которая на основе сформированного файла миграции создаст таблицы в БД:

```
python manage.py migrate
```

```
Terminal: Local × +  

(venv) C:\Users\Anatoly\PycharmProjects\World_books\WebBooks>python manage.py migrate  

Operations to perform:  

  Apply all migrations: admin, auth, catalog, 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 catalog.0001_initial... OK  

  Applying catalog.0002_auto_20200807_1424... OK  

  Applying sessions.0001_initial... OK
```

Рис. 8.25. Создание таблиц в БД на основе миграции для сайта «Мир книг»

catalog_genre (db0)	catalog_language (db0)	catalog_status (db0)
Структура Данные Огранич >>	Структура Данные Огранич >>	Структура Данные Огранич >>
Имя таблицы: genre WITHOUT ROWID	Имя таблицы: lang WITHOUT ROWID	Имя таблицы: status WITHOUT ROWID
Имя Тип данных Первичный ключ	Имя Тип данных Первичный ключ	Имя Тип данных Первичный ключ
1 id integer	1 id integer	1 id integer
2 name varchar (200)	2 lang varchar (20)	2 name varchar (20)

catalog_book (db0)	catalog_author (db0)	catalog_book_author (db0)
Структура Данные Огранич >>	Структура Данные Огранич >>	Структура Данные Огранич >>
Имя таблицы: book WITHOUT ROWID	Имя таблицы: thor WITHOUT ROWID	Имя таблицы: bthor WITHOUT ROWID
Имя Тип данных Первичный ключ	Имя Тип данных Первичный ключ	Имя Тип данных Первичный ключ
1 id integer	1 id integer	1 id integer
2 title varchar (200)	2 first_name varchar (100)	2 book_id integer
3 summary text	3 last_name varchar (100)	3 author_id integer
4 isbn varchar (13)	4 date_of_birth date	4 date_of_death date
5 genre_id integer	5	
6 language_id integer		

Рис. 8.26. Таблицы в БД, созданные на основе миграции для сайта «Мир книг»

Если все было сделано правильно, то в окне терминала PyCharm мы получим сообщение об успешном завершении всех операций, связанных с созданием таблиц в БД (рис. 8.25).

Если теперь с помощью менеджера SQLiteStudio открыть базу данных, то мы увидим, что Django действительно создал все таблицы, которые были спроектированы в моделях данных (рис. 8.26).

Теоретически, после этого можно переходить к созданию форм и HTML-страниц нашего сайта, однако перед этим мы познакомимся с еще одним важным элементом Django — встроенным администратором сайта.

8.5. Административная панель Django Admin

Теперь, когда модели для сайта «Мир книг» созданы, добавим некоторые «настоящие» данные о книгах, воспользовавшись для этого административной панелью Django Admin. По ходу дела мы разберемся с тем, как зарегистрировать в ней модели, как войти и внести данные, а также рассмотрим некоторые способы дальнейшего улучшения вида административной панели.

Приложение Django Admin способно использовать ранее разработанные модели данных для автоматического формирования некоторой части сайта, предназначеннной для создания, просмотра, обновления и удаления записей. Это может сэкономить достаточно много времени в процессе разработки сайта, упрощая тестирование ваших моделей на предмет правильности данных. Это также может быть полезным для управления данными на стадии публикации. Разработчики Django рекомендуют использовать это приложение только для управления данными на уровне разработчиков или владельцев сайтов (администраторов, специалистов внутри организации, владеющей сайтом). Это связано с тем, что модельно-ориентированный подход не является лучшим интерфейсом для внешних пользователей, которые, кроме всего прочего, неумелыми действиями могут нарушить структуру моделей данных.

Все необходимые настройки, которые необходимо включить в административное приложение веб-сайта, были сделаны автоматически, когда создавался веб-проект. Остается только добавить в приложение Django Admin разработанные модели данных, т. е. просто зарегистрировать их. После регистрации моделей нужно создать «суперпользователя» (главного администратора сайта), и уже он сможет войти на сайт и создавать в БД записи тестовых данных (книги, авторы, экземпляры книг, жанры и т. д.). Это будет полезным для тестирования представлений и шаблонов, создаваемых при разработке интерфейса сайта для внешних пользователей.

8.5.1. Регистрация моделей данных в Django Admin

Для регистрации моделей в административной части сайта нужно открыть файл `admin.py` по пути: `World_books\WebBooks\catalog\admin.py`.

Изначально в нем ничего нет, кроме одной строки импорта модуля `django.contrib.admin` (рис. 8.27).

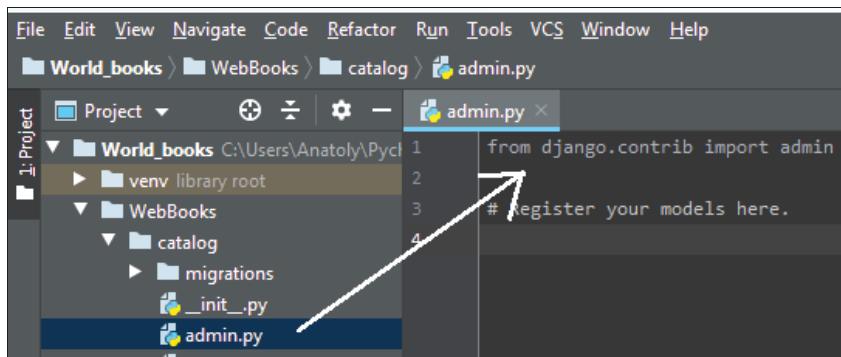


Рис. 8.27. Модуль регистрации моделей admin.py

Зарегистрируем модели, вставив в нижнюю часть этого файла следующий код:

```

from .models import Author, Book, Genre, Language, Status, BookInstance

admin.site.register(Author)
admin.site.register(Book)
admin.site.register(Genre)
admin.site.register(Language)
admin.site.register(Status)
admin.site.register(BookInstance)

```

Этот код просто импортирует указанные модели и затем вызывает модуль `admin.site.register` для регистрации каждой из них.

8.5.2. Работа с данными в Django Admin

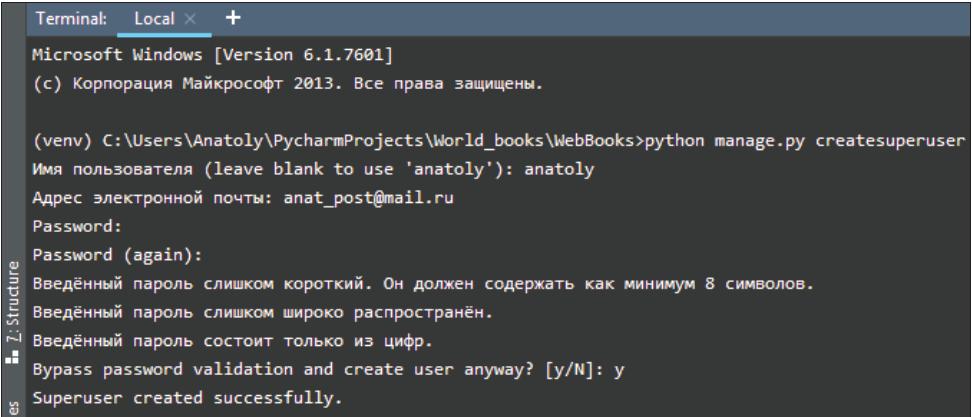
Чтобы войти в административную панель Django, необходимо иметь учетную запись пользователя со статусом `Staff` (персонал). Для просмотра и создания записей пользователю также понадобится разрешение на управление всеми нашими объектами. Прежде всего создайте учетную запись `superuser` (суперпользователь или главный администратор), которая даст полный доступ к сайту и все необходимые разрешения.

Для создания суперпользователя воспользуйтесь модулем `manage.py` и выполните следующую команду в окне терминала PyCharm:

```
python manage.py createsuperuser
```

После ввода этой команды Django автоматически сформирует имя пользователя (по умолчанию это имя компьютера). Можно оставить предложенное имя и нажать клавишу `<Enter>`, а можно набрать другое имя суперпользователя и подтвердить его также нажатием клавиши `<Enter>`.

Затем вам будет предложено ввести электронный адрес суперпользователя и дважды ввести пароль для входа суперпользователя в административную панель (рис. 8.28).



```
Terminal: Local × +
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт 2013. Все права защищены.

(venv) C:\Users\Anatoly\PycharmProjects\World_books\WebBooks>python manage.py createsuperuser
Имя пользователя (leave blank to use 'anatoly'): anatoly
Адрес электронной почты: anat_post@mail.ru
Password:
Password (again):
Введённый пароль слишком короткий. Он должен содержать как минимум 8 символов.
Введённый пароль слишком широк распространён.
Введённый пароль состоит только из цифр.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Рис. 8.28. Создание суперпользователя сайта «Мир книг» в окне терминала PyCharm

Если введенный вами пароль не удовлетворит требованиям системы (слишком простой, слишком короткий и пр.), будет выдано соответствующее предупреждение. После проверки валидности пароля нужно будет нажать клавишу <Y>, после чего вы получите сообщение, что суперпользователь успешно создан (см. рис. 8.28).

Теперь можно запустить сервер и протестировать вход на административную панель сайта. Для этого выполните в окне терминала PyCharm команду:

```
python manage.py runserver
```

Перейдя на страницу сайта <http://127.0.0.1:8000/>, мы получим на главной странице сайта ранее созданное сообщение (рис. 8.29).

Для входа в панель администратора сайта нужно в адресной строке набрать ссылку на соответствующую страницу: /admin (в нашем случае: <http://127.0.0.1:8000/admin>). После этого откроется окно для идентификации суперпользователя (рис. 8.30).

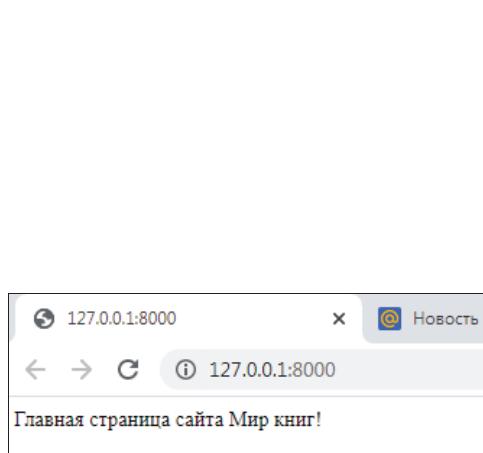


Рис. 8.29. Страница сайта «Мир книг»

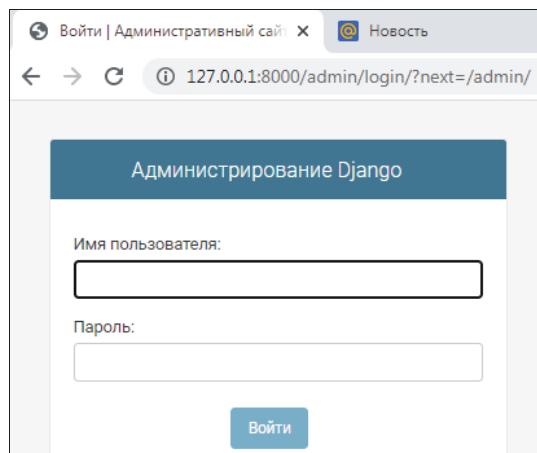


Рис. 8.30. Окно для входа суперпользователя на страницу администратора сайта «Мир книг»

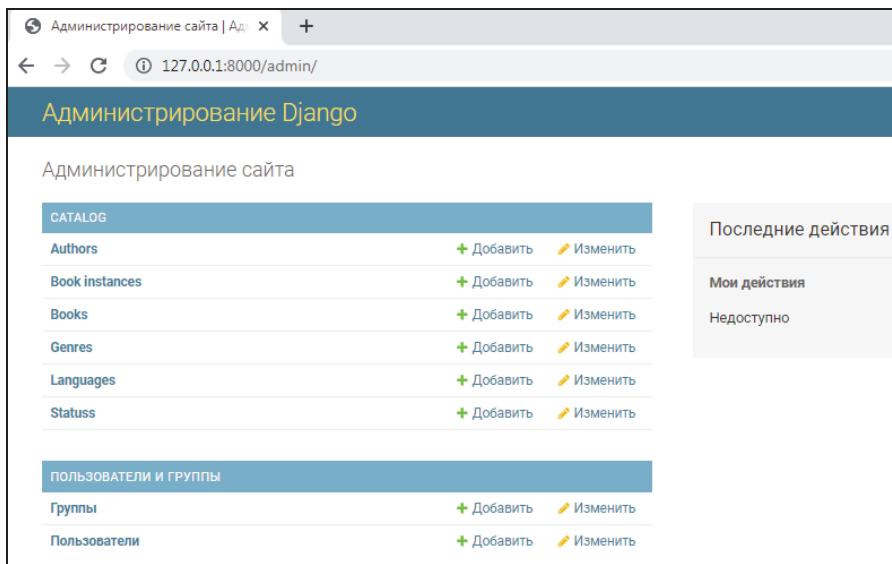


Рис. 8.31. Главное окно администратора сайта «Мир книг»

Введите в соответствующие поля этого окна имя суперпользователя и его пароль, и будет открыто главное окно администратора сайта (рис. 8.31).

В окне администрирования сайта отображаются все модели по установленному приложению. Здесь можно щелкнуть на названии модели, чтобы получить список всех связанных записей, а затем щелкнуть на этих записях для их редактирования. Вы также можете непосредственно перейти по ссылкам **Добавить** или **«Изменить**, расположенным рядом с каждой моделью, чтобы начать создание или изменение записи этого типа.

Итак, мы зарегистрировались, создали учетную запись суперпользователя и вошли с ее помощью в панель администрирования сайта.

Приступим теперь к формированию базовых справочников. Щелкнем мышью на ссылке **Добавить** рядом с моделью **Genres**, описывающей жанры книг, — откроется окно для ввода жанров (рис. 8.32).

Рис. 8.32. Окно администратора сайта для ввода жанров книг

Введем в этом окне несколько жанров: Детективы, Поэзия, Приключения, Проза, Фантастика, нажимая после ввода очередного жанра клавишу <Enter>. После ввода всех жанров получим следующее окно (рис. 8.33).

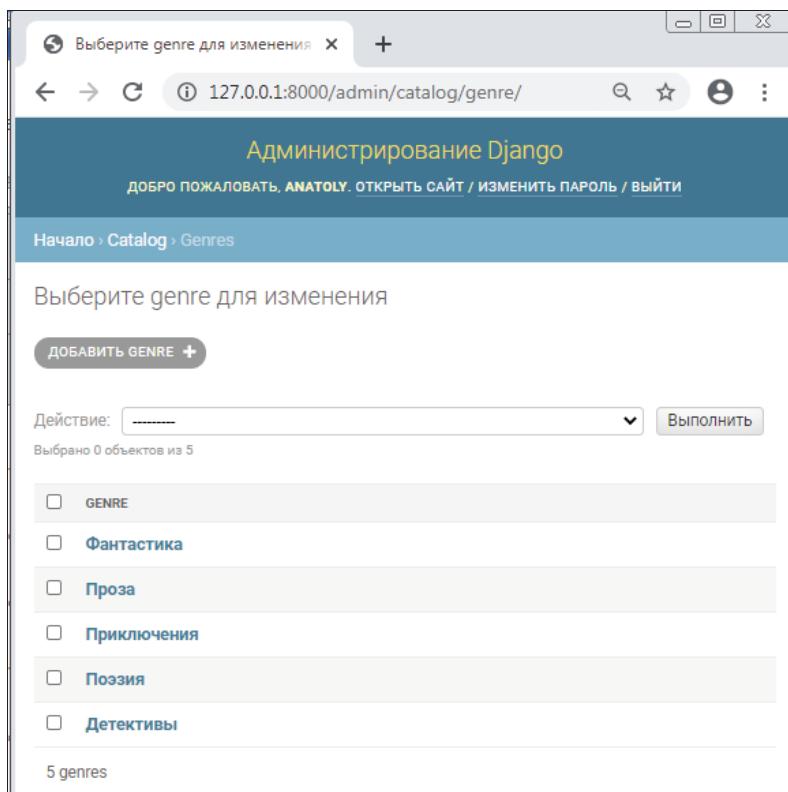


Рис. 8.33. Окно администратора сайта после ввода пяти жанров книг

Если теперь вернуться в главное окно администратора сайта, то вам будут показаны последние действия, которые выполнялись с моделями (рис. 8.34).

Теперь аналогичным образом сформируем справочник языков, на которых написаны книги: Английский, Немецкий, Французский, Русский. Пополним справочник статусов книг: На складе, В заказе, Продана.

Введем сведения о некоторых авторах: Александр Беляев, Александр Пушкин, Лев Толстой, Илья Ильф, Евгений Петров. Поскольку модель `Autors` имеет несколько полей, то при вводе сведений об авторах мы получим окно несколько иного вида (рис. 8.35).

В этом окне для ввода дат будет автоматически подключаться календарь. Дату можно либо выбрать из календаря, либо ввести с клавиатуры. Если при ручном вводе даты будет сделана ошибка, то Django выдаст соответствующее предупреждение (рис. 8.36).

Сформировав базовые справочники, можно перейти к заполнению данными связанных с ними таблиц. Введем в базу данных несколько книг. Для этого щелкнем

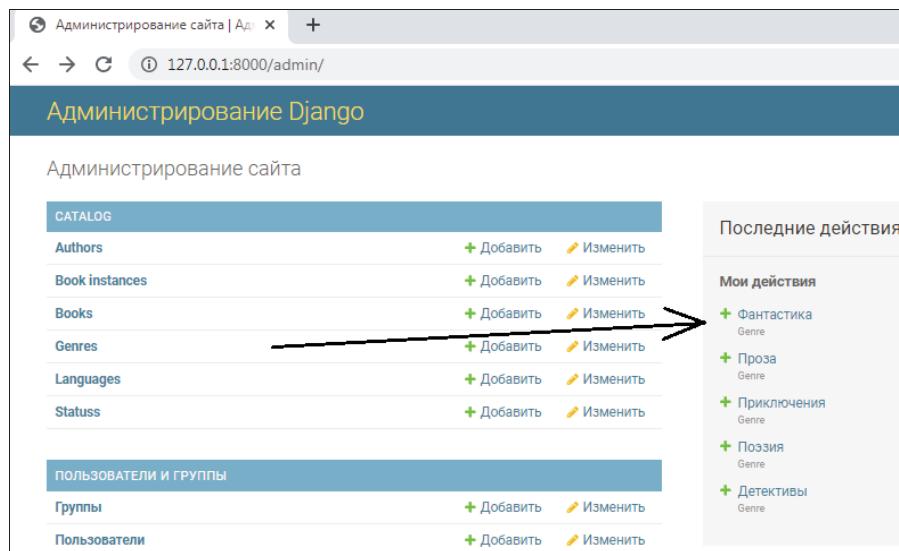


Рис. 8.34. Ведение журнала действий администратора сайта

Рис. 8.35. Окно администратора сайта для ввода сведений об авторах

The screenshot shows the 'Добавить author' (Add author) page in the Django admin interface. At the top, there's a red-bordered error message: 'Пожалуйста, исправьте ошибку ниже.' (Please correct the error below). Below it, the 'Имя автора:' field contains 'Лев' and has a placeholder 'Введите имя автора'. The 'Фамилия автора:' field contains 'Толстой' and has a placeholder 'Введите фамилию автора'. Under 'Дата рождения:', the date '26.08.1828' is shown with a link to 'Сегодня | 📅' and a placeholder 'Введите дату рождения'. Under 'Дата смерти:', the date '07.11ю1910' is shown with a link to 'Сегодня | 📅' and a placeholder 'Введите дату смерти'.

Рис. 8.36. Сообщение об ошибке в окне администратора сайта при вводе ошибочных данных

мышью на ссылке **Добавить** рядом с моделью **Books** — откроется следующее окно (рис. 8.37).

В этом окне видно, что на текущий момент в БД нет ни одной книги, но при этом присутствует кнопка **ДОБАВИТЬ BOOK**, позволяющая войти в режим ввода сведений о книгах. После нажатия на эту кнопку будет открыто основное окно для ввода сведений о книгах (рис. 8.38).

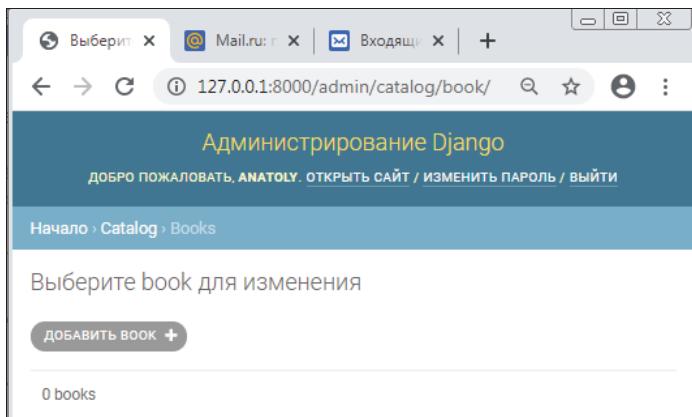


Рис. 8.37. Начальное окно администратора сайта для добавления новой книги

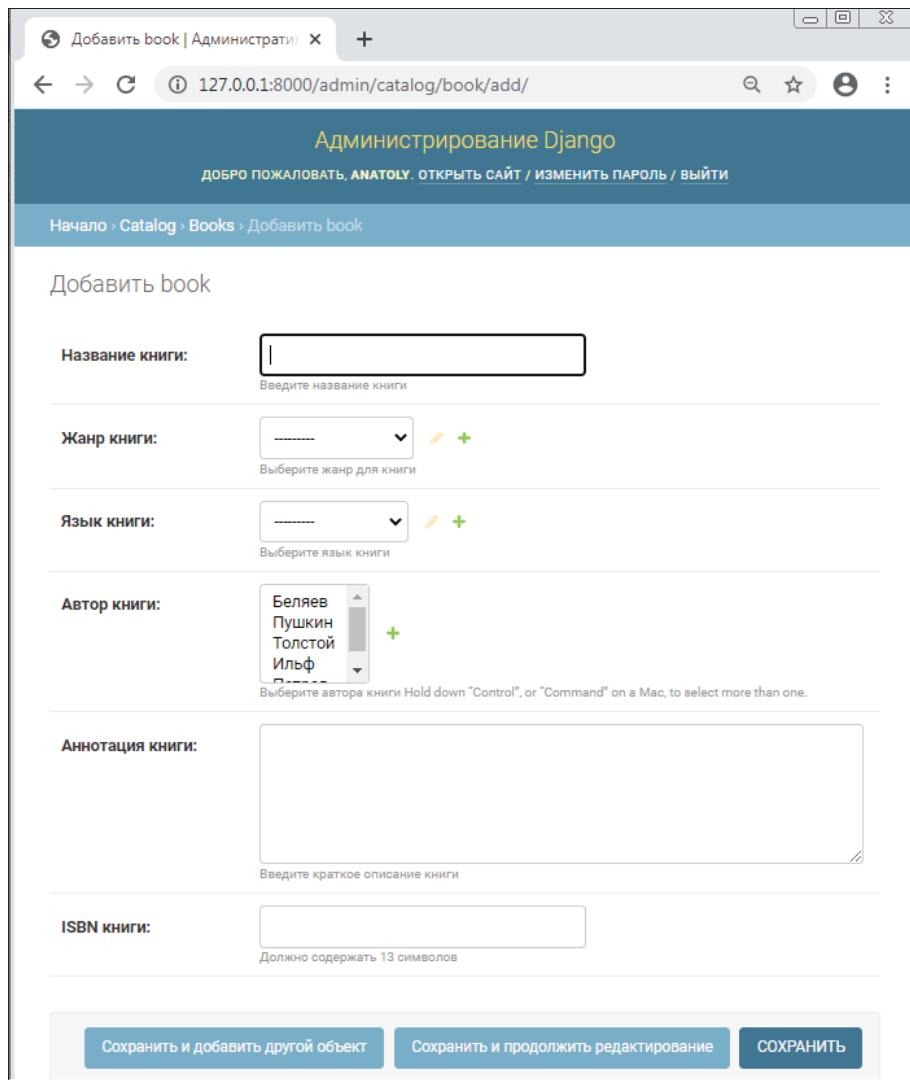


Рис. 8.38. Основное окно администратора сайта для добавления информации о книгах

Вспомним, что модель «Книги» связана с моделями «Жанр книги» и «Язык книги» связью «один-ко-многим». То есть один жанр — много книг, один язык — много книг. В связи с этим для выбора жанра и языка Django использует поле выбора в виде выпадающего списка. А вот книги и авторы имеют связь «многие-ко-многим». Поэтому для выбора автора (авторов) книги используется поле в виде простого списка, в котором можно одновременно выбрать несколько авторов. Рядом с этим полем имеется подсказка — для выбора нескольких авторов необходимо щелкнуть в списке на фамилии автора, удерживая при этом клавишу `<Ctrl>`. Рядом с полями ввода жанра книги, языка и автора книги имеется значок . Если на нем щелкнуть мышью, то появится новое окно, в котором можно пополнить соответствующий справочник.

Введем в этом окне хотя бы по одной книге каждого автора. А для Ильфа и Петрова введем две книги, чтобы проверить, как будет работать связь между таблицами «многие-ко-многим». Результаты ввода сведений о книгах представлены на рис. 8.39.

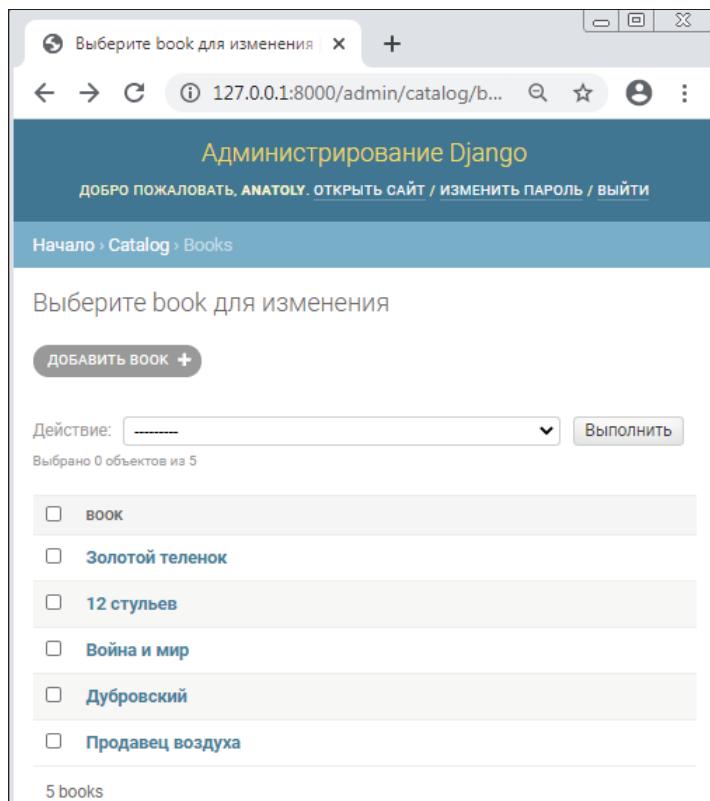


Рис. 8.39. Окно администратора сайта с добавленной информацией о пяти книгах

Если щелкнуть мышью на названии книги, то будет показано окно с полной информацией об этой книге (рис. 8.40).

Итак, мы успешно завели в БД несколько книг. Теперь введем сведения о конкретных экземплярах книг и их статусе. Для этого в главном окне администратора сайта (см. рис. 8.31) щелкнем на ссылке **Добавить** рядом с моделью **Book Instances**, после чего откроется соответствующее окно (рис. 8.41).

Здесь можно задать параметры для каждого отдельного экземпляра книги и указать статус этого экземпляра. Для тестирования нашего приложения введем информацию о нескольких экземплярах каждой книги. Результаты ввода данных об экземплярах книг представлены на рис. 8.42.

Итак, мы заполнили информацией все наши модели, а вернее, соответствующие таблицы в БД. Мы в этом можем убедиться, если откроем содержимое таблиц БД с помощью менеджера SQLiteStudio (рис. 8.43).

Администрирование Django

ДОБРО ПОЖАЛОВАТЬ, ANATOLY. ОТКРЫТЬ САЙТ / ИЗМЕНИТЬ ПАРОЛЬ / ВЫЙТИ

Начало > Catalog > Books > Золотой теленок

Изменить book

ИСТОРИЯ

Название книги: Золотой теленок
Введите название книги

Жанр книги: Проза Выберите жанр для книги

Язык книги: Русский Выберите язык книги

Автор книги: Пушкин
Толстой
Ильф
Петров Выберите автора книги Hold down "Control", or "Command" on a Mac, to select more than one.

Аннотация книги: "Золотой теленок" Ильфа и Петрова - продолжение невероятно остроумного романа "Двенадцать стульев". С непревзойденной ловкостью неунывающие жулики и авантюристы продолжают наказывать глупых и жадных. А читатели из поколения в поколение следят за приключениями героев, от души веселясь и симпатизируя очаровательным мошенникам.
Переведенный на пятьдесят языков и выдержавший несколько...
Введите краткое описание книги

ISBN книги: 978517098229
Должно содержать 13 символов

Удалить **Сохранить и добавить другой объект** **СОХРАНИТЬ**
Сохранить и продолжить редактирование

Рис. 8.40. Окно администратора сайта с информацией о книге «Золотой теленок»

Таким образом, используя административную панель, мы смогли сформировать структуру таблиц БД и заполнить их информацией через веб-интерфейс, при этом не создав ни одной HTML-страницы и не использовав SQL-запросы. Весь необходимый интерфейс был нам предоставлен внутренней структурой Django. И это еще не все — в Django имеется возможность изменить настройки и внешний вид этого интерфейса, знакомству с чем и будет посвящен следующий раздел.

Администрирование Django
ДОБРО ПОЖАЛОВАТЬ, ANATOLY. ОТКРЫТЬ САЙТ / ИЗМЕНИТЬ ПАРОЛЬ / ВЫТИ

Начало > Catalog > Book instances > Добавить book instance

Добавить book instance

Book:

Инвентарный номер:
Введите инвентарный номер экземпляра

Издательство:
Введите издательство и год выпуска

Статус экземпляра книги:
Изменить состояние экземпляра

Дата окончания статуса: Сегодня

Сохранить и добавить другой объект Сохранить и продолжить редактирование СОХРАНИТЬ

Рис. 8.41. Окно администратора сайта для ввода информации об экземплярах книг

Администрирование Django
ДОБРО ПОЖАЛОВАТЬ, ANATOLY. ОТКРЫТЬ САЙТ / ИЗМЕНИТЬ ПАРОЛЬ / ВЫТИ

Начало > Catalog > Book instances

Выберите book instance для изменения

Действие:	<input type="button"/>	<input type="button" value="Выполнить"/>	Выбрано 0 объектов из 7
<input type="checkbox"/> BOOK INSTANCE			
<input type="checkbox"/> №7 Золотой теленок Продана			
<input type="checkbox"/> №6 12 стульев В заказе			
<input type="checkbox"/> №5 Война и мир На складе			
<input type="checkbox"/> №4 Дубровский На складе			
<input type="checkbox"/> №3 Продавец воздуха В заказе			
<input type="checkbox"/> №2 Продавец воздуха На складе			
<input type="checkbox"/> №1 Продавец воздуха На складе			

7 book instances

Рис. 8.42. Окно администратора сайта после ввода информации об экземплярах книг

The screenshot shows the MySQL Workbench interface with six database tables:

- catalog_genre (db0)**: Contains genres like Детективы, Поэзия, Приключения, Проза, Фантастика.
- catalog_language (db0)**: Contains languages like Английский, Немецкий, Французский, Русский.
- catalog_status (db0)**: Contains statuses like На складе, В заказе, Продана.
- catalog_book (db0)**: Contains books like Продавец воздуха, Дубровский, Война и мир, 12 стульев, Золотой теленок.
- catalog_author (db0)**: Contains authors like Александр Беляев, Александр Пушкин, Лев Толстой, Илья Ильф, Евгений Петров.
- catalog_bookinstance (db0)**: Contains book instances with details like imprint (ACT, 1917), due back date, book id, status id, and inventory number.
- catalog_book_author (db0)**: A junction table linking books and authors.

Рис. 8.43. Содержимое таблиц БД после ввода необходимой информации через панель администрирования сайта

8.6. Изменение конфигурации административной панели Django

Административная панель Django обеспечивает эффективную помощь разработчикам сайтов, используя информацию из зарегистрированных моделей данных:

- каждая модель имеет набор записей в виде строк, создаваемых методом `__str__()` модели, и связанных с представлением (view) для их редактирования. По умолчанию в верхней части этого представления находится меню действий, которое может быть использовано для удаления нескольких записей за раз;
- формы для редактирования и добавления записей содержат все поля модели, которые расположены вертикально в порядке их объявления в модели.

Однако в административной панели можно настроить интерфейс пользователя для упрощения ее использования. Вот некоторые доступные настройки:

□ List views (список представлений):

- добавление дополнительных отображаемых полей или информации для каждой записи;
- добавление фильтров для отбора записей по разным критериям (например, статус выдачи книги);
- добавление дополнительных вариантов выбора в меню действий и места расположения этого меню на форме.

□ Detail views (подробное представление):

- выбор отображаемых полей, их порядка, группирования и т. д.;
- добавление связанных полей к записи (например, возможности добавления и редактирования записей книг при создании записи автора).

В этом разделе как раз будут рассмотрены некоторые желательные изменения для совершенствования интерфейса пользователя нашего сайта «Мир книг». В частности, включение дополнительной информации в списки моделей `Book` и `Author`, а также улучшение расположения элементов соответствующих представлений редактирования.

8.6.1. Регистрация класса `ModelAdmin`

Для изменения отображения модели в пользовательском интерфейсе административной панели необходимо определить класс `ModelAdmin` (он описывает расположение элементов интерфейса, где `Model` — наименование модели) и зарегистрировать его для использования с этой моделью.

Начнем с модели `Author`. Откройте файл `admin.py` в каталоге приложения (по пути `World_books\WebBooks\catalog\admin.py`).

Закомментируйте исходную регистрацию этой модели, используя префикс #:

```
# admin.site.register(Author)
```

Теперь добавьте новый класс `AuthorAdmin` и зарегистрируйте его так, как показано здесь:

```
# Определения к классу администратор
class AuthorAdmin(admin.ModelAdmin):
    pass

# Зарегистрируйте класс admin с соответствующей моделью
admin.site.register(Author, AuthorAdmin)
```

Затем добавьте классы `ModelAdmin` для моделей `Book` и `BookInstance`. И снова прежде закомментируйте ранее сделанную регистрацию:

```
# admin.site.register(Book)
# admin.site.register(BookInstance)
```

В этот раз для создания и регистрации новых моделей мы воспользуемся декоратором `@register` (он делает то же самое, что и метод `admin.site.register()`):

```
# Регистрируем классы администратора для книг
@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    pass

# Регистрируем классы администратора для экземпляра книги

@admin.register(BookInstance)
class BookInstanceAdmin(admin.ModelAdmin):
    pass
```

Пока что все наши admin-классы пустые (см. pass), поэтому в интерфейсе администраторской панели ничего не изменилось. Добавим код для задания особенностей интерфейса моделей.

8.6.2. Настройка отображения списков

В текущий момент приложение «Мир книг» отображает всех авторов, используя имя объекта, возвращаемое методом `_str_()` модели. Это приемлемо, когда есть только несколько авторов. Однако если их большое количество, то возможно появление дублирующих записей. Чтобы как-то различить авторов или просто отобразить дополнительную информацию о каждом авторе, можно использовать для добавления дополнительных полей кортеж `list_display`.

Заменим класс `AuthorAdmin` кодом, приведенным в листинге 8.9. Названия полей, которые будут отображаться в списке, приведены в кортеже `list_display` в требуемом порядке (это те же имена, что и в исходной модели).

Листинг 8.9

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('last_name', 'first_name', 'date_of_birth', 'date_of_death')
```

Если теперь перезапустить сайт и перейти к списку авторов, то указанные поля должны отображаться в виде таблицы (рис. 8.44).

Для нашей модели, описывающей книги (`Book`), где отображалось только название книги, добавим отображение полей: `genre`, `language` и `author`.

Поля `genre` и `language` имеют внешний ключ (`ForeignKey`) связи «один-ко-многим», поэтому они будут представлены значением `_str_()` для связанной записи. Замените класс `BookAdmin` на версию, приведенную в листинге 8.10.

Листинг 8.10

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'genre', 'language', 'display_author')
```

<input type="checkbox"/>	ФАМИЛИЯ АВТОРА	ИМЯ АВТОРА	ДАТА РОЖДЕНИЯ	ДАТА СМЕРТИ
<input type="checkbox"/>	Петров	Евгений	30 декабря 1902 г.	2 июля 1942 г.
<input type="checkbox"/>	Ильф	Илья	3 октября 1897 г.	13 апреля 1937 г.
<input type="checkbox"/>	Толстой	Лев	26 августа 1828 г.	7 ноября 1910 г.
<input type="checkbox"/>	Пушкин	Александр	26 мая 1799 г.	10 февраля 1837 г.
<input type="checkbox"/>	Беляев	Александр	16 марта 1884 г.	6 января 1942 г.

5 authors

Рис. 8.44. Сведения об авторах, показанные в виде таблицы с помощью кортежа `list_display`

К сожалению, мы не можем напрямую поместить поле `author` в `list_display`, т. к. между авторами и книгами имеется связь «многие-ко-многим» (`ManyToManyField`). Вместо этого мы определим функцию `display_author` в модели данных `Book` для получения строкового представления информации об авторах (листинг 8.11).

Листинг 8.11

```
def display_author(self):
    return ', '.join([author.last_name for author in
                     self.author.all()])
display_author.short_description = 'Авторы'
```

Эту функцию необходимо вставить в модель `Book` в файле `models.py` (рис. 8.45).

После этих изменений нужно сделать миграцию данных, перезапустить наш сервер и снова войти в административную панель. Если теперь нажать на ссылку со списком книг, то мы получим такую страницу (рис. 8.46).

8.6.3. Добавление фильтров к спискам

Если на экране в списке присутствует множество элементов, то имеется возможность отфильтровать эти данные по некоторым признакам. Это выполняется путем указания необходимых признаков в атрибуте `list_filter`. Откроем файл `admin.py` и добавим атрибуты `list_filter` в классы `BookAdmin` (книги) и `BookInstanceAdmin` (экземпляры книг), как показано в листинге 8.12.

```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
World_books > WebBooks > catalog > models.py
Project World_books C:\Users\...
  -> venv library root
    -> WebBooks
      -> migrations
        -> __init__.py
        -> admin.py
        -> apps.py
        -> models.py <-- Selected
        -> tests.py
        -> urls.py
      -> views.py
    -> WebBooks
      -> __init__.py
      -> asgi.py
      -> settings.py
      -> urls.py
      -> wsgi.py
    -> db.sqlite3
    -> manage.py
External Libraries
Scratches and Consoles

class Book(models.Model):
    title = models.CharField(max_length=200, help_text="Введите название книги",
                             verbose_name="Название книги")
    genre = models.ForeignKey('Genre', on_delete=models.CASCADE,
                             help_text="Выберите жанр для книги",
                             verbose_name="Жанр книги",
                             null=True)
    language = models.ForeignKey('Language', on_delete=models.CASCADE,
                             help_text="Выберите язык книги",
                             verbose_name="Язык книги",
                             null=True)
    author = models.ManyToManyField('Author', help_text="Выберите автора книги",
                                 verbose_name="автор книги")
    summary = models.TextField(max_length=1000, help_text="Введите краткое описание",
                               verbose_name="Аннотация книги")
    isbn = models.CharField(max_length=13, help_text="Должно содержать 13 символов",
                           verbose_name="ISBN книги")

    def display_author(self):
        return ', '.join([author.last_name for author in self.author.all()])
    display_author.short_description = 'Авторы'

    def __str__(self):
        return self.title

```

Рис. 8.45. Добавление функции `display_author` в модель данных `Book`

НАЗВАНИЕ КНИГИ	ЖАНР КНИГИ	ЯЗЫК КНИГИ	АВТОРЫ
Золотой теленок	Проза	Русский	Ильф, Петров
12 стульев	Проза	Русский	Ильф, Петров
Война и мир	Проза	Русский	Толстой
Дубровский	Проза	Русский	Пушкин
Продавец воздуха	Фантастика	Русский	Беляев

5 books

Рис. 8.46. Развёрнутый список книг на основе модернизированной модели данных `Book`

Листинг 8.12

```
@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'genre', 'language', 'display_author')
    list_filter = ('genre', 'author')

@admin.register(BookInstance)
class BookInstanceAdmin(admin.ModelAdmin):
    list_filter = ('book', 'status')
```

Здесь к классу, описывающему книги, добавлен фильтр на жанр и на авторов книг. Если теперь нажать на ссылку со списком книг, то мы получим такую страницу, на которой справа появятся два фильтра: на названия книг и на их статус (рис. 8.47).

The screenshot shows the Django Admin interface for the 'Catalog' app's 'Books' model. The main page title is 'Выберите book для изменения'. A 'ДОБАВИТЬ BOOK +' button is visible. Below it, a table lists five books:

<input type="checkbox"/>	Название книги	Жанр книги	Язык книги	Авторы
<input type="checkbox"/>	Золотой теленок	Проза	Русский	Ильф, Петров
<input type="checkbox"/>	12 стульев	Проза	Русский	Ильф, Петров
<input type="checkbox"/>	Война и мир	Проза	Русский	Толстой
<input type="checkbox"/>	Дубровский	Проза	Русский	Пушкин
<input type="checkbox"/>	Продавец воздуха	Фантастика	Русский	Беляев

Below the table, it says '5 books'. To the right, there are two filter panels:

- ФИЛЬТР ЖАНР КНИГИ** (Genre Filter):
 - Все
 - Детективы
 - Поэзия
 - Приключения
 - Проза
 - Фантастика
- ФИЛЬТР автор книги** (Author Filter):
 - Все
 - Беляев
 - Пушкин
 - Толстой
 - Ильф
 - Петров

Рис. 8.47. Развернутый список книг с возможностью фильтрации по жанрам и авторам

Если мы теперь в жанрах выберем **Проза**, а в авторах — **Ильф**, то в списке останутся две книги, которые соответствуют этим фильтрам (рис. 8.48).

Если в панели администратора нажать на ссылку со списком экземпляров книг, то мы получим страницу, на которой справа появятся соответствующие фильтры (рис. 8.49).

Администрирование Django
ДОБРО ПОЖАЛОВАТЬ, ANATOLY. ОТКРЫТЬ САЙТ / ИЗМЕНИТЬ ПАРОЛЬ / ВЫЙТИ

Начало > Catalog > Books

Выберите book для изменения

Действие:

Выбрано 0 объектов из 2

<input type="checkbox"/> НАЗВАНИЕ КНИГИ	ЖАНР КНИГИ	ЯЗЫК КНИГИ	АВТОРЫ
<input type="checkbox"/> Золотой теленок	Проза	Русский	Ильф, Петров
<input type="checkbox"/> 12 стульев	Проза	Русский	Ильф, Петров

2 books

ФИЛЬТР

Жанр книги

- Все
- Детективы
- Поэзия
- Приключения
- Проза
- Фантастика

автор книги

- Все
- Беляев
- Пушкин
- Толстой
- Ильф
- Петров

Рис. 8.48. Развёрнутый список книг с фильтром по жанрам Проза и авторам Ильф

Администрирование Django
ДОБРО ПОЖАЛОВАТЬ, ANATOLY. ОТКРЫТЬ САЙТ / ИЗМЕНИТЬ ПАРОЛЬ / ВЫЙТИ

Начало > Catalog > Book instances

Выберите book instance для изменения

Действие:

Выбрано 0 объектов из 7

<input type="checkbox"/> BOOK INSTANCE
<input type="checkbox"/> №7 Золотой теленок Продана
<input type="checkbox"/> №6 12 стульев В заказе
<input type="checkbox"/> №5 Война и мир На складе
<input type="checkbox"/> №4 Дубровский На складе
<input type="checkbox"/> №3 Продавец воздуха В заказе
<input type="checkbox"/> №2 Продавец воздуха На складе
<input type="checkbox"/> №1 Продавец воздуха На складе

7 book instances

ФИЛЬТР

book

- Все
- Продавец воздуха
- Дубровский
- Война и мир
- 12 стульев
- Золотой теленок

Статус экземпляра книги

- Все
- На складе
- В заказе
- Продана

Рис. 8.49. Развёрнутый список экземпляров книг с фильтром по названиям экземпляров и их статусом

8.6.4. Формирование макета с подробным представлением элемента списка

По умолчанию в административной панели в списках отображаются все поля по вертикали в порядке их объявления в модели. Но можно изменить порядок их следования, указать, какие поля нужно отобразить (или исключить), отобразить ли поля горизонтально или вертикально, и даже определить, какие виджеты редактирования использовать в формах администратора.

Модели в нашем проекте «Мир книг» достаточно просты, поэтому нет необходимости менять макет отображения полей, но мы все равно внесем некоторые изменения, чтобы показать, как это можно сделать. Обновим наш класс `AuthorAdmin` в файле `admin.py`: добавим в него атрибут `fields` и в нем укажем поля, которые должны отображаться в форме в порядке их следования. Поля в детальном отображении сведений об авторах по умолчанию будут отображаться по вертикали, но два поля (поля с датами) отобразим горизонтально, для чего дополнительно сгруппируем их в кортеже (листинг 8.13).

Листинг 8.13

```
class AuthorAdmin(admin.ModelAdmin):  
    list_display = ('last_name', 'first_name')  
    fields = ['first_name', 'last_name',  
              ('date_of_birth', 'date_of_death')]
```

The screenshot shows the Django Admin interface for the 'Authors' model. The title bar says 'Выберите author для изменения' (Select an author for change). There is a dropdown menu labeled 'Действие:' (Action:) with a 'Выполнить' (Execute) button. Below this, there is a table with five rows, each containing a checkbox, the author's last name, and their first name. The table has two columns: 'ФАМИЛИЯ АВТОРА' (Last Name) and 'ИМЯ АВТОРА' (First Name). The data is as follows:

	ФАМИЛИЯ АВТОРА	ИМЯ АВТОРА
<input type="checkbox"/>	Петров	Евгений
<input type="checkbox"/>	Ильф	Илья
<input type="checkbox"/>	Толстой	Лев
<input type="checkbox"/>	Пушкин	Александр
<input type="checkbox"/>	Беляев	Александр

At the bottom left, it says '5 authors'. On the right, there is a 'Добавить автора' (Add author) button.

Рис. 8.50. Усеченный список авторов книг

Перезагрузим наше приложение, перейдем на страницу вывода информации об авторах и получим форму в виде таблицы с горизонтальным расположением полей со сведениями об авторах: фамилии и имени (рис. 8.50).

Теперь щелкнем мышью на фамилии одного из авторов — например, **Петров**. После этого появится страница с более детальной информацией о нем. При этом поля будут расположены так, как они приведены в атрибуте `fields` (рис. 8.51).

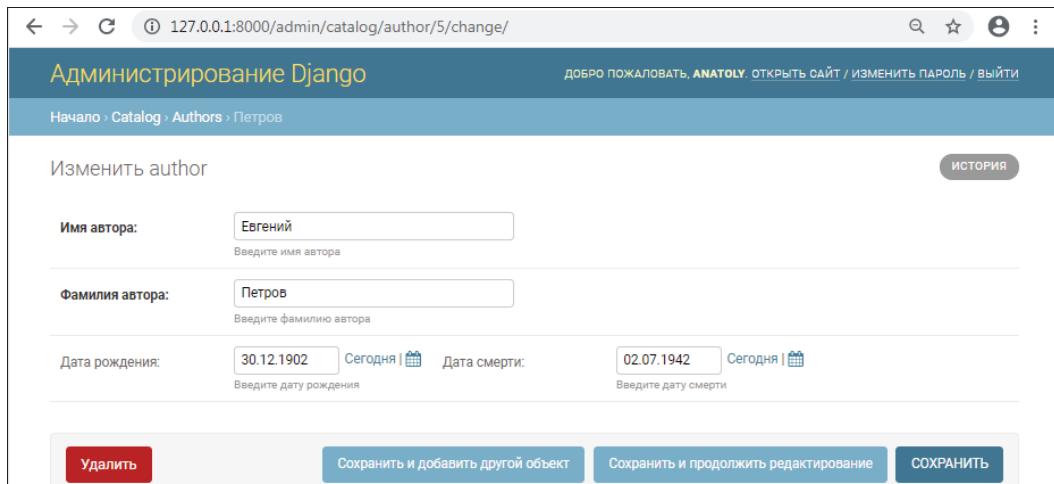


Рис. 8.51. Детальная информация об авторе книги

Вы также можете использовать атрибут `exclude` для объявления списка полей, которые будут исключены из формы (все остальные поля модели продолжат отображаться).

8.6.5. Разделение страницы на секции с отображением связанной информации

В панели администрирования Django есть возможность добавлять на веб-страницы разделы или секции (`sections`) для группировки связанный информации в модели в форме детализации, используя атрибут `fieldsets`.

Так, в модели с данными об экземплярах книг (`BookInstance`) мы имеем информацию о каждом конкретном экземпляре книги (поля `name`, `imprint`, `inv_nom`), а также о статусе экземпляра и дате, когда действие статуса заканчивается (`status`, `due_back`). Мы можем разделить эти поля, разместив их в разные секции. Откроем файл `admin.py` и следующим образом изменим код класса `BookInstanceAdmin` (листинг 8.14).

Листинг 8.14

```
@admin.register(BookInstance)
class BookInstanceAdmin(admin.ModelAdmin):
    list_filter = ('book', 'status')
```

```
fieldsets = (
    ('Экземпляр книги', {
        'fields': ('book', 'imprint', 'inv_nom')
    }),
    ('Статус и окончание его действия', {
        'fields': ('status', 'due_back')
    }),
)
```

Здесь страницу с отображением данных о конкретном экземпляре книги мы разбили на две секции. Каждая секция имеет свой заголовок 'Экземпляр книги' и 'Статус и окончание его действия' (если заголовок не нужен, то вместо него пишем None). В каждой секции сформирован кортеж из полей.

Перезапускаем сайт и переходим сначала к списку экземпляров книг (рис. 8.52).

Рис. 8.52. Форма со списком экземпляров книг

Теперь на этой форме щелкнем мышью на одном из показанных экземпляров. В результате откроется новое окно с подробными сведениями о выбранном экземпляре с разбивкой на секции (рис. 8.53).

Рис. 8.53. Форма с данными об экземпляре книги с разбивкой на секции

8.6.6. Встроенное редактирование связанных записей

Достаточно часто необходимо добавлять связанные записи одновременно с отображением записи главной таблицы. Например, имеет смысл одновременно видеть как информацию о книге, так и сведения о конкретных экземплярах этой книги. Для этого необходимо объявить новый класс `inlines` и указать для него тип расположения: либо `TabularInline` (горизонтальное расположение), либо `StackedInline` (вертикальное расположение).

Открываем файл `admin.py` и в него добавляем код, формирующий класс `BooksInstanceInline`. Затем дописываем строку в класс `BookAdmin`, как это показано в листинге 8.15.

Листинг 8.15

```
class BooksInstanceInline(admin.TabularInline):
    model = BookInstance
```

```
@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'display_genre')
    inlines = [BooksInstanceInline]
```

Эти изменения в коде представлены на рис. 8.54.

Перезапускаем сайт и переходим сначала к списку книг (рис. 8.55).

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help
World_books > WebBooks > catalog > admin.py
Project C:\Users\library root
World_books
  venv
  WebBooks
    catalog
      migrations
      __init__.py
      admin.py
      apps.py
      models.py
      tests.py
      urls.py
      views.py
admin.py x views.py x tests.py x models.py x manage.py x
17 admin.site.register(Author, AuthorAdmin)
18
19
20 class BooksInstanceInline(admin.TabularInline):
21     model = BookInstance
22
23 @admin.register(Book)
24
25 class BookAdmin(admin.ModelAdmin):
26     list_display = ('title', 'genre', 'language', 'display_author')
27     list_filter = ('genre', 'author')
28     inlines = [BooksInstanceInline]
29
```

Рис. 8.54. Объявление и использование нового класса BooksInstanceInline

Действие:	НАЗВАНИЕ КНИГИ	ЖАНР КНИГИ	ЯЗЫК КНИГИ	АВТОРЫ
<input type="checkbox"/>	Золотой теленок	Проза	Русский	Ильф, Петров
<input type="checkbox"/>	12 стульев	Проза	Русский	Ильф, Петров
<input type="checkbox"/>	Война и мир	Проза	Русский	Толстой
<input type="checkbox"/>	Дубровский	Проза	Русский	Пушкин
<input type="checkbox"/>	Продавец воздуха	Фантастика	Русский	Беляев

Выберите book для изменения ДОБАВИТЬ BOOK +

Выполнено 0 объектов из 5

ФИЛЬР

Жанр книги

Все
Детективы
Поэзия
Приключения
Проза
Фантастика

автор книги

Все
Беляев
Пушкин

Рис. 8.55. Форма со списком книг

Теперь на этой форме щелкнем мышью на одной из книг. В результате откроется новое окно с подробными сведениями о выбранной книге, а также будут показаны связанные записи со сведениями обо всех экземплярах этой книги. Поскольку форма достаточно большая, то продемонстрируем ее содержание в виде двух рисунков. На рис. 8.56 представлена верхняя часть формы, где мы можем видеть и редактировать всю информацию о выбранной книге.

Рис. 8.56. Верхняя часть формы для ввода и редактирования информации о книге

А на рис. 8.57 показана нижняя часть формы, где мы можем добавлять и изменять сведения об экземплярах этой книги.

Применительно к этой странице разработчики Django реализовали функциональный и достаточно удобный интерфейс для разработчиков сайтов. Здесь можно одновременно модифицировать данные в главной таблице, в связанной таблице, а также во всех подключенных вспомогательных справочниках. При этом пользователи Django используют уже готовые программные модули и не тратят время на разработку интерфейса. При активации мышью знаков автоматически появляются подсказки, которые были предусмотрены при формировании моделей данных (рис. 8.58).

ISBN книги: 9785171001698
Должно содержать 13 символов

BOOK INSTANCES			
ИНВЕНТАРНЫЙ НОМЕР	ИЗДАТЕЛЬСТВО	СТАТУС ЭКЗЕМПЛЯРА КНИГИ	ДАТА ОКОНЧАНИЯ СТАТУСА
№1 Продавец воздуха На складе	ACT, 1917	На складе	30.11.2020 Сегодня <input type="button" value="удалить"/>
№2 Продавец воздуха На складе	ACT, 1917	На складе	30.11.2020 Сегодня <input type="button" value="удалить"/>
№3 Продавец воздуха В заказе	ACT, 1917	В заказе	28.08.2020 Сегодня <input type="button" value="удалить"/>
		-----	Сегодня <input type="button" value="удалить"/>
		-----	Сегодня <input type="button" value="удалить"/>
		-----	Сегодня <input type="button" value="удалить"/>

+ Добавить еще один Book instance

Рис. 8.57. Нижняя часть формы для ввода и редактирования информации об экземплярах книг

BOOK INSTANCES

ИНВЕНТАРНЫЙ НОМЕР	ИЗДАТЕЛЬСТВО
№1 Продавец воздуха На складе	Ведите инвентарный номер экземпляра
№1	ACT, 1917

Рис. 8.58. Фрагмент нижней части формы для ввода и редактирования информации об экземплярах книг с подсказкой

Рядом с полями справочников, к которым подключена родительская и дочерняя таблицы БД, имеются значки, при активации которых можно войти в режим редактирования или пополнения справочников (рис. 8.59).

Изменить book

Название книги: Продавец воздуха
Введите название книги

Жанр книги: Фантастика
Выберите жанр для книги

Язык книги: Русский
Выберите язык книги

СТАТУС ЭКЗЕМПЛЯРА КНИГИ

На складе |
На складе |
В заказе |

a

б

Рис. 8.59. Значки для входа в режим модификации справочников для родительской (а) и дочерней (б) таблиц БД

Рядом с полями, которые в модели данных имеют тип «Дата», располагается значок, при активации которого автоматически появляется панель календаря, позволяющая выбрать нужную дату (рис. 8.60); при этом сохраняется режим ввода даты с клавиатуры.

На этом мы закончим знакомство с административной панелью Django.

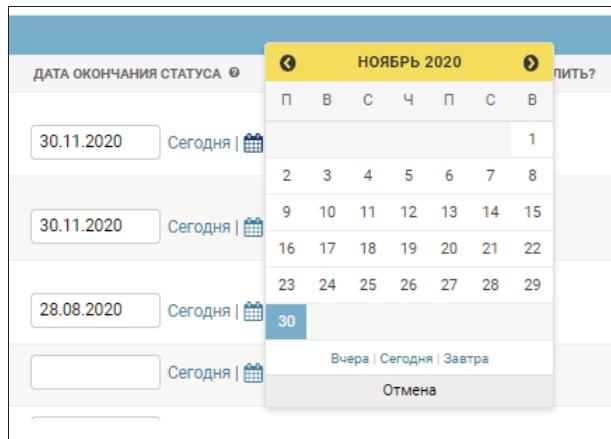
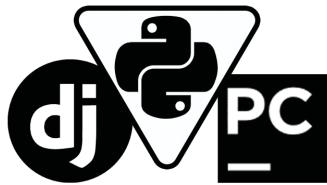


Рис. 8.60. Возможность выбора даты с использованием встроенного календаря

8.7. Краткие итоги

В этой главе мы создали проект сайта и разработали структуру таблиц для хранения данных. Затем создали модели данных и на основе этих моделей в автоматическом режиме сформировали набор соответствующих таблиц в самой БД. Мы также познакомились с возможностями административной панели Django. Узнали о настройках модуля администрирования сайта как в самом простом виде, так и в улучшенной форме. Мы создали суперпользователя и узнали о том, как перемещаться по сайту администратора, просматривать, удалять и обновлять записи. Мы проверили работу всех моделей данных, для чего ввели сведения о нескольких книгах и их экземплярах, жанрах, авторах и языках.

В административной панели Django есть еще возможность управлять правами пользователей сайта, но с этими возможностями мы познакомимся немного позже. А пока перейдем к рассмотренной в следующей главе теме, которая касается создания интерфейса пользователя.



ГЛАВА 9

Пример создания веб-интерфейса для пользователей сайта «Мир книг»

В предыдущей главе мы рассмотрели вопросы создания структуры и моделей данных сайта «Мир книг» и использования административной панели Django для работы с информацией, которая будет храниться на сайте. Теоретически, административная панель Django — это по своей сути почти готовый сайт, с которым могут работать удаленные пользователи. Однако для пользователей лучше разработать иной интерфейс. Во-первых, он должен быть более красочным и привлекательным, а во-вторых, нельзя допустить бесконтрольную возможность манипулирования информацией со стороны удаленных пользователей. Неумелыми действиями можно так испортить данные, что сайт окажется неработоспособным. Эта глава посвящена как раз теме, связанной с разработкой HTML-страниц сайта для внешних пользователей. При этом больше внимания будет уделено технологическим вопросам и меньше — дизайну разрабатываемых страниц.

Мы рассмотрим здесь следующие вопросы:

- последовательность создания пользовательских страниц сайта и формирования их перечня;
- создание URL-преобразований (маршрутов), чтобы URL-адреса ассоциировались сервером с определенными представлениями (views);
- создание представлений (views), чтобы на основе запросов пользователей выбирать из БД информацию, на ее основе формировать ответы и превращать их в HTML-страницы, которые будут отображаться в браузере пользователей;
- создание шаблона для главной страницы сайта «Мир книг»;
- возможности отображения на HTML-страницах списков и детальной информации об элементах списков.

9.1. Последовательность создания пользовательских страниц сайта «Мир книг»

Итак, в предыдущих главах мы создали модели данных и на их основе сформировали в БД соответствующие таблицы, которые заполнили тестовыми данными.

Теперь мы готовы создать код нашей первой страницы — домашней страницы сайта «Мир книг». Эта страница будет достаточно простой, на ней мы покажем количество записей в каждой модели и боковую навигационную панель со ссылками на другие страницы сайта (главное меню). В результате мы приобретем практический навык написания простых URL-преобразований, получения записей из базы данных, применения шаблонов для вывода данных из БД.

Вернемся к структуре приложений, написанных на Django (рис. 9.1).

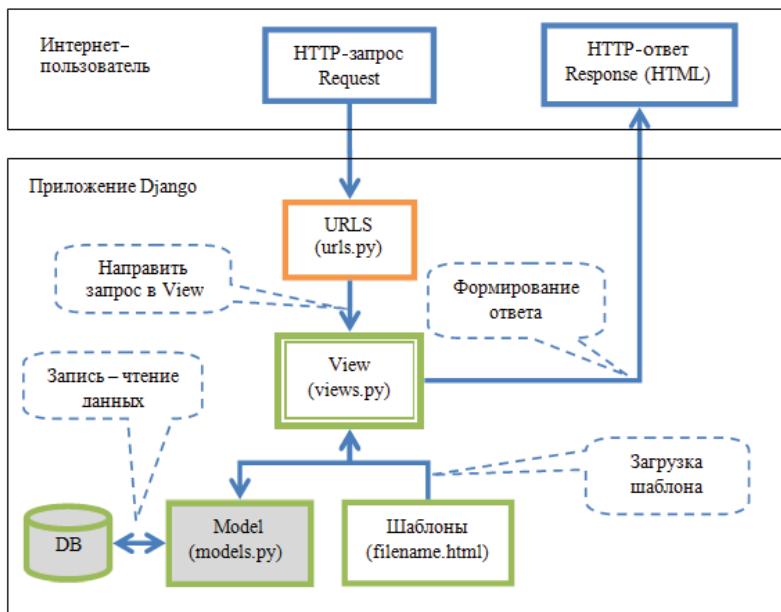


Рис. 9.1. Структура приложений, написанных на Django

Здесь в затененных квадратах показаны элементы, которые мы уже реализовали, — это модели данных и сами данные на уровне таблиц БД. Теперь пришло время написать код, который будет показывать пользователям информацию, содержащуюся в БД. Первое, что нам необходимо сделать, — это определить, какую информацию мы бы хотели показывать на наших страницах. Затем установить URL-адреса для получения соответствующих ресурсов. После чего создать URL-преобразования, представления (функции или классы) и шаблоны соответствующих страниц.

9.2. Определение перечня и URL-адресов страниц сайта «Мир книг»

Для конечных пользователей первая версия сайта «Мир книг» будет работать в режиме read-only (только для чтения). В этом случае нам надо создать страницы, которые будут показывать списки авторов и книг, а также детальную информацию о них. Сформируем URL-адреса, которые понадобятся для наших страниц:

- catalog/** — домашняя (индексная) страница;
- catalog/books/** — список всех книг;
- catalog/authors/** — список всех авторов;
- catalog/book/<id>** — детальная информация для определенной книги со значением первичного ключа, равным **<id>**. Например: **/catalog/book/3** для id = 3;
- catalog/author/<id>** — детальная информация для определенного автора со значением первичного ключа, равным **<id>**. Например: **/catalog/author/11** для автора с id = 11.

Первые три URL-адреса используются для показа домашней страницы, списков книг и авторов. Содержание этих страниц будет полностью зависеть от того, что находится в базе данных.

Последние два URL-адреса служат для показа детальной информации об определенной книге или авторе. В себе они содержат соответствующее значение идентификатора (он показан как **<id>**). URL-преобразование получает эту информацию и передает ее в представление (view), которое применяет ее для запроса к базе данных. Для кодирования и применения этой информации в URL-адресе нам понадобится только одно URL-преобразование, соответствующее представление и шаблон страницы для показа любой книги или автора.

В Django можно конструировать URL-адреса любым удобным способом. Можно закодировать информацию в теле URL-адреса, как показано ранее, или использовать URL-адрес типа GET (например: **/book/?id=6**). Независимо от способа формирования URL-адресов, они должны быть понятными, логичными и читабельными. Документация Django рекомендует кодировать информацию в теле URL-адреса — на практике это приводит к улучшению структуры сайта.

9.3. Создание главной страницы сайта «Мир книг»

Первой страницей, которую мы создадим, будет главная страница сайта (**catalog/**). Это небольшая статичная HTML-страница, показывающая полученные из базы данных «количество» различных записей. Для того чтобы проделать эту работу, мы вначале создадим URL-преобразование, затем представление и шаблон. Рекомендуется уделить большее внимание этому разделу, поскольку описанная здесь последовательность действий будет использоваться при создании остальных страниц сайта.

9.3.1. Создание URL-преобразования

Итак, приступим к созданию URL-преобразования. Внутри нашего приложения **catalog** откроем файл **urls.py** и поместим в него следующий код (листинг 9.1), как показано на рис. 9.2.

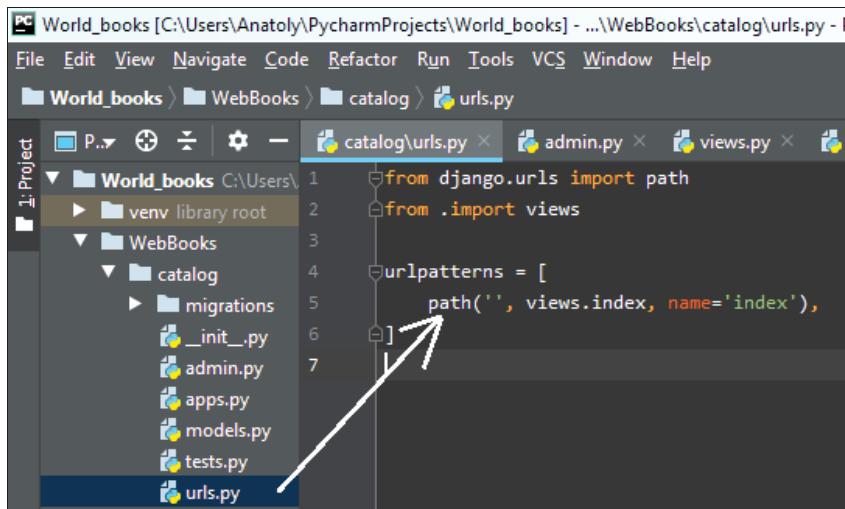


Рис. 9.2. Создание URL-преобразования для перехода на главную страницу сайта «Мир книг»

Листинг 9.1

```

from django.urls import path
from .import views

urlpatterns = [
    path('', views.index, name='index'),
]

```

Здесь функция `path()` определяет URL-шаблон (в нашем случае это пустая строка в апострофах) и функцию отображения (`views`), которая будет вызвана, если окажется, что введенный адрес соответствует этому шаблону. А `views.index` — это функция с именем `index()`, которая находится в файле `views.py`. Иными словами, если в запросе, поступившем от внешнего пользователя, присутствует пустая строка, то этот запрос будет адресован к функции `index()`. В свою очередь, в этой функции прописывается обращение к БД для загрузки данных и вызов страницы, которая эти данные вернет пользователю.

Кроме того, в функции `path()` определен параметр `name`, который уникально объявляет, что мы имеем дело с частным URL-преобразованием. После такого объявления можно использовать это имя для «обратного» (`reverse`) преобразования — т. е. для динамического создания URL-адреса, указывающего на определенный ресурс. Например, если имеется такое символическое имя, мы можем ссылаться на нашу домашнюю страницу при помощи создания следующей ссылки внутри какого-либо HTML-шаблона:

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

Конечно, можно жестко указывать прямую ссылку на домашнюю страницу так:

```
<a href="/catalog/">Home</a>)
```

Но в этом случае если по какой-либо причине будет изменен адрес домашней страницы (например, на `/catalog/index`), то такие ссылки перестанут корректно работать. Так что применение «обратного» URL-преобразования — более гибкий и эффективный подход.

9.3.2. Создание представления (view)

Представление (view) — это функция, которая обрабатывает HTTP-запрос, получает информацию из базы данных (если это необходимо) и полученные из БД данные использует при формировании ответной HTML-страницы. Затем функция представления возвращает сгенерированную страницу пользователю в виде HTTP-ответа. В нашем случае все эти действия будет выполнять функция с именем `index()`: она получит из БД информацию о количестве записей о книгах (`Book`), о количестве экземпляров книг (`BookInstance`) и их доступности, а также о количестве авторов (`Author`), а затем передаст эти данные в шаблон страницы. В завершение эта функция сгенерирует ответную страницу и передаст ее в браузер пользователя.

Откроем файл `catalog\views.py`. В предыдущих главах мы внесли в него самый простой код, говорящий о том, что мы якобы попали на главную страницу сайта (листинг 9.2).

Листинг 9.2

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return HttpResponse("Главная страница сайта Мир книг!")
```

Добавим в верхнюю часть этого кода строку подключения наших моделей данных (листинг 9.3).

Листинг 9.3

```
from .models import Book, Author, BookInstance, Genre
```

Заменим ранее созданные программные строки функции `index()` на следующий код (листинг 9.4).

Листинг 9.4

```
def index(request):
    # Генерация "количество" некоторых главных объектов
    num_books = Book.objects.all().count()
    num_instances = BookInstance.objects.all().count()
```

```
# Доступные книги (статус = 'На складе')
# Здесь метод 'all()' применен по умолчанию.
num_instances_available =
    BookInstance.objects.filter(status__exact=2).count()
# Авторы книг,
num_authors = Author.objects.count()

# Отрисовка HTML-шаблона index.html с данными
# внутри переменной context
return render(request, 'index.html',
              context={'num_books': num_books,
                        'num_instances': num_instances,
                        'num_instances_available':
                            num_instances_available,
                        'num_authors': num_authors,
                        'num_visits': num_visits},
              )
```

В первой части функции `index()` обеспечивается подсчет количества записей в БД при помощи вызова метода `objects.all().count()` для классов моделей `Book`, `BookInstance` и `Author`. Похожим образом мы получаем список объектов `BookInstance`, которые имеют статус 'На складе'.

Во второй части функции `index()` вызывается функция `render()`, которая в качестве ответа пользователю создает и возвращает HTML-страницу `index.html`. На самом деле эта функция как бы обертывает в себя вызовы нескольких функций, тем самым существенно упрощая процесс разработки. В качестве параметров ей передаются: объект `request` (запрос типа `HttpRequest`), шаблон HTML-страницы с метками (`placeholders`), которые будут замещены данными, а также переменная `context`, представляющая собой словарь Python, содержащий данные из БД, которые и будут замещать в шаблоне эти метки.

Теперь нужно создать шаблон HTML-страницы, в которой можно показать пользователю ответ на его запрос. Это мы сделаем в следующем разделе, а также более подробно поговорим там и о шаблонах, и о переменной `context`.

9.3.3. Создание базового шаблона сайта и шаблона для главной страницы сайта «Мир книг»

Шаблон — это текстовый файл, который определяет структуру и расположение данных на HTML-странице. Кроме того, в нем размещают специальные метки (`placeholders`), которые используются для показа реального содержимого, т. е., в нашем случае, информации из базы данных. По умолчанию Django ищет файлы шаблонов в каталоге `templates` приложения. Например, внутри функции `index()`, которую мы создали в файле `catalog\views.py`, вызов вложенной функции `render()` будет пытаться найти файл `\WebBooks\catalog\templates\index.html` и в случае неудачи генерирует ошибку о том, что файл не найден. Для того чтобы этого не случилось, сначала

чала нужно создать папку (каталог) `templates`, а затем в этой папке создать пока еще пустой файл с именем `index.html`. Сделав эти шаги в PyCharm, мы увидим следующий результат (рис. 9.3).

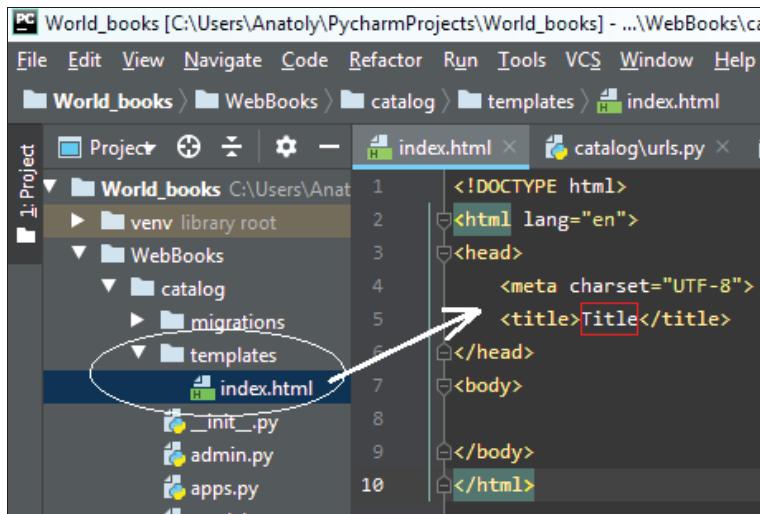


Рис. 9.3. Создание в PyCharm каталога `templates` и файла `index.html`

Шаблон главной страницы нашего сайта должен соответствовать стандарту разметки HTML для разделов `head` (заголовок) и `body` (тело). Кроме того, на странице должны быть разделы: для навигации по сайту (главное меню), что обеспечит переход на другие страницы сайта (этот раздел мы создадим чуть позже), и раздел для показа некоторого вводного текста. Большая часть этой структуры останется одинаковой для всех страниц нашего сайта. Таким образом, чтобы избежать копирования одной и той же информации, язык создания шаблонов Django позволяет сформировать один базовый шаблон, а затем создавать на его основе другие страницы сайта, замещая только те части базового шаблона, которые являются специфическими для каждой страницы. Будем и мы придерживаться этого правила: сначала сформируем базовый шаблон для всего сайта, а потом на его основе будем создавать все остальные страницы.

Типовой базовый шаблон имеет следующую структуру:

```
{% block head %}
{% endblock %}
{% block content %}
    {% block page %}
        {% endblock page %}
        {% block sidebar %}
        {% endblock %}
    {% endblock content %}
    {% block footer %}
    {% endblock %}
```

Здесь присутствуют несколько блоков: блок-заголовок, блок-контент, внутри которого находятся: блок-страница и блок-боковая панель. Последним идет блок-подвал.

Для базового шаблона создадим в каталоге `\WebBooks\catalog\templates\` новый файл с именем `base_generic.html` и добавим в него следующий код (листинг 9.5).

Листинг 9.5

```
<!DOCTYPE html>
<html lang="en">
<head>
    {% block title %}<title>Мир книг</title>{% endblock %}
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
                                initial-scale=1">
    <link rel="stylesheet"
          href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
          bootstrap.min.css">
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/
                jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/
                js/bootstrap.min.js"></script>

    <!-- Добавление статического CSS-файла -->
    {% load static %}
    <link rel="stylesheet" href="{% static 'css/styles.css' %}">
    {% block head %}
        
        <font size=7, color="blue">Сайт "Мир книг"</font>
    {% endblock %}
</head>

<body>
<div class="container-fluid">

    <div class="row">
        <div class="col-sm-2">
            {% block sidebar %}
            <ul class="sidebar-nav">
                <li><a href="{% url 'index' %}">Главная страница</a></li>
                <li><a href="">Все книги</a></li>
                <li><a href="">Все авторы</a></li>
            </ul>
            {% endblock %}
        </div>
```

```
<div class="col-sm-10 ">
    {% block content %}{% endblock %}
    {% block footer %}
        {% block copyright %}
            <p>Copyright ООО "Люди и книги", 2020. Все права защищены</p>
        {% endblock %}
    {% endblock %}
</div>
</div>
</body>
</html>
```

В этом файле содержится HTML-код, в котором объявлены следующие блоки: `title` (название), `head` (заголовок, шапка), `sidebar` (боковая панель), `content` (содержание) и `footer` (подвал). В блок `title` мы добавили текст с названием сайта: Мир книг — этот текст можно впоследствии заменить. В блок `head` поместили изображение с логотипом сайта ('`images/logotip.jpg`') и надпись: Сайт "Мир книг". Создали боковую панель навигации по страницам сайта. Пока боковая панель содержит ссылки перехода на главную страницу, на списки всех книг и авторов. Панель навигации также можно менять в процессе работы над сайтом.

В шаблоне присутствует еще один блок, описывающий подвал страницы, — `block footer` и вложенный в него блок `copyright`. В нем содержится строка, закрепляющая авторское право на этот сайт.

Здесь есть еще два дополнительных шаблонных тега: `url` и `load static`. Они будут описаны в следующих разделах.

Кроме того, наш шаблон включает в себя код JavaScript и CSS от Bootstrap для лучшего размещения элементов и формирования внешнего вида HTML-страницы. Bootstrap — это открытый и бесплатный HTML-, CSS- и JS-фреймворк, который используется веб-разработчиками для быстрого создания дизайнов сайтов и веб-приложений. Обратиться к Bootstrap или любому другому фреймворку для создания дизайна клиентской части сайта — весьма продуктивный способ повышения привлекательности страниц. Применение этих решений обеспечивает корректный показ страниц сайта на устройствах с различными разрешениями экрана и позволяет повысить уровень привлекательности страниц для пользователей. Мы не станем концентрироваться на дизайне страниц (хотя косвенно коснемся этого вопроса), а направим большую часть своих усилий на реализацию взаимодействия пользователей с БД и серверной частью нашего сайта.

Что касается дизайна, то наш базовый шаблон ссылается на локальный файл CSS-стилей — `styles.css`, который предоставляет нашему сайту возможность использования дополнительных стилей. Создадим этот файл в каталоге `\WebBooks\catalog\static\css\` и добавим в него следующий код (листинг 9.6), как показано на рис. 9.4.

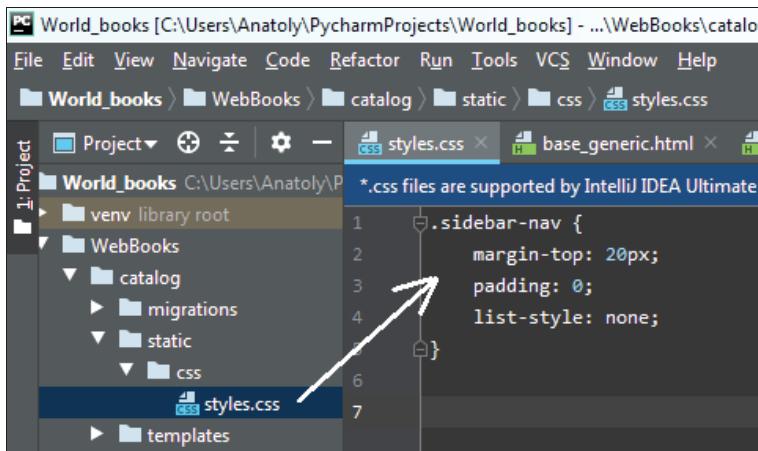


Рис. 9.4. Создание в PyCharm локального файла стилей styles.css

Листинг 9.6

```
.sidebar-nav {
    margin-top: 20px;
    padding: 0;
    list-style: none;
}
```

Теперь мы можем перейти к созданию главной (ее еще называют *домашней*) страницы нашего сайта. Мы уже создали заготовку для этой страницы — файл \WebBooks\catalog\templates\index.html. Внесем в него следующий код (листинг 9.7).

Листинг 9.7

```
{% extends "base_generic.html" %}

{% block content %}
<h1>Главная страница</h1>

<p>Добро пожаловать в <em>Мир книг</em>, Это очень простой
веб-сайт, написанный на Django. Разработан на Python
в качестве учебного примера с использованием PyCharm.</p>

<h2>Динамический контент</h2>

<p>База данных сайта Мир книг содержит следующее
количество записей :</p>
<ul>
<li><strong>Количество книг:</strong> {{ num_books }}</li>
<li><strong>Количество экземпляров книг:</strong>
{{ num_instances }}</li>
```

```
<li><strong>Количество экземпляров книг в заказе:<br/>
    </strong> {{ num_instances_available }}</li>
<li><strong>Количество авторов книг:<br/>
    </strong> {{ num_authors }}</li>
</ul>
{%- endblock %}
```

Как можно видеть, в первой строке мы строим эту страницу путем расширения базового шаблона `base_generic.html`. Затем замещаем содержимое блока `content` базового шаблона новым содержимым из текущего шаблона.

В текущем шаблоне главной страницы имеется раздел динамический контент. В этом разделе объявлены метки (переменные шаблона): `num_books`, `num_instances`, `num_instances_available` и `num_authors`. Эти метки соответствуют следующим значениям, которые в представлении (`view`) будут получены из БД: Количество книг, Количество экземпляров книг, Количество экземпляров книг в заказе, Количество авторов книг. Эти переменные объявляются при помощи двойных фигурных скобок (`{{ num_books }}`). В этом же коде теги, выделяющие функции шаблона, помещаются в одинарные фигурные скобки со знаками процента (`{% extends "base_generic.html" %}`).

Важно отметить, что переменные в шаблоне должны иметь те же имена, что и имена передаваемых ключей из словаря `context`, которая передается из содержащейся в представлении (`view`) функции вызова `render()` (листинг 9.8).

Листинг 9.8

```
context={'num_books':num_books,
         'num_instances':num_instances,
         'num_instances_available':num_instances_available,
         'num_authors':num_authors}
```

При выдаче шаблона в браузере пользователя вместо этих ключей будут подставлены соответствующие им значения.

Любой веб-проект с большой вероятностью будет использовать статические ресурсы, включая код JavaScript, CSS и изображения. В связи с тем, что расположение этих файлов может быть неизвестно (или может измениться), Django позволяет в шаблоне указать относительное расположение этих файлов при помощи глобального значения `STATIC_URL`. По умолчанию значение параметра `STATIC_URL` установлено в `'/static/'` в файле `settings.py` (рис. 9.5). Впрочем, вы можете установить любое другое место расположения статичных файлов.

Обратите внимание: в коде базового шаблона `base_generic.html` (см. листинг 9.5) имеется тег `load`. По сути, это функция, которая загружает статическую библиотеку `static`:

```
<!-- Добавление статического CSS-файла -->
{%- load static %}
<link rel="stylesheet" href="{% static 'css/styles.css' %}">
```

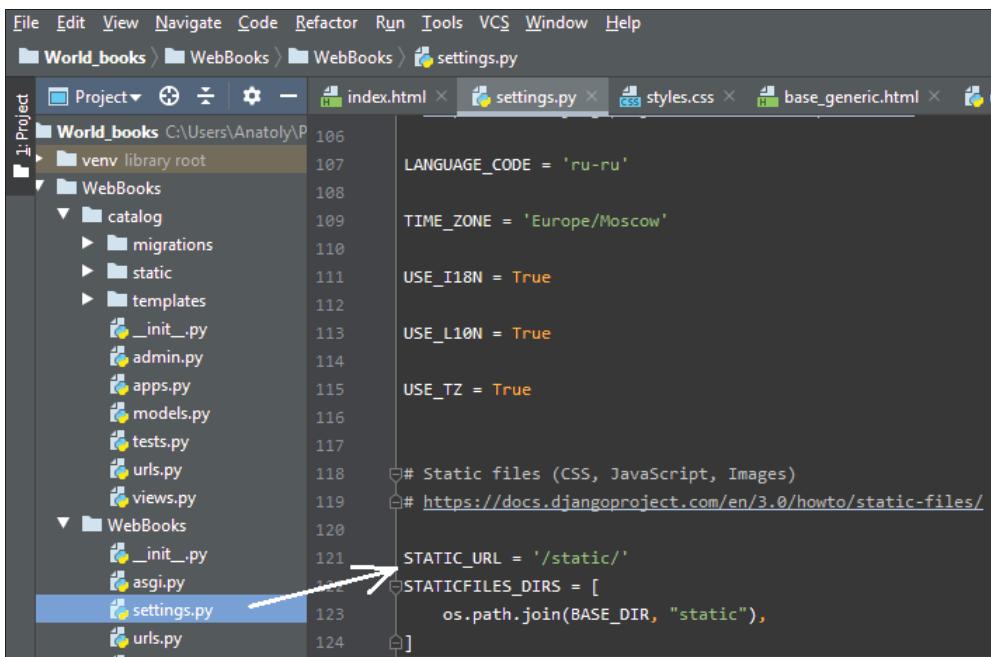


Рис. 9.5. Задание расположения каталога со статичными файлами в настройках файла settings.py

Когда статическая библиотека уже загружена, можно использовать тег шаблона static, который указывает относительный путь (URL) к интересующему нас статичному файлу. Таким способом можно загрузить на страницу и изображение. У нас это сделано с помощью следующего программного кода:

```

{% block head %}

<font size=7, color="blue">Сайт "Мир книг"</font>
{% endblock %}

```

В коде базового шаблона base_generic.html также имеется и тег url:

```
<li><a href="{% url 'index' %}">Главная страница</a></li>
```

Этот тег ищет в файле urls.py связанное значение переменной, указанной здесь в качестве ее параметра 'index', а затем возвращает интернет-адрес (URL), который вы можете использовать для ссылки на соответствующие ресурсы.

Поскольку в базовом шаблоне (в шапке базовой страницы) мы указали на наличие логотипа, нам теперь нужно создать каталог static\images и поместить туда изображение логотипа — файл с именем logotip.jpg. Результаты этих действий показаны на рис. 9.6.

К настоящему моменту у нас сделано все необходимое, чтобы показать главную страницу нашего сайта. Запускаем сервер командой:

```
python manage.py runserver
```

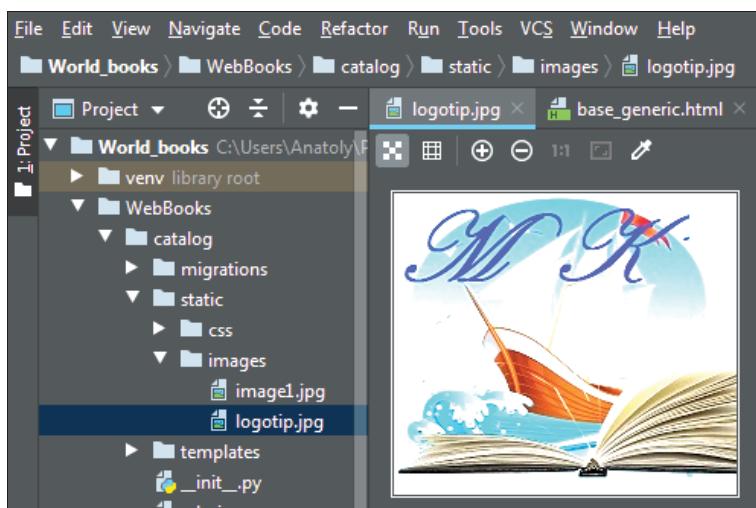


Рис. 9.6. Размещение файла с изображением логотипа сайта
в каталоге со статичными файлами

и вводим в адресной строке браузера адрес: <http://127.0.0.1:8000/>. Если все настроено правильно, наш сайт должен выглядеть так, как показано на рис. 9.7.

Сейчас у нас нет возможности воспользоваться ссылками на страницы **Все книги** и **Все авторы**, потому что URL-адреса, представления и шаблоны для этих страниц еще не созданы. Мы просто объявили метки для соответствующих ссылок в базовом шаблоне `base_generic.html`, но сами шаблоны страниц и соответствующие представления для них пока не создали.

Рис. 9.7. Главная страница сайта «Мир книг»

Тем не менее мы сформировали домашнюю страницу для нашего сайта, которая показывает количество некоторых записей из базы данных и содержит ссылки на другие, еще не созданные страницы. Кроме того, мы получили информацию об URL-преобразованиях, представлениях и простейших запросах к базе данных. При этом были использованы наши модели данных. В дальнейшем изложении на основе полученных здесь знаний мы создадим недостающие страницы сайта. А пока рассмотрим возможность отображения детальной информации об элементе списка на примере списка книг — т. е. разберемся, как, имея список книг, получить сведения о каждой книге.

9.4. Отображение списков и детальной информации об элементе списка

В этом разделе мы продолжим разработку сайта «Мир книг», добавляя в него новые страницы с выдачей подробной информации о книгах и их авторах. Мы познакомимся с обобщенными базовыми классами визуализации данных и покажем, как они могут существенно сократить количество кода, который потребовалось бы написать в обычной ситуации. Кроме того, мы более подробно рассмотрим настройки URL-адресов.

Процесс создания новых страниц похож на процесс создания главной страницы сайта, описанный в предыдущем разделе. Нам все так же надо будет создать URL-преобразования, представления и шаблоны страниц. Основное различие заключается в том, что для показа страниц с подробной информацией о книге или авторе перед нами встанет дополнительная задача получения данных из шаблона URL-адреса и передачи его представлению. Для этих страниц мы используем совершенно иной тип представления, основанный на применении обобщенных классов показа списка и детальной информации об элементе этого списка. Это может существенно сократить количество кода, необходимого для отображения страниц и сделает программный код более простым для написания и сопровождения.

Завершающая часть этого раздела посвящена демонстрации постраничного показа данных (*pagination*) при использовании обобщенного класса отображения списка.

Итак, приступим к созданию страницы с отображением списка книг, имеющихся в БД. Страница со списком книг покажет все книги и будет доступна по адресу: `catalog/books/`. Такая страница в каждой записи станет выводить название книги и имя ее автора, при этом каждое название книги будет являться гиперссылкой, нажав на которую пользователь перейдет на страницу подробной информации об этой книге. Наша страница будет иметь ту же структуру, что и все остальные страницы сайта. То есть для ее создания мы воспользуемся базовым шаблоном сайта (`base_generic.html`), который был рассмотрен в предыдущем разделе.

И начнем мы с сопоставления URL-адресов. Для этого откроем файл `\WebBooks\WebBooks\urls.py` и добавим в него строки, выделенные в листинге 9.9 серым фоном и полужирным шрифтом.

Листинг 9.9

```
from django.contrib import admin
from django.urls import path
from catalog import views
from django.conf.urls import url

urlpatterns = [
    path('', views.index, name='index'),
    path('admin/', admin.site.urls),
    url(r'^books/$', views.BookListView.as_view(), name='books'),
]
```

Практически так же, как и для главной страницы сайта, функция `url()` определяет здесь регулярное выражение (`r'^books/$'`), связывающее URL-адрес с функцией представления (отображения) данных `views.BookListView.as_view()`, которая будет вызвана, если окажется, что URL-адрес соответствует шаблону регулярного выражения. Кроме того, здесь определяется имя для такого сопоставления — `name='books'`.

Приведенный шаблон регулярного выражения сопоставления URL-адреса полностью соответствует строке `books/`. Здесь символ (^) является маркером начала строки, а символ (\$) — маркером конца строки. В соответствии с настройками нашего веб-приложения, URL-адрес уже должен содержать каталог `/catalog` — таким образом, полный адрес страницы на самом деле будет иметь вид: `/catalog/books/`.

Функция представления данных имеет несколько иной формат, чем мы использовали ранее. Это связано с тем, что здесь представление (`view`) реализуется через класс. Мы будем наследовать все возможности существующей общей функции `view()`, которая уже делает большую часть того, что мы хотим. В нашем случае такой подход достаточно эффективен, потому что при нем отпадает необходимость писать свою собственную функцию в представлении `view` с нуля.

В Django при использовании обобщенных классов получения данных доступ к соответствующей функции отображения информации организуется при помощи вызова метода `as_view()`. В результате выполняется вся запрограммированная работа по созданию экземпляра класса и гарантируется вызов правильных методов для входящих HTTP-запросов от пользователей.

Конечно, мы могли бы достаточно просто реализовать показ списка книг при помощи обычной функции, как это делалось для главной страницы сайта. То есть просто выполнить запрос на получение списка всех книг из базы данных, после чего вызвать функцию `render()`, которой передать этот список и отправить его в соответствующий шаблон страницы. Тем не менее мы все же воспользуемся *обобщенным классом отображения списка* — т. е. классом, который наследуется от существующего класса отображения списка данных `ListView`. Этот класс уже реализует функционал того, что нам нужно, и, следуя лучшим практикам Django, мы сможем

создать более эффективный список с данными из БД при помощи меньшего количества кода, меньшего количества повторений и с гораздо лучшей поддержкой.

Теперь откроем файл `catalog\views.py` и опишем в нем класс `BookListView` с помощью следующего кода (листинг 9.10).

Листинг 9.10

```
from django.views import generic

class BookListView(generic.ListView):
    model = Book
```

Вот, собственно, и все! Обобщенное представление выполнит запрос к базе данных, получит все записи заданной модели (`Book`), а затем заполнит соответствующий шаблон для выдачи списка книг (файл `book_list.html`), который будет располагаться в каталоге `\WebBooks\catalog\templates\catalog`.

Внутри этого шаблона можно будет получить доступ к списку книг при помощи переменной шаблона `object_list` (в нашем случае — `book_list`, поскольку имя этой переменной формируется по шаблону: `model_name_list`).

ПРИМЕЧАНИЕ

На первый взгляд приведенный у нас путь к файлу шаблона: `\WebBooks\catalog\templates\catalog\book_list.html` содержит опечатку. Ведь по факту шаблон должен находиться по пути: `\WebBooks\catalog\templates\book_list.html`.

Однако обобщенное представление ищет файл шаблона по пути: `\application_name\the_model_name_list.html` (в нашем случае: `catalog\book_list.html`), т. е. внутри каталога приложения `\application_name\templates\` (в нашем примере это `\catalog\templates\`).

Теперь создадим этот шаблон для выдачи списка книг — HTML-файл `book_list.html` в каталоге `\WebBooks\catalog\templates\catalog` и внесем в него следующий код (листинг 9.11).

Листинг 9.11

```
{% extends "base_generic.html" %}

{% block content %}
<h1>Список книг в БД</h1>
{% if book_list %}
<ul>
    {% for book in book_list %}
        <li>
            <a href="{{ book.get_absolute_url }}">
                {{ book.title }}</a>
                ({{ book.genre }})
        </li>
    {% endfor %}
</ul>
{% endif %}
```

```
{% endfor %}  
</ul>  
{% else %}  
    <p>В базе данных нет книг</p>  
{% endif %}  
{% endblock %}
```

Здесь, как и в шаблоне главной страницы, в первой строке мы расширяем наш базовый шаблон, а затем определяем блок с именем `content`. Внутри этого блока в цикле формируется и выдается список книг.

Шаблон использует теги организации цикла `for` и `endfor` для того, чтобы «пробежаться» по списку книг. На каждой итерации (в каждом цикле) в переменную шаблона `book` передается информация текущего элемента списка. В каждой строке списке содержатся всего два поля: название книги и ее жанр.

Внутри цикла с помощью тегов `if`, `else` и `endif` проверяется, определена ли переменная `book_list` и содержит ли она данные. Если список не пуст, мы выполняем итерации по списку книг. Если список пуст (случай `else`), мы выводим текст, поясняющий, что в наличии книг нет.

Код внутри цикла создает экземпляр для каждой книги — т. е. ее название преобразуется в ссылку, которая может использоваться для перехода на страницу с полной информацией об этой книге:

```
<a href="{{ book.get_absolute_url }}">  
    {{ book.title }}</a>  
    ({{ book.genre }})
```

Доступ к полям соответствующей записи о книге осуществляется при помощи шаблона `модель.поле` (`book.title`, `book.genre`), где фрагмент, который идет после имени модели `book`, является именем поля.

Кроме того, внутри нашего шаблона мы можем вызывать функции модели (в нашем случае мы вызываем `Book.get_absolute_url()`) для получения URL-адреса, который будет использоваться для перехода на страницу показа детальной информации о книге.

Теперь откройте файл базового шаблона по пути: `\WebBooks\catalog\templates\base_generic.html` и вставьте следующий фрагмент кода:

```
{% url 'books' %}
```

в URL-ссылку для пункта `Все книги`, как показано в листинге 9.12 (выделено серым фоном и полужирным шрифтом).

Листинг 9.12

```
<li><a href="{% url 'index' %}">Главная страница</a></li>  
<li><a href="{% url 'books' %}">Все книги</a></li>  
<li><a href="">Все авторы</a></li>
```

Тем самым мы создали переход на страницу с книгами (мы можем смело это сделать, поскольку у нас имеется соответствующее «книжное» URL-преобразование).

Посмотрим, что же у нас получилось. Запустите наш веб-сайт и на главной странице в левом меню щелкните мышью на ссылке **Все книги** (рис. 9.8). Если все сделано правильно, то будет выдана страница с полным списком книг, которые занесены в БД (рис. 9.9).

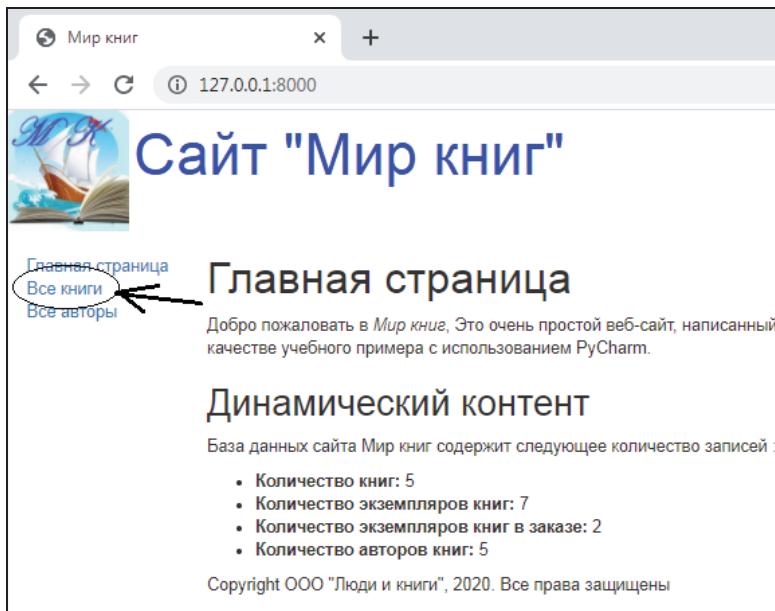


Рис. 9.8. Переход к странице **Все книги** на главной странице сайта «Мир книг»

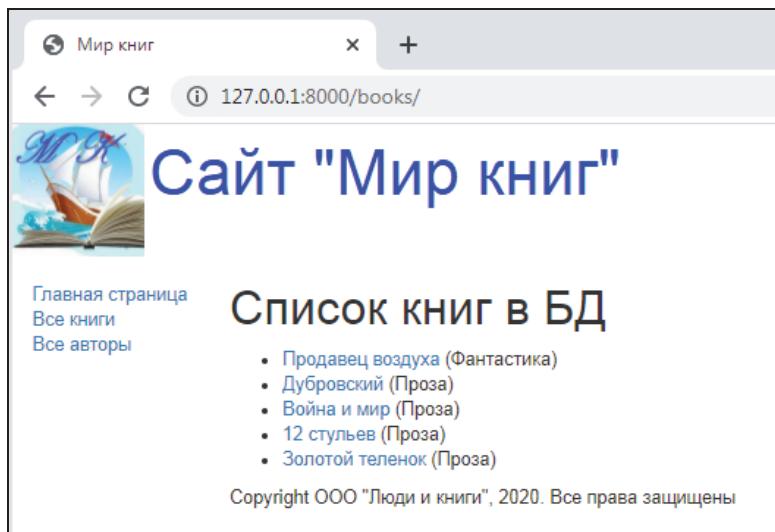


Рис. 9.9. Страница **Все книги** сайта «Мир книг»

На этой странице названия книг показаны синим цветом, а это признак того, что они являются ссылками для перехода на страницу с детальной информацией о книге. Однако если здесь щелкнуть мышью на такой ссылке (на название книги), то ничего не произойдет, поскольку мы еще не создали страницу для показа полной (детальной) информации о книге. Приступим к этому процессу.

Доступ к странице с детальной информацией о книге осуществляется при помощи URL-адреса `catalog/book/<id>` (где `<id>` является первичным ключом в БД для этой книги). В детальную информацию о книге в дополнение к полям модели `Book` (автор, краткое содержание, ISBN, язык и жанр) мы также добавим информацию о доступных экземплярах книги (`BookInstances`), включая их статус, ожидаемой дате изменения статуса, издаельстве (`imprint`) и инвентарном номере (`inv_nom`). Эта информация должна позволить пользователям не просто получить сведения о книге, но и убедиться, имеется ли она в наличии и является ли доступной.

Откроем файл `WebBooks\WebBooks\urls.py` и добавим фрагмент '`book-detail`' в URL-преобразование, как показано в листинге 9.13 (выделено серым фоном и полужирным шрифтом).

Листинг 9.13

```
from django.contrib import admin
from django.urls import path
from catalog import views
from django.conf.urls import url

urlpatterns = [
    path('', views.index, name='index'),
    path('admin/', admin.site.urls),
    url(r'^books/$', views.BookListView.as_view(), name='books'),
    url(r'^book/(?P<pk>\d+)$', views.BookDetailView.as_view(),
        name='book-detail'),
]
```

Здесь функция `url()` определяет шаблон, связывающий URL-адрес с обобщенным классом, который обеспечивает вывод детальной информации о книге, а также имя для этой связи.

В нашем случае мы применяем регулярное выражение для «настоящего шаблона», а не просто для текстовой строки. Это регулярное выражение сопоставляет любой URL-адрес с шаблоном, который начинается с `book/` и за которым до конца строки следуют одна или более цифр — до маркера конца строки (`$`). В процессе выполнения этого преобразования оно «захватывает» цифры и передает их в функцию представления как параметр с именем `pk` (от английского `primary key`, первичный ключ).

Как было отмечено ранее, наш URL-адрес в реальности выглядит так: `catalog/book/<digits>` (здесь `digits` — идентификатор записи в БД с информацией о книге). По-

скольку мы заранее не знаем, о какой книге запросит пользователь детальную информацию, то в этом случае вынуждены использовать функцию и регулярное выражение.

Немного отвлечемся от создания нашей HTML-страницы и более подробно рассмотрим синтаксис регулярных выражений.

Шаблоны регулярных выражений являются невероятно мощным инструментом преобразования. Пока мы немного говорили о них, поскольку сопоставляли URL-адреса с простыми строками (а не с шаблонами). Шаблоны регулярных выражений на первый взгляд не очень понятны и выглядят пугающе для начинающих, однако они очень эффективны. Поэтому, не поддаваясь панике, мы будем использовать достаточно простые шаблоны, и при этом хорошо задокументированные.

В первую очередь необходимо знать, что регулярные выражения (URL-шаблоны) объявляются при помощи строкового литерала (т. е. они заключены в кавычки: `r'<ваше регулярное выражение>'`). Главные элементы синтаксиса объявления шаблонов приведены в табл. 9.1.

Таблица 9.1. Главные элементы синтаксиса объявления URL-шаблонов

Символ	Значение
<code>^</code>	Соответствует началу строки
<code>\$</code>	Соответствует концу строки
<code>\d</code>	Соответствует цифре (0, 1, 2, ... 9)
<code>\w</code>	Соответствует любому символу из алфавита в верхнем или нижнем регистре, цифре или символу подчеркивания (<code>_</code>)
<code>+</code>	Соответствует одному или более предыдущему символу. Например, для соответствия одной или более цифр вы должны использовать: <code>\d+</code> . Для одного и более символа а вы можете использовать: <code>a+</code>
<code>*</code>	Соответствует отсутствию вообще или присутствию одного или более предыдущего символа. Например, для соответствия «ничему» или слову (т. е. любому символу) вы можете использовать: <code>\w*</code>
<code>()</code>	Захват части шаблона внутри скобок. Любое захваченное значение будет передано отображению как безымянный параметр (если захватывается множество шаблонов, то соответствующие параметры будут поставляться в порядке их объявления).
<code>(?P<name>...)</code>	Захват части шаблона (обозначенной ...) как именованной переменной (в нашем случае <code><name></code>). Захваченные значения передаются в представление с определенным именем. Таким образом, ваше представление должно объявить аргумент с тем же самым именем!
<code>[]</code>	Соответствует одному символу из множества. Например, <code>[abc]</code> будет соответствовать либо 'a', либо 'b', либо 'c'. <code>[-\w]</code> будет соответствовать либо символу '-', либо любому другому словарному символу

Чтобы более детально познакомиться с синтаксисом шаблонов, рассмотрим несколько реальных примеров их применения (табл. 9.2).

Таблица 9.2. Примеры синтаксиса объявления URL-шаблонов

Шаблон	Описание
r'^book/ (?P<pk>\d+) \$'	<p>Это регулярное выражение применяется в нашем URL-преобразовании выдачи информации о деталях книги. Оно соответствует строке, которая начинается с <code>book/ (^book/)</code>, затем содержит одну или более цифр (<code>\d+</code>) и завершается цифрой (и только цифрой).</p> <p>Оно также захватывает все цифры (<code>?P<pk>\d+</code>) и передает их в отображение в параметре с именем '<code>pk</code>'. Захваченные значения всегда передаются как строка!</p> <p>Например, если шаблону соответствует строка <code>book/1234</code>, в представление в переменную <code>pk</code> будет передано значение <code>pk='1234'</code>.</p>
r'^book/ (\d+) \$'	<p>Этот шаблон соответствует тем же самым URL-адресам, что и в предыдущем случае. Захваченная информация будет отправлена в представление как безымянный параметр</p>
r'^book/ (?P<stub>[-\w]+) \$'	<p>Этот шаблон соответствует строке, которая начинается с <code>book/ (^book/)</code>, затем идут один или более символов либо '-' или словарные символы (<code>[-\w]+</code>). Он также захватывает это множество символов и передает их в представление в параметре с именем '<code>stub</code>'.</p> <p>Показанный здесь шаблон весьма типичен для «стаба». «Стабы» являются «дружественными» URL-адресами — первичными ключами для данных. Вы могли бы применить «стаб», если бы захотели, чтобы URL-адрес вашей книги был более информативным. Например, адрес <code>/catalog/book/секреты-садоводства</code>, выглядит немного лучше, чем <code>/catalog/book/33</code></p>

Можно захватить (указать) несколько шаблонов в одном преобразовании и тем самым закодировать в URL-адресе много различной информации.

Вернемся к процессу создания нашей страницы для показа детальной информации о книгах. Откройте файл `catalog\views.py` и добавьте в него код, описывающий класс `BookDetailView` (листинг 9.14).

Листинг 9.14

```
class BookDetailView(generic.DetailView):
    model = Book
```

Все, что надо теперь сделать, — это создать шаблон с именем `book_detail.html` по пути: `WebBooks\catalog\templates\catalog\`. Представление передаст ему информацию из базы данных для определенной записи книги `Book`, выделенной при помощи URL-преобразования. Внутри шаблона вы можете получить доступ к списку книг при помощи переменной с именем `object` (в нашем случае это объект `book`). Итак, открываем файл `book_detail.html` и пишем в нем следующий код (листинг 9.15).

Листинг 9.15

```

{% extends "base_generic.html" %}

{% block content %}
    <h1>Название книги: {{ book.title }}</h1>

    <p><strong>Жанр:</strong> {{ book.genre }}</p>
    <p><strong>Аннотация:</strong> {{ book.summary }}</p>
    <p><strong>ISBN:</strong> {{ book.isbn }}</p>
    <p><strong>Язык:</strong> {{ book.language }}</p>

    {% for author in book.author.all %}
        <p><strong>Автор:</strong>
            <a href="">{{ author.first_name }}<br>
                {{ author.last_name }}</a></p>
    {% endfor %}

<div style="margin-left:20px;margin-top:20px">
    <h4>Количество экземпляров книг в БД</h4>
    {% for copy in book.bookinstance_set.all %}
        <hr><p class="`% if copy.status == 1 %` text-success
            {% elif copy.status == 2 %` text-danger
            {% else %` text-warning
            {% endif %}"> {{ copy.get_status_display }}</p>
        <p><strong>Издательство:</strong> {{copy.imprint}}</p>
        <p class="text-muted"><strong>Инвентарный номер:</strong> {{copy.id}}</p>
        <p><strong>Статус экземпляра книги в БД:</strong> {{copy.status}}</p>
    {% endfor %}
</div>
{% endblock %}
<p><strong>Автор:</strong> <a href="">{{ author.first_name }}<br>
    {{author.last_name }}</a></p>

```

Почти все, что используется в этом шаблоне, нам уже известно из содержания предыдущих разделов. Здесь мы берем за основу и расширяем наш базовый шаблон (`base_generic.html`) и переопределяем блок `content`. Мы используем условие `if` для показа того или иного содержимого и циклы `for`, чтобы «пробежаться» по элементам (объектам) в соответствующих списках. Мы получаем доступ к полям модели при помощи «дот-нотации».

Интересной функцией, которую мы не использовали ранее, является здесь функция `book.bookinstance_set.all()`. Этот метод автоматически сконструирован Django, чтобы вернуть множество записей экземпляров книг (`BookInstance`), связанных с конкретной книгой (`Book`):

```

{% for copy in book.bookinstance_set.all %}
    <!-- итерации по каждому экземпляру книги -->
{% endfor %}

```

А автоматически он создан потому, что в модели данных на стороне «многие» связи «один-ко-многим» объявлено поле `ForeignKey`. Поскольку мы ничего не объявили на другой стороне связи («один») этой модели (т. е. модель `Book` «ничего не знает» про модель `BookInstance`), то она не имеет никакой возможности (по умолчанию) для получения множества соответствующих записей. Для того чтобы обойти эту проблему, Django самостоятельно конструирует соответствующую функцию «обратного просмотра» (`reverse lookup`), которой можно воспользоваться. Имя этой функции создается в нижнем регистре и состоит из имени модели, в которой был объявлен `ForeignKey` (в нашем случае это `bookinstance`), за которым следует выражение `_set` (т. е. функция, созданная для модели `Book`, будет иметь имя `book.bookinstance_set`). Мы здесь также используем параметр `all()` для получения всех записей из таблицы БД.

К настоящему моменту мы создали все необходимое для показа детальной информации о каждой книге. Запускаем наш сервер (`python manage.py runserver`) и открываем браузер: <http://127.0.0.1:8000/>. На главной странице в меню выбираем опцию **Все книги** и получаем страницу со списком книг. Если мы теперь щелкнем мышью на названии книги — например, на **Продавец воздуха**, то будет выдана страница с полной информацией об этой книге (рис. 9.10).

The screenshot shows a web page titled "Сайт \"Мир книг\"". On the left, there is a sidebar with links: "Главная страница", "Все книги", and "Все авторы". The main content area has a title "Название книги: Продавец воздуха". Below it, the book's details are listed:
Жанр: Фантастика
Аннотация: Увлекательная, захватывающая история противостояния бесстрашного молодого метеоролога Клименко и полубезумного, гениального авантюриста Бейли, научившегося сжигать воздух из атмосферы и пускать полученные шариками воздушного газогидрата на продажу. Преступная деятельность Бейли приводит к радикальным изменениям климата. Земля начинает терять атмосферу. Как же остановить "продавца воздуха", пока он не погубил всю жизнь на планете?..
ISBN: 9785171001698
Язык: Русский
Автор: Александр Беляев
Количество экземпляров книг в БД
Издательство: АСТ, 1917
Инвентарный номер: 1
Статус экземпляра книги в БД: На складе
Издательство: АСТ, 1917
Инвентарный номер: 2
Статус экземпляра книги в БД: На складе
Издательство: АСТ, 1917
Инвентарный номер: 3
Статус экземпляра книги в БД: В заказе
Copyright ООО "Люди и книги", 2020. Все права защищены.

Рис. 9.10. Страница с детальной информацией о книге

Обратите внимание: фамилия и имя автора показаны здесь синим цветом, а это признак того, что они являются ссылкой для перехода на страницу сайта с информацией об авторах. Пока эта ссылка не работает, т. к. мы еще не создали соответствующую страницу и не реализовали механизм доступа к ней. Это мы сделаем несколько позже.

А теперь выдадим сведения о книге, у которой несколько авторов — например, «12 стульев». В этом случае в шаблоне страницы с использованием цикла будет выдана информация о двух авторах книги (рис. 9.11).

The screenshot shows a web browser window with the URL `127.0.0.1:8000/book/4`. The page title is "Сайт \"Мир книг\"". On the left, there's a sidebar with links: "Главная страница", "Все книги", and "Все авторы". The main content area has a header "Название книги: 12 стульев". Below it, under "Жанр:", it says "Проза". The "Аннотация:" section contains text about the book's publication in 1928 and its unique restoration. Under "ISBN:", it lists "9785170990573". Under "Язык:", it says "Русский". Under "Автор:", it lists "Илья Ильф" and "Евгений Петров". A section titled "Количество экземпляров книг в БД" follows. Below that, "Издательство:" is listed as "АСТ, 1915". "Инвентарный номер:" is "6". "Статус экземпляра книги в БД:" is "В заказе". At the bottom, a copyright notice reads "Copyright ООО \"Люди и книги\", 2020. Все права защищены".

Рис. 9.11. Страница с детальной информацией о книге двух авторов

Если в БД всего несколько записей об объекте, то наша страница вывода списка книг будет выглядеть отлично. Тем не менее, когда в БД появятся десятки или сотни записей, страница станет значительно дольше загружаться и окажется слишком длинной для комфорtnого просмотра. Решением этой проблемы может стать добавление постраничного вывода (Pagination) к отображению списка книг. При этом на каждую страницу будет выводиться ограниченное количество элементов.

Django имеет отличный встроенный механизм для организации постраничного вывода. Даже более того, он встроен в обобщенный класс отображения списков — так что нам не придется проделывать большой объем работы, чтобы воспользоваться возможностями постраничного вывода.

Откроем файл catalog\views.py и добавим поле paginate_by (листиг 9.16).

Листинг 9.16

```
class BookListView(generic.ListView):
    model = Book
    paginate_by = 3
```

Как только в базе данных появится более трех записей, представление начнет формировать постраничный вывод данных (страницы будут передаваться шаблону). К различным страницам можно будет получить доступ при помощи параметров GET-запроса. Например, к странице 2 можно получить доступ, используя URL-адрес: /catalog/books/?page=2.

Теперь, когда задан вывод данных постранично, нам надо добавить функционал переключения между страницами в шаблон страницы. Поскольку мы хотели бы использовать этот механизм для всех списков на сайте, то мы пропишем его в базовом шаблоне сайта.

Откроем файл \WebBooks\catalog\templates\base_generic.html и после блока content вставим блок, отвечающий за постраничный вывод (листиг 9.17).

Листинг 9.17

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

{% block pagination %}
    {% if is_paginated %}
        <div class="pagination">
            <span class="page-links">
                {% if page_obj.has_previous %}
                    <a href="{{ request.path }}?page={{ page_obj.previous_page_number }}>Предыдущая</a>
                {% endif %}
                <span class="page-current">
                    Страница {{ page_obj.number }} из
                    {{ page_obj.paginator.num_pages }}.
                </span>
                {% if page_obj.has_next %}
                    <a href="{{ request.path }}?page={{ page_obj.next_page_number }}>Следующая</a>
                {% endif %}
            </span>
        </div>
    {% endif %}
    {% endblock %}
```

Этот код в первую очередь проверяет, включен ли механизм постраничного вывода для страницы, и, если это так, добавляет на страницу ссылки **Следующая** и **Предыдущая**, а также номер текущей страницы.

Параметр `page_obj` является объектом типа `Paginator`, который станет создаваться каждый раз, когда будет применяться постраничный вывод данных для текущей страницы. Он позволяет получить всю информацию о текущей странице, о предыдущих страницах, количестве всего страниц и т. п.

Мы здесь используем выражение `{{ request.path }}` для получения URL-адреса текущей страницы, чтобы создать ссылки на соответствующие страницы. Обратите внимание, что этот вызов не зависит от объекта `page_obj` и может использоваться отдельно.

Если мы теперь загрузим страницу со списком книг, то под ним увидим блок организации постраничного вывода данных (рис. 9.12). Нажмите в нем на ссылку **Следующая** — будет выведена следующая страница со списком книг и появится ссылка для возврата на предыдущую страницу (рис. 9.13).

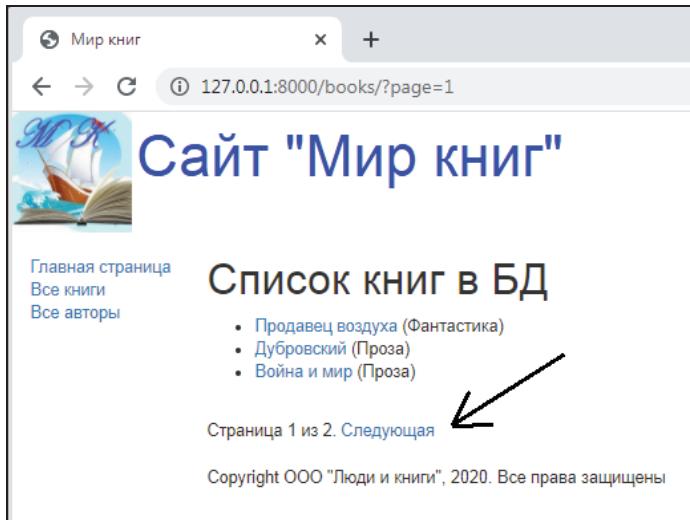


Рис. 9.12. Блок организации постраничного вывода данных

Нам осталось реализовать механизм показа списка авторов. Страница со списком авторов будет доступна по адресу: `catalog/authors/`. Начнем мы с сопоставления URL-адресов, для чего откроем файл `\WebBooks\WebBooks\urls.py` и добавим в него строки, выделенные в листинге 9.18 серым фоном и полужирным шрифтом.

Листинг 9.18

```
from django.contrib import admin
from django.urls import path
from catalog import views
from django.conf.urls import url
```

```
urlpatterns = [
    path('', views.index, name='index'),
    path('admin/', admin.site.urls),
    url(r'^books/$', views.BookListView.as_view(), name='books'),
    url(r'^book/(?P<pk>\d+)/$', views.BookDetailView.as_view(),
        name='book-detail'),
    url(r'^authors/$', views.AuthorListView.as_view(),
        name='authors'),
]
```

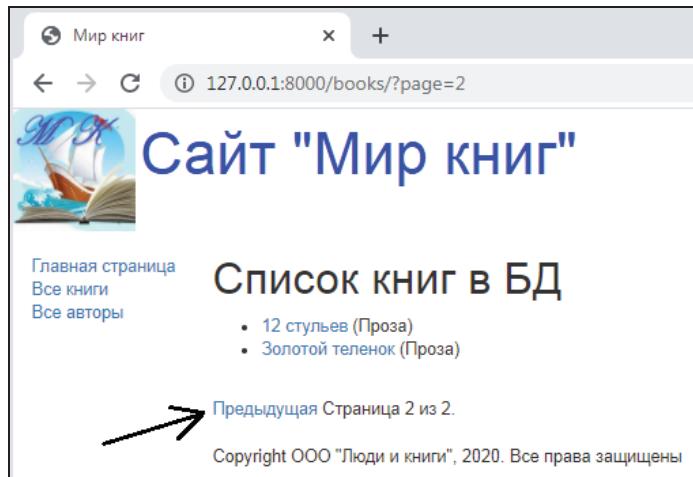


Рис. 9.13. Листание списка страниц с данными с возможностью возврата на предыдущие страницы

Теперь откроем файл catalog\views.py и опишем в нем класс AuthorsListView (листинг 9.19).

Листинг 9.19

```
class AuthorListView(generic.ListView):
    model = Author
    paginate_by = 4
```

На следующем шаге создадим в каталоге \WebBooks\catalog\templates\catalog\ HTML-файл author_list.html (шаблон для выдачи списка авторов книг) и внесем в него следующий код (листинг 9.20).

Листинг 9.20

```
{% extends "base_generic.html" %}

{% block content %}
<h1>Список авторов в БД</h1>
```

```
{% if author_list %}
    {% for author in author_list.all %}
        <p>{{ author.first_name}} {{author.last_name}},
        <strong>Родился-</strong>{{author.date_of_birth}},
        <strong>Умер-</strong>{{author.date_of_death}}</p>
    {% endfor %}
    {% else %}
        <p>В базе данных нет авторов</p>
    {% endif %}
{% endblock %}
```

Остался последний шаг. Откройте файл базового шаблона: \WebBooks\catalog\templates\base_generic.html и вставьте выражение:

```
{% url 'authors' %}
```

в URL-ссылку для пункта Все авторы, как показано в листинге 9.21 (выделено серым фоном и полужирным шрифтом).

Листинг 9.21

```
<li><a href="{% url 'index' %}">Главная страница</a></li>
<li><a href="{% url 'books' %}">Все книги</a></li>
<li><b><a href="{% url 'authors' %}">Все авторы</a></b></li>
```

Тем самым мы создали переход на страницу с авторами книг (теперь мы можем смело это сделать, поскольку у нас имеется соответствующее URL-преобразование).

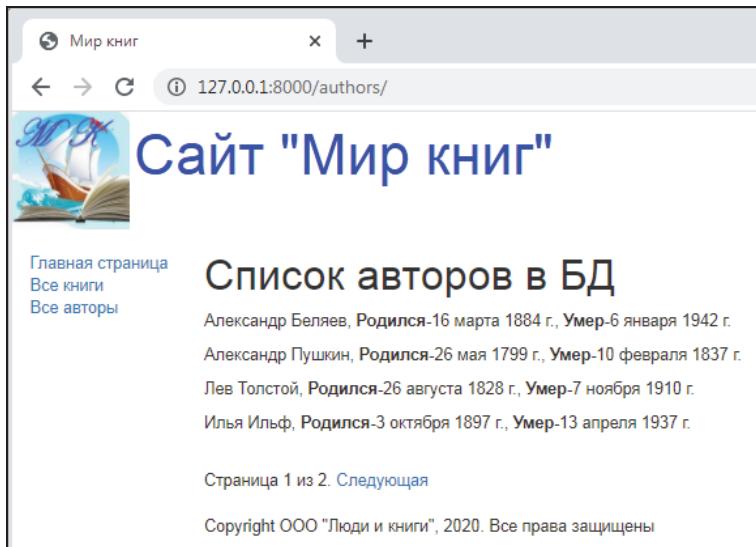


Рис. 9.14. Страница сайта с отображением списка авторов книг

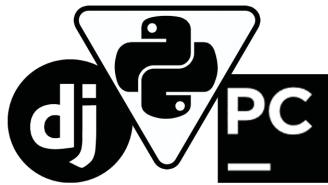
Для выдачи списка авторов у нас все сделано. Загружаем главную страницу нашего сайта и в левом меню щелкаем мышью на ссылке **Все авторы** — откроется страница с полным списком авторов (рис. 9.14).

Следует обратить внимание, что список авторов разбит на страницы. Это автоматически сработал блок разбивки данных на страницы, который мы ранее создали в базовом шаблоне.

9.5. Краткие итоги

В этой главе мы рассмотрели применение обобщенных классов для отображения списка данных и разобрались с показом детальной информации об отдельных элементах списка (на примере показа списка названий книг и детальной информации о книге). Кроме того, мы многое узнали о шаблонах преобразования URL-адресов, построенных на основе регулярных выражений, а также о том, как можно передавать данные из URL-адреса в представление (view). Мы изучили несколько приемов применения шаблонов и в самом конце главы показали, как осуществлять постраничный вывод длинных списков.

В следующей главе мы расширим наш сайт «Мир книг», введя в него поддержку пользовательских аккаунтов, и таким образом продемонстрируем аутентификацию, разграничение уровней доступа и познакомимся с понятием *сессии*. А также разрабатываем пользовательские формы для ввода данных в БД со стороны удаленных пользователей.



ГЛАВА 10

Расширение возможностей для администрирования сайта «Мир книг» и создание пользовательских форм

В предыдущих главах мы создали сайт «Мир книг», который позволяет пользователям получать из каталога списки книг и авторов. И сейчас каждый посетитель сайта имеет доступ к одним и тем же страницам и типам информации из базы данных.

Однако хотелось бы предоставить пользователю индивидуальные услуги, которые зависят от его предпочтений и предыдущего опыта использования сайта и его настроек. Например, возможность добавлять на страницы сайта свою информацию, что актуально для интернет-магазинов, социальных сетей, блогов и прочих подобных ресурсов.

В этой главе мы расширим возможности нашего сайта «Мир книг», добавив в него счетчик посещений домашней страницы, организованный на основе сессий. Этот относительно простой пример покажет, как с помощью сессий реализовать анализ поведения на сайте анонимных пользователей.

Мы также организуем на сайте «Мир книг» поддержку пользовательских аккаунтов и таким образом продемонстрируем аутентификацию, разграничение уровней доступа, сессии и пользовательские формы.

В этой главе будут рассмотрены следующие вопросы:

- что такое сессии в Django;
- создание страниц для авторизации пользователей в Django;
- организация проверки подлинности входа пользователя в систему;
- добавление к сайту страницы для создания заказов на книги;
- что такое формы и управление формами в Django;
- как создать форму для ввода и обновления информации на сайте на основе класса `Form()`;
- как создать форму для ввода и обновления информации на сайте на основе класса `ModelForm()`.

10.1. Сессии в Django

Сессии позволяют хранить и получать произвольные данные, запрошенные на основе индивидуального поведения пользователя на сайте. Разберемся, что же такое сессии.

Все взаимодействия между браузерами пользователей и серверами осуществляются при помощи протокола HTTP, который не сохраняет состояние таких взаимодействий (stateless). Это означает, что сообщения между клиентом и сервером являются полностью независимыми друг от друга — т. е. не существует какого-либо представления «последовательности» или поведения системы в зависимости от предыдущих сеансов взаимодействия. Так что если вы хотите создать сайт, который будет отслеживать взаимодействия с клиентом, то вам нужно реализовать это самостоятельно.

Сессии представляют собой механизм, который использует Django (как и весь остальной Интернет) для отслеживания состояния взаимодействий между сайтом и каким-либо браузером. Сессии позволяют хранить произвольные данные браузера и извлекать их в тот момент, когда между браузером и сайтом устанавливается соединение. Данные получаются и сохраняются в сессии при помощи соответствующего «ключа».

Django для этого использует файлы cookie. Они содержат специальный идентификатор сессии, который выделяет каждый браузер в соответствующую сессию. Реальные данные сессии по умолчанию хранятся в базе данных сайта. Впрочем, у вас есть возможность настроить Django так, чтобы сохранять данные сессий в других местах (кэше, файлах, «безопасных» cookie). Но все же хранение по умолчанию является хорошей и безопасной возможностью.

Сессии становятся доступны автоматически в тот момент, когда вы создаете «скелет» сайта. Необходимые настройки конфигурации выполняются в разделах INSTALLED_APPS и MIDDLEWARE файла проекта WebBooks\WebBooks\settings.py (рис. 10.1).

Доступ к переменной session можно получить в соответствующем представлении через параметр request (HttpRequest передается как первый аргумент в каждое представление). Переменная session представляет собой связь с определенным пользователем (если быть точным — с определенным браузером пользователя), который определяется при помощи идентификатора (`id`) сессии, получаемого из файла cookie браузера.

Переменная (или поле) `session` является объектом-словарем, в который можно делать записи неограниченное число раз. С ним вы можете выполнять любые стандартные операции: чтение, очистку всех данных, проверку наличия ключа, циклы по данным и т. п. Далее представлены фрагменты кода, которые показывают, как получать, задавать и удалять некоторые данные при помощи ключа `my_car`, связанного с текущей сессией (браузером):

```
# Получение значения сессии при помощи ключа('my_car').  
# Если такого ключа нет, то возникнет ошибка KeyError  
my_car = request.session['my_car']
```

```
# Получение значения сессии. Если значения не существует,
# то вернется значение по умолчанию ('mini')
my_car = request.session.get('my_car', 'mini')

# Передача значения в сессию
request.session['my_car'] = 'mini'

# Удаление значения из сессии
del request.session['my_car']
```

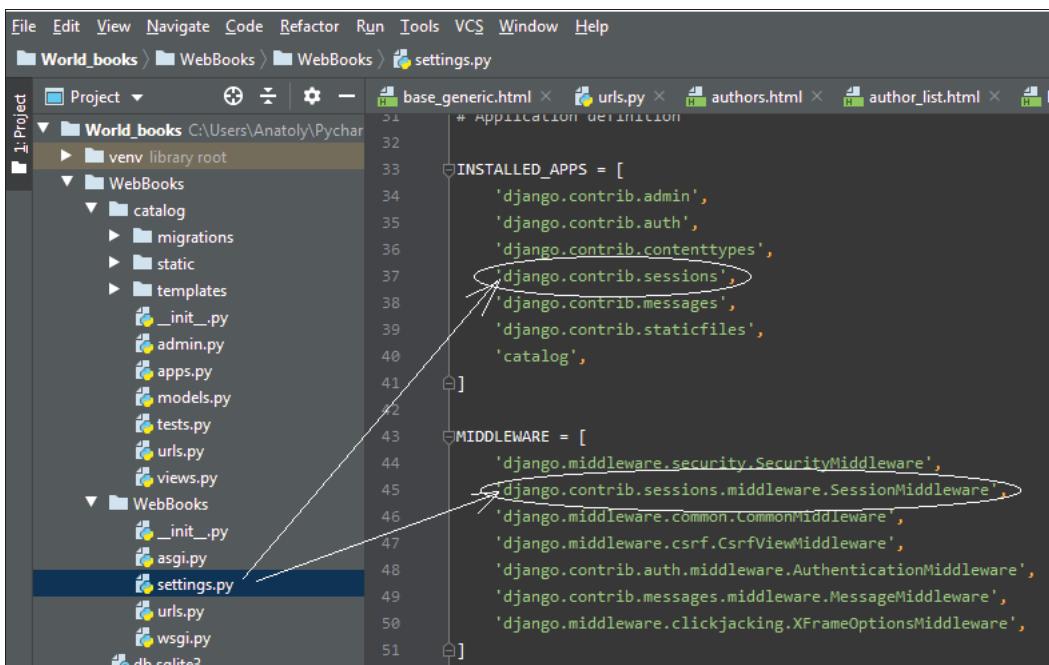


Рис. 10.1. Разделы конфигурирования сессий в файле settings.py

Это API имеет и другие методы, которые большей частью используются для управления файлами cookie, связанными с сессией. Например, существуют методы проверки того, что cookie поддерживаются клиентским браузером, другие методы служат для установки и проверки предельных дат жизни cookie, а также для очистки хранилища от просроченных сессий.

По умолчанию Django сохраняет сессии в базе данных и отправляет соответствующие cookie клиенту только тогда, когда сессия была изменена или удалена. Если вы обновляете какие-либо данные при помощи ключа сессии, как показано в предыдущем фрагменте кода, тогда вам не надо беспокоиться о процессе сохранения! Например:

```
# Это присваивание распознается как обновление сессии
# и данные будут сохранены
request.session['my_car'] = 'mini'
```

Если же вы обновляете информацию внутри данных сессии, тогда Django не распознает эти изменения и не выполняет сохранение данных. Например, если вы изменили параметр `wheels` внутри переменной `my_car`, как показано далее, то надо явно указывать, что сессия была изменена:

```
# Объект сессии модифицируется неявно.  
# Изменения НЕ БУДУТ сохранены!  
request.session['my_car']['wheels'] = 'alloy'
```

Но можно указать и явное сохранение данных сессии:

```
# Явное указание, что данные изменены.  
# Сессия будет сохранена, cookie обновлены (если необходимо)  
request.session.modified = True
```

Можно изменить поведение сессий таким образом, чтобы они записывали любое свое изменение в базу данных и отправляли cookie при каждом запросе, путем установки значения:

```
SESSION_SAVE_EVERY_REQUEST = True
```

в файле настроек проекта `WebBooks\WebBooks\settings.py`.

В качестве примера использования сессий мы обновим наш сайт так, чтобы сообщать пользователю количество совершенных им визитов на главной странице сайта «Мир книг». Для этого откройте файл `\WebBooks\catalog\views.py` и добавьте в него строки, выделенные серым фоном и полужирным шрифтом (листинг 10.1).

Листинг 10.1

```
def index(request):  
    # Генерация "количество" некоторых главных объектов  
    num_books = Book.objects.all().count()  
    num_instances = BookInstance.objects.all().count()  
    # Доступные книги (статус = 'На складе')  
    num_instances_available =  
        BookInstance.objects.filter(status__exact=2).count()  
    num_authors = Author.objects.count() # Метод 'all()' применен по умолчанию.  
  
    # Количество посещений этого view, подсчитанное в переменной session  
    num_visits = request.session.get('num_visits', 0)  
    request.session['num_visits'] = num_visits + 1  
  
    # Отрисовка HTML-шаблона index.html с данными внутри переменной context  
    return render(request, 'index.html',  
                 context={'num_books': num_books,  
                           'num_instances': num_instances,  
                           'num_instances_available': num_instances_available,  
                           'num_authors': num_authors,  
                           'num_visits': num_visits},  
                 )
```

В первую очередь мы формируем здесь переменную 'num_visits' из сессии и устанавливаем для нее значение 0 (если оно не было установлено ранее). Затем каждый раз при обращении к задействованному представлению (view) мы увеличиваем значение этой переменной на единицу и сохраняем его в сессии (до следующего посещения этой страницы пользователем). В завершение переменная num_visits передается в шаблон главной страницы через переменную context. Для показа значения этой переменной добавьте одну строчку кода (выделенную в листинге 10.2 серым фоном и полужирным шрифтом) в шаблон главной страницы сайта \WebBooks\catalog\templates\index.html.

Листинг 10.2

```
<p>База данных сайта Мир книг содержит следующее количество записей:</p>
<ul>
    <li><strong>Количество книг:</strong> {{ num_books }}</li>
    <li><strong>Количество экземпляров книг:</strong> {{ num_instances }}</li>
    <li><strong>Количество экземпляров книг в заказе:</strong>
        {{ num_instances_available }}</li>
    <li><strong>Количество авторов книг:</strong> {{ num_authors }}</li>
</ul>
<p>Количество посещений данной страницы - {{ num_visits }}</p>
```

Запустим наш сервер и несколько раз выполним вход на главную страницу. Значение количества посещений будет изменяться всякий раз при каждом входе на главную страницу (рис. 10.2).

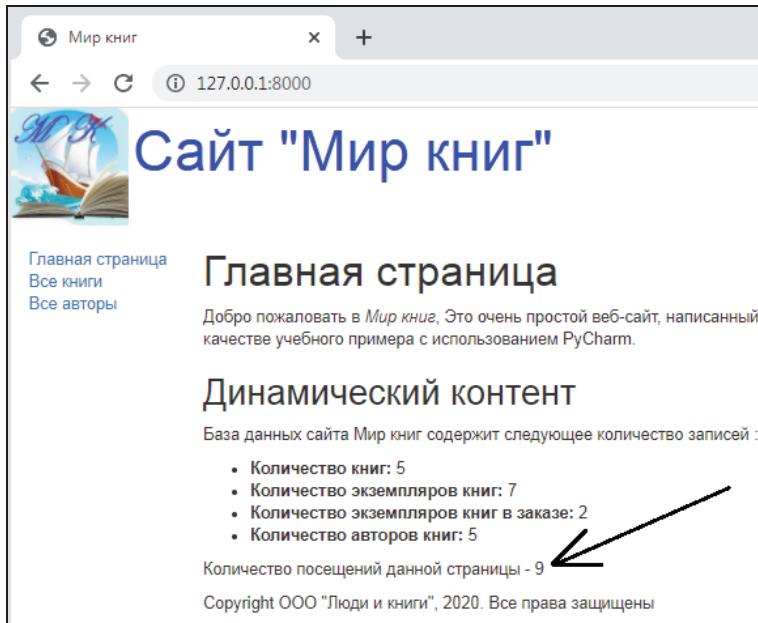


Рис. 10.2. Счетчик посещения страницы сайта на базе переменной session

10.2. Аутентификация и авторизация пользователей в Django

В этом разделе мы продемонстрируем возможность входа пользователя на сайт с использованием его собственного аккаунта. Кроме того, мы покажем, как реализовать контроль действий пользователя в зависимости от того, вошел он на сайт с собственным паролем или нет, а также имеет ли он соответствующий уровень прав доступа (permissions) к элементам сайта. Для того чтобы показать все это, мы расширим сайт «Мир книг», добавив в него страницы для входа/выхода, а также страницы просмотра/редактирования книг, специфические для пользователя и для персонала.

10.2.1. Немного об аутентификации пользователей в Django

Django предоставляет систему аутентификации и авторизации (permission) пользователя, реализованную на основе сессий, с которыми мы познакомились в предыдущем разделе. Система аутентификации и авторизации позволяет проверять учетные данные пользователей и определять, какие действия они могут выполнять. Django имеет встроенные модели для отдельных «Пользователей» и «Групп» пользователей (более чем один пользователь с одинаковыми правами), а также саму систему прав доступа.

Мы реализуем аутентификацию пользователя путем создания страниц входа/выхода и добавления разграничения доступа к моделям данных, а также продемонстрируем контроль доступа к некоторым страницам. Мы применим авторизацию для показа пользователям и сотрудникам, сопровождающим сайт, списков книг, которые находятся в заказе.

Система аутентификации является очень гибкой и позволяет формировать свои собственные URL-адреса, формы, отображения, а также шаблоны страниц для зарегистрированных пользователей. Но в этом разделе мы воспользуемсястроенными в Django методами показа данных и построения форм аутентификации, а также методами построения страниц входа и выхода. Для этого понадобится создать шаблоны соответствующих страниц, но это будет довольно просто.

Система аутентификации пользователей была подключена к сайту автоматически, когда мы создали его проект (при помощи команды `django-admin startproject`), так что сейчас нам ничего делать не надо. Таблицы базы данных для пользователей и модели авторизации были созданы, когда мы в первый раз выполнили команду миграции (`python manage.py migrate`). Соответствующие настройки сделаны в параметрах `INSTALLED_APPS` и `MIDDLEWARE` файла проекта `settings.py` (рис. 10.3).

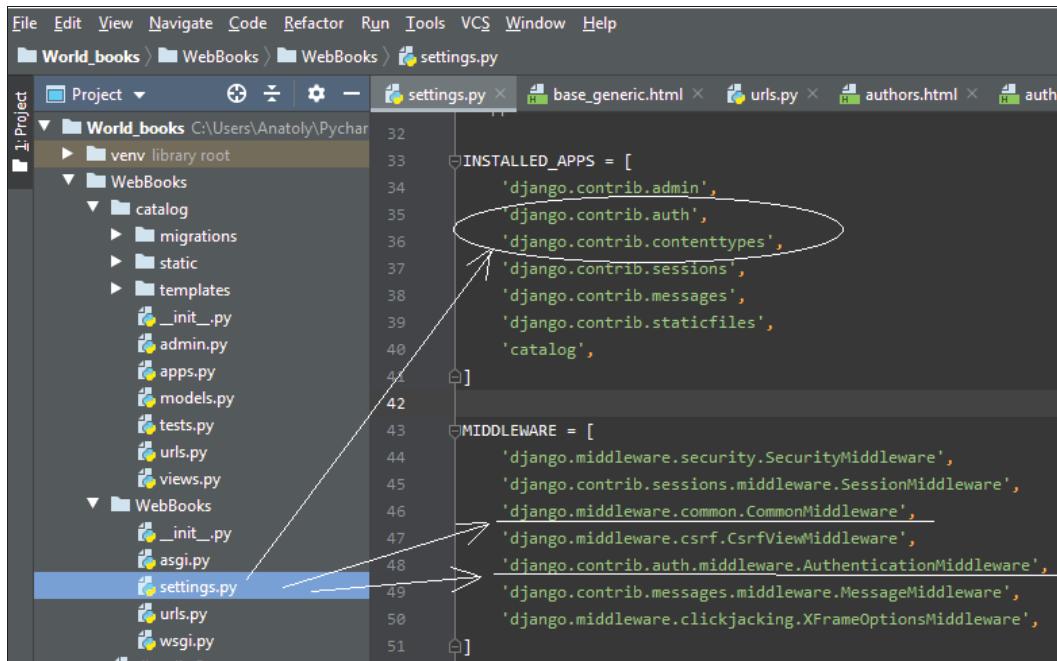


Рис. 10.3. Параметры настройки авторизации пользователей в файле settings.py

10.2.2. Создание отдельных пользователей и групп пользователей

Мы уже создали первого пользователя, когда рассматривали административную панель сайта. Это был суперпользователь, созданный при помощи команды `python manage.py createsuperuser`. Наш суперпользователь уже авторизован и имеет все необходимые уровни доступа и к данным, и к функциям. Таким образом, нам необходимо создать тестового пользователя.

Можно создавать пользователей и программным способом. Например, в том случае, если вы разрабатываете интерфейс, который позволяет пользователям создавать их собственные аккаунты (при этом вы не должны предоставлять доступ пользователям к административной панели вашего сайта). Вот образец такого программного кода:

```

from django.contrib.auth.models import User

# Создать пользователя и сохранить его в базе данных
user = User.objects.create_user('myusername',
                               'myemail@crazymail.com',
                               'mypassword')

# Обновить поля и сохранить их снова
user.first_name = 'Михаил'
user.last_name = 'Петров'
user.save()

```

Тем не менее мы создадим группу, а затем пользователя в этой группе с помощью административной панели сайта. Несмотря на то что у нас пока нет никаких разрешений на добавление к нашей библиотеке каких-либо читателей, если мы захотим это сделать в будущем, то будет намного проще добавлять их к уже созданной группе с заданной аутентификацией. И административная панель сайта предоставляет наиболее быстрый способ создания соответствующих групп и аккаунтов.

Итак, запускаем сервер разработки и переходим к административной панели сайта (<http://127.0.0.1:8000/admin/>). Заходим на сайт при помощи параметров аккаунта суперпользователя (имени пользователя и пароля). Самая верхняя страница панели администратора показывает все наши модели. Для того чтобы увидеть записи в разделах **Authentication** и **Authorisation**, вы можете нажать на ссылку **Группы** или **Пользователи** в разделе **ПОЛЬЗОВАТЕЛИ И ГРУППЫ** (рис. 10.4).

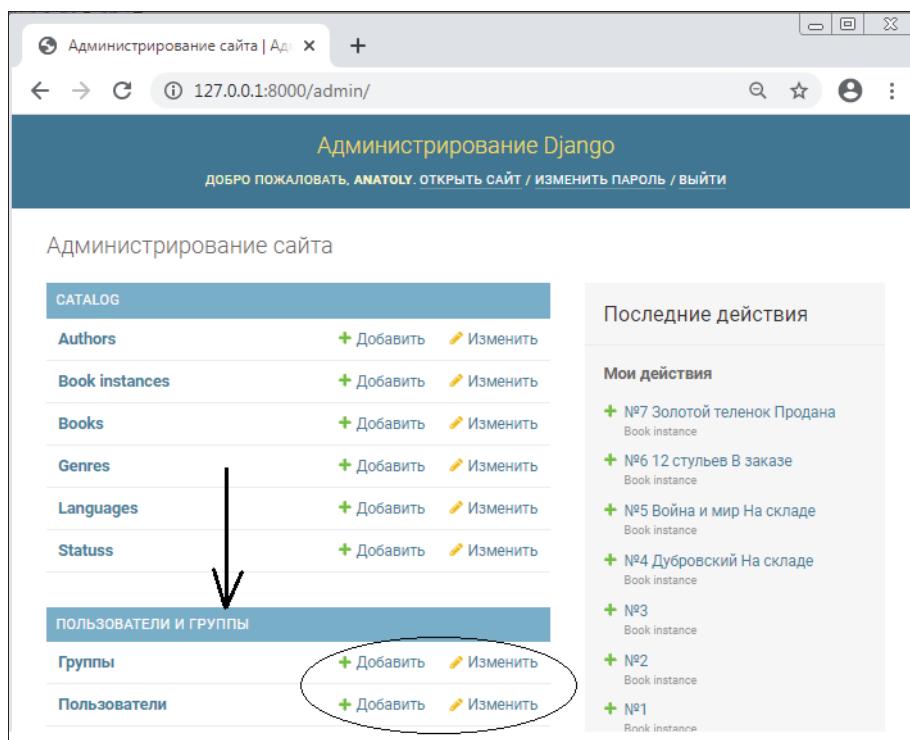


Рис. 10.4. Разделы административной панели сайта для создания групп и пользователей

В первую очередь давайте создадим новую группу — нажмите для этого на ссылку **+Добавить** в строке **Группы**. Введите для нее имя: Пользователи сайта (рис. 10.5). Для этой группы не нужны какие-либо разрешения, поэтому просто нажмите кнопку **СОХРАНИТЬ**, и вы перейдете к списку групп.

Теперь давайте создадим пользователя. Для этого возвращаемся на главную страницу административной панели и нажимаем ссылку **+Добавить** в строке **Пользователи**, после чего откроется окно со списком пользователей (рис. 10.6).

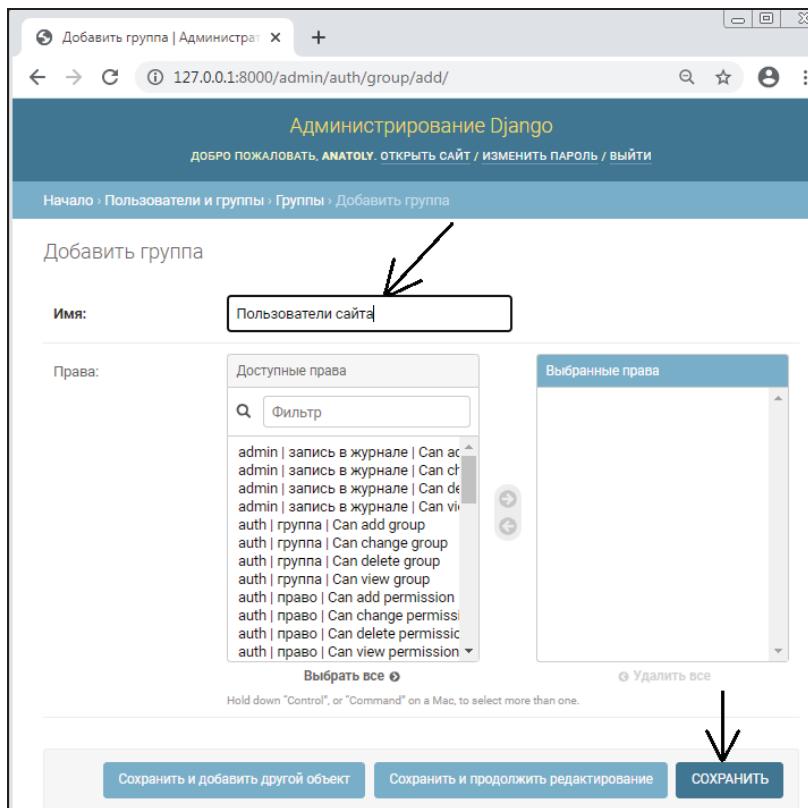


Рис. 10.5. Создание группы пользователей в административной панели сайта

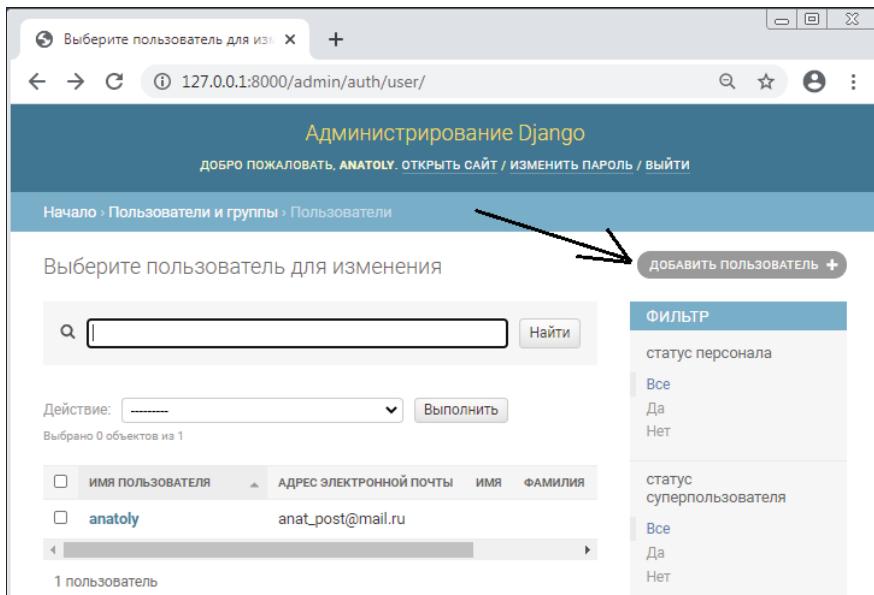


Рис. 10.6. Окно со списком пользователей в административной панели сайта

В этом окне нажмите на кнопку **ДОБАВИТЬ ПОЛЬЗОВАТЕЛЯ +**. Будет выполнен переход к диалоговому окну ввода сведений о новом пользователе (рис. 10.7).

Введите здесь в полях **Имя пользователя**, **Пароль** и **Подтверждение пароля** соответствующие данные для нашего тестового пользователя и нажмите кнопку **СОХРАНИТЬ** для завершения процесса создания пользователя.

The screenshot shows the 'Добавить пользователя' (Add user) page in the Django Admin interface. The URL in the browser is 127.0.0.1:8000/admin/auth/user/add/. The page title is 'Администрирование Django'. Below it, there are links: 'ДОБРО ПОЖАЛОВАТЬ, ANATOLY. ОТКРЫТЬ САЙТ / ИЗМЕНИТЬ ПАРОЛЬ / ВЫЙТИ'. The main heading is 'Начало > Пользователи и группы > Пользователи > Добавить пользователя'. The form itself has a title 'Добавить пользователя' and a note: 'First, enter a username and password. Then, you'll be able to edit more user options.' It contains four input fields: 'Имя пользователя:' (username), 'Пароль:' (password), 'Подтверждение пароля:' (password confirmation), and a note below the confirmation field: 'Для подтверждения введите, пожалуйста, пароль ещё раз.' At the bottom are three buttons: 'Сохранить и добавить другой объект' (Save and add another object), 'Сохранить и продолжить редактирование' (Save and continue editing), and a large blue button 'СОХРАНИТЬ' (Save).

Рис. 10.7. Окно для ввода новых пользователей в административной панели сайта

Административная часть сайта создаст нового пользователя и немедленно перенаправит вас на страницу изменения параметров пользователя (рис. 10.8), где вы сможете ввести для него некоторую персональную информацию:

- изменить имя пользователя (входное имя на сайт);
- добавить имя пользователя (по паспорту);
- добавить фамилию пользователя;
- ввести его электронный адрес.

Как можно видеть, мы задали пользователю имя: **Тестер_1**. Если прокрутить страницу с параметрами пользователя вниз, то можно увидеть блок, в котором можно задать для него права доступа (рис. 10.9).

Администрирование Django

ДОБРО ПОЖАЛОВАТЬ, ANATOLY. ОТКРЫТЬ САЙТ / ИЗМЕНИТЬ ПАРОЛЬ / ВЫЙТИ

[Начало](#) · [Пользователи и группы](#) > [Пользователи](#) > Тестер_1

✓ The пользователь "Тестер_1" was added successfully. Вы можете снова изменить этот объект ниже.

Изменить пользователь

Имя пользователя: Тестер_1
Обязательное поле. Не более 150 символов. Только буквы, цифры и символы @/./+/-.

Пароль:
алгоритм: pbkdf2_sha256 итерации: 180000 соль: LnXkMB***** хэш: NhmeSe*****
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Персональная информация

Имя:

Фамилия:

Адрес электронной почты:

Рис. 10.8. Блок с персональными данными пользователя в административной панели сайта

Права доступа

Активный
Отметьте, если пользователь должен считаться активным. Уберите эту отметку вместо удаления учётной записи.

Статус персонала
Отметьте, если пользователь может входить в административную часть сайта.

Статус суперпользователя
Указывает, что пользователь имеет все права без явного их назначения.

Группы: Доступные группы Выбранные группы +

Группы, к которым принадлежит данный пользователь. Пользователь получит все права, указанные в каждой из его/её групп. Hold down "Control", or "Command" on a Mac, to select more than one.

Права пользователя: Доступные права пользователя Выбранные права пользователя

Индивидуальные права данного пользователя. Hold down "Control", or "Command" on a Mac, to select more than one.

Рис. 10.9. Блок задания прав доступа пользователей в административной панели сайта

В этом блоке можно определить статус пользователя и признак активности (может быть установлен только соответствующий флаг). Кроме того, можно определить группу для пользователя и необходимые параметры доступа.

В самой нижней части страницы изменения параметров пользователя имеется блок **Важные даты**, в котором можно внести даты, относящиеся к этому пользователю (дату подключения к сайту и дату последнего входа), а также расположена кнопочная панель для сохранения введенных изменений и перехода на другие страницы (рис. 10.10).

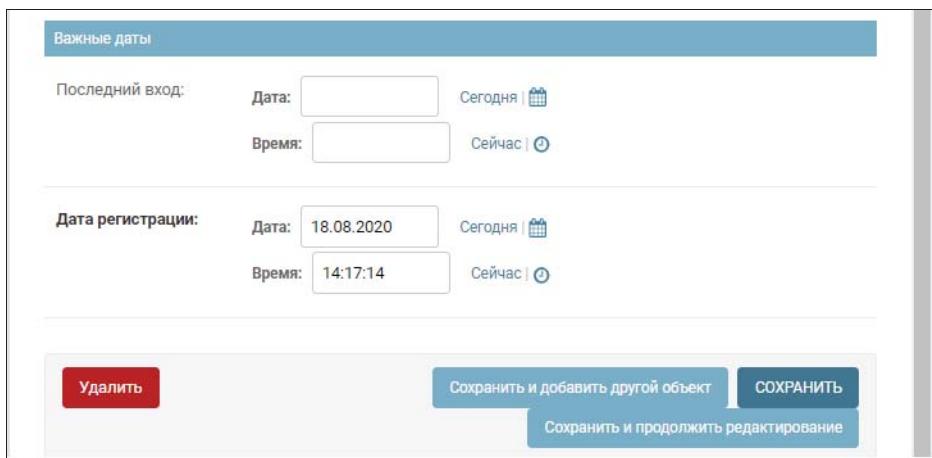


Рис. 10.10. Блок **Важные даты** и кнопочная панель в административной панели сайта

Вот и все! Теперь у нас есть учетная запись «обычного пользователя» сайта, которую можно использовать для тестирования работы сайта (после того, как мы добавим страницы входа пользователя в систему).

10.2.3. Создание страницы регистрации пользователя при входе на сайт

Django предоставляет почти все, что нужно для создания страниц аутентификации входа, выхода из системы и управления паролями. Этот набор включает в себя URL-адреса, представления (views) и формы, но не включает шаблоны (придется создать свой собственный шаблон самостоятельно).

ПРИМЕЧАНИЕ

Мы можем поместить страницы аутентификации, включая URL-адреса и шаблоны, в папку *catalog* нашего единственного приложения. Однако при наличии нескольких приложений лучше отделить от него вход в систему и обеспечить доступность страницы авторизации пользователей для всего сайта. Именно так мы и поступим.

Начнем с создания ссылок на URL-адреса. Добавьте в нижнюю часть файла проекта *WebBooks\WebBooks\urls.py* следующий код (листинг 10.3).

Листинг 10.3

```
from django.urls import path, include

# Добавление URL-адреса для входа в систему
urlpatterns += [
    path('accounts/', include('django.contrib.auth.urls')),
]
```

Следующий шаг — создайте каталог (папку) `registration` в папке `templates` (рис. 10.11).

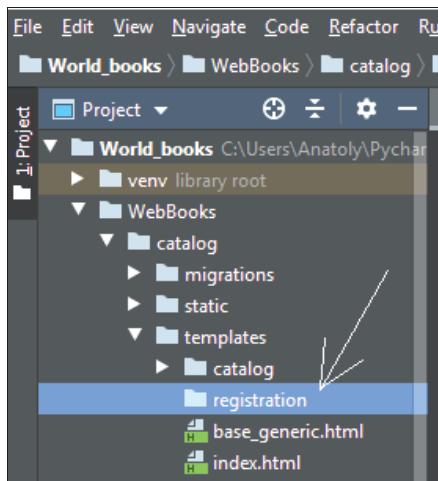


Рис. 10.11. Создание каталога `registration` в папке `templates`

Затем в эту папку добавьте шаблон — файл `login.html` (листинг 10.4).

Листинг 10.4

```
{% extends "base_generic.html" %}

{% block content %}

{% if form.errors %}
<p>Ваше имя пользователя и пароль не совпадают. Пожалуйста,  
попробуйте еще раз.</p>
{% endif %}

{% if next %}
{% if user.is_authenticated %}
<p>Вы не имеете доступа к этой странице. Войдите в систему  
с другими параметрами</p>
{% else %}
<p> Войдите в систему, чтобы увидеть эту страницу.</p>
```

```
{% endif %}  
{% endif %}  
  
<form method="post" action="{% url 'login' %}">  
  {% csrf_token %}  
<table>  
  
<tr>  
  <td>{{ form.username.label_tag }}</td>  
  <td>{{ form.username }}</td>  
</tr>  
  
<tr>  
  <td>{{ form.password.label_tag }}</td>  
  <td>{{ form.password }}</td>  
</tr>  
</table>  
  
<input type="submit" value="Вход" />  
<input type="hidden" name="next" value="{{ next }}" />  
</form>  
  
  {# Предполагается, что вы настроили представление password_reset в своем URLconf #}  
<p><a href="{% url 'password_reset' %}"> Утерян пароль?</a></p>  
  
{% endblock %}
```

Этот шаблон имеет сходство с теми шаблонами, что мы создавали раньше. Он расширяет наш базовый шаблон и переопределяет блок контента. Остальная часть кода — это довольно-таки стандартный код обработки формы, о котором мы поговорим в следующем разделе. Все, что вам нужно сделать, — это показать форму, в которую можно ввести имя пользователя и пароль, а если будут введены недопустимые значения, предложить (когда страница обновится) ввести правильные.

Чтобы сделать наш шаблон доступным, откроем настройки проекта (файл `settings.py`) и обновим в секции `TEMPLATES` (рис. 10.12) строку `'DIRS':`

```
os.path.join(BASE_DIR, 'templates')
```

Если теперь перейти по адресу: `http://127.0.0.1:8000/accounts/login/`, то вы должны увидеть страницу входа с полями идентификации пользователя (рис. 10.13).

Введите здесь входные данные пользователя, и вы будете перенаправлены на другую страницу сайта — по умолчанию это страница профиля пользователя по адресу: `http://127.0.0.1:8000/accounts/profile/`. Но поскольку страницу с профилем мы еще не сделали, то вам будет выдано сообщение об ошибке с информацией, что страница `profile` отсутствует (рис. 10.14).

Ликвидируем этот недочет. Откройте настройки проекта (файл `settings.py`) и добавьте в конец файла следующий код (листинг 10.5).

```

File Edit View Navigate Code Refactor Run Tools VCS Window Help
World_books > WebBooks > WebBooks > settings.py
Project World_books C:\Users\Anatoly\PycharmProjects\World_books
  venv library root
    WebBooks
      catalog
      migrations
      static
      templates
        __init__.py
        admin.py
        apps.py
        models.py
        tests.py
        urls.py
        views.py
    WebBooks
      __init__.py
      asgi.py
      settings.py
settings.py
46     'django.middleware.common.CommonMiddleware',
47     'django.middleware.csrf.CsrfViewMiddleware',
48     'django.contrib.auth.middleware.AuthenticationMiddleware',
49     'django.contrib.messages.middleware.MessageMiddleware',
50     'django.middleware.clickjacking.XFrameOptionsMiddleware'
51   ]
52
53 ROOT_URLCONF = 'WebBooks.urls'
54
55 TEMPLATES = [
56   {
57     'BACKEND': 'django.template.backends.django.DjangoTemplates',
58     'DIRS': [os.path.join(BASE_DIR, 'templates')], ←
59     'APP_DIRS': True, →
60     'OPTIONS': {
61       'context_processors': [
62         'django.template.context_processors.debug',
63         'django.template.context_processors.request',
64       ],
65     },
66   },
67 ]

```

Рис. 10.12. Обновление в секции TEMPLATES строки 'DIRS'

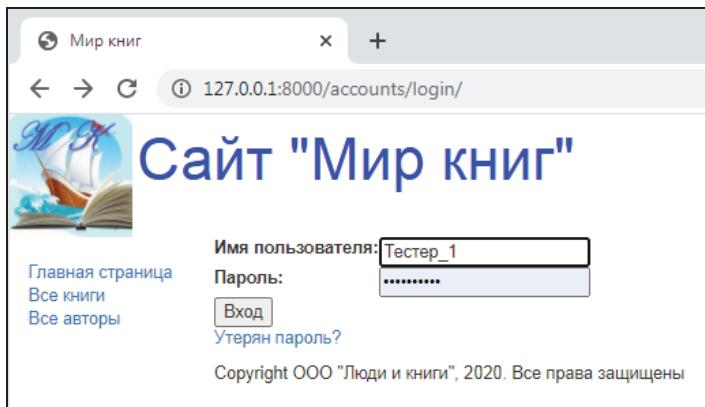


Рис. 10.13. Страница входа с идентификацией пользователя

Листинг 10.5

```
# Переадресация на главную страницу сайта после входа в систему
LOGIN_REDIRECT_URL = '/'
```

Теперь, если повторить вход в систему, то ошибка больше выдаваться не будет, и пользователь будет успешно перенаправлен на домашнюю страницу сайта.

Если же вы перейдете по URL-адресу выхода: <http://127.0.0.1:8000/accounts/logout/>, то увидите, что пользователь выведен из системы, но это по умолчанию окно панели администратора сайта (рис. 10.15).

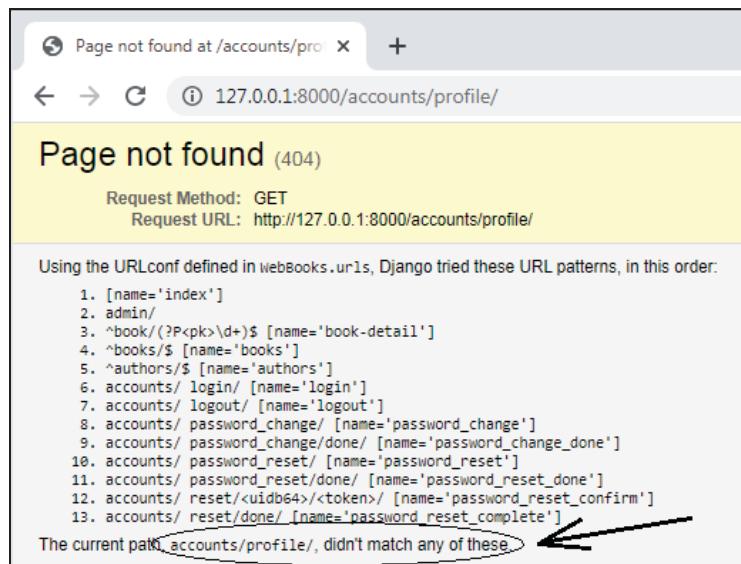


Рис. 10.14. Сообщение об отсутствии страницы profile

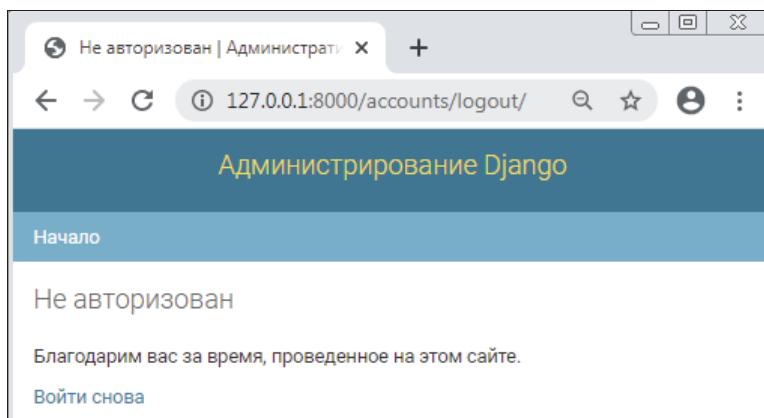


Рис. 10.15. Окно с сообщением об успешном выходе из системы панели администратора сайта

Тем не менее это не то, что нам нужно (такое окно должны получать только пользователи, у которых есть статус `is_staff`, — т. е. персонал, обслуживающий сайт). Чтобы этого не происходило, создадим шаблон страницы выхода из системы (файл `templates\registration\logged_out.html`) и внесем в этот файл следующий код (листинг 10.6).

Листинг 10.6

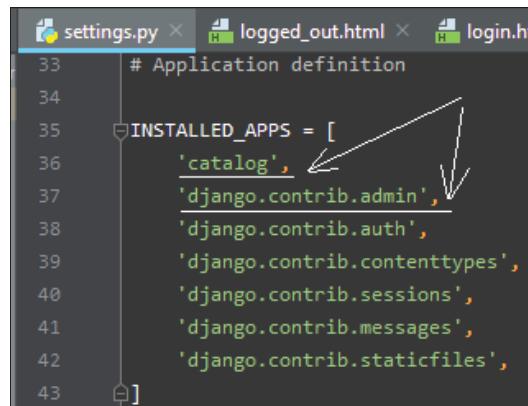
```
{% extends "base_generic.html" %}

{% block content %}
<p>Выход из системы!</p>
```

```
<a href="{% url 'login'%}"> Нажмите здесь, чтобы снова войти в систему.</a>
{% endblock %}
```

Этот шаблон очень прост. Он показывает сообщение о том, что вы вышли из системы, и предоставляет ссылку, которую вы можете нажать, чтобы вернуться на экран входа в систему.

Попробуем снова перейти на страницу выхода из системы. Если вы опять видите страницу выхода из системы панели администратора сайта (см. рис. 10.15) вместо своей собственной страницы выхода, то пугаться не стоит. Проверьте настройку `INSTALLED_APPS` вашего проекта в файле `settings.py` и убедитесь, что строка: `'django.contrib.admin'` стоит после строки с вашим приложением: `'catalog'` (рис. 10.16).



```
# Application definition

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

Рис. 10.16. Правильный порядок описания приложений в настройках `INSTALLED_APPS`

Если строка: `'django.contrib.admin'` стоит выше, чем строка: `'catalog'`, то поменяйте их местами. Дело в том, что оба шаблона расположены по одному и тому же относительному пути, и загрузчик шаблонов Django воспользуется первым най-

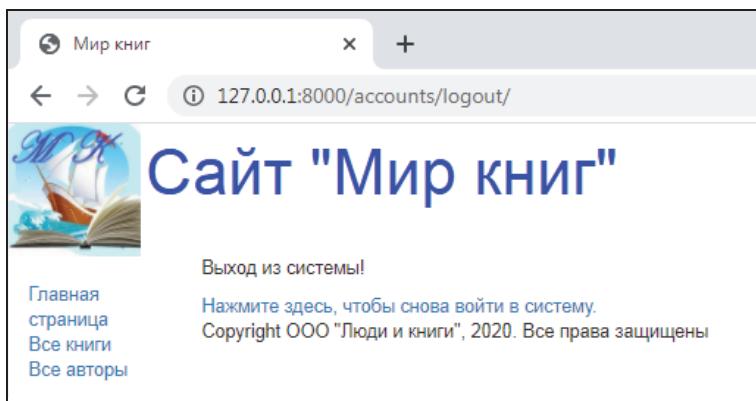


Рис. 10.17. Страница успешного выхода из системы на основе шаблона `logged_out.html`

денным. Если все было сделано правильно, то при запросе по адресу: <http://127.0.0.1:8000/accounts/logout/> будет загружен страница на основе вашего шаблона `logged_out.html` (рис. 10.17).

10.2.4. Создание страницы для сброса пароля пользователя

Довольно часто интернет-пользователи забывают свой пароль входа на тот или иной сайт. Для таких забывчивых пользователей необходимо предусмотреть возможность восстановления доступа к своему аккаунту путем смены пароля. Система сброса пароля по умолчанию использует электронную почту, чтобы отправить пользователю ссылку на сброс. Для этого на сайте необходимо создать форму, чтобы получить адрес электронной почты пользователя, отправить ему электронное письмо, разрешить ввести новый пароль и сделать отметку о завершении процесса.

Для этих целей создадим в каталоге `\templates\registration\` новую форму (шаблон) `password_reset_form.html` (рис. 10.18).

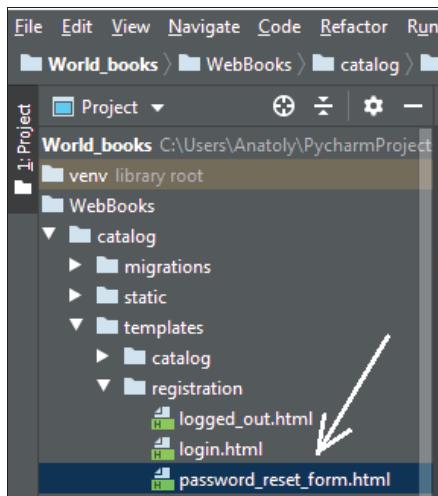


Рис. 10.18. Создание файла шаблона (формы) для сброса пароля пользователя

В созданный файл внесем следующий код (листинг 10.7).

Листинг 10.7

```
{% extends "base_generic.html" %}  
{% block content %}  
  
<form action="" method="post">{% csrf_token %}  
    {% if form.email.errors %} {{ form.email.errors }} {% endif %}  
    <label>Ваш e-mail</label><br>  
    <p>{{ form.email }}</p>  
    <input type="submit" class="btn btn-default btn-lg" value="Сброс пароля" />
```

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

Затем создадим в каталоге `\templates\registration\` форму `password_reset_done.html` (рис. 10.19), которая будет показана пользователю после того, как он введет адрес электронной почты, и внесем в нее следующий код (листиng 10.8).

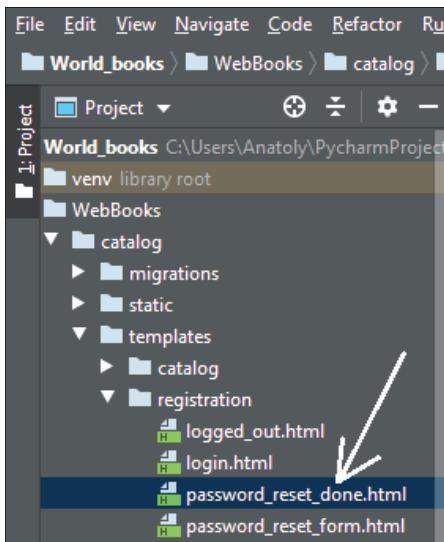


Рис. 10.19. Создание файла шаблона (формы) для смены пароля пользователя

Листинг 10.8

```
{% extends "base_generic.html" %}
{% block content %}
<p>Мы отправили вам по электронной почте инструкции  
по установке пароля. Если они не получены в течение  
нескольких минут, то проверьте папку со спамом.</p>
{% endblock %}
```

Следующий шаблон предоставляет собой текст письма, которое мы отправим пользователю по электронной почте, — в нем содержится ссылка на сброс пароля. Для него в том же каталоге `\templates\registration\` создадим файл `password_reset_email.html` и внесем в него следующий код (листиng 10.9).

Листинг 10.9

Для смены пароля пользователя с электронной почтой {{ email }} перейдите по ссылке ниже:
{{ protocol}}://{{ domain }}{{ url 'password_reset_confirm' uidb64=uid token=token }}

И это еще не все — теперь необходимо дать пользователю возможность ввести новый пароль. По нажатию на ссылку в полученном электронном письме пользова-

тель должен попасть на страницу сайта, где сможет это сделать. Чтобы предоставить ему такую возможность, в том же каталоге `\templates\registration\` создадим шаблон `password_reset_confirm.html` и внесем в него следующий код (листинг 10.10).

Листинг 10.10

```
{% extends "base_generic.html" %}

{% block content %}

    {% if validlink %}
        <p>Пожалуйста, введите (и подтвердите) свой новый пароль.</p>
        <form action="" method="post">
            {% csrf_token %}
            <table>
                <tr>
                    <td>{{ form.new_password1.errors }}</td>
                    <label for="id_new_password1"> Новый пароль:</label></td>
                <td>{{ form.new_password1 }}</td>
                </tr>
                <tr>
                    <td>{{ form.new_password2.errors }}</td>
                    <label for="id_new_password2"> Подтвердите
                        пароль:</label></td>
                <td>{{ form.new_password2 }}</td>
                </tr>
                <tr>
                    <td></td>
                    <td><input type="submit" value=" Сменить пароль " /></td>
                </tr>
            </table>
        </form>
    {% else %}
        <h1> Ошибка при смене пароля!!! </h1>
        <p> Ссылка на сброс пароля была недействительной, возможно, потому, что
            она уже использовалась. Пожалуйста, запросите новый сброс пароля.</p>
    {% endif %}

    {% endblock %}
```

Ну и, наконец, создадим в том же каталоге `\templates\registration\` последний шаблон `password_reset_complete.html`, с помощью которого уведомим пользователя об успешном завершении смены пароля, и внесем в него следующий код (листинг 10.11).

Листинг 10.11

```
{% extends "base_generic.html" %}

{% block content %}
```

```
<h1> Пароль был успешно изменен!</h1>
<p><a href="{% url 'login' %}">Повторить вход в систему?</a></p>

{% endblock %}
```

Теперь вы сможете проверить функцию смены пароля по ссылке на странице входа в систему.

ПРИМЕЧАНИЕ

Имейте в виду, что Django отправляет электронные письма для смены пароля только на те адреса пользователей, которые хранятся в базе данных в его аккаунте!

Следует также отметить, что система смены пароля требует, чтобы ваш сайт поддерживал электронную почту, описание организации которой выходит за рамки книги. Так что эта часть сайта пока еще не будет работать полноценно, однако разработчик имеет возможность протестировать работу всех созданных шаблонов страниц.

Чтобы разрешить их тестирование, поместите в конец файла `settings.py` следующую строку (листинг 10.12).

Листинг 10.12

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Это обеспечит просмотр любых писем, отправленных на консоль (чтобы вы могли скопировать ссылку на сброс пароля с консоли). Получить дополнительную информацию об отправке с сайта писем пользователям по электронной почте можно в оригинальной документации на Django.

Проверим работу созданных нами шаблонов и форм. Для этого запустим наш сайт и войдем на страницу входа пользователя: <http://127.0.0.1:8000/accounts/login/> (рис. 10.20).

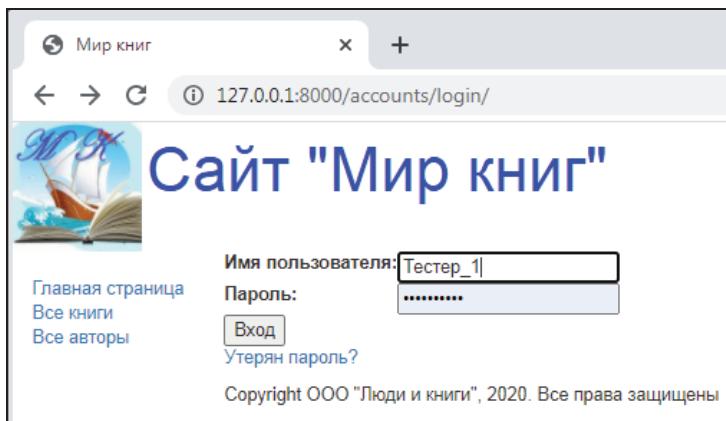


Рис. 10.20. Страница входа в аккаунт пользователя

Щелкнем на этой странице на ссылке **Утерян пароль?** — вам будет выдана страница с предложением ввести свой электронный адрес для смены пароля (рис. 10.21).

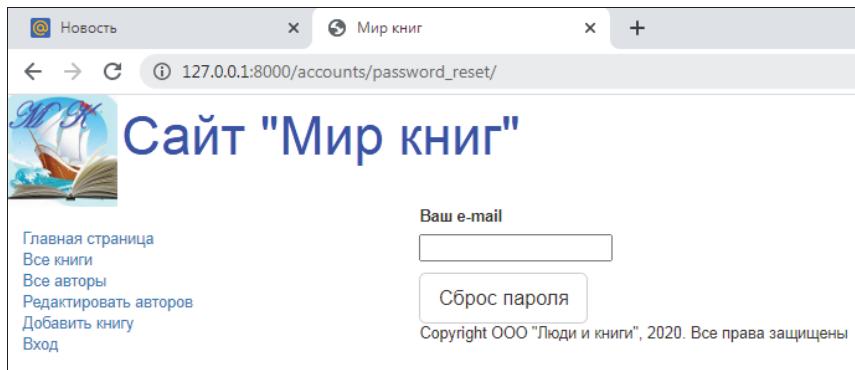


Рис. 10.21. Страница ввода электронного адреса пользователя для смены пароля

Введем электронный адрес пользователя и нажмем на кнопку **Сброс пароля** — появится страница с сообщением, что пользователю на его электронный адрес отправлено письмо с инструкциями по смене пароля (рис. 10.22).

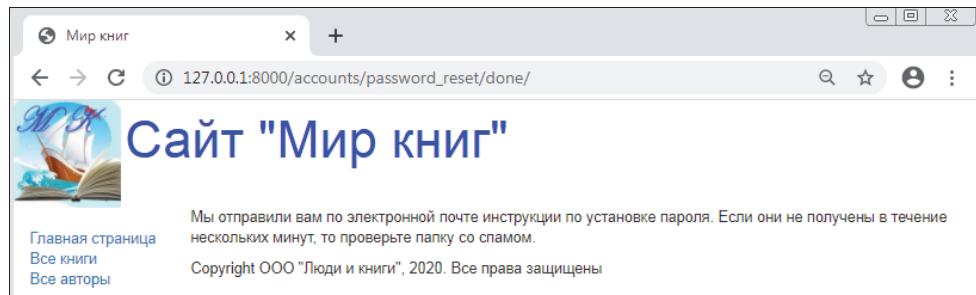


Рис. 10.22. Страница, сообщающая пользователю об отправке ему инструкции по смене пароля

Поскольку наш сайт еще не настроен на отправку электронной почты с внешнего сервера на реальный адрес, то для тестирования работы приложения инструкцию о смене пароля отправьте в окно терминала PyCharm (рис. 10.23).

A screenshot of a PyCharm terminal window titled 'Terminal: Local'. The text in the terminal reads: 'Для смены пароля пользователя с электронной почтой anat_post@mail.ru перейдите по ссылке ниже: http://127.0.0.1:8000/accounts/reset/Mg/5j6-194a2fdd2e1c274c8dbd/'.

Рис. 10.23. Инструкция для смены пароля, отправленная пользователю по электронной почте

Щелкните мышью на ссылке, представленной в полученной пользователем инструкции в окне терминала PyCharm, — вы вернетесь на наш сайт, где будет загружена страница для ввода нового пароля (рис. 10.24).

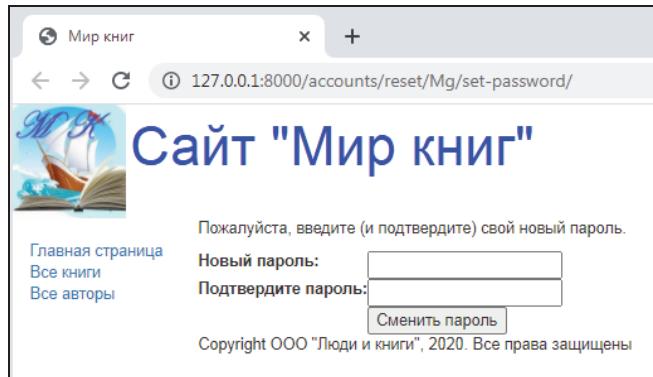


Рис. 10.24. Страница для смены пароля пользователя

На этой странице введем новый пароль, его подтверждение и нажмем на кнопку **Сменить пароль**. Если все было сделано правильно, то откроется окно с сообщением об успешной смене пароля (рис. 10.25). Нажмите в этом окне на ссылку **Повторить вход в систему?**, и вы вернетесь на сайт «Мир книг».

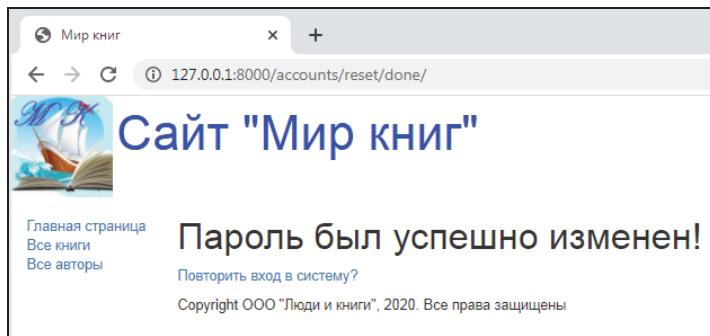


Рис. 10.25. Страница с сообщением об успешной смене пароля

В базе данных пароли пользователей хранятся в зашифрованном виде, и даже главный администратор сайта не может их знать. В этом можно убедиться, открыв с помощью менеджера SQLite таблицу БД с аккаунтами пользователей (рис. 10.26).

auth_user (db0)	
Структура Данные Ограничения Индексы Триггеры DDL	
Табличный вид Форма	
1	pbkdf2_sha256\$180000\$BeG91aBeyta\$C2XQmRuSVntXN4Ug+ULY7NxFV6w962v6rWvcitJAhIA=
2	pbkdf2_sha256\$180000\$RBKbAmksb1dq\$CeMsib8c4eEta3U1gT8xWEL13K7PutB+0NrFNKRuc=

Рис. 10.26. Зашифрованные пароли в базе данных сайта

10.3. Проверка подлинности входа пользователя в систему

При желании можно получить информацию о текущем статусе зарегистрированных в системе пользователей. Для этого в шаблонах присутствует переменная `{{user}}` (она добавляется в контекст шаблона по умолчанию при настройке проекта).

Обычно сначала проверяется переменная шаблона `{{user.is_authenticated}}`, позволяющая определить, имеет ли пользователь право видеть конкретный контент. Чтобы продемонстрировать это, мы обновим нашу боковую панель и покажем в ней ссылку **Вход** (если пользователь вышел из системы) и ссылку **Выход** (если он вошел в систему).

Откройте базовый шаблон `\catalog\templates\base_generic.html` и добавьте следующий текст (листинг 10.13) в блок `sidebar` непосредственно перед тегом `endblock` (рис. 10.27).

Листинг 10.13

```
<ul class="sidebar-nav">
  ...
  {% if user.is_authenticated %}
    <li>Пользователь: {{ user.get_username }}</li>
    <li><a href="{% url 'logout'%}?next={{request.path}}">Logout</a></li>
  {% else %}
    <li><a href="{% url 'login'%}?next={{request.path}}">Login</a></li>
  {% endif %}
</ul>
```

```
World_books [C:\Users\Anatoly\PycharmProjects\World_books] - ...\\WebBooks\\catalog\\templates\\base_generic.html - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
World_books > WebBooks > catalog > templates > base_generic.html
Project C:\Users\Anatoly\PycharmProjects\World_books
library root
books
catalog
migrations
static
templates
catalog
author_list.html
authors.html
book_detail.html
book_list.html
bookinstance_list.html
registration
base_generic.html
index.html



{% block sidebar %}



  {% endblock %}


```

Рис. 10.27. Блок проверки входа/выхода пользователя на сайт в боковой панели базового шаблона

Как можно видеть, в этом коде используются теги шаблона `if...else...endif` для условного отображения текста в зависимости от того, является ли значение `{{user.is_authenticated}}` истинным. Если пользователь аутентифицирован, то мы знаем, что это вошел зарегистрированный пользователь, поэтому мы вызываем `{{user.get_username}}`, чтобы отобразить его имя.

Затем мы создаем URL-адрес для входа и выхода из системы, используя тег шаблона URL-адреса и имена соответствующих конфигураций `url`. Обратите также внимание на то, как мы добавили фрагмент: `?next={{request.path}}` в конец блока `url`. Это означает, что следующий URL-адрес содержит адрес (URL) текущей страницы в конце связанного URL-адреса. После успешного входа пользователя в систему представления будут использовать значение `next`, чтобы перенаправить пользователя обратно на страницу, где он изначально нажал ссылку входа в систему или выхода из нее.

Протестируем то, что мы сейчас сделали. Если теперь загрузить главную страницу сайта, то мы увидим, что на боковой панели появилась дополнительная ссылка с надписью **Вход** (рис. 10.28). Щелкните мышью на ссылке **Вход**, и вы попадете на страницу идентификации пользователя (рис. 10.29).

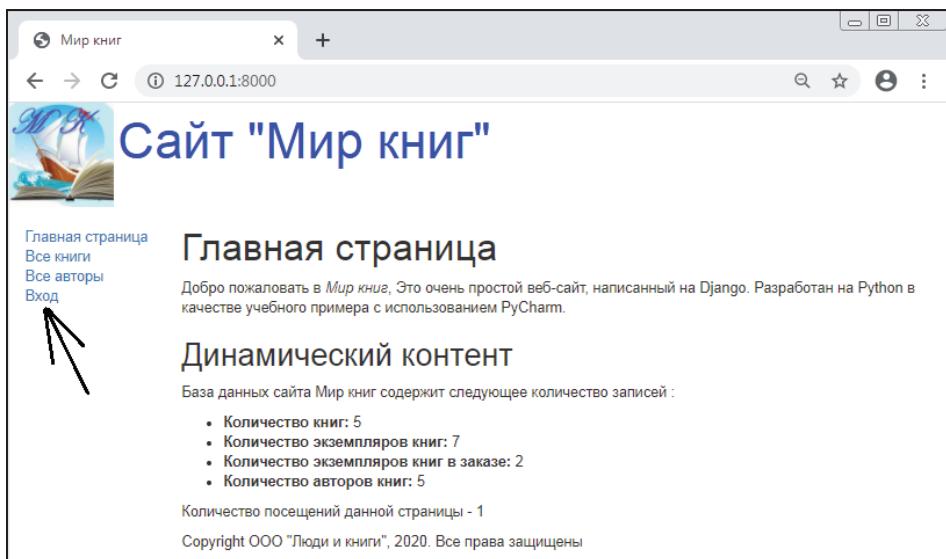


Рис. 10.28. Ссылка **Вход** на главной странице сайта, ведущая на страницу идентификации пользователя

Введите на этой странице имя и пароль нашего тестового пользователя и нажмите на ссылку **Вход** — мы опять попадем на главную страницу сайта, но при этом в боковой панели появится имя вошедшего пользователя, а ссылка **Вход** будет заменена на ссылку **Выход** (рис. 10.30).

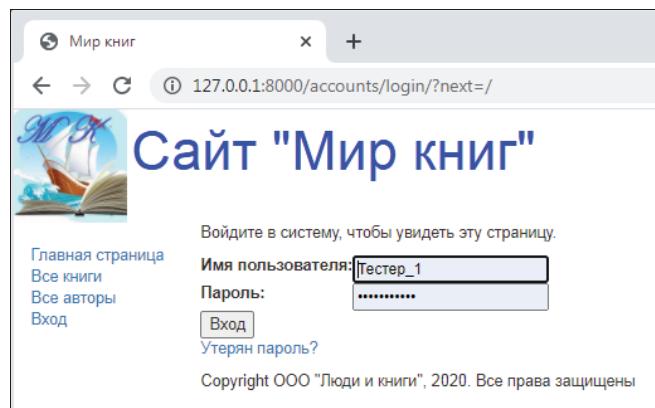


Рис. 10.29. Страница идентификации пользователя, куда он попадает, нажав на ссылку **Вход** на главной странице сайта

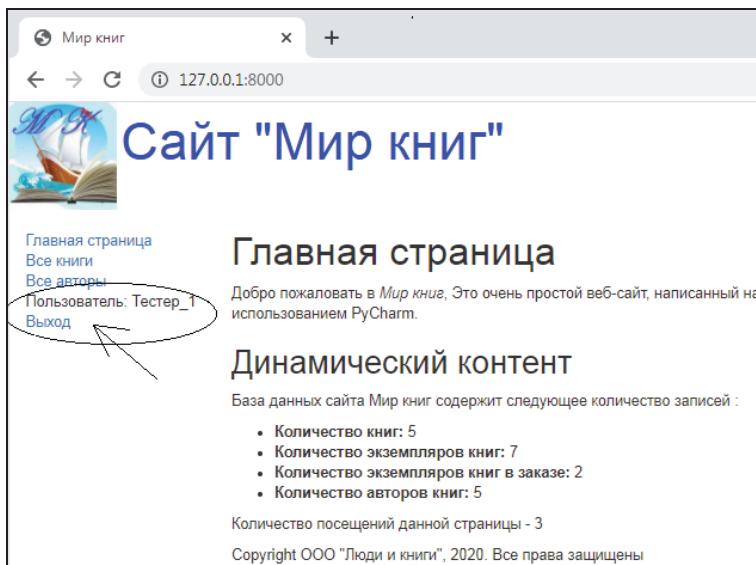


Рис. 10.30. Главная страница сайта после успешной идентификации пользователя

10.4. Формирование страниц сайта для создания заказов на книги

До сих пор у нас для пользователей не была реализована возможность делать заказы на книги (через интернет-магазин или библиотеку). Давайте реализуем такую возможность, представив, что сайт «Мир книг» — это некий аналог электронной библиотеки для удаленного пользователя.

Начнем мы с того, что расширим модель `BookInstance` (экземпляры книг) так, чтобы получить возможность использования приложения `Django Admin` для оформления заказа (выдачи) книг нашему тестовому пользователю.

Прежде всего, мы должны предоставить пользователям возможность взять книгу на какое-то время. В модели `BookInstance` уже есть поле для обозначения статуса экземпляра книги (`status`) и поле для хранения даты возврата книги (`due_back`), но у нас пока нет связи между этой моделью и моделью «пользователи». Мы создадим такую связь с помощью поля `ForeignKey` («один-ко-многим») — т. е. «один пользователь — несколько экземпляров книг». Кроме того, нужно будет создать простой механизм для проверки, не просрочил ли пользователь возврат полученной книги.

Импортируем модель User из модели `django.contrib.auth.models`, которая была создана в административной панели. Для этого откроем файл `catalog\models.py` и добавим строку кода (листинг 10.14) чуть ниже предыдущей строки импорта в верхней части файла (рис. 10.31).

Листинг 10.14

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

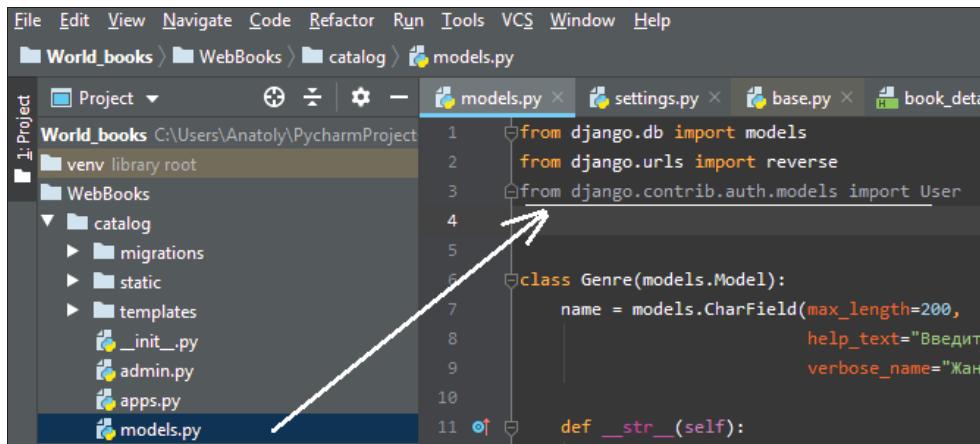


Рис. 10.31. Импортируем модель User из django.contrib.auth.models

Добавим в модель BookInstance поле `borrower` (заемщик, получатель) — в классе BookInstance (листинг 10.15) добавленный код выделен серым фоном и полужирным шрифтом.

Листинг 10.15

```
imprint = models.CharField(max_length=200,
                           help_text="Введите издательство и год выпуска",
                           verbose_name="Издательство")
status = models.ForeignKey('Status', on_delete=models.CASCADE,
                           null=True,
                           help_text='Изменить состояние экземпляра',
                           verbose_name='Статус экземпляра книги')
due_back = models.DateField(null=True, blank=True,
                           help_text="Введите конец срока статуса",
                           verbose_name="Дата окончания статуса")
borrower = models.ForeignKey(User, on_delete=models.SET_NULL,
                            null=True, blank=True,
                            verbose_name="Заказчик",
                            help_text='Выберите заказчика книги')
```

Теперь добавим в эту модель свойство, которое сможем использовать в наших шаблонах, чтобы показать, просрочен ли конкретный экземпляр книги. Хотя мы могли бы сделать такую проверку в самом шаблоне, использование свойства представляется более эффективным. Так что добавим в верхнюю часть файла catalog\models.py импорт модуля для обработки дат (листинг 10.16) и определение свойства внутри класса BookInstance (листинг 10.17).

Листинг 10.16

```
from datetime import date
```

Листинг 10.17

```
@property
def is_overdue(self):
    if self.due_back and date.today() > self.due_back:
        return True
    return False
```

Здесь сначала делается проверка — не является ли значение due_back (дата возврата) пустым. При пустом значении поля due_back пользователю вместо нужной страницы будет выдано сообщение об ошибке (пустые значения несопоставимы). А затем проверяется условие — не превысило ли значение даты возврата книги текущей даты. В зависимости от значений этих дат будут возвращаться значения True или False.

Так как мы обновили наши модели, то нужно внести изменения в проект, а именно — создать файл миграции и внести соответствующие изменения в саму базу данных. Для этого в окне терминала PyCharm последовательно выполним следующие команды:

```
python manage.py makemigrations
python manage.py migrate
```

Теперь откроем файл catalog\admin.py и добавим поле borrower в класс BookInstanceAdmin — как в list_display, так и в поле fieldsets. После этих изменений код для класса BookInstanceAdmin будет выглядеть следующим образом (листинг 10.19).

Листинг 10.19

```
@admin.register(BookInstance)
class BookInstanceAdmin(admin.ModelAdmin):
    list_display = ('book', 'status', 'borrower', 'due_back', 'id')
    list_filter = ('status', 'due_back')

    fieldsets = (
        (None, {
            'fields': ('book', 'imprint', 'inv_nom')
        }),
        ('Availability', {
            'fields': ('status', 'due_back', 'borrower')
        }),
    )
```

Эти изменения сделают поле borrower видимым в разделе Admin, так что мы сможем при необходимости назначить пользователя User для экземпляра книги BookInstance.

Теперь, когда возможно оформить выдачу книги конкретному пользователю, зайдем в панель администратора и закрепим за нашим тестовым пользователем несколько экземпляров книг в **Book Instances**: установим тестовому пользователю значение поля borrowed, присвоим значение status **В заказе** и назначим сроки возврата как на будущие, так и на прошедшие даты.

Итак, запускаем наш сайт, входим на страницу административной панели с паролем суперпользователя (см. рис. 8.31) и щелкаем мышью на ссылке **Book Instances** — на открывшейся странице в таблице со списком экземпляров книг появилась колонка **Заказчик** (рис. 10.32).

Как можно видеть, статус **В заказе** имеют здесь две книги. Загрузим страницу с экземпляром книги «12 стульев», имеющей статус **В заказе**, — на открывшейся странице с информацией об этом экземпляре книги появилось новое поле: **Заказчик**. В выпадающем списке выберем для этой книги в качестве заказчика пользователя с именем **Тестер_1** и сделаем этот заказ просроченным (выберем любую дату из прошедшего периода). В итоге получим следующую страницу (рис. 10.33). По нажатию на кнопку **СОХРАНИТЬ** будет выполнен возврат на предыдущую страницу, где мы увидим внесенные нами изменения (рис. 10.34).

Проделаем ту же процедуру с другой книгой, имеющей статус **В заказе**, но установим для нее дату возврата из будущего периода.

Теперь создадим представление (view) для получения списка всех книг, которые были заказаны пользователем. Мы воспользуемся для этого одним и тем же общим

Администрирование Django
ДОБРО ПОЖАЛОВАТЬ, ANATOLY. ОТКРЫТЬ САЙТ / ИЗМЕНИТЬ ПАРОЛЬ / ВЫЙТИ

Начало > Catalog > Book Instances

Выберите book instance для изменения

Действие: Выполнить Выбрано 0 объектов из 7

<input type="checkbox"/>	НАЗВАНИЕ КНИГИ	СТАТУС ЭКЗЕМПЛЯРА КНИГИ	ЗАКАЗЧИК	ДАТА ОКОНЧАНИЯ СТАТУСА	ID
<input type="checkbox"/>	Золотой теленок	Продана	-	3 августа 2020 г.	7
<input type="checkbox"/>	12 стульев	В заказе	-	19 августа 2020 г.	6
<input type="checkbox"/>	Война и мир	На складе	-	30 ноября 2020 г.	5
<input type="checkbox"/>	Дубровский	На складе	-	31 августа 2020 г.	4
<input type="checkbox"/>	Продавец воздуха	В заказе	-	28 августа 2020 г.	3
<input type="checkbox"/>	Продавец воздуха	На складе	-	30 ноября 2020 г.	2
<input type="checkbox"/>	Продавец воздуха	На складе	-	30 ноября 2020 г.	1

7 book instances

ФИЛЬР

Статус экземпляра книги

- Все
- На складе
- В заказе
- Продана

Дата окончания статуса

- Любая дата
- Сегодня
- Последние 7 дней
- Этот месяц
- Этот год
- Дата не указана
- Дата указана

Рис. 10.32. Колонка Заказчик в списке экземпляров книг

Администрирование Django
ДОБРО ПОЖАЛОВАТЬ, ANATOLY. ОТКРЫТЬ САЙТ / ИЗМЕНИТЬ ПАРОЛЬ / ВЫЙТИ

Начало > Catalog > Book Instances > №6 12 стульев В заказе

Изменить book instance

История

Название книги: 12 стульев

Издательство: АСТ, 1915

Инвентарный номер: №6

Статус и окончание его действия

Статус экземпляра книги: В заказе

Дата окончания статуса: 19.05.2020

Заказчик: Тестер_1

Удалить Сохранить и добавить другой объект Сохранить и продолжить редактирование СОХРАНИТЬ

Рис. 10.33. Выбор заказчика на конкретный экземпляр книги

<input type="checkbox"/>	НАЗВАНИЕ КНИГИ	СТАТУС ЭКЗЕМПЛЯРА КНИГИ	ЗАКАЗЧИК	ДАТА ОКОНЧАНИЯ СТАТУСА
<input type="checkbox"/>	Золотой теленок	Продана	-	3 августа 2020 г.
<input type="checkbox"/>	12 стульев	В заказе	Тестер_1	19 мая 2020 г.
<input type="checkbox"/>	Война и мир	На складе	-	30 ноября 2020 г.

Рис. 10.34. Результаты изменения поля ЗАКАЗЧИК и даты окончания статуса экземпляра книги

классом, с которым уже знакомы, но на этот раз также импортируем и используем класс `LoginRequiredMixin`, так что только вошедший со своим паролем пользователь сможет вызвать это представление. Мы также объявим шаблон имени `template_name` вместо того, чтобы использовать значение по умолчанию, потому что у нас может быть несколько разных списков записей `BookInstance` с разными представлениями и шаблонами.

Реализуем намеченные изменения, добавив в файл `catalog\views.py` следующий код (листинг 10.19).

Листинг 10.19

```
from django.contrib.auth.mixins import LoginRequiredMixin

class LoanedBooksByUserListView(LoginRequiredMixin, generic.ListView):
    """
    Универсальный класс представления списка книг,
    находящихся в заказе у текущего пользователя.
    """
    model = BookInstance
    template_name ='catalog/bookinstance_list_borrowed_user.html'
    paginate_by = 10

    def get_queryset(self):
        return BookInstance.objects.filter(
            borrower=self.request.user).
            filter(status__exact='2').order_by('due_back')
```

Чтобы ограничить наш запрос только объектами BookInstance для текущего пользователя, мы повторно реализуем здесь запрос `get_queryset()`. Обратите внимание, что '2' — это сохраненный код для справочника статусов **В заказе**, и мы сортируем книги в заказе по дате `due_back`, чтобы сначала отображались самые старые элементы из списка.

Затем откроем файл `\catalog\urls.py` и добавим адрес `url()`, который укажет на созданное в листинге 10.19 представление (код из листинга 10.20 просто добавляется в конец этого файла).

Листинг 10.20

```
urlpatterns += [
    url(r'^mybooks/$', views.LoanedBooksByUserListView.as_view(), name='my-borrowed'),
]
```

Теперь нам нужно создать шаблон для страницы, на которой будут показаны книги со статусом **В заказе**. Сначала создайте файл для этого шаблона: `\catalog\templates\catalog\bookinstance_list_borrowed_user.html`, а затем запишите в него следующий код (листинг 10.21).

Листинг 10.21

```
{% extends "base_generic.html" %}

{% block content %}
<h1>Взятые книги</h1>

{% if bookinstance_list %}
<ul>
    {% for bookinst in bookinstance_list %}
        <li class="{% if bookinst.is_overdue %}"
            text-danger{% endif %}">
            <a href="{% url 'book-detail' bookinst.book.pk %}">
                {{bookinst.book.title}}</a>
            ({{ bookinst.due_back }})
        </li>
    {% endfor %}
</ul>

{% else %}
<p>Нет книг со статусом В ЗАКАЗЕ.</p>
{% endif %}
{% endblock %}
```

Этот шаблон очень похож на тот, который мы создали ранее для объектов книги (`Book`) и авторы (`Author`). Единственное различие — в нем мы проверяем работу

добавленного нами в модель метода `bookinst.is_overdue`. Он используется для изменения цвета на красный для тех дат, которые показывают просроченный срок завершения заказов.

Протестируем работу сайта после внесенных изменений — запускаем наш сервер и видим главную страницу сайта (рис. 10.35). Пока на ней нет ничего необычного — ведь мы еще не прошли авторизацию в качестве зарегистрированного пользователя. Нажмите на ссылку **Вход** и пройдите авторизацию как пользователь с именем **Тестер_1** (рис. 10.36).

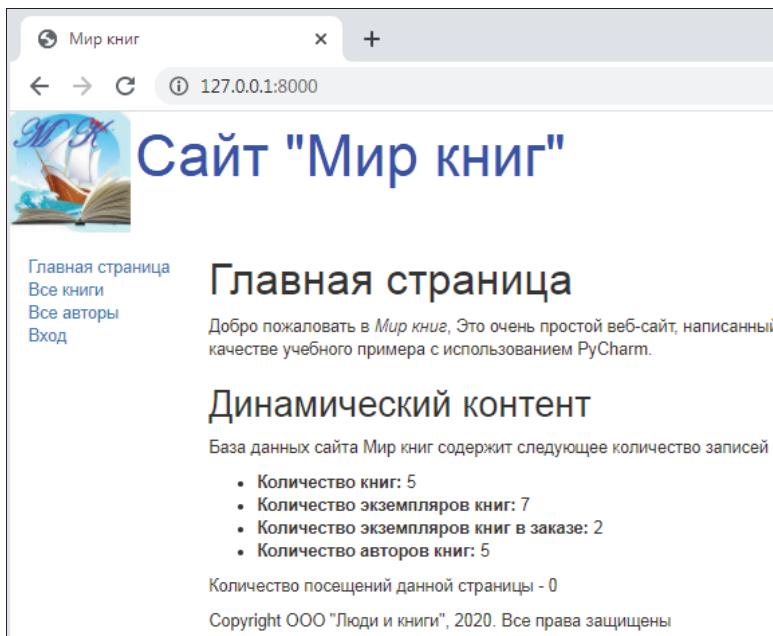


Рис. 10.35. Главная страница сайта «Мир книг»
с возможностью входа зарегистрированных пользователей

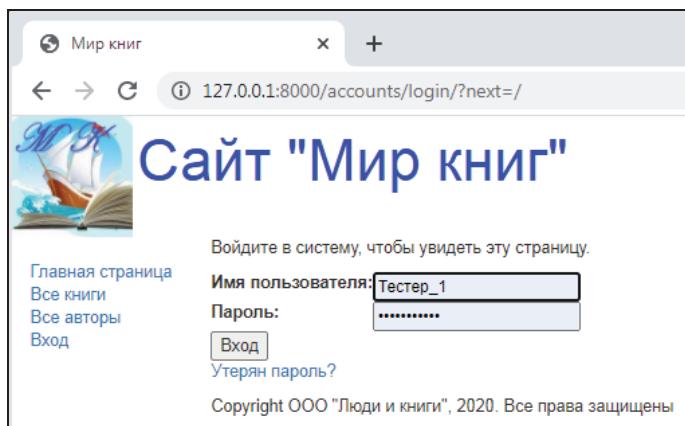


Рис. 10.36. Страница сайта «Мир книг» для входа зарегистрированных пользователей

Если теперь — после авторизации — снова нажать на ссылку **Вход**, то вы вернетесь на главную страницу сайта, но уже с возможностями зарегистрированного пользователя (рис. 10.37).

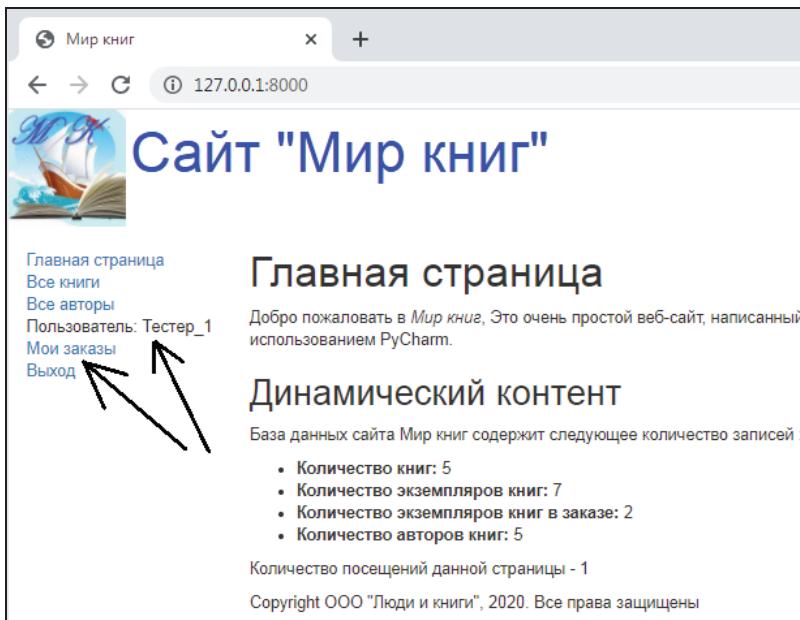


Рис. 10.37. Страница сайта «Мир книг» после входа авторизированного пользователя

На этой странице мы видим как минимум два признака того, что пользователь прошел авторизацию: появилось имя пользователя (**Тестер_1**), и появилась ссылка на страницу **Мои заказы** для этого пользователя. Если теперь щелкнуть мышью на ссылке **Мои заказы**, то откроется новая страница с книгами, которые заказал именно этот пользователь (рис. 10.38). Причем книга с просроченным сроком возврата будет выделена красным цветом.

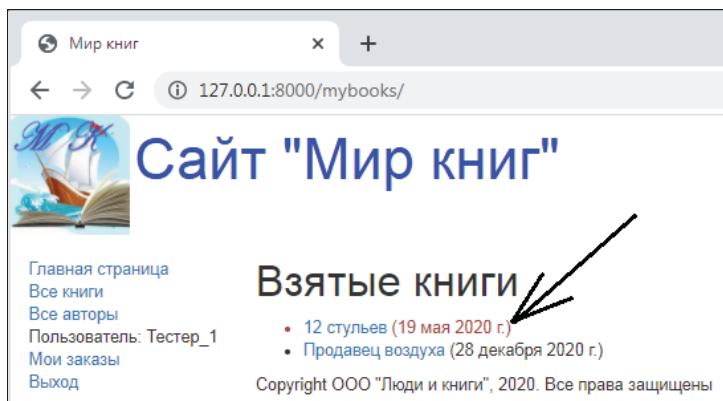


Рис. 10.38. Страница сайта с перечнем книг со статусом **В заказе** пользователя Тестер_1

Если на этой странице нажать кнопку **Выход**, а потом перейти на главную страницу, то ссылка **Мои заказы** исчезнет, т. к. в системе будет находиться уже не авторизованный пользователь.

10.5. Работа с формами

В этом разделе мы рассмотрим возможности работы с HTML-формами и, в частности, познакомимся с самым простым способом построения формы для создания, обновления и удаления экземпляров модели. При этом мы расширим сайт «Мир книг» с тем, чтобы у нас появилась возможность добавлять и редактировать книги и их авторов, используя собственные формы, а не средства приложения администратора.

Форма HTML — это группа из одного или нескольких полей (вернее, их виджетов) на веб-странице, которая используется для ввода данных со стороны пользователей для последующей отправки их на сервер. Формы являются гибким механизмом сбора пользовательских данных, поскольку имеют целый набор виджетов для ввода различных типов данных: текстовые поля, флажки, переключатели, установщики дат, изображения, файлы и т. п. Формы предоставляют безопасный способ взаимодействия пользовательского клиента и сервера, поскольку позволяют отправлять данные в POST-запросах, обеспечивая защиту от межсайтовой подделки запроса (Cross Site Request Forgery, CSRF).

Пока мы не создавали формы, но встречались с ними в административной панели Django — например, использовалистроенную форму для редактирования сведений о книгах, состоящую из нескольких списков выбора и текстовых полей.

Работа с формами может быть достаточно сложной. Разработчикам надо описать форму на языке HTML и проверить ее правильность. Необходимо также проверять на стороне сервера введенные пользователем данные (а возможно, и на стороне клиента). В случае возникновения ошибок надо снова показать пользователю форму и при этом указать на то, что пошло не так. В случае же успеха внести изменения в БД и проинформировать об этом пользователя. Приятно сознавать, что Django берет большую часть этой работы на себя. Он предоставляет фреймворк, который позволяет создать форму и ее поля программно, а затем использовать эти объекты и для генерации непосредственно кода HTML-формы, и для управления процессом корректного взаимодействия пользователя с формой.

В этом разделе вы познакомитесь с несколькими способами создания форм и работы с ними. В том числе вы узнаете, как применение обобщенных классов взаимодействия с формой может значительно уменьшить необходимый объем работы по написанию программного кода. Кроме того, мы дополним возможности нашего сайта «Мир книг», добавив в него страницы для ввода, редактирования, удаления книг и авторов, а также расширим стандартные возможности административной части сайта.

10.5.1. Краткий обзор форм в Django

Рассмотрим простейшую форму HTML, имеющую поле для ввода имени и связанную с этим полем текстовую метку (рис. 10.39).

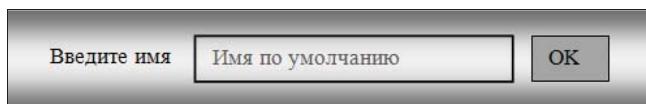


Рис. 10.39. Простейшая форма

Форма описывается на языке HTML в виде набора элементов, расположенных внутри парных тегов `<form>...</form>`. Любая форма содержит как минимум одно поле-тег `input` типа `type="submit"`. Например, форма, представленная на рис. 10.39, будет создана на основе следующего кода (листинг 10.22).

Листинг 10.22

```
<form action="/team_name_url/" method="post">
    <label for="team_name">Введите имя</label>
    <input id="team_name" type="text" name="name_field"
           value="Имя по умолчанию">
    <input type="submit" value="OK">
</form>
```

Здесь у нас только одно поле для ввода имени, но форма может иметь любое количество элементов ввода и связанных с ними текстовых меток. Атрибут элемента `type` определяет, какого типа виджет будет показан в той или иной строке. Атрибуты `name` и `id` используются для однозначной идентификации поля в коде JavaScript, CSS и HTML, в то время как `value` содержит значение для поля, когда оно показывается в первый раз. Текстовая метка добавляется при помощи тега `label` (строка с текстом Введите имя в листинге 10.22) и имеет атрибут `for` со значением идентификатора `id` того поля, с которым связана эта текстовая метка.

Элемент `input` типа `type="submit"` будет по умолчанию показан как кнопка **OK**, нажав на которую пользователь отправляет введенные им данные на сервер (в нашем случае — только значение поля с идентификатором `team_name`). Атрибуты формы `action` и `method` определяют, каким методом будут отправлены данные на сервер (атрибут `method`) и куда (атрибут `action`).

Рассмотрим более подробно два последних элемента:

- `action ==` это ресурс (URL-адрес), куда будут отправлены данные для обработки. Если значение не установлено (т. е. значением поля является пустая строка), тогда данные будут отправлены в представление `view` (функцию, или класс), которое сформировало текущую страницу;

- method — определяет метод, используемый для отправки данных (post или get):
 - метод post должен использоваться тогда, когда необходимо внести изменения в базу данных на сервере. Применение этого метода должно повысить уровень защиты от межсайтовой подделки запроса (CSRF);
 - метод get должен использоваться только для форм, действия с которыми не приводят к изменению базы данных (например, для поисковых запросов). Кроме того, этот метод рекомендуется применять для создания внешних ссылок на ресурсы сайта.

Роль сервера в первую очередь заключается в выводе начального состояния формы (либо содержащей пустые поля, либо с установленными начальными значениями). После того как пользователь нажмет на кнопку **OK**, сервер получит все данные формы и должен провести их проверку на корректность. В том случае, если форма содержит неверные данные, сервер должен снова отобразить пользователю форму, показав при этом поля с правильными данными, а также сообщения о том, что нужно исправить в полях с ошибочными данными. В тот момент, когда сервер получит запрос с правильными данными, он должен выполнить все необходимые действия (например, сохранить данные в БД, вернуть результата поиска, загрузить файлы и т. п.) и в случае необходимости проинформировать пользователя об успешном завершении операции.

Как видите, создание HTML-формы, проверка и возврат данных, перерисовка введенных значений, а также выполнение желаемых действий с правильными данными может потребовать весьма больших усилий, чтобы все это «заработало». Django делает этот процесс намного проще, беря на себя создание некоторых «тяжелых» и повторяющихся фрагментов кода.

10.5.2. Управление формами в Django

Для управления формами Django использует тот же механизм, которые мы изучали в предыдущих разделах, — представление (view):

- получает запрос от пользователя и выполняет необходимые действия, включающие чтение данных из БД;
- генерирует и возвращает пользователю HTML-страницу из шаблона, в который передаются полученные из БД данные.

При использовании форм серверной части системы надо дополнитель но обработать данные, предоставленные пользователем, и в случае возникновения ошибок снова перерисовать исходную страницу. Схема, представленная на рис. 10.40, демонстрирует процесс взаимодействия пользователя с формой в Django.

В соответствии с этой схемой, главными действиями, которые берут на себя формы Django, являются:

- показ формы по умолчанию при первом запросе со стороны пользователя:
 - форма может содержать пустые поля (например, если вы создаете новую запись в базе данных), или эти поля могут иметь начальные значения (напри-

- мер, если вы изменяете запись или хотите заполнить ее каким-либо начальным значением);
- форма в начальный момент является несвязанной, потому что не ассоциируется с какими-либо введенными пользователем данными (хотя и может иметь начальные значения);

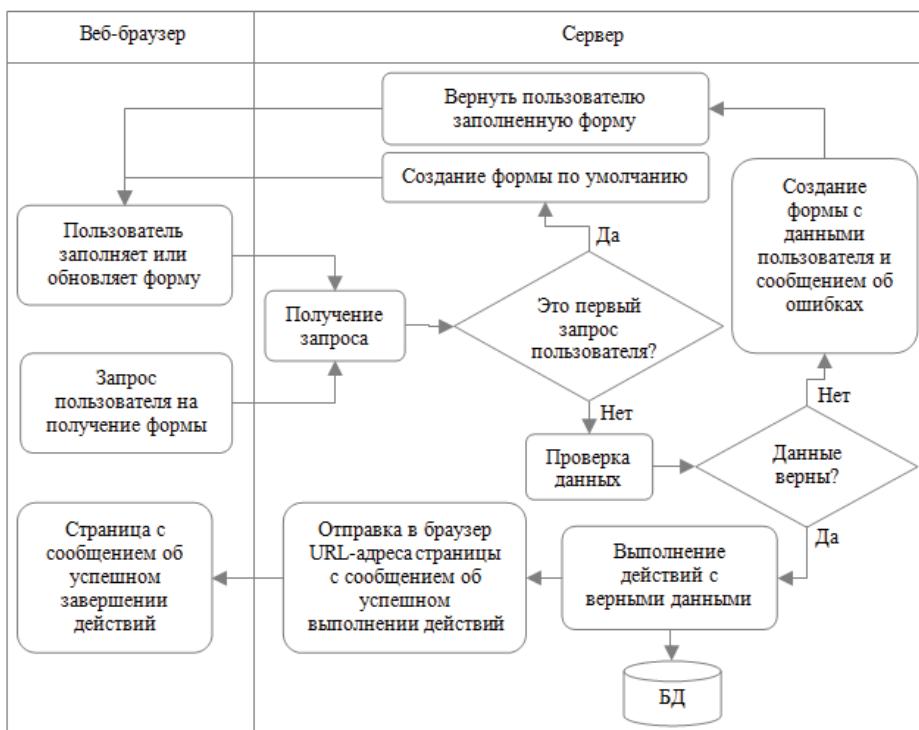


Рис. 10.40. Процесс взаимодействия пользователя с формой Django

- получение данных из формы со стороны клиента и связывание их с формой (классом формы) на стороне сервера:
 - связывание данных с формой означает, что данные, введенные пользователем, а также возможные ошибки при ее обновлении будут относиться именно к этой форме, а не к какой-либо иной;
- очистка и проверка данных:
 - очистка данных — это их проверка на наличие недопустимых символов или вставок в поля ввода (с удалением символов, которые потенциально могут использоваться для отправки вредоносного содержимого на сервер) с последующей конвертацией очищенных данных в подходящие типы данных Python;
 - при проверке данных анализируются значения полей (например, правильность введенных дат, их диапазон и т. п.);

- если какие-либо данные являются неверными, то обновление формы, но на этот раз с уже введенными пользователем данными и сообщениями об ошибках;
- если все данные верны, то исполнение необходимых действий (например, сохранение данных в БД, отправка писем, возврат результата поиска, загрузка файла и т. д.);
- когда все действия успешно завершены, перенаправление пользователя на другую страницу.

Django предоставляет несколько инструментов и приемов, которые помогают пользователю во время выполнения этих задач. Наиболее важным из этих инструментов является класс `Form`, который упрощает генерацию HTML-формы, а также очистку и проверку данных. В следующем разделе мы рассмотрим процесс работы с формами на основе практического примера по созданию страницы, которая позволит пользователям сайта обновлять информацию о книгах.

10.5.3. Форма для ввода и обновления информации об авторах книг на основе класса `Form()`

Создадим форму, с помощью которой пользователи смогут добавлять, редактировать и удалять сведения об авторах книг. Поскольку каждая форма определяется в виде отдельного класса, который расширяет класс `forms.Form`, то создадим в каталоге `World_Books\WebBooks\catalog` файл `forms.py` (рис. 10.41) и запишем в него следующий код (листинг 10.23).

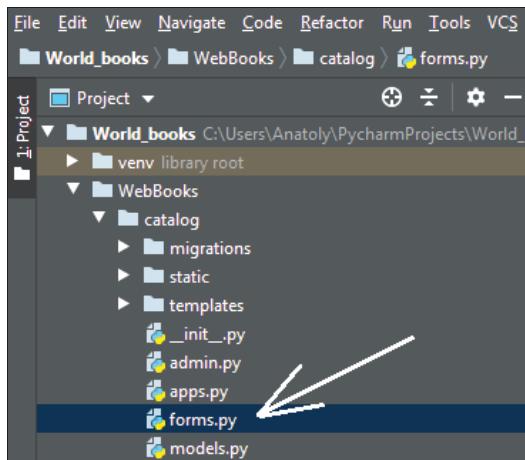


Рис. 10.41. Создание файла `forms.py`

Листинг 10.23

```

from django import forms
from datetime import date

class AuthorsForm(forms.Form):
    first_name = forms.CharField(label="Имя автора")
  
```

```
last_name = forms.CharField(label="Фамилия автора")
date_of_birth = forms.DateField(label="Дата рождения",
    initial=format(date.today()),
    widget=forms.widgets.DateInput(attrs={'type': 'date'}))
date_of_death = forms.DateField(label="Дата смерти",
    initial=format(date.today()),
    widget=forms.widgets.DateInput(attrs={'type': 'date'}))
```

Здесь мы создаем класс формы AuthorsForm(). В нем определены четыре поля для ввода данных. Поля first_name (имя) и last_name (фамилия) имеют тип forms.CharField и будут генерировать текстовое поле input type="text". Поля date_of_birth (дата рождения) и date_of_death (дата смерти) имеют тип forms.DateField и будут генерировать поле для работы с датами input type="date". Первые два поля предназначены для ввода текста, а вторые два поля — для ввода дат. Чтобы даты при первой загрузке формы не были пустыми, мы инициировали для них начальное значение — текущая дата (date.today()). Кроме того, мы для них сменили виджет — теперь это календарь, в котором можно выбрать нужную дату.

Затем в файл views.py добавим следующий код для вызова страницы, где пользователь сможет добавить нового автора, — это будет функция с именем authors_add() (листинг 10.24).

Листинг 10.24

```
from django.http import *
from django.shortcuts import render
from .forms import AuthorsForm

# получение данных из БД и загрузка шаблона authors_add.html
def authors_add(request):
    author = Author.objects.all()
    authorsform = AuthorsForm()
    return render(request, "catalog/authors_add.html",
                  {"form": authorsform, "author": author})
```

Здесь мы определили две переменные: author, в которую загрузили всех авторов из БД, и authorsform, в которую поместили созданную нами ранее форму. Затем этот созданный нами объект "form" (форма) и сведения об авторах в переменных form и author передали в шаблон authors_add.html.

Создадим теперь в каталоге World_Books\WebBooks\catalog\templates\catalog\ файл шаблона authors_add.html (рис. 10.42) и запишем в него следующий код (листинг 10.25).

Листинг 10.25

```
{% extends "base_generic.html" %}
% block content %}
```

```

<h3>Список авторов в БД</h3>





```

В первой строке этого кода мы подключили базовый шаблон `base_generic.html`. Затем в блоке `content` создали два встроенных блока. В первом с помощью цикла `for`

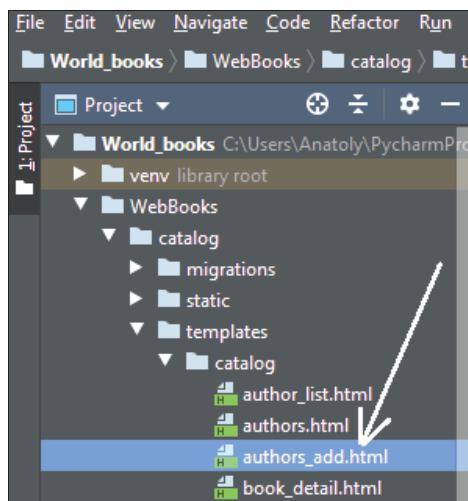


Рис. 10.42. Создание файла `authors_add.html`

вывели всех авторов, которые уже внесены в БД. А также создали две ссылки: `edit1` и `delete`. Это ссылки на страницы для редактирования сведений об авторе и для удаления автора из БД соответственно. Сами страницы мы создадим несколько позже.

Во втором блоке вызывается форма, с помощью которой можно внести в БД нового автора книги. Здесь метод имеет тип POST (`method="POST"`), а это значит, что данные формы будут отправлены на сервер после нажатия кнопки **Сохранить**, и на сервере с этими данными будут выполнены действия `action="/create/"` — а это, по сути, обращение к функции в файле `view.py`, с помощью которой будут обработаны полученные данные.

Теперь нужно создать ссылку для того, чтобы загрузить этот шаблон для ввода в БД сведений об авторах книг. Создадим такую ссылку в меню нашей базовой страницы, для чего откроем файл `base_generic.html` и добавим одну строку в блок боковой панели `sidebar-nav` (в листинге 10.26 это последняя строка, выделенная серым фоном и полужирным шрифтом).

Листинг 10.26

```
<ul class="sidebar-nav">
    <li><a href="{% url 'index' %}">Главная страница</a></li>
    <li><a href="{% url 'books' %}">Все книги</a></li>
    <li><a href="{% url 'authors' %}">Все авторы</a></li>
    <li><a href="{% url 'authors_add' %}">Редактировать авторов</a></li>
```

Остался последний шаг — необходимо по этой ссылке загрузить шаблон `authors_add.html`. Откроем файл `urls.py` и добавим в него следующую строку (листинг 10.19), как показано на рис. 10.43.

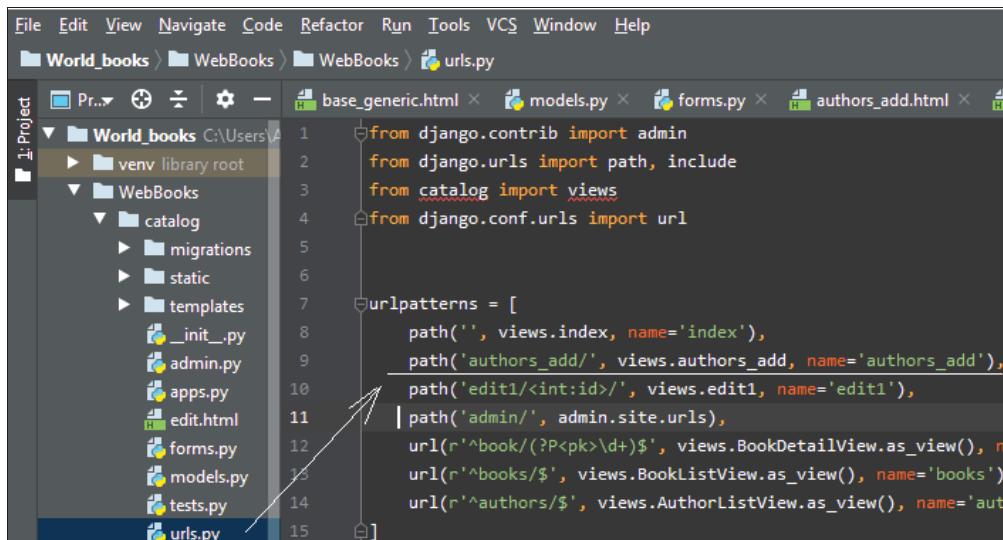


Рис. 10.43. Создание в файле `view.py` маршрута для вызова функции `authors_add`

Листинг 10.27

```
path('authors_add/', views.authors_add, name='authors_add'),
```

Итак, у нас все готово для вывода на экран формы для добавления авторов в БД. Для начала посмотрим, как она выглядит. Запустим наше приложение и загрузим главную страницу сайта. Теперь на ней появилась новая ссылка **Редактировать авторов** (рис. 10.44). Если все было сделано правильно, то при переходе по этой ссылке будет загружена наша форма (рис. 10.45).

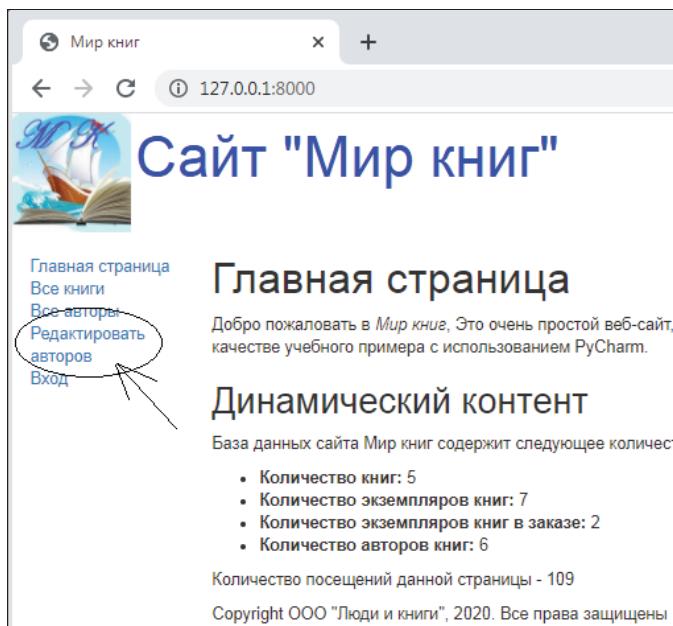


Рис. 10.44. Ссылка на главной странице сайта для вызова формы ввода сведений об авторах

В соответствии с шаблоном нашей формы, в верхней части страницы мы видим перечень авторов, которые уже занесены в БД, а в нижней части формы — блок для добавления в БД авторов книг. Пока еще при нажатии на кнопку **Сохранить** или на ссылки **Изменить** и **Удалить** ничего не произойдет, т. к. мы еще не реализовали соответствующий функционал. Дополним нашу форму, реализовав заложенную в нее функции, а именно: сохранение данных в БД о новых авторах, удаление авторов из БД и редактирования сведений об авторах.

Откройте файл `views.py` и добавьте в него код для сохранения в БД сведений об авторах (обработка нажатия кнопки **Сохранить**) — функцию с именем `create()` (листинг 10.28).

Листинг 10.28

```
# сохранение данных об авторах в БД
def create(request):
    if request.method == "POST":
        author = Author()
        author.first_name = request.POST.get("first_name")
        author.last_name = request.POST.get("last_name")
        author.date_of_birth = request.POST.get("date_of_birth")
        author.date_of_death = request.POST.get("date_of_death")
        author.save()
    return HttpResponseRedirect("/authors_add/")
```

Сайт "Мир книг"

Список авторов в БД

Id	Имя	Фамилия	Изменить	Удалить
1	Александр	Беляев	Изменить	Удалить
2	Александр	Пушкин	Изменить	Удалить
3	Лев	Толстой	Изменить	Удалить
4	Илья	Ильф	Изменить	Удалить
5	Евгений	Петров	Изменить	Удалить
13	Василий	Петровский	Изменить	Удалить

Добавить в БД автора книги

Имя автора:

Фамилия автора:

Дата рождения:

Дата смерти:

Copyright ООО "Люди и книги", 2020. Все права защищены

Рис. 10.45. Форма для ввода сведений об авторах

Здесь сначала делается проверка, что данные отправлены методом POST. Затем создается объект author и соответствующим полям присваиваются те значения, которые отправлены из пользовательской формы. Далее вызывается метод save(), который сохраняет (записывает) данные в БД, и мы возвращаемся на исходную страницу (authors_add).

Продолжим работать с файлом views.py и запишем в него код для удаления из БД сведений об авторах (обработка нажатия ссылки Удалить) — функцию с именем delete() (листинг 10.29).

Листинг 10.29

```
# удаление авторов из БД
def delete(request, id):
    try:
        author = Author.objects.get(id=id)
        author.delete()
        return HttpResponseRedirect("/authors_add/")
    except Author.DoesNotExist:
        return HttpResponseNotFound("<h2>Автор не найден</h2>")
```

В эту функцию передается идентификатор записи в БД (`id`). Чтобы перехватить возможную ошибку при удалении данных, используется блок `try:...except`. Затем в переменную `author` заносится идентификатор удаляемой записи и методом `delete()` эта запись удаляется из БД. После этого делается возврат на исходную страницу (`authors_add`).

И наконец, в файле `views.py` пишем код для изменения в БД сведений об авторах (обработка нажатия ссылки **Изменить**) — функцию с именем `edit1()` (листинг 10.30).

Листинг 10.30

```
# изменение данных в БД
def edit1(request, id):
    author = Author.objects.get(id=id)
    if request.method == "POST":
        author.first_name = request.POST.get("first_name")
        author.last_name = request.POST.get("last_name")
        author.date_of_birth = request.POST.get("date_of_birth")
        author.date_of_death = request.POST.get("date_of_death")
        author.save()
        return HttpResponseRedirect("/authors_add/")
    else:
        return render(request, "edit1.html", {"author": author})
```

Эта функция имеет некоторое сходство с функцией `create()`, однако в ней обрабатываются два метода: POST и GET. Если получен GET-запрос, то пользователь отправляется на страницу `edit1.html`, где сможет отредактировать данные об авторах. Если же получен POST-запрос, то осуществляется запись в БД изменений об авторе книги, после чего пользователь возвращается на исходную страницу (`authors_add`).

Нам не хватает еще шаблона страницы, на которой пользователь сможет отредактировать сведения об авторах. Создадим в каталоге `World_Books\WebBooks\catalog\templates\` файл такого шаблона с именем `edit1.html` (рис. 10.46) и запишем в него следующий код (листиング 10.31).

Листинг 10.31

```
{% extends "base_generic.html" %}

{% block content %}

<h3>Список авторов в БД</h3>
<h4>{{author.first_name}} {{author.last_name}}</h4>

<form method="POST">
    {% csrf_token %}
    <p> <label>Измените имя</label><br>
        <input type="text" name="first_name"
               value="{{author.first_name}}"/>
    </p>
    <p> <label>Измените фамилию</label><br>
        <input type="text" name="last_name"
               value="{{author.last_name}}"/>
    </p>
    <p> <label>Измените дату рождения</label><br>
        <input type="date" name="date_of_birth"
               value="{{author.date_of_birth}}"/>
    </p>
    <p> <label>Измените дату смерти</label><br>
        <input type="date" name="date_of_death"
               value="{{author.date_of_death}}"/>
    </p>
    <input type="submit" value="Сохранить" />
</form>

{% endblock %}
```

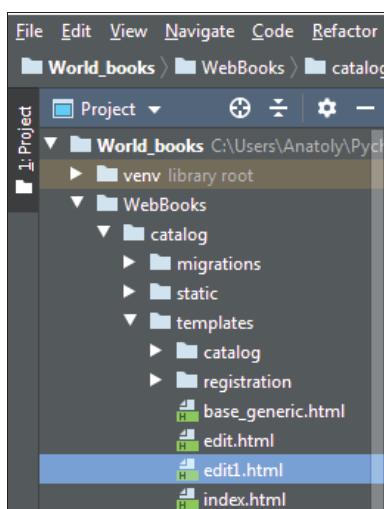


Рис. 10.46. Создание файла edit1.html

Остался последний шаг — внести изменения в файл urls.py, прописав в шаблоне URL-адресов соответствующие маршруты (листинг 10.32 и рис. 10.47).

Листинг 10.32

```
path('edit1/<int:id>', views.edit1, name='edit1'),
path('create/', views.create, name='create'),
path('delete/<int:id>', views.delete, name='delete'),
```

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help
World_books > WebBooks > WebBooks > urls.py
Project urls.py base_generic.html forms.py authors_add.html edit1.html
World_books C:\Users\Anatoly\P...
  venv library root
    WebBooks
      catalog
        migrations
        static
        templates
          catalog
          registration
            base_generic.html
            edit.html
            edit1.html
            index.html
        __init__.py
from django.conf.urls import url
urlpatterns = [
    path('', views.index, name='index'),
    path('authors_add/', views.authors_add, name='authors_add'),
    path('edit1/<int:id>', views.edit1, name='edit1'),
    path('create/', views.create, name='create'),
    path('delete/<int:id>', views.delete, name='delete'),
    path('admin/', admin.site.urls),
    url(r'^book/(\?P<pk>\d+)$', views.BookDetailView.as_view(), name='book'),
    url(r'^books/$', views.BookListView.as_view(), name='books'),
    url(r'^authors/$', views.AuthorListView.as_view(), name='authors')
]
```

Рис. 10.47. Создание маршрутов для функций edit1(), create() и delete() в файле urls.py

Запустим наше приложение, загрузим главную страницу сайта и перейдем по ссылке **Редактировать авторов**. На экране появится форма, с помощью которой можно внести в БД нового автора. Для примера внесем в БД автора Михаил Шолохов. Следует отметить, что при вводе дат будет использоваться встроенный календарь (рис. 10.48). При этом нужную дату пользователь может либо ввести с клавиатуры, либо выбрать из календаря.

По нажатию на кнопку **Сохранить** будет выполнен возврат на предыдущую страницу, где сразу появятся сведения о добавленном авторе (рис. 10.49).

Если на этой странице в строке какого-либо автора нажать на ссылку **Удалить**, то запись с данными об этом авторе будет из БД удалена.

Теперь в строке какого-либо автора нажмем на ссылку **Изменить** — будет загружена страница с возможностью изменения данных об этом авторе (рис. 10.50).

Итак, мы познакомились с формами Django и научились с использованием форм добавлять в БД данные, а также изменять и удалять существующие записи.

Список авторов в БД

Id ----Имя---- Фамилия

1	Александр	Беляев	Изменить , Удалить
2	Александр	Пушкин	Изменить , Удалить
3	Лев	Толстой	Изменить , Удалить
4	Илья	Ильф	Изменить , Удалить
5	Евгений	Петров	Изменить , Удалить
13	Василий	Петровский	Изменить , Удалить

Добавить в БД автора книги

Имя автора:

Фамилия автора:

Дата рождения: Календарь

Дата смерти: Календарь

Copyright ООО "Люди и книги", 2020. Все права защищены

Рис. 10.48. Ввод даты на форме с использованием встроенного календаря

Мир книг

← → ⌛ 127.0.0.1:8000/authors_add/

Сайт "Мир книг"

Главная страница
Все книги
Все авторы
Редактировать авторов
Вход

Список авторов в БД

Id ----Имя---- Фамилия

1	Александр	Беляев	Изменить , Удалить
2	Александр	Пушкин	Изменить , Удалить
3	Лев	Толстой	Изменить , Удалить
4	Илья	Ильф	Изменить , Удалить
5	Евгений	Петров	Изменить , Удалить
13	Василий	Петровский	Изменить , Удалить
14	Михаил	Шолохов	Изменить , Удалить

Добавить в БД автора книги

Имя автора:

Фамилия автора:

Дата рождения: Календарь

Дата смерти: Календарь

Copyright ООО "Люди и книги", 2020. Все права защищены

Рис. 10.49. Исходная форма после добавления в БД нового автора

Мир книг

127.0.0.1:8000/edit1/13/

Сайт "Мир книг"

Главная страница Все книги Все авторы Редактировать авторов Вход

Текущие сведения об авторе
Василий Петровский (3 августа 2020 г. - 20 августа 2020 г.)

Изменить сведения об авторе книги

Измените имя

Измените фамилию

Измените дату рождения

Измените дату смерти

Copyright ООО "Люди и книги", 2020. Все права защищены

Рис. 10.50. Страница для редактирования сведений об авторе книги

10.5.4. Форма для ввода и обновления информации о книгах на основе класса *ModelForm()*

Использование класса формы `Form()`, описанное в предыдущем разделе, представляет собой довольно гибкий способ создания форм любой структуры в связке с любой моделью или моделями.

Тем не менее если нужна форма для отображения полей одиночной модели, то это можно сделать проще. Ведь модель уже содержит большую часть информации, которая необходима для построения формы: сами поля, текстовые метки, текст «помощи» и пр. И чтобы не дублировать для создаваемой формы информацию из модели, лучше воспользоваться классом `ModelForm()`, который позволяет создавать формы непосредственно из модели данных. Класс `ModelForm()` может использоваться в представлениях (*views*) точно так же, как и «классический» класс формы `Form()`.

Итоговая форма при использовании класса `ModelForm()` будет содержать те же поля, что подключенная модель данных. Все, что необходимо сделать внутри создаваемого класса, — это добавить класс `Meta` и связать его с моделью. А затем в поле `fields` указать поля модели данных, которые необходимо включить в форму.

В качестве примера создадим на основе класса `ModelForm()` форму для редактирования сведений о книгах и начнем с создания класса `BookModelForm()`. Для этого откроем файл `forms.py` и добавим в него следующий код (листинг 10.33).

Листинг 10.33

```
from django.forms import ModelForm
from .models import Book

class BookModelForm(ModelForm):
    class Meta:
        model = Book
        fields = ['title', 'genre', 'language', 'author', 'summary', 'isbn']
```

Здесь мы выполняем импорт базового класса `ModelForm()` и модели данных `Book`. Затем объявляем класс `Meta` с указанием имени модели, на базе которой будет строиться форма (`Book`), и в атрибуте `fields` указываем те поля, которые из этой модели будут отображаться на форме.

Для каждого поля модели вы можете определить значения меток, виджетов, текстов сообщений об ошибках и т. д., если они не были заданы в модели. Их так же можно переопределить в классе `Meta` при помощи словаря, содержащего поле, которое надо изменить, и его новое значение. В качестве примера приведем синтаксис кода, который мы могли бы использовать для изменения параметров поля для ввода аннотации к книге (листинг 10.34).

Листинг 10.34

```
fields = ['summary',]
labels = { 'summary': _('Аннотация'), }
help_texts = { 'summary': _('Не более 1000 символов'), }
```

Алгоритм управления формой, который мы использовали в нашей функции представления в предыдущем разделе, является примером достаточно общего подхода к работе с формами. Однако Django старается упростить большую часть такой работы путем применения обобщенных классов для создания, редактирования и удаления записей в БД на основе моделей данных.

В этом разделе мы воспользуемся обобщенными классами создания страниц, через которые можно будет создавать, редактировать и удалять записей о книгах (`Book`). Подобную страницу мы видели в административной части сайта.

Откройте файл `WebBooks\catalog\views.py` и добавьте в него следующий код (листинг 10.35).

Листинг 10.35

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.urls import reverse_lazy
from .models import Book

class BookCreate(CreateView):
    model = Book
```

```
fields = '__all__'
success_url = reverse_lazy('books')

class BookUpdate(UpdateView):
    model = Book
    fields = '__all__'
    success_url = reverse_lazy('books')

class BookDelete(DeleteView):
    model = Book
    success_url = reverse_lazy('books')
```

Как можно видеть, на основе базовых классов мы создали три собственных класса: для ввода в БД новой книги (`BookCreate()`), для обновления сведений о книге (`BookUpdate()`) и для удаления книги из БД (`BookDelete()`) и каждый класс связали с моделью данных (`Book`).

Для классов `BookCreate()` и `BookUpdate()` мы определили показываемые на форме поля. Здесь можно было применить тот же синтаксис, что и для класса `ModelForm()` (указание всех необходимых полей), но мы подключили все поля при помощи инструкции `fields = '__all__'`. При этом, если надо, можно воспользоваться полем `exclude` (вместо поля `fields`), чтобы определить поля модели, которые не нужно включать в форму. Здесь также можно указать начальные значения для каждого поля, применяя словарь пар «имя_поля/значение».

По умолчанию наши классы в представлении (`view`) перенаправляют пользователя на страницу, указанную в параметре `success_url`. Эта страница, на которую будет перенаправлен пользователь в случае успешного завершения операции, покажет ему созданные или отредактированные данные. Кстати, при помощи параметра `success_url` можно задать и альтернативное перенаправление.

Классу `BookDelete()` не нужно отображать поля, так что их не требуется и декларировать. Тем не менее все равно указать для него параметр `success_url` надо, потому что для Django не очевидно, что делать после успешного выполнения операции удаления записи. Здесь с параметром `success_url` мы используем функцию `reverse_lazy()` — для перехода на страницу списка книг после удаления одной из них. Функция `reverse_lazy()` — это более «ленивая» версия функции `reverse()`.

На следующем шаге мы создадим соответствующие шаблоны. Наши представления (`views`) `BookCreate` и `BookUpdate` по умолчанию будут использовать шаблоны с именем `model_name_form.html`. Сuffixикс в этом имени можно поменять на какой-нибудь другой при помощи поля `template_name_suffix` вашего представления (`view`). Для этого вы можете применить, например, следующий синтаксис: `template_name_suffix = '_other_suffix'`. Однако мы не станем это делать, а создадим файл шаблона HTML-страницы с тем именем, которое в Django используется по умолчанию.

Итак, создайте в каталоге `WebBooks\catalog\templates\catalog` файл `book_form.html` и внесите в него следующий код (листинг 10.36).

Листинг 10.36

```
{% extends "base_generic.html" %}

{% block content %}

<form action="" method="post">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" value="Сохранить" />

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

Код этого шаблона напоминает наши предыдущие формы и прорисовку полей при помощи таблицы. Заметьте, что мы снова используем выражение `{% csrf_token %}`. Кроме того, следует отметить, что код шаблона для класса `ModelForm()` короче и проще, чем код шаблона для класса `Form()`.

Представление (`view`) `BookDelete` по умолчанию будет искать шаблон с именем формата `model_name_confirm_delete.html`. Поэтому создайте в каталоге `WebBooks\catalog\templates\catalog\` файл шаблона с именем `book_confirm_delete.html` и внесите в него следующий код (листинг 10.37).

Листинг 10.37

```
{% extends "base_generic.html" %}

{% block content %}

<h1>Удаление книги</h1>
<p> Вы уверены, что хотите удалить книгу: {{ book }}?</p>

<form action="" method="POST">
    {% csrf_token %}
    <input type="submit" action="" value="Да, удалить" />
</form>

{% endblock %}
```

В заключение откроем файл конфигураций URL-адресов (`WebBooks\catalog\urls.py`) и добавим в него следующий код (листинг 10.38).

Листинг 10.38

```
urlpatterns += [
    url(r'^book/create/$', views.BookCreate.as_view(),
        name='book_create'),
```

```
url(r'^book/update/(?P<pk>\d+)$', views.BookUpdate.as_view(),
     name='book_update'),
url(r'^book/delete/(?P<pk>\d+)$', views.BookDelete.as_view(),
     name='book_delete'),
]
```

Здесь нет ничего нового! Как можно видеть, представления (`view`) являются классами, поэтому они должны вызываться через метод `.as_view()`. Шаблоны URL-адресов для каждого случая также должны быть вам понятны. Когда поступает запрос `book/create/` (ввести сведения о новой книге), то мы просто обращаемся к представлению `views.BookCreate`. Когда же поступает запрос на обновление сведений о книге, которая уже занесена в БД (`views.BookUpdate`), или об удалении конкретной книги (`views.BookDelete`), то мы обязаны использовать первичный ключ этой книги в БД (`pk`), чтобы выполнить соответствующую операцию именно с этой книгой.

Страницы для создания, обновления и удаления книги теперь готовы к использованию, осталось создать ссылки на эти страницы из интерфейса сайта. Обращение к странице ввода в БД информации о новой книге сделаем из главного меню сайта. Для этого откроем базовый шаблон (файл `base_generic.html`) и в блок боковой панели `sidebar-nav` добавим следующий код (в листинге 10.39 это последняя строка, выделенная серым фоном и полужирным шрифтом).

Листинг 10.39

```
{% block sidebar %}


- <a href="{% url 'index' %}">Главная страница</a></li>
- <a href="{% url 'books' %}">Все книги</a></li>
- <a href="{% url 'authors' %}">Все авторы</a></li>
- <a href="{% url 'authors_add' %}">Редактировать
авторов</a></li>
- <a href="{% url 'book_create' %}">Добавить книгу</a></li>

```

Если теперь запустить наш сайт, то на его первой странице появится ссылка **Добавить книгу** (рис. 10.51). Щелкните мышью на этой ссылке, и откроется созданная нами форма. Введите в эту форму сведения о новой книге — например: «Евгений Онегин» (рис. 10.52), нажмите на кнопку **Сохранить** и перейдите в открывшемся окне по ссылке **Все книги** — наша вновь введенная книга появилась в БД (рис. 10.53).

Теперь нам необходимо создать ссылки на страницы, где можно либо отредактировать сведения о книге, либо удалить книгу из БД. Поскольку эти действия мы должны выполнить для конкретной книги, то эти ссылки нужно вставить в шаблон страницы **Список книг в БД**. Шаблон для этой страницы у нас носит имя `book_list.html`. Открываем этот шаблон и добавляем в него следующий код (в листинге 10.40 это строки, выделенные серым фоном и полужирным шрифтом).

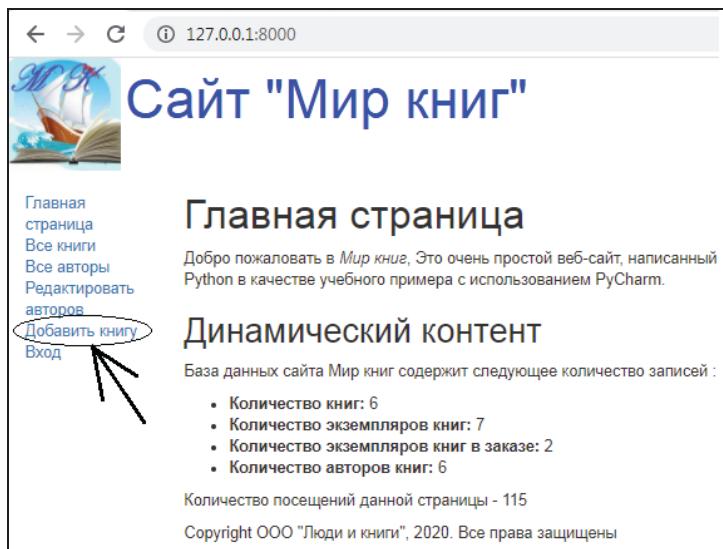


Рис. 10.51. Главная страница сайта со ссылкой на страницу добавления новой книги

The screenshot shows a form for adding a new book at the URL 127.0.0.1:8000/book/create/. The sidebar on the left is identical to the one in the previous screenshot. The main form fields are: Название книги: "Евгений Онегин" (input field), Жанр книги: "Поззия" (dropdown menu), Язык книги: "Русский" (dropdown menu), Автор книги: "Пушкин" (dropdown menu with options Беляев, Пушкин, Толстой, Ильф), Аннотация книги: "А.С. Пушкин, не дожидаясь завершения своего произведения, печатал его по главам. Читатели, сопереживая героям, с нетерпением ждали появления очередной главы. Критика бурлила. "Евгений Онегин" стал центральным событием в литературной и общественной жизни пушкинской поры. И, как показало время, - центральным событием в русской культуре в целом." (text area), ISBN книги: "9785699984589" (input field), and Сохранить (button). The footer copyright notice is "Copyright ООО \"Люди и книги\", 2020. Все права защищены".

Рис. 10.52. Форма для добавления в БД новой книги

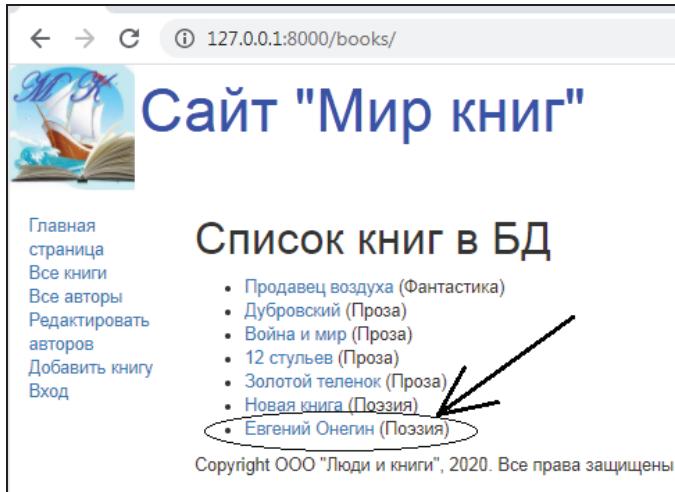


Рис. 10.53. Итог работы формы добавления в БД новой книги

Листинг 10.40

```
% extends "base_generic.html" %

{% block content %}
<h1>Список книг в БД</h1>
{% if book_list %}
<ul>
    {% for book in book_list %}
<li>
    <a href="{{ book.get_absolute_url }}>{{ book.title }}</a>
    {{book.genre}}
    <a href="/book/update/{{book.id}}">Изменить</a> ,
    <a href="/book/delete/{{book.id}}">Удалить</a>
</li>
    {% endfor %}
</ul>
{% else %}
<p>В базе данных нет книг</p>
{% endif %}
{% endblock %}
```

Если теперь на главной странице сайта щелкнуть мышью на ссылке **Все книги**, то на странице **Список книг в БД** рядом с названием книги появятся две ссылки: **Изменить** и **Удалить** (рис. 10.54).

Щелкните мышью на ссылке **Изменить** рядом с какой-либо из книг — откроется форма, в которую из БД автоматически будет подгружена вся информация об этой книге (рис. 10.55). Информацию о книге на этой странице можно изменить и сохранить в БД обновленные данные, нажав на кнопку **Сохранить**.

The screenshot shows a web browser window with the URL 127.0.0.1:8000/books/. The title of the page is "Сайт \"Мир книг\"". On the left, there is a sidebar with links: Главная страница, Все книги, Все авторы, Редактировать авторов, Добавить книгу, Вход. Below the sidebar, the main content area has a heading "Список книг в БД" with a downward arrow pointing to it. A list of books is shown with edit and delete links: Продавец воздуха (Фантастика), Изменить, Удалить; Дубровский (Проза), Изменить, Удалить; Война и мир (Проза), Изменить, Удалить; 12 стульев (Проза), Изменить, Удалить; Золотой теленок (Проза), Изменить, Удалить; Новая книга (Поззия), Изменить, Удалить; Евгений Онегин (Поззия), Изменить, Удалить. At the bottom of the page, a copyright notice reads: Copyright ООО "Люди и книги", 2020. Все права защищены.

Рис. 10.54. Ссылки на страницы изменения и удаления информации о книге

The screenshot shows a web browser window with the URL 127.0.0.1:8000/book/update/5. The title of the page is "Сайт \"Мир книг\"". On the left, there is a sidebar with links: Главная страница, Все книги, Все авторы, Редактировать авторов, Добавить книгу, Вход. The main content area contains fields for editing book information: Название книги: "Золотой теленок" (input field), Жанр книги: "Проза" (dropdown menu), Язык книги: "Русский" (dropdown menu), Автор книги: dropdown menu showing Ильф, Петров, Петровский, Аннотация книги: large text area containing a summary of the book, ISBN книги: "978517098229" (input field), and a note that it must contain 13 symbols. At the bottom, there is a "Сохранить" button and a copyright notice: Copyright ООО "Люди и книги", 2020. Все права защищены.

Рис. 10.55. Страница для изменения информации о книге

Щелкните теперь рядом с какой-либо из книг на ссылке **Удалить** — откроется форма, в которой будут присутствовать название выбранной книги и кнопка с надписью **Да, удалить** (рис. 10.56). Если здесь щелкнуть мышью на кнопке **Да, удалить**, то сведения о книге будут удалены из БД, и пользователь вернется на страницу со списком книг.

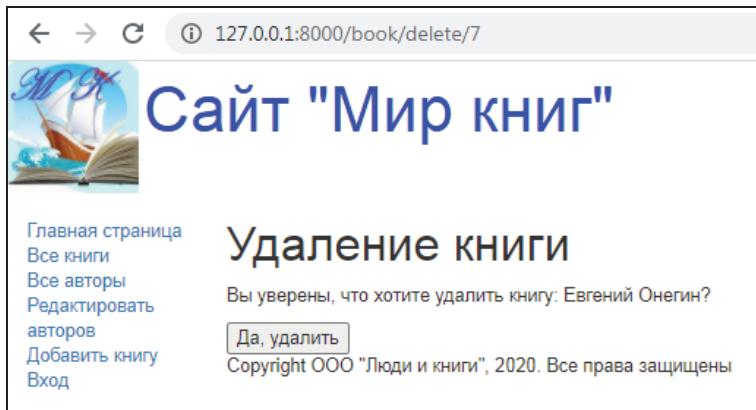


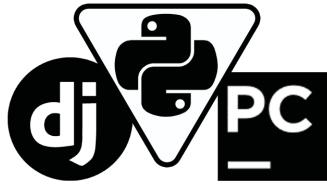
Рис. 10.56. Страница для удаления из БД информации о книге

10.6. Краткие итоги

Создание и управление формами — это довольно сложная и трудоемкая работа. Однако Django делает этот процесс намного проще за счет встроенных в него готовых механизмов создания, прорисовки и проверки форм. Более того, Django предоставляет обобщенные классы редактирования форм, которые могут выполнять практически любую работу по созданию, редактированию и удалению записей в БД, связанных с отдельной моделью данных. При этом автоматически генерируются и исполняются SQL-запросы к базе данных.

Сайты в процессе развития и разработки становятся все сложнее. С ростом количества страниц сайта возрастает количество внутренних взаимодействий между компонентами, в результате чего внесение изменений в одну часть приложения может нарушить работу других его частей. Исходя из этого, разработчику приходится постоянно вручную тестировать и проверять работу всех элементов сайта. Одним из способов, который позволяет проверять корректность внесенных изменений, является автоматическое тестирование приложения. В Django реализована возможность автоматизации тестирования создаваемого сайта при помощи специального фреймворка. К сожалению, из-за ограниченного объема книги мы не сможем рассмотреть автоматическое тестирование приложений. Однако вы можете ознакомиться с этим процессом в оригинальной документации на Django.

Итак, мы создали достаточно простой сайт, который позволяет пользователям через веб-интерфейс работать с удаленной базой данных. Теперь необходимо развернуть наш сайт в сети Интернет. Решению этой задачи посвящена следующая глава.



ГЛАВА 11

Публикация сайта в сети Интернет

В предыдущих главах были созданы формы сайта «Мир книг», его общий интерфейс, главная страница, а также страницы для ввода, редактирования и удаления данных из БД со стороны удаленного пользователя.

Теперь, когда мы создали и протестируем свой сайт, необходимо развернуть его на публичном веб-сервере, чтобы он стал доступен через Интернет удаленным пользователям. Эта глава дает общее представление о том, каким образом подойти к поиску хостинга для размещения сайта и что нужно сделать, чтобы подготовить свой сайт к публикации.

Мы рассмотрим здесь следующие вопросы:

- как подготовить инфраструктуру сайта перед его публикацией в сети Интернет;
- как выбрать хостинг-провайдера для своего сайта;
- какие настройки в проекте нужно сделать перед публикацией сайта;
- как создать и для чего нужен репозиторий своего приложения на сайте GitHub;
- почему для развертывания в сети Интернет желательно использовать веб-сервер Gunicorn;
- как развернуть сайт на интернет-ресурсе Heroku.

11.1. Подготовка инфраструктуры сайта перед публикацией в сети Интернет

Когда разработка сайта завершена, его необходимо разместить на удаленном сервере для публичного доступа. До настоящего момента мы работали в локальном рабочем окружении. При этом использовали веб-сервер Django на собственном компьютере (или в локальной сети) и запускали сайт с небезопасными настройками своей рабочей среды. Перед тем как разместить сайт на публичном сервере, необходимо:

- сделать несколько изменений в настройках проекта;
- выбрать (настроить) окружение для хостинга приложения Django;

- выбрать (настроить) окружение для размещения статических файлов;
- в целях обслуживания сайта настроить инфраструктуру для его развертывания.

В этом разделе сделан обзор мероприятий по выбору хостинга, описан процесс подготовки сайта к публичному размещению, а также приведен практический пример размещения сайта «Мир книг» на облачном сервисе Heroku.

11.1.1. Окружение развертывания сайта в сети Интернет

Окружение развертывания — это среда, которую предоставляет сервер, на котором вы будете размещать свой веб-сайт для публичного доступа. Такое окружение включает в себя следующие элементы:

- технические средства сервера, на которых будет работать сайт;
- операционную систему на сервере (Linux, Windows);
- языки программирования времени выполнения (скриптовые) и библиотеки, которые использует ваш сайт;
- веб-сервер, используемый для обслуживания страниц и другого контента (Gunicorn, Nginx, Apache);
- сервер приложений, который передает динамические запросы между сайтом Django и веб-сервером;
- базу данных, которую будет использовать сайт для хранения информации.

Сервер может быть и вашим собственным (с подключением к Интернету по скоростному каналу), но более общим подходом является применение облачных решений. При этом ваш код будет запускаться на удаленном компьютере (возможно и виртуальном) на серверной площадке data-центра. Хостинг-провайдер обычно предоставляет доступ к своим компьютерным ресурсам (процессору, оперативной памяти, памяти на жестких носителях, базе данных и т. д.) и соединение с Интернетом за некоторую плату.

Такой тип удаленного доступа к вычислительным средствам называется «Инфраструктура как Сервис» (Infrastructure as a Service, или IaaS). Множество IaaS-поставщиков предлагают услуги по предустановке какой-либо операционной системы, на которую вы сможете установить необходимые для вашего рабочего окружения компоненты. Другие поставщики предлагают вам выбрать уже готовые полноценные рабочие окружения, возможно включающие в себя Django и настроенный веб-сервер.

Конечно, проще развернуть сайт в уже готовом окружении. Это позволит сделать настройку вашего веб-сайта достаточно простой задачей с минимальным изменением конфигурации. Однако в готовом окружении может быть либо недостаточное количество доступных опций, либо они будут использовать устаревшую операционную систему. Поэтому в некоторых случаях более предпочтительно установить необходимые вам компоненты самостоятельно.

Ряд провайдеров поддерживают Django как часть своего предложения «Платформа как Сервис» (Platform as a Service, или PaaS). При этом виде хостинга вам не нужно беспокоиться о большей части окружения (веб-сервер, сервер приложений, СУБД и т. п.), т. к. сама платформа берет это на себя (включая все, касающееся развития вашего приложения). В таком случае развертывание приложения является довольно простой задачей — необходимо сконцентрироваться только на своем приложении, а не на внешней инфраструктуре.

Некоторые разработчики выбирают более гибкое решение, предоставляемое IaaS, в то время как другие предпочитают иметь наименьшие накладные расходы и простое масштабирование, предоставляемое PaaS. Когда вы только начинаете работать с веб-приложениями, система типа PaaS представляется более удобной, и именно ее мы будем использовать.

11.1.2. Выбор хостинг-провайдера

Существует более 100 хорошо известных хостинг-провайдеров, которые либо активно поддерживают Django, либо работают с ним (их список можно увидеть по ссылке: <https://djangofriendly.com/index.html>). Эти поставщики предоставляют различные типы окружений (IaaS, PaaS) и различные уровни доступа к вычислительным и сетевым ресурсам за разную цену.

Отметим несколько моментов, на которые надо обратить внимание при выборе хостинга:

- насколько ваш сайт является требовательным к вычислительным ресурсам;
- уровень поддержки горизонтального масштабирования (добавление большего количества компьютеров) и вертикального масштабирования (переход на более мощные технические средства);
- где находятся data-центры и, следовательно, откуда будет получен более быстрый доступ к сетевым ресурсам;
- время непрерывной работы хостинга, а также время и количество простоя;
- инструменты, которые предоставляются для управления сайтом (SFTP и FTP), простота и безопасность их использования;
- наличие встроенных фреймворков для мониторинга вашего сервера;
- наличие ограничений (блокировка электронной почты, количество часов «живого» времени за определенную плату, количество места для данных и т. п.);
- наличие преимуществ (провайдеры могут предложить бесплатные доменные имена и поддержку сертификатов SSL, которые часто необходимо оплачивать);
- что будет с сайтом после окончания времени бесплатного (пробного) использования хостинга, какова стоимость платного обслуживания сайта и т. д.?

Существует достаточно много компаний, которые предоставляют пробные бесплатные тарифы типа Evaluation (для пробы), Developer (разработка) или Hobbyist (хобби). Такие сервисы могут быть доступны лишь в течение определенного пе-

риода времени. Тем не менее они являются отличным решением для тестирования сайтов с небольшим трафиком в реальном окружении, а также могут в случае необходимости предоставлять простой доступ к платным ресурсам. Наиболее популярными провайдерами являются Heroku, Python Anywhere, Amazon Web Services, Microsoft Azure и ряд других. Многие провайдеры предлагают базовый тариф (Basic), предоставляющий достаточный уровень вычислительной мощности с небольшим количеством ограничений. Примерами таких провайдеров могут служить Digital Ocean и Python Anywhere. Необходимо помнить, что цена не является единственным критерием выбора. Если ваш сайт будет успешен, то может так случиться, что самым важным элементом при выборе услуг хостинга станет возможность масштабирования.

11.2. Подготовка веб-сайта к публикации

Основа нашего сайта была создана при помощи инструментов Django Admin и `manage.py`, которые настроены таким образом, чтобы сделать разработку сайта как можно проще. Однако многие настройки, которые хранятся в файле проекта `settings.py`, перед публикацией сайта должны быть изменены. Это необходимо либо для повышения безопасности, либо для улучшения производительности. Общепринятым подходом является создание отдельного файла `settings.py` для публикации, который импортирует важные настройки из внешних файлов или из переменных окружения. Доступ к этому файлу должен быть ограничен, даже если остальная часть исходного кода размещена в публичном хранилище.

Критически важными являются следующие настройки файла `settings.py`:

- `DEBUG` — при развертывании сайта должна быть установлен `False` (`DEBUG = False`). При этом прекратится вывод важной отладочной информации;
- `SECRET_KEY` — это большое случайное число, применяемое для защиты от CSRF (важно, чтобы ключ не указывался в исходном коде и/или не запрашивался с другого сервера). Django рекомендует размещать значение ключа либо в переменной окружения, либо в файле с доступом только на чтение.

Указанную настройку `SECRET_KEY` можно установить с помощью следующего кода (листинг 11.1).

Листинг 11.1

```
# Чтение SECRET_KEY из переменной окружения
import os
SECRET_KEY = os.environ['SECRET_KEY']
#ИЛИ чтение ключа из файла
with open('/etc/secret_key.txt') as f:
    SECRET_KEY = f.read().strip()
```

Давайте изменим приложение таким образом, чтобы читать `SECRET_KEY` и `DEBUG` из переменных окружения, если те определены, или в противном случае использовать для них значения по умолчанию.

Откройте файл `\WebBooks\settings.py`, закомментируйте значение `SECRET_KEY` и добавьте новые строки, выделенные в листинге 11.2 серым фоном и полужирным шрифтом.

Листинг 11.2

```
# SECURITY WARNING: keep the secret key used in production secret!
# SECRET_KEY = 'dhp40_!05cp071e9pd5e5+3_90fev*vq-obx^3^hv8cx0=1#!k'
```

```
import os
SECRET_KEY = os.environ.get('DJANGO_SECRET_KEY',
'cg#p$g+j9tax!#a3cup@1$8obt2_+&k3qt+pmu)5%asj6yjpkg!')
```

В процессе разработки сайта никаких переменных окружения определено не было, таким образом, будут применены значения по умолчанию (не имеет значения, какой ключ вы использовали в процессе разработки, поскольку при развертывании проекта вы будете использовать другой ключ).

Затем закомментируйте строку с настройкой `DEBUG`, а под ней добавьте новую (выделена в листинге 11.3 серым фоном и полужирным шрифтом).

Листинг 11.3

```
# SECURITY WARNING: don't run with debug turned on in production!
# DEBUG = True
DEBUG = bool(os.environ.get('DJANGO_DEBUG', True))
```

Значение `DEBUG` будет `True` по умолчанию и станет `False` в том случае, если переменная окружения `DJANGO_DEBUG` будет проинициализирована пустой строкой, т. е. когда `DJANGO_DEBUG = ''`.

ПРИМЕЧАНИЕ

Было бы более понятно, если бы мы могли просто установить для `DJANGO_DEBUG` значение `True` или `False` напрямую, а не использовать «любую строку» или «пустую строку» (соответственно). К сожалению, значения переменных среды хранятся как строки Python, и единственной строкой, которая распознается, как `False`, является пустая строка (например: `bool('') == False`).

С полным перечнем всех настроек для развертывания сайта можно ознакомиться по ссылке: <https://docs.djangoproject.com/en/1.10/howto/deployment/checklist/>.

Кроме того, вы можете получить список необходимых настроек, выполнив в терминале команду:

```
python manage.py check --deploy
```

11.3. Пример размещения веб-сайта на сервисе Heroku

Одним из самых известных и популярных облачных сервисов, который предоставляет услуги PaaS, является Heroku. Первоначально он поддерживал только приложения Ruby, но в настоящее время на этом сервисе можно размещать веб-приложения, созданные в различных средах программирования, включая Django.

Сервис Heroku рекомендуется использовать по следующим причинам:

- у Heroku есть возможность бесплатного размещения сайта (хотя и с некоторыми ограничениями);
- как PaaS-сервис, Heroku имеет большой набор веб-инфраструктуры для различных клиентов (индивидуальных разработчиков, небольших бизнес-компаний, критически нагруженных приложений, крупномасштабных компаний). Это в значительной степени снимает нагрузку с разработчиков, потому что в таких условиях у них отсутствует необходимость беспокоиться о серверах и системном программном обеспечении, балансирах нагрузки, обратных прокси и прочей веб-инфраструктуре (все это Heroku предоставляет единым пакетом).

Тем не менее у этого сервиса есть некоторые ограничения, которые, однако, не влияют на ваше приложение. Например:

- Heroku предоставляет только недолговечное хранилище, поэтому загруженные пользователем файлы на нем нельзя безопасно хранить;
- бесплатный уровень склонен «засыпать», и ваше веб-приложение станет неактивным, если в течение получаса на него не будут поступать запросы (после этого сайт может «просыпаться» на несколько секунд, чтобы ответить на запрос пользователя);
- бесплатный уровень ограничивает время, в течение которого ваш сайт будет работать (количество часов в месяц, не включая время, когда сайт «спит»). Это нормально для сайта с низким уровнем загрузки («демонстрационным»), но не подходит, если требуется 100% времени безотказной работы.

Сервис Heroku идеально подходит для апробации и тестирования функций разработанного сайта, его использование также упрощает настройку и масштабирование при переходе с бесплатного использования на платные тарифы.

Heroku запускает сайты Django внутри изолированных контейнеров, носящих название «dynos». Это, по сути, виртуальные Linux-контейнеры, которые предоставляют необходимое окружение для вашего приложения. Такие dynos полностью изолированы и имеют *эфемерную* файловую систему («короткоживущая» файловая система, которая полностью очищается и обновляется каждый раз, когда dynos перезапускается). Единственной сущностью, которую предоставляет dynos по умолчанию, является приложение по конфигурированию переменных. Heroku внутри себя использует «балансировщик» загрузки (для распределения веб-трафика среди

всех dynos). Поскольку контейнеры dynos изолированы друг от друга, то Heroku может масштабировать приложение горизонтально, просто добавляя больше dynos.

На эфемерную файловую систему вы не можете напрямую установить необходимые для вашего приложения сервисы (т. е. базы данных, очереди, системы кэширования, хранения, сервисы электронной почты и пр.). Взамен этого Heroku предоставляет сервисы, доступные как независимые «дополнения (add-ons), либо от самого Heroku, либо от сторонних разработчиков. В тот момент, когда ваше приложение запускается в системе, dynos получает доступ к таким сервисам, используя информацию, содержащуюся в переменных настройки вашего приложения.

Для того чтобы выполнять ваше приложение в Heroku, необходимо иметь возможность сконфигурировать под вас соответствующее окружение и зависимости. В случае приложений Django нужно предоставить Heroku соответствующую информацию именно о вашем приложении в нескольких текстовых файлах:

- runtime.txt — язык программирования и его версия;
- requirements.txt — необходимые для Python компоненты, включая Django;
- Procfile — список процессов, которые будут выполнены при старте веб-приложения. Для Django это обычно сервер веб-приложений Gunicorn;
- wsgi.py — файл с конфигурацией для вызова вашего приложения Django в окружении Heroku.

Разработчики взаимодействуют с Heroku при помощи специального клиентского приложения (терминала), который похож на bash-скрипт UNIX. Это приложение позволяет загружать код, находящийся в Git-репозитории, контролировать выполняемые процессы, смотреть «логи» (журналы), устанавливать конфигурационные переменные и многое другое.

Для того чтобы заставить приложение работать с Heroku, необходимо разместить разработанное веб-приложение в Git-репозитории, добавить упомянутые ранее файлы, подключить дополнение (add-on) базы данных и выполнить настройки для правильной работы со статическими файлами.

Когда мы выполним все, что необходимо для нашего сайта, мы можем создать аккаунт Heroku, получить доступ к клиенту Heroku и использовать его для установки нашего веб-сайта.

ПРИМЕЧАНИЕ

Рекомендации, приведенные далее, соответствуют процессу работы с Heroku на момент подготовки книги. Если Heroku значительно изменит этот процесс, вы можете воспользоваться соответствующим описанием по адресу: <https://devcenter.heroku.com/articles/getting-started-with-python>.

11.3.1. Создание репозитория приложения на GitHub

Сервис Heroku тесно интегрирован с системой управления версиями исходного кода приложения Git, используя ее для загрузки и синхронизации любых изменений, которые вы вносите в «живую» систему. Он делает это, добавляя новый «удален-

ный» репозиторий Heroku с именем `heroku`. Во время разработки вы используете GitHub для хранения изменений в вашем «master»-репозитории. Когда вы хотите развернуть свой сайт, то переносите все изменения проекта из репозитория GitHub в репозиторий Heroku.

ПРИМЕЧАНИЕ

GitHub (<https://github.com/>) — это платформа для разработчиков приложений. Вы можете размещать там и просматривать программный код, управлять проектами и создавать программное обеспечение вместе с 50 миллионами разработчиков со всего мира. Если вы обладаете большим опытом разработки программного обеспечения, используете GitHub, и у вас уже есть Git-репозиторий, вы можете пропустить следующие материалы, связанные с созданием учетной записи Git.

Существует множество способов работы с Git, но одним из самых простых является создать учетную запись в GitHub, сформировать там свой репозиторий, а затем выполнить его синхронизацию с вашим локальным репозиторием.

Начнем мы с того, что посетим сайт <https://github.com/> и создадим в нем свой аккаунт.

ПРИМЕЧАНИЕ

Сайт GitHub — англоязычный. Однако некоторые веб-браузеры позволяют отображать страницы сайтов в переводе на русский язык, так что далее представлены страницы этого сайта в русскоязычном варианте, предложенном браузером.

После загрузки главной страницы сайта выберите опцию **зарегистрироваться** (рис. 11.1) — откроется окно для ввода сведений о пользователе (рис. 11.2).

В этом окне введите **Имя пользователя**, адрес своей электронной почты и пароль для входа на сайт. Подтвердите свой аккаунт и нажмите на кнопку **Регистрация** в нижней части окна.

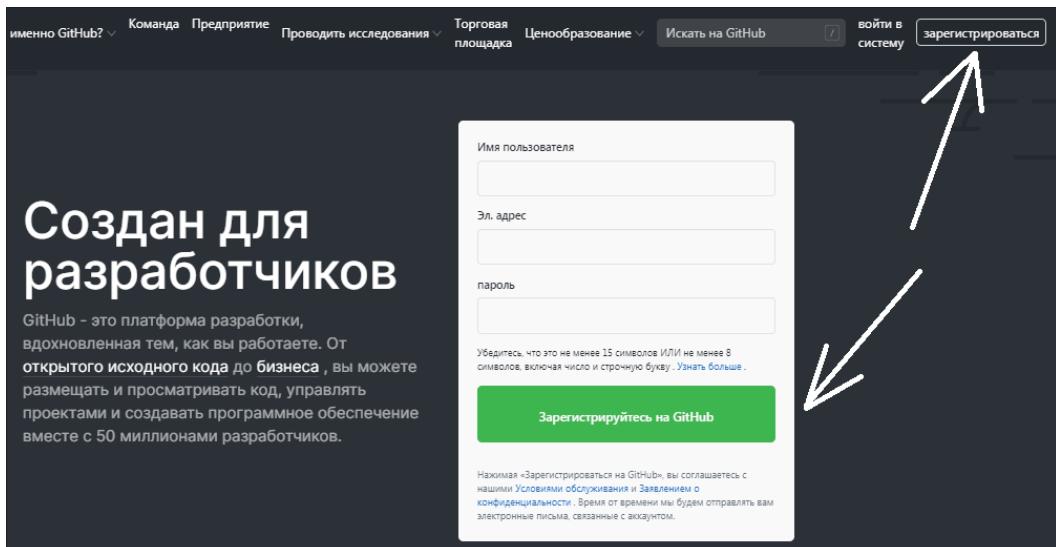


Рис. 11.1. Главная страница сайта github.com

Создать аккаунт

При создании учетной записи возникли проблемы.

Имя пользователя *

Имя пользователя не может быть пустым

Электронная почта *

Электронная почта не может быть пустым

Пароль не может быть пустым мволов ИЛИ не менее 8 символов, включая число и строчную букву, у этого больше:

Настройки электронной почты

Присылайте мне периодические обновления продуктов, объявления и предложения.

подтвердите ваш аккаунт

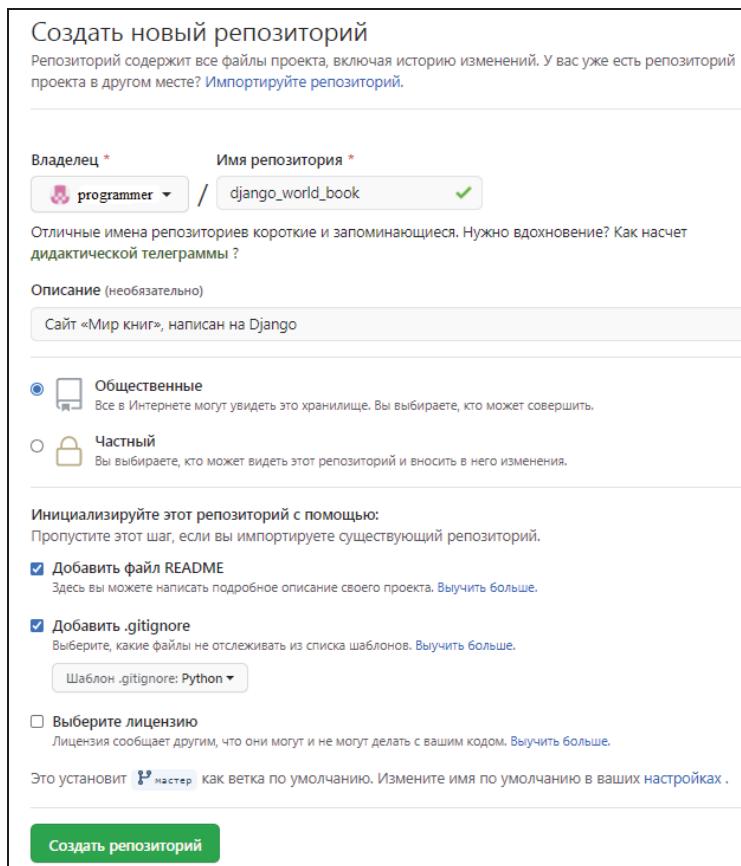
Рис. 11.2. Окно регистрации нового пользователя на сайте github.com

После успешной регистрации вы сможете войти в личный кабинет, используя данные своего аккаунта, при этом в левой верхней части окна вы увидите кнопку **Создать репозиторий**. Нажмите на нее и заполните все поля в открывшейся форме (рис. 11.3). Хотя некоторые из них не являются обязательными, рекомендуются их все же заполнить:

- введите имя нового репозитория (например: `django_world_book`) и комментарий к репозиторию (например: Сайт «Мир книг», написан на Django);
- установите флажки **Добавить файл README** (Add a README file) и **Добавить .gitignore** (Add.gitignore);
- в списке **Шаблон .gitignore** выберите **Python**;
- выберите также подходящую вам лицензию из списка **Выберите лицензию** (Add license). Если вы не знаете, для чего эта опция, то оставьте ее без изменения.

В завершение нажмите кнопку **Создать репозиторий** (Create repository), после чего ваш репозиторий будет создан (рис. 11.4).

На странице вашего репозитория нажмите на зеленую кнопку с надписью **Код** (Code) и скопируйте интернет-адрес (URL) из открывшегося диалогового окна (рис. 11.5).

Рис. 11.3. Окно создания репозитория на сайте github.com

Изучите Git и GitHub без кода!
Используя руководство Hello World, вы создадите ветку, напишете комментарии и откроете пулреквест.

[Прочтите руководство](#)

Действия | Проекты | Вики | Безопасность | Insights | Настройки

Мастер | 1 файл | 0 тегов | Перейти в файл | Добавить файл | Код |

APostolit	Начальная фиксация	3283548	34 минуты назад	1 совершает
.gitignore	Начальная фиксация		34 минуты назад	
README.md	Начальная фиксация		34 минуты назад	

README.md

django_world_book

Сайт «Мир книг», написан на Django

Около

Сайт «Мир книг», написан на Django
Прочти меня

Релизы

Релизов не опубликовано
Создать новую версию

Пакеты

Пакетов не опубликовано
Опубликуйте свой первый пакет

Рис. 11.4. Репозиторий `django_world_book` на сайте github.com

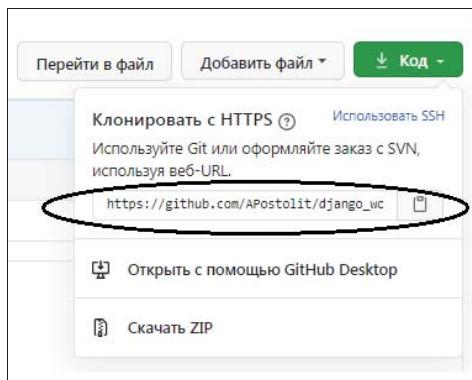


Рис. 11.5. Ссылка на репозиторий пользователя на сайте github.com

Этот адрес имеет следующую структуру:

https://github.com/<имя_пользователя>/<имя_репозитория>.git

Здесь <имя_пользователя> — это идентификатор пользователя, а <имя_репозитория> — имя созданного вами репозитория. В моем случае была сформирована следующая адресная ссылка:

https://github.com/APostolit/django_world_book.git

По ссылке, сформированной по вашим данным, вы всегда можете войти в свой удаленный репозиторий на сайте github.com.

Когда ваш удаленный репозиторий будет создан, его необходимо загрузить на ваш компьютер, на котором велась разработка сайта (создать локальный репозиторий).

Сначала установите на свой компьютер приложение Git. Вы можете найти нужную версию этого приложения для своей платформы по ссылке: <https://git-scm.com/downloads> (рис. 11.6).

Поскольку мы вели разработку сайта на Windows, то установим Windows-версию этого приложения. На момент подготовки книги была доступна версия 2.28.0 — соответственно, на компьютер был скачан файл Git-2.28.0-64-bit.exe. Выполните установку этого приложения.

Затем откройте командную строку (запустите терминал администратора Git — файл git-cmd.exe) и в ее окне выполните следующую команду, используя URL-ссылку, которая была получена на GitHub:

```
git clone https://github.com/<имя_пользователя>/<имя_репозитория>.git
```

Для нашего конкретного примера эта команда будет выглядеть следующим образом (рис. 11.7):

```
git clone https://github.com/APostolit/django_world_book.git
```

В результате в каталоге (папке), откуда была запущена эта команда (в нашем случае это каталог C:\Program Files\Git), будет создан новый каталог. Для нашего проекта имя этого каталога: django_world_book (рис. 11.8).



Рис. 11.6. Сайт для загрузки приложения Git

```
C:\Program Files\Git>git clone https://github.com/APostolit/django_world_book.git
Cloning into 'django_world_book'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), 1.62 KiB / 184.00 KiB, done.

C:\Program Files\Git>
```

Рис. 11.7. Выполнение команды в окне терминала администратора Git

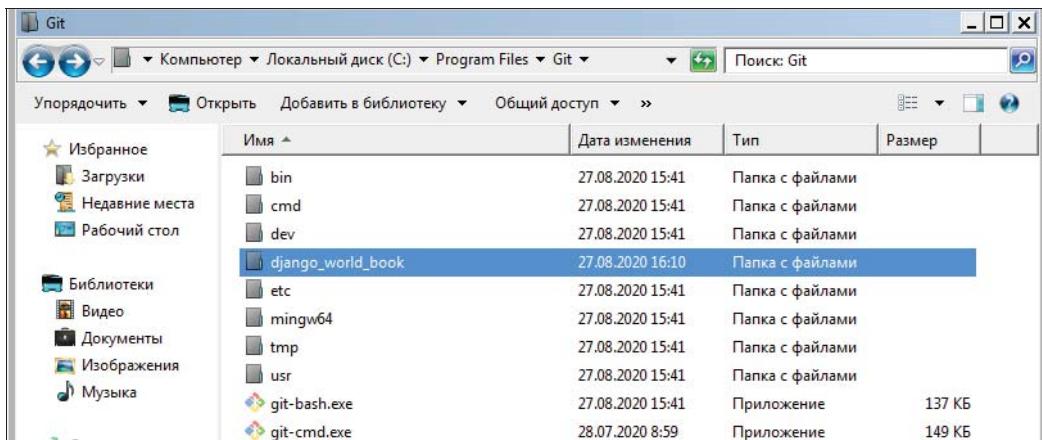


Рис. 11.8. Папка с именем django_world_book, созданная администратором git

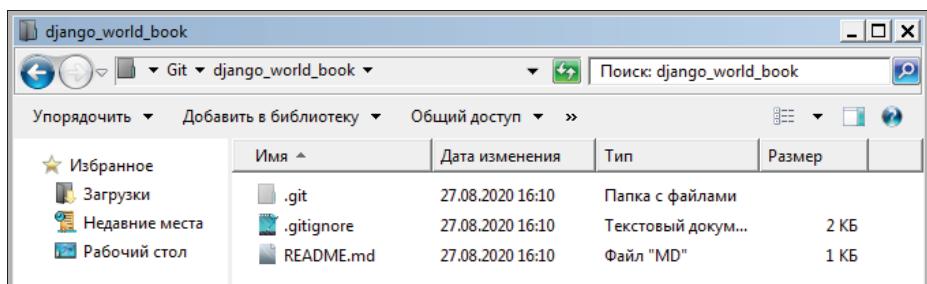


Рис. 11.9. Файлы, созданные администратором Git по умолчанию

Если открыть этот каталог, то вы увидите, что в нем по умолчанию создано несколько файлов (рис. 11.9).

Теперь надо перенести наше Django-приложение в папку локального репозитория. Для этого скопируйте в нее папку `WebBooks` с рабочими файлами нашего приложения. Обратите внимание: копирование это осуществляется БЕЗ внешней папки проекта, в которой эти файлы находятся (рис. 11.10).

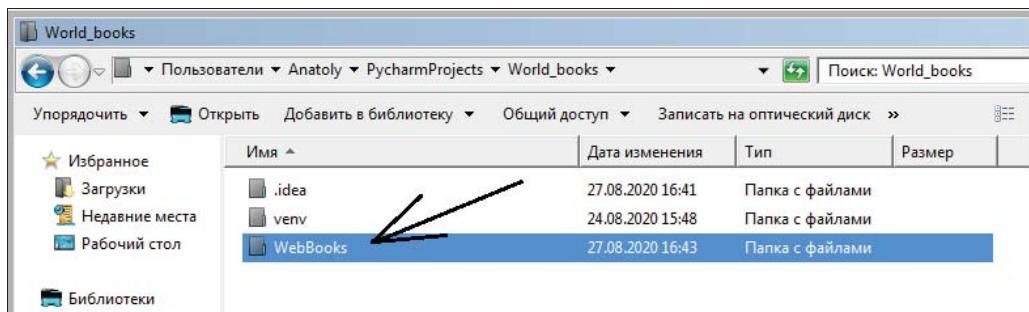


Рис. 11.10. Папка проекта, которую нужно скопировать в локальный репозиторий

Затем войдите в папку локального репозитория и откройте файл `.gitignore` в любом текстовом редакторе (в нем указаны файлы вашего приложения, которые не должны быть загружены в глобальный репозиторий Git). Добавьте в самый конец этого файла (рис. 11.11) строки, приведенные в листинге 11.4, и сохраните эти изменения.

Листинг 11.4

```
# Text backup files
*.bak

#Database
*.sqlite3
```

Возвратитесь в окно терминала администратора Git и выполните команду:

```
cd django_world_book
```

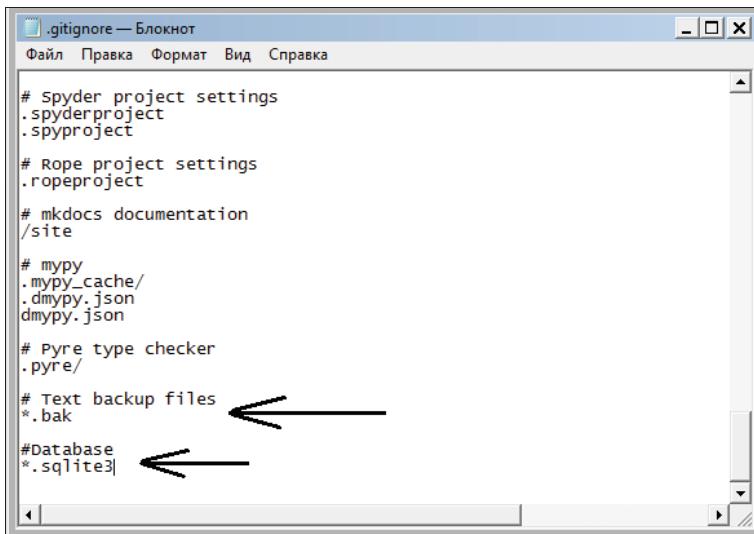


Рис. 11.11. Внесение изменений в файл .gitignore

Тем самым мы в окне терминала переместились в папку нашего локального репозитория с именем `django_world_book` (рис. 11.12).

```

Администратор: git-cmd.exe - Ярлык

C:\Program Files\Git>git clone https://github.com/APostolit/django_world_book.git
Cloning into 'django_world_book'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 <delta 0>, reused 0 <delta 0>, pack-reused 0
Unpacking objects: 100% (4/4), 1.62 KiB / 184.00 KiB/s, done.

C:\Program Files\Git>cd django_world_book
C:\Program Files\Git\django_world_book>

```

Рис. 11.12. Перемещение в папку с локальным репозиторием `django_world_book`

Введите в командной строке терминала команду `add` с флагом `(-A)`:

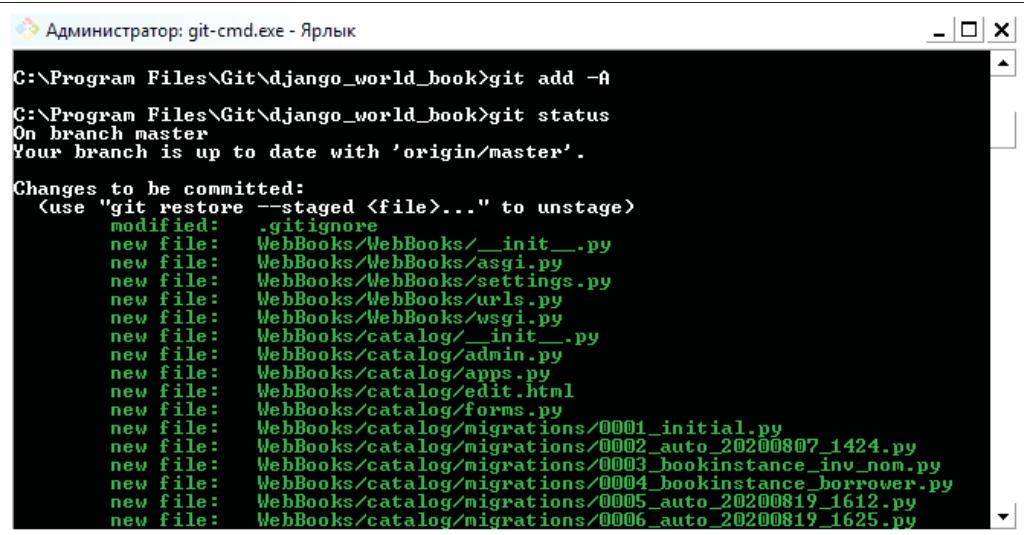
```
git add -A
```

Эта команда внесет необходимые изменения в локальный репозиторий.

Теперь в окне терминала введите команду:

```
git status
```

Эта команда помогает убедиться, что все файлы, которые мы собираемся добавить в глобальный репозиторий, верны, поскольку нам нужно добавить в глобальный репозиторий только исходные файлы проекта, а не бинарные файлы, временные файлы и пр. В консоль будет выведено сообщение, аналогичное представленному на рис. 11.13.



```
C:\Program Files\Git\django_world_book>git add -A
C:\Program Files\Git\django_world_book>git status
On branch master
Your branch is up to date with 'origin/master'.

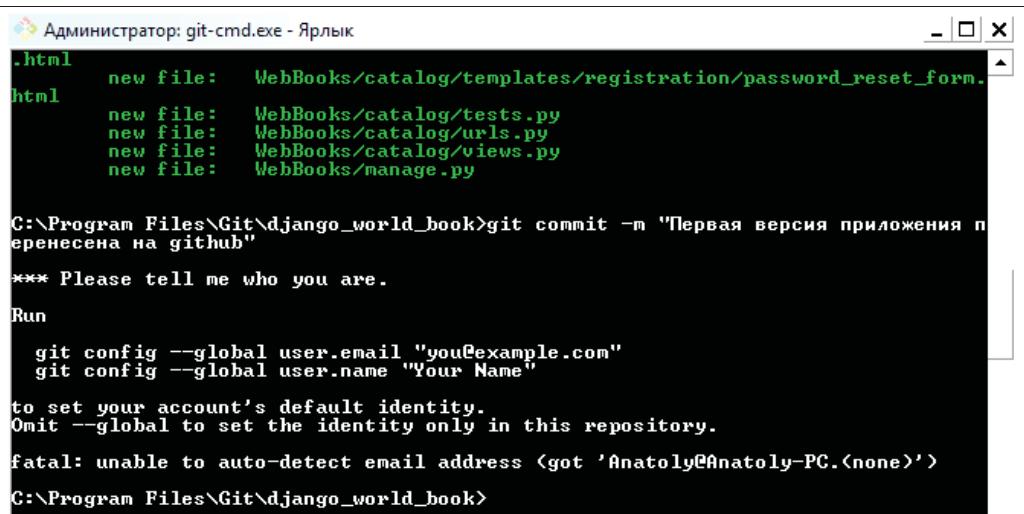
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   .gitignore
    new file:   WebBooks/WebBooks/__init__.py
    new file:   WebBooks/WebBooks/asgi.py
    new file:   WebBooks/WebBooks/settings.py
    new file:   WebBooks/WebBooks/urls.py
    new file:   WebBooks/WebBooks/wsgi.py
    new file:   WebBooks/catalog/__init__.py
    new file:   WebBooks/catalog/admin.py
    new file:   WebBooks/catalog/apps.py
    new file:   WebBooks/catalog/edit.html
    new file:   WebBooks/catalog/forms.py
    new file:   WebBooks/catalog/migrations/0001_initial.py
    new file:   WebBooks/catalog/migrations/0002_auto_20200807_1424.py
    new file:   WebBooks/catalog/migrations/0003_bookinstance_inv_nom.py
    new file:   WebBooks/catalog/migrations/0004_bookinstance_borrower.py
    new file:   WebBooks/catalog/migrations/0005_auto_20200819_1612.py
    new file:   WebBooks/catalog/migrations/0006_auto_20200819_1625.py
```

Рис. 11.13. Сообщение о статусе файлов, загруженных в локальный репозиторий django_world_book

Следующей командой зафиксируем файлы в локальном репозитории:

`git commit -m "Первая версия приложения перенесена на github"`

Если вы ранее не регистрировали свое имя и электронный адрес в конфигурации Git, то вам будет выдано следующее сообщение, предлагающее в ней зарегистрироваться (рис. 11.14).



```
.html      new file:  WebBooks/catalog/templates/registration/password_reset_form.html
          new file:  WebBooks/catalog/tests.py
          new file:  WebBooks/catalog/urls.py
          new file:  WebBooks/catalog/views.py
          new file:  WebBooks/manage.py

C:\Program Files\Git\django_world_book>git commit -m "Первая версия приложения перенесена на github"
*** Please tell me who you are.
Run
  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"
to set your account's default identity.
Omit --global to set the identity only in this repository.
fatal: unable to auto-detect email address (got 'Anatoly@Anatoly-PC.<none>')

C:\Program Files\Git\django_world_book>
```

Рис. 11.14. Предложение зарегистрировать свои параметры в глобальной конфигурации Git

В этом случае в окне терминала выполните следующие команды регистрации своего имени и электронного адреса в конфигурации Git (используя свой личный идентификатор на Git и свой личный электронный адрес):

```
git config --global user.name "<ваш идентификатор на git>"  
git config --global user.email <ваш электронный адрес в формате  
exampl@example.com>.
```

После регистрации этих параметром командой `commit` зафиксируйте файлы в локальном репозитории (рис. 11.15):

```
git commit -m "Первая версия приложения перенесена на github"
```

```
C:\>Program Files\Git\django_world_book>git commit -m "Первая версия приложения перенесена на github"  
[master 9af9b2a] Первая версия приложения перенесена на github  
 50 files changed, 1283 insertions(+)  
 create mode 100644 WebBooks/WebBooks/__init__.py  
 create mode 100644 WebBooks/WebBooks/asgi.py  
 create mode 100644 WebBooks/WebBooks/settings.py  
 create mode 100644 WebBooks/WebBooks/urls.py  
 create mode 100644 WebBooks/WebBooks/wsgi.py  
 create mode 100644 WebBooks/catalog/__init__.py  
 create mode 100644 WebBooks/catalog/admin.py  
 create mode 100644 WebBooks/catalog/apps.py  
 create mode 100644 WebBooks/catalog/edit.html  
 create mode 100644 WebBooks/catalog/forms.py  
 create mode 100644 WebBooks/catalog/migrations/0001_initial.py  
 create mode 100644 WebBooks/catalog/migrations/0002_auto_20200807_1424.py  
 create mode 100644 WebBooks/catalog/migrations/0003_bookinstance_inv_nom.py  
 create mode 100644 WebBooks/catalog/migrations/0004_bookinstance_borrower.py  
 create mode 100644 WebBooks/catalog/migrations/0005_auto_20200819_1612.py  
 create mode 100644 WebBooks/catalog/migrations/0006_auto_20200819_1625.py  
 create mode 100644 WebBooks/catalog/migrations/0007_auto_20200821_1918.py  
 create mode 100644 WebBooks/catalog/migrations/0008_auto_20200824_2019.py  
 create mode 100644 WebBooks/catalog/migrations/__init__.py  
 create mode 100644 WebBooks/catalog/models.py  
 create mode 100644 WebBooks/catalog/static/css/styles.css  
 create mode 100644 WebBooks/catalog/static/images/bg.jpg  
 create mode 100644 WebBooks/catalog/static/images/image1.jpg  
 create mode 100644 WebBooks/catalog/static/images/logotip.jpg  
 create mode 100644 WebBooks/catalog/templates/base_generic.html  
 create mode 100644 WebBooks/catalog/templates/catalog/author_list.html  
 create mode 100644 WebBooks/catalog/templates/catalog/authors.html  
 create mode 100644 WebBooks/catalog/templates/catalog/authors_add.html  
 create mode 100644 WebBooks/catalog/templates/catalog/book_confirm_delete.html  
 create mode 100644 WebBooks/catalog/templates/catalog/book_detail.html  
 create mode 100644 WebBooks/catalog/templates/catalog/book_form.html  
 create mode 100644 WebBooks/catalog/templates/catalog/book_list.html  
 create mode 100644 WebBooks/catalog/templates/catalog/book_renew_librarian.html  
 create mode 100644 WebBooks/catalog/templates/catalog/bookinstance_list_borrowe
```

Рис. 11.15. Результаты выполнения команды `commit`

Наконец, запустите команду синхронизации своего локального репозитория с глобальным репозиторием на сайте GitHub (рис. 11.16):

```
git push origin master
```

Когда эта операция завершится, можно проверить, выполнился ли перенос нашего проекта с локального репозитория вашего компьютера в глобальный репозиторий на сайте GitHub. Для этого вернитесь на страницу сайта GitHub, где вы создали свой репозиторий и, при необходимости, обновите страницу своего аккаунта. Если все было сделано правильно, то вы убедитесь, что наше приложение загружено (рис. 11.17).

Если в проект были внесены изменения, то обновить файлы в репозитории теперь можно простым повторением последовательности команд `add`, `commit`, `push` в окне терминала администратора Git.

```
C:\Program Files\Git\django_world_book>git push origin master
Username for 'https://github.com': APostolit
Password for 'https://APostolit@github.com':
Enumerating objects: 61, done.
Counting objects: 100% (61/61), done.
Delta compression using up to 4 threads
Compressing objects: 100% (56/56), done.
Writing objects: 100% (59/59), 161.20 KiB / 9.48 MiB/s, done.
Total 59 <delta 8>, reused 0 <delta 0>, pack-reused 0
remote: Resolving deltas: 100% (8/8), completed with 1 local object.
To https://github.com/APostolit/django_world_book.git
 3203548..9af9b2a master -> master

C:\Program Files\Git\django_world_book>
```

Рис. 11.16. Результаты выполнения команды push

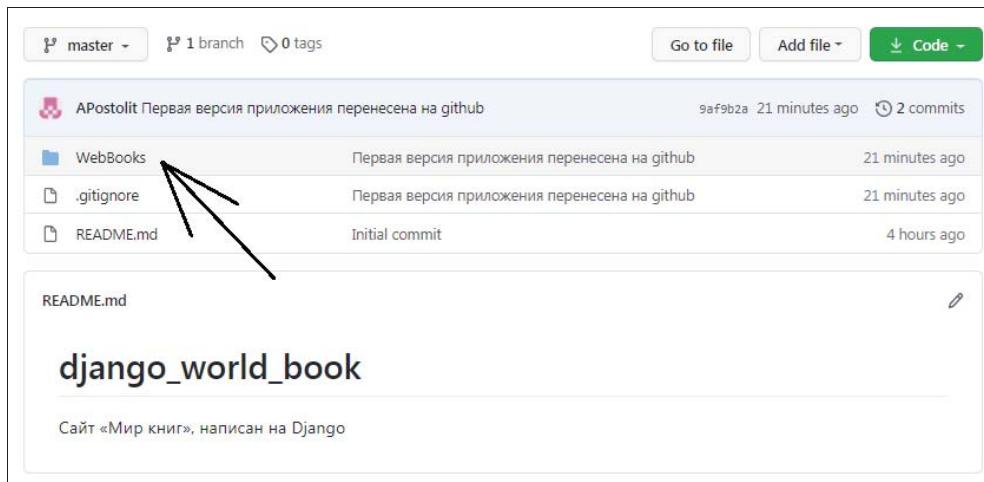


Рис. 11.17. Приложение WebBooks («Мир книг») в глобальном репозитории сайта GitHub

11.3.2. Подключение веб-сервера Gunicorn

Теперь рассмотрим изменения, которые необходимо сделать в нашем приложении WebBooks, чтобы оно работало в Интернете после публикации на сервисе Heroku.

Создайте файл с именем Procfile (без расширения) в «корне» нашего локального GitHub-репозитория (рис. 11.18), объявив в нем типы процессов и точки входа в приложение (листинг 11.5).

Листинг 11.5

```
web: gunicorn webhook.wsgi --log-file -
```

Здесь параметр `web:` сообщает Heroku, что это будет динамический веб-процесс с HTTP-трафиком и что этот процесс будет использовать сервер `gunicorn`, который является популярным сервером для веб-приложений (этот сервер используют сервисы Heroku).

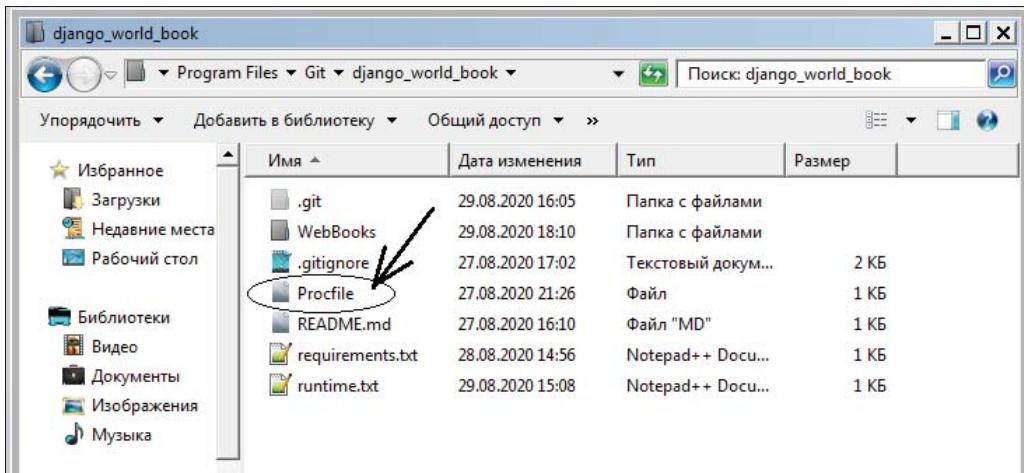


Рис. 11.18. Создание файла с именем Procfile в локальном репозитории сайта GitHub

ВЕБ-СЕРВЕР GUNICORN

Gunicorn — это автономный веб-сервер с обширной функциональностью. Он изначально поддерживает различные фреймворки, что делает его чрезвычайно простым в прямой замене многих серверов разработки. Технически Gunicorn работает подобно Unicorn — популярному веб-серверу приложений Ruby. Они оба используют так называемую модель *pre-fork* (это значит, что главный процесс управляет инициированными рабочими процессами различного типа). Особенности сервера Gunicorn заключаются в следующем:

- запускает любое приложение WSGI Python;
- служит заменой серверам Paster/Pyramid (сервер разработки Django), web2py и пр.;
- поставляется с различными конфигурациями и типами процессов;
- автоматически управляет процессами;
- поддерживает HTTP/1.0 и HTTP/1.1 с помощью синхронных и асинхронных процессов;
- поддерживает SSL;
- расширяется с помощью специальных точек входа;
- поддерживает Python 2.6+ и 3.x.

Gunicorn представляет собой HTTP-сервер, который рекомендуется для запуска Django-приложений на Heroku. Это чистый Python HTTP-сервер для WSGI-приложений, который может запускать множество параллельных Python-процессов в пределах одного динамического процесса. WSGI (Web Server Gateway Interface) — это стандарт взаимодействия между Python-программой, выполняющейся на стороне сервера, и самим веб-сервером.

Мы будем запускать сервер Gunicorn, используя конфигурационную информацию из модуля `wsgi.py`, расположенного в каталоге `\WebBooks\WebBooks\`. Этот файл был создан автоматически, когда Django создавал наш проект с помощью команды: `django-admin startproject`.

Gunicorn нам также понадобится для обслуживания нашего приложения в течение разработки. Установить Gunicorn локально можно из командной строки, используя пакетный менеджер pip:

```
pip install gunicorn
```

Или это можно сделать через интерфейс PyCharm в окне изменения установок **Settings** (рис. 11.19).

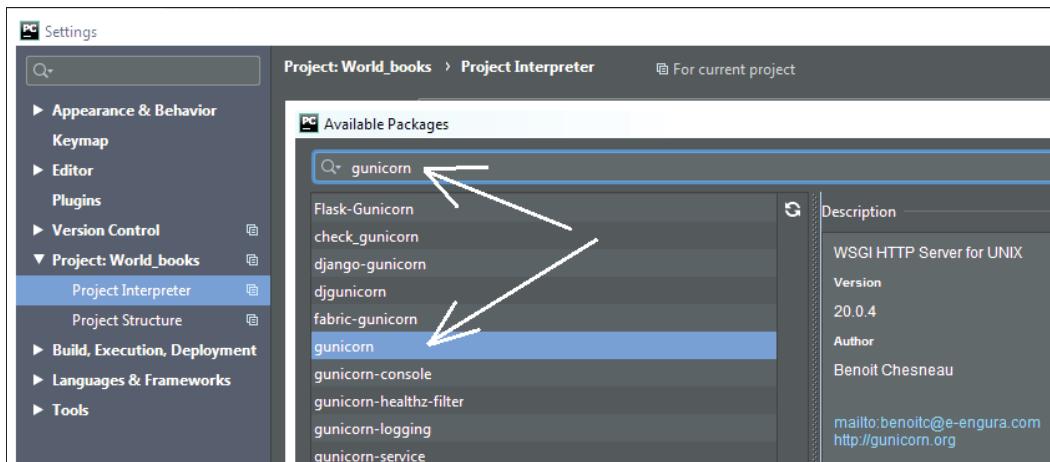


Рис. 11.19. Установка веб-сервера Gunicorn на локальный компьютер разработчика в PyCharm

11.3.2. Пакеты для обеспечения доступа к базе данных на Heroku

Теперь нам надо выполнить настройки, которые позволят серверу увидеть нашу базу данных. Мы не можем использовать на Heroku по умолчанию базу данных SQLite, потому что она основана на файлах. Они будут удаляться из эфемерной файловой системы сервера каждый раз, когда приложение перезагружается (после изменения самого приложения или переменных его конфигурации).

Механизм Heroku для обработки этой ситуации заключается в использовании настроек базы данных и настройки веб-приложения с использованием информации из переменной конфигурации среды. Существует множество опций базы данных, но мы воспользуемся бесплатным уровнем доступа к базе данных Postgres Heroku. Для этого есть несколько причин: это, как уже отмечено, бесплатно, это поддерживается Django, это будет автоматически добавляться в наши новые приложения при использовании бесплатного уровня доступа.

Информация о подключении базы данных предоставляется ботом Heroku web-dyno с использованием конфигурационной переменной `DATABASE_URL`. Вместо того чтобы жестко кодировать эту информацию в Django, Heroku рекомендует разработчикам использовать пакет `dj-database-url`. В дополнение к пакету `dj-database-url` нам

также потребуется установить пакет `psycopg2`, поскольку Django нуждается в этом пакете, чтобы взаимодействовать с базой данных Postgres.

Установите пакет `dj-database-url` локально, чтобы он стал частью наших требований к настройкам Heroku на удаленном сервере. Для этого в окне терминала PyCharm выполните следующую команду:

```
pip install dj-database-url
```

В настройках **Settings** нашего инструментария PyCharm (рис. 11.20) можно убедиться, что модуль `dj-database-url` появился среди прочих модулей нашего проекта.

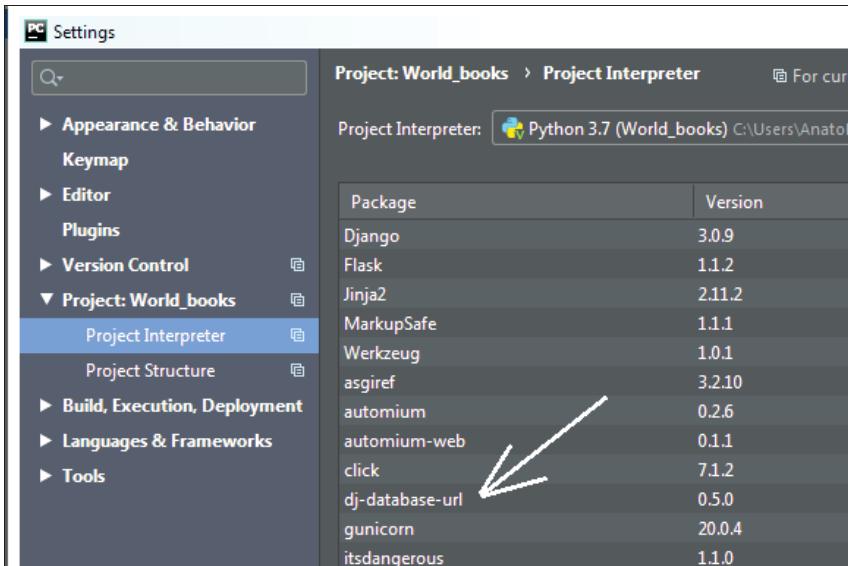


Рис. 11.20. Установка пакета `dj-database-url` на локальный компьютер разработчика в PyCharm

Теперь откройте файл `\WebBooks\settings.py` и в нижнюю его часть допишите следующий код (листинг 11.6).

Листинг 11.6

```
# Heroku: Обновление конфигурации базы данных из $DATABASE_URL.
import dj_database_url
db_from_env = dj_database_url.config(conn_max_age=500)
DATABASES['default'].update(db_from_env)
```

Естественно, строку `import dj_database_url` следует затем перенести в верхнюю часть этого файла, где присутствуют остальные строки импорта модулей.

ПРИМЕЧАНИЕ

Мы по-прежнему будем использовать БД SQLite во время разработки и модификации нашего приложения, поскольку переменная среды `DATABASE_URL` не будет установлена на нашем компьютере, где мы ведем разработку.

Значение `conn_max_age=500` делает соединение постоянным, что намного эффективнее, чем воссоздавать его в каждом цикле запросов. Однако это не обязательно, и при необходимости этот параметр можно удалить.

11.3.3. Настройка доступа к базе данных Postgres на Heroku

При работе с Heroku в сети Интернет Django понадобится модуль `psycopg2`, обеспечивающий его взаимодействие с базами данных Postgres, и это нужно прописать в файле требований к внешней среде (`requirements.txt`). Однако для разработки и модификации вашего приложения на локальном компьютере Django будет по-прежнему использовать базу данных SQLite (локально по умолчанию), поскольку переменная среды `DATABASE_URL` не задана в нашей локальной среде.

ПРИМЕЧАНИЕ

Вы можете полностью перейти на Postgres и задействовать эту бесплатную базу данных и для разработки приложения, и для развертывания его на Heroku. Чтобы установить модуль `psycopg2` и использовать его в системе локально, нужно выполнить следующую команду:

```
pip install psycopg2
```

Однако этого можно не делать — нам не обязательно, чтобы СУБД PostgreSQL была активна на локальном компьютере. Heroku будет достаточно, если сообщить в файле `requirements.txt`, что эта база будет использоваться на удаленном сервере. Однако мы выполним эту процедуру, поскольку она обеспечит на следующих шагах автоматическое добавление модуля `psycopg2` в файл `requirements.txt`.

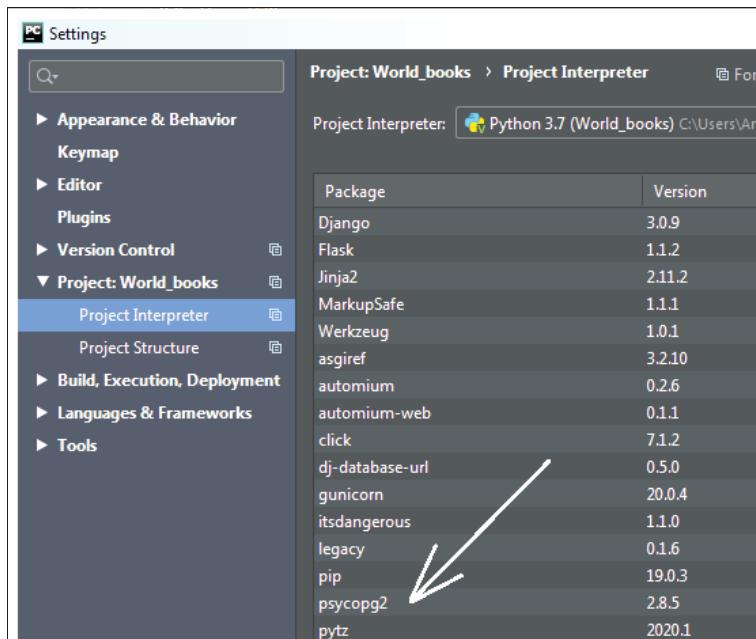


Рис. 11.21. Проверка успешной установки модуля `psycopg2` на локальный компьютер в PyCharm

Итак, добавим модуль psycopg2 к нашему проекту с помощью следующей команды в окне терминала PyCharm:

```
pip install psycopg2
```

Убедимся, что этот модуль успешно добавлен к проекту (рис. 11.21).

11.3.4. Обслуживание статичных файлов на Heroku

Во время разработки приложения мы использовали локальный веб-сервер разработки Django для обслуживания наших статических файлов (CSS, JavaScript, изображений и т. п.). При развертывании сайта в сети Интернет за обработку этих файлов будет отвечать внешний веб-сервер и система CDN (Content Delivery Network).

ПРИМЕЧАНИЕ

CDN — это географически распределенная сетевая инфраструктура, обеспечивающая быструю доставку контента пользователям веб-сервисов и сайтов. Входящие в состав CDN серверы географически располагаются таким образом, чтобы сделать время ответа для пользователей сайта минимальным.

Чтобы упростить размещение статических файлов отдельно от веб-приложения, Django предоставляет средство сбора сведений об этих файлах для развертывания (переменную параметров `collectstatic`, определяющую, где файлы должны собираться при запуске приложения). Шаблоны Django извлекают информацию о месте размещения статических файлов из переменной параметров `STATIC_URL`, так что этот параметр и ряд других нужно изменить, если статические файлы перемещаются на другой хост или сервер.

К параметрам, которые следует скорректировать, относятся следующие:

- `STATIC_URL` — это базовое расположение URL, из которого будут загружены статические файлы (например, на CDN). Этот параметр также используется для переменной статического шаблона, доступ к которому осуществляется в нашем базовом шаблоне;
- `STATIC_ROOT` — это абсолютный путь к каталогу, в котором инструмент `collectstatic` Django будет собирать любые статические файлы, упомянутые в наших шаблонах. После их сбора они затем будут загружены в единую группу;
- `STATICFILES_DIRS` — в этом списке указаны дополнительные каталоги, в которых инструмент коллективного поиска Django `collectstatic` должен искать статические файлы.

Откройте файл `\WebBooks\settings.py` и добавьте в конец файла следующий код (листинг 11.7).

Листинг 11.7

```
# Статичные файлы (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.10/howto/static-files/
```

```
# Абсолютный путь к каталогу, в котором collectstatic
# будет собирать статические файлы для развертывания.
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

Выполните еще несколько обязательных действий. Скопируйте каталог static в корневой каталог нашего приложения (World_books\WebBooks\), после чего в окне терминала PyCharm выполните команду:

```
python manage.py collectstatic
```

В результате в каталоге World_books\WebBooks\ будет создан новый каталог staticfiles. В нем сформируются подкаталоги и файлы, которые будут перенесены на Heroku (рис. 11.22).

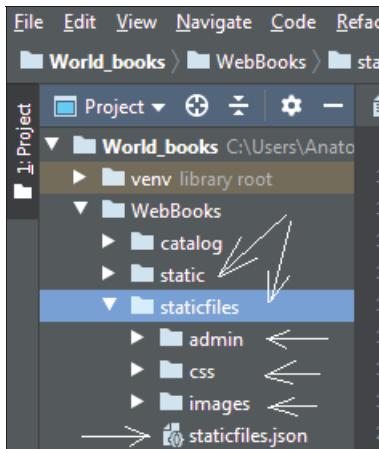


Рис. 11.22. Создание в PyCharm каталога staticfiles

11.3.5. Подключение библиотеки WitheNoise

Существует множество способов обслуживания статических файлов на серверах в сети Интернет. Heroku рекомендует использовать для этого библиотеку WhiteNoise непосредственно из веб-сервера Gunicorn.

ПРИМЕЧАНИЕ

Heroku автоматически вызывает `collectstatic` и готовит ваши статические файлы для использования в WhiteNoise после того, как будет загружено ваше приложение.

Для установки библиотеки WhiteNoise на ваш локальный компьютер выполните в терминальном окне PyCharm следующую команду:

```
pip install whitenoise
```

Убедимся, что этот модуль успешно добавлен к проекту (рис. 11.23).

Чтобы ваше приложение начало использовать библиотеку WhiteNoise, откройте файл \WebBooks\settings.py, найдите в нем параметр MIDDLEWARE и добавьте строку:

```
'whitenoise.middleware.WhiteNoiseMiddleware'
```

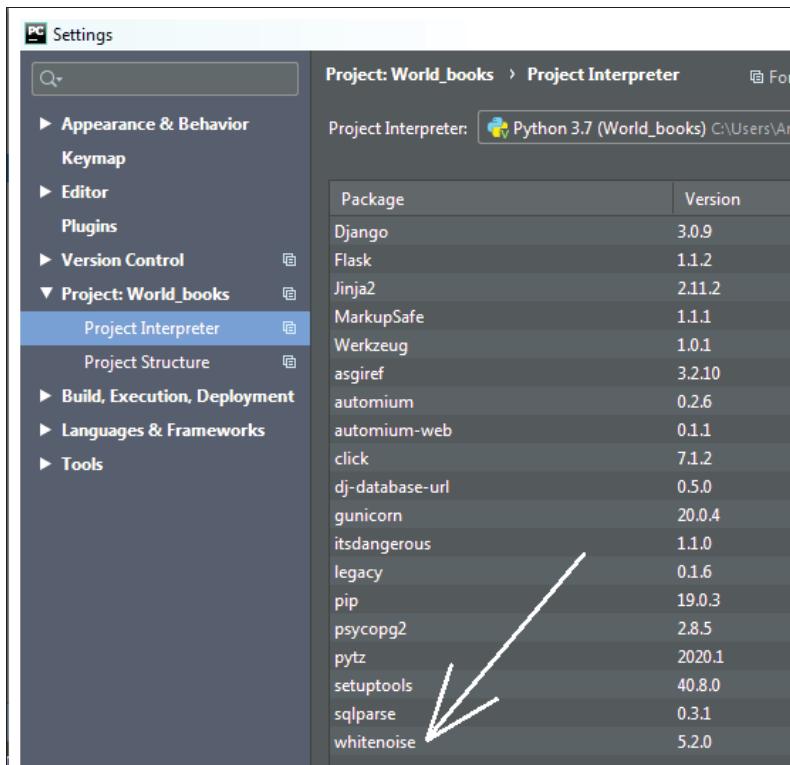


Рис. 11.23. Проверка успешной установки модуля whitenoise на локальный компьютер в PyCharm

В верхнюю часть списка ниже строки:

```
'django.middleware.security.SecurityMiddleware',
```

Результаты этих действий показаны на рис. 11.24.

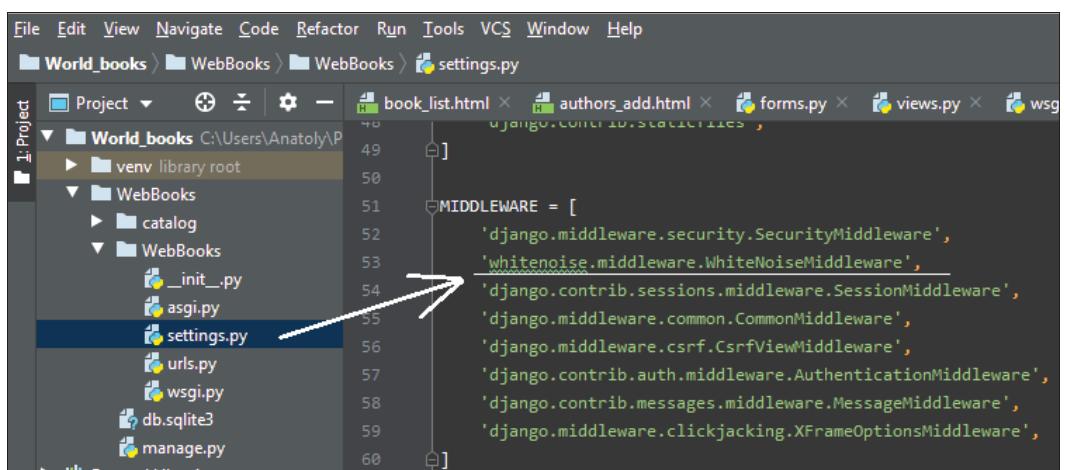


Рис. 11.24. Включение в конфигурацию Django пакета WhiteNoise на локальном компьютере в PyCharm

При желании мы можем уменьшить размер статических файлов при их обслуживании на внешнем сайте (это повысит эффективность его работы). Просто добавьте в конец файла `WebBooks\settings.py` следующий код (листинг 11.8).

Листинг 11.8

```
# Упрощенная обработка статических файлов.  
# https://warehouse.python.org/project/whitenoise/  
STATICFILES_STORAGE =  
    'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

11.3.6. Задание требований к Python

Требования, предъявляемые к версии Python, с которым будет работать веб-приложение, как уже отмечалось ранее, должны храниться в файле `requirements.txt`, расположенному в корневом каталоге вашего репозитория. Имея эти требования, Heroku автоматически выполнит необходимые настройки при создании среды для вашего приложения. Вы можете создать этот файл с помощью следующей команды `pip` из командной строки (напомним еще раз: файл `requirements.txt` должен находиться в корневом каталоге вашего приложения):

```
pip freeze > requirements.txt
```

ПРИМЕЧАНИЕ

`pip freeze requirements.txt` — это команда, позволяющая создать текстовый документ, в котором указаны все установленные и необходимые для работы Python-приложения программные пакеты (чаще всего — для Django).

Список всех пакетов можно посмотреть, выполнив команду `pip freeze` (стандартный вывод производится на экран терминала). Обычно эту информацию сохраняют в файл, который оставляется в корне приложения и переносится на другой сервер вместе с ним.

Убедимся, что этот файл успешно добавлен к проекту (рис. 11.25).

Файл `requirements.txt` должен как минимум содержать проведенные далее строки (номера версий на вашем компьютере могут отличаться):

```
dj-database-url==0.5.0  
Django==3.0.9  
gunicorn==20.0.4  
psycopg2==2.8.5  
whitenoise==5.2.0
```

ПРИМЕЧАНИЕ

Убедитесь, что строка для `psycopg2` присутствует в этом файле! Даже если вы не установили это приложение локально, вы должны добавить такую строку в файл `requirements.txt`.

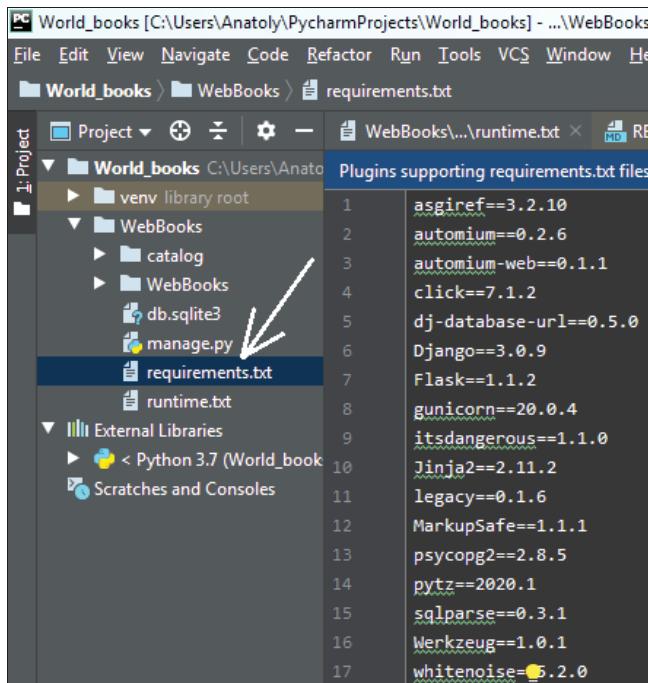


Рис. 11.25. Проверка успешного создания файла requirements.txt в PyCharm

11.3.7. Настройка среды выполнения

Файл runtime.txt сообщает Heroku о том, какой язык программирования нужно использовать при работе с нашим приложением. Создайте файл runtime.txt в «корне» нашего приложения и добавьте в него следующий текст:

```
python-3.7.9
```

ПРИМЕЧАНИЕ

Heroku одновременно поддерживает несколько версий Python (на момент подготовки книги это версии: 2.7.18, 2.6.12, 3.7.9, 3.8.5). Информацию о поддерживаемых версиях можно получить на сайте по адресу: <https://devcenter.heroku.com/articles/python-support#supported-python-runtimes>.

Heroku будет использовать версию Python, которая наиболее подходит для вашего приложения (независимо от значения версии, указанной в этом файле).

Затем перенесите все сделанные вами изменения в GitHub и снова скопируйте папку WebBooks с вашим сайтом из среды разработки в каталог с локальным репозиторием Git.

В эту же папку скопируйте файл manage.py. Здесь очень важно проверить, чтобы файлы с именами manage.py, Procfile, requirements.txt и runtime.txt находились в корневом каталоге локального репозитория. В нашем случае это каталог django_world_book (рис. 11.26).

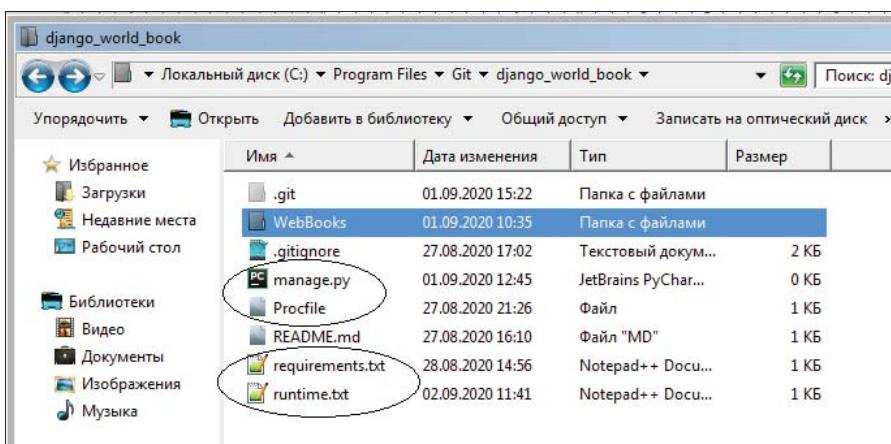


Рис. 11.26. Место файлов manage.py, Procfile, requirements.txt и runtime.txt в локальном репозитории

Если они находятся в другом месте, переместите их в этот каталог. Если этого не сделать, то приложение из-за ошибки не будет переноситься на сайт Heroku.

Затем запустите на выполнение файл git-cmd.exe (административный терминал Git) внутри вашего локального репозитория Git и в окне этого терминала последовательно выполните следующие команды (рис. 11.27):

```
cd django_world_book
git add -A
git commit -m "Added files for deployment in heroku 20.08.2020"
git push origin master
```

```
C:\> cd django_world_book
C:\Program Files\Git\django_world_book>git add -A
C:\Program Files\Git\django_world_book>git commit -m "Added files for deployment in heroku 29.08.2020"
[master 9d24e03] Added files for deployment in heroku 29.08.2020
 5 files changed, 51 insertions(+)
 create mode 100644 WebBooks/WebBooks/requirements.txt
 create mode 100644 WebBooks/WebBooks/runtime.txt
 create mode 100644 WebBooks/requirements.txt
 create mode 100644 WebBooks/runtime.txt

C:\Program Files\Git\django_world_book>git push origin master
Username for 'https://github.com': APostolit
Password for 'https://APostolit@github.com':
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 1.39 KiB / 712.00 KiB/s, done.
Total 7 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/APostolit/django_world_book.git
  f427a99..9d24e03 master -> master

C:\Program Files\Git\django_world_book>
```

Рис. 11.27. Обновление параметров проекта на сайте GitHub

После этого желательно снова перейти на сайт github.com с вашим приложением и убедиться, что все изменения, которые вы сделали на текущий момент, попали в глобальный репозиторий и там присутствуют все важные файлы с настройками (рис. 11.28).

Теперь все подготовлено для развертывания вашего сайта на Heroku.

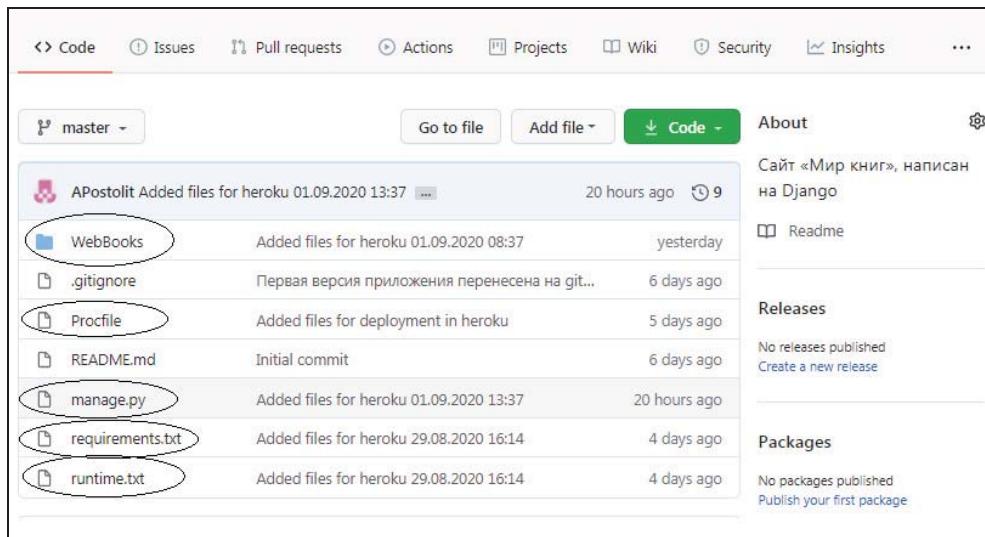


Рис. 11.28. Проверка наличия обновлений параметров проекта в глобальном репозитории на сайте github.com

11.3.8. Получение аккаунта на Heroku

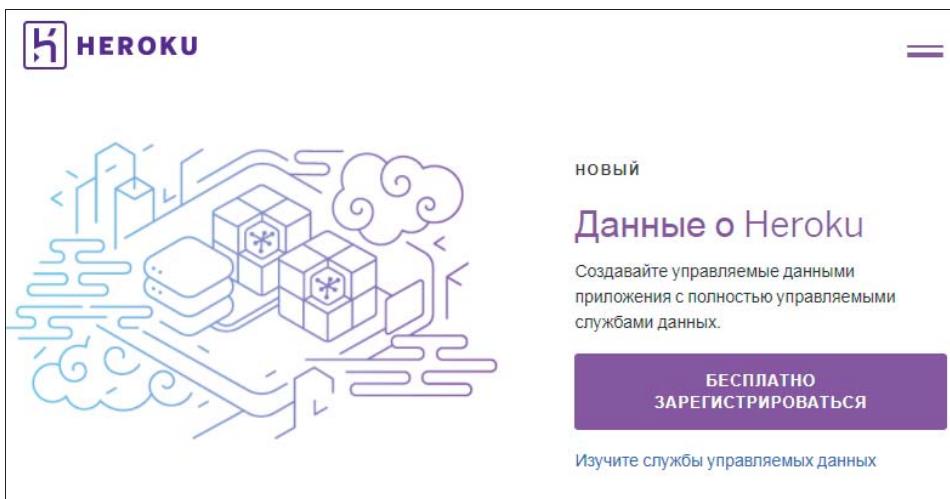
Чтобы начать использовать Heroku, сначала нужно создать на этом интернет-ресурсе свою учетную запись. Для этого перейдите на сайт www.heroku.com и нажмите кнопку **БЕСПЛАТНО ЗАРЕГИСТРИРОВАТЬСЯ** (рис. 11.29) — или в английском варианте страницы: **SIGN UP FOR FREE**.

В открывшуюся форму регистрации введите свои данные и нажмите кнопку **СОЗДАТЬ БЕСПЛАТНЫЙ АККАУНТ** (рис. 11.30) — или в английском варианте страницы: **CREATE FREE ACCOUNT**.

После успешной регистрации вы можете перейти на главную страницу сайта для работы с приложениями по адресу: <https://dashboard.heroku.com/apps> (рис. 11.31).

Для дальнейшей работы нам потребуется загрузить и установить на своем компьютере клиент Heroku (рис. 11.32). Скачать нужную версию клиента можно по ссылке: <https://devcenter.heroku.com/articles/getting-started-with-python#set-up>.

После установки этого приложения вы можете использовать клиентское приложение Heroku — отдавать команды из его командной оболочки. В Windows запустите командную строку (cmd.exe), чтобы получить доступ к командной оболочке Windows. В открывшемся окне наберите команду `heroku login` для входа в интерфейс

Рис. 11.29. Фрагмент главной страницы сайта heroku.comA registration form for Heroku. It consists of several input fields with red asterisks indicating required information: "Имя" (Name), "Фамилия" (Last Name), "Адрес электронной почты" (Email Address), "Название компании" (Company Name), "Роль" (Role), "Страна" (Country), and "Основной язык разработки" (Primary Development Language). Each field has a corresponding text input box. Below these fields is a reCAPTCHA verification section with a checkbox labeled "I'm not a robot", the reCAPTCHA logo, and links for "Privacy" and "Terms". At the bottom of the form is a large blue button with the text "СОЗДАТЬ БЕСПЛАТНЫЙ АККАУНТ" (Create Free Account).Рис. 11.30. Регистрационная форма на сайте heroku.com

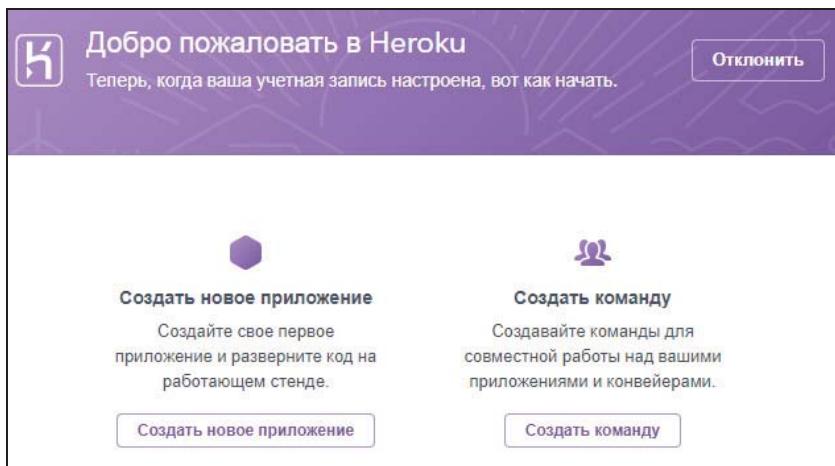


Рис. 11.31. Главная страница на сайте heroku.com

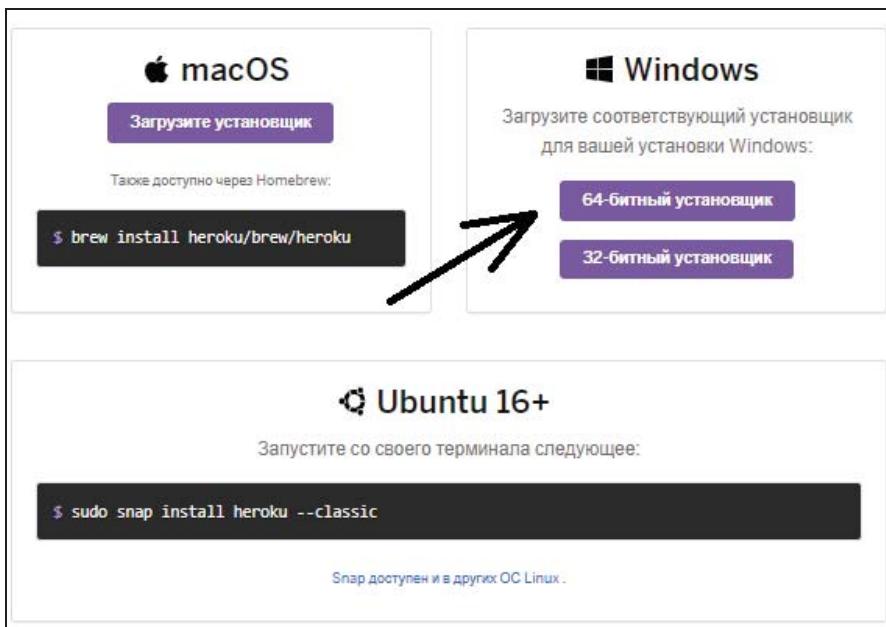


Рис. 11.32. Страница сайта heroku.com для загрузки клиента Heroku

командной строки Heroku (CLI, Command Line Interface) и нажмите клавишу <Enter> — в окне терминала командной строки появится строчка с предложением нажать любую клавишу (рис. 11.33).

Нажмите любую клавишу, и загрузится страница сайта Heroku с единственной кнопкой (рис. 11.34).

Нажмите здесь кнопку **Log in** — появится сообщение об успешном входе и необходимости закрытия этой страницы (рис. 11.35).

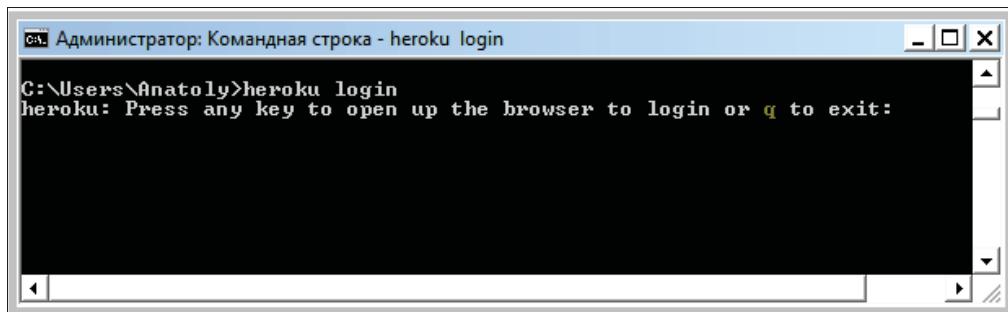
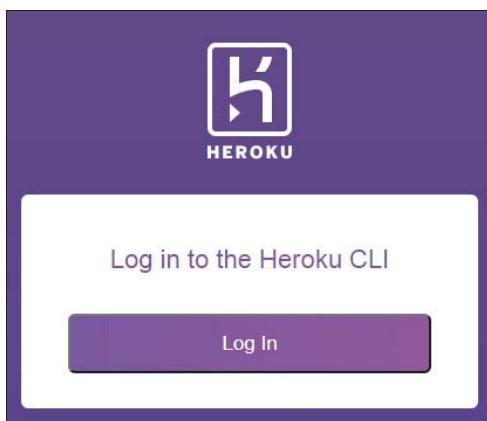
Рис. 11.33. Ввод команды `heroku login` для входа в Heroku CLI

Рис. 11.34. Страница сайта Heroku с кнопкой для входа в Heroku CLI

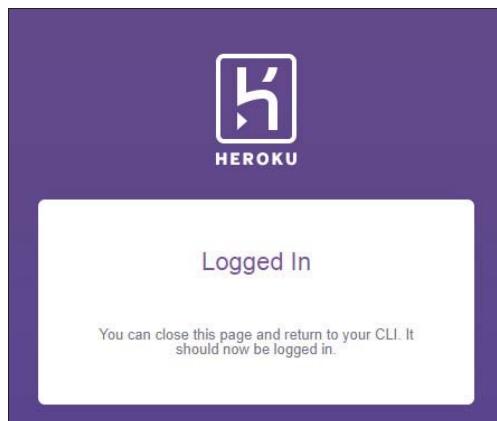


Рис. 11.35. Страница сайта Heroku с информацией об успешном входе в Heroku CLI

Закройте эту страницу сайта, и вы возвратитесь к окну терминала командной строки Windows. Там будет выдано сообщение об успешном входе в режим работы с Heroku с показом электронного адреса пользователя, который осуществил вход (рис. 11.36).



Рис. 11.36. Окно терминала командной строки с информацией об успешном входе в Heroku CLI

11.3.9. Создание и запуск приложения на Heroku

Теперь мы можем взаимодействовать с Heroku, используя различные команды. Введите в окне терминала строку: heroku help, и вы получите список возможных команд (рис. 11.37).

```
C:\>Users\Anatoly>heroku help
CLI to interact with Heroku

VERSION
heroku/7.42.13 win32-x64 node-v12.16.2

USAGE
$ heroku [COMMAND]

COMMANDS
access      manage user access to apps
addons      tools and services for developing, extending, and operating
            your app
apps        manage apps on Heroku
auth        check 2fa status
authorizations OAuth authorizations
autocomplete display autocomplete installation instructions
buildpacks   scripts used to compile apps
certs       a topic for the ssl plugin
ci          run an application test suite on Heroku
clients     OAuth clients on the platform
config      environment variables of apps
container   Use containers to build and deploy Heroku apps
domains    custom domains for apps
drains      forward logs to syslog or HTTPS
features   add/remove app features
git         manage local git repository for app
help        display help for heroku
keys        add/remove account ssh keys
labs        add/remove experimental features
local      run Heroku app locally
logs       display recent log output
maintenance enable/disable access to app
members    manage organization members
notifications display notifications
orgs       manage organizations
pg          manage postgresql databases
pipelines  manage pipelines
plugins    list installed plugins
ps          Client tools for Heroku Exec
psql      open a psql shell to the database
```

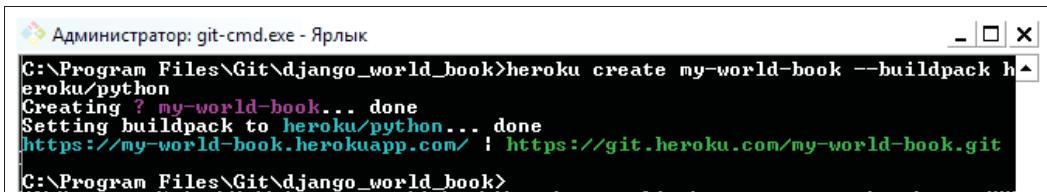
Рис. 11.37. Список возможных команд для взаимодействия с Heroku

Чтобы создать приложение, нужно выполнить команду create, находясь в корневом каталоге нашего локального репозитория. Будет создано приложение с именем по умолчанию (сайт сам случайным образом сгенерирует имя — например: stil-cliffs-67252), однако имя приложения можно задать и самому.

Итак, запустим на выполнение файл git-cmd.exe (административный терминал Git) внутри нашего локального репозитория Git и в окне этого терминала выполним команду (рис. 11.38):

```
heroku create my-world-book --buildpack heroku/python
```

В этой команде мы задали два параметра: имя создаваемого приложения (`my-world-book`) и язык, который использует наше приложение (`heroku/python`). Соответственно, нашему приложению на удаленном репозитории Heroku будет дано имя: `my-world-book`.



```
Administrator: git-cmd.exe - Ярлык
C:\Program Files\Git\django_world_book>heroku create my-world-book --buildpack heroku/python
Creating ? my-world-book... done
Setting buildpack to heroku/python... done
https://my-world-book.herokuapp.com/ | https://git.heroku.com/my-world-book.git
C:\Program Files\Git\django_world_book>
```

Рис. 11.38. Создание указателя на удаленный репозиторий Heroku

Если теперь перейти в свой аккаунт на сайте Heroku, то мы увидим созданный удаленный репозиторий и в нем ссылку на наше приложение: **my-world-book** (рис. 11.39).

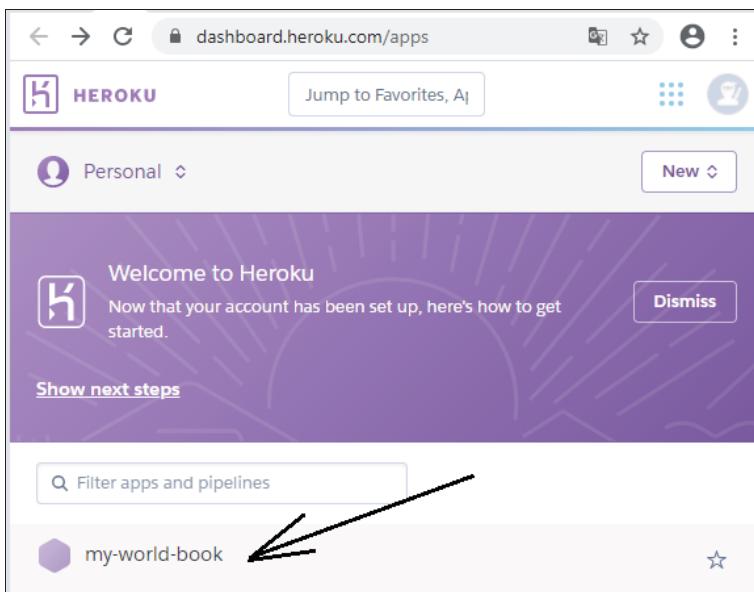


Рис. 11.39. Ссылка **my-world-book** на приложение в удаленном репозитории Heroku

Пройдя по ссылке **my-world-book**, мы попадем на страницу своего удаленного репозитория, внутри которого пока еще нет нашего приложения. И сейчас мы можем передать наше приложение в репозиторий Heroku, выполнив следующую команду (рис. 11.40):

```
git push heroku master
```

Теоретически приложение должно на сайте запуститься, но оно не будет работать должным образом, потому что мы еще не настроили таблицы базы данных для использования нашим приложением. Для этого нужно использовать команду `heroku run` и запустить среду `one off dyno` для выполнения операции переноса таблиц базы данных. Введите в окне терминала следующую команду:

```
heroku run python manage.py migrate
```

```
C:\Program Files\Git\django_world_book>git push heroku master
Enumerating objects: 79, done.
Counting objects: 100% (79/79), done.
Delta compression using up to 4 threads
Compressing objects: 100% (73/73), done.
Writing objects: 100% (79/79), 164.66 KiB / 6.86 MiB/s, done.
Total 79 (delta 15), reused 0 (delta 0), pack-reused 0
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: !     Python has released a security update! Please consider upgrading
to python-3.7.9
remote:     Learn More: https://devcenter.heroku.com/articles/python-runtimes
remote: -----> Installing python-3.7.8
remote: -----> Installing pip 20.1.1, setuptools 47.1.1 and wheel 0.34.2
remote: -----> Installing SQLite3
remote: -----> Installing requirements with pip
remote:     Collecting asgiref==3.2.10
remote:         Downloading asgiref-3.2.10-py3-none-any.whl (19 kB)
remote:     Collecting automaton==0.2.6
remote:         Downloading automaton-0.2.6.zip (29 kB)
remote:     Collecting automium-web==0.1.1
remote:         Downloading automium_web-0.1.1.zip (460 kB)
remote:     Collecting click==7.1.2
remote:         Downloading click-7.1.2-py2.py3-none-any.whl (82 kB)
remote:     Collecting dj-database-url==0.5.0
remote:         Downloading dj_database_url-0.5.0-py2.py3-none-any.whl (5.5 kB)
remote:     Collecting Django==3.0.9
remote:         Downloading Django-3.0.9-py3-none-any.whl (7.5 MB)
remote:     Collecting Flask==1.1.2
remote:         Downloading Flask-1.1.2-py2.py3-none-any.whl (94 kB)
remote:     Collecting gunicorn==20.0.4
remote:         Downloading gunicorn-20.0.4-py2.py3-none-any.whl (77 kB)
remote:     Collecting itsdangerous==1.1.0
remote:         Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
remote:     Collecting Jinja2==2.11.2
remote:         Downloading Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
remote:     Collecting legacy==0.1.6
remote:         Downloading legacy-0.1.6-py2.py3-none-any.whl (4.3 kB)
remote:     Collecting MarkupSafe==1.1.1
remote:         Downloading MarkupSafe-1.1.1-cp37-cp37m-manylinux1_x86_64.whl (27 kB)
remote:     Collecting psycopg2==2.8.5
remote:         Downloading psycopg2-2.8.5.tar.gz (380 kB)
remote:     Collecting pytz==2020.1
```

Рис. 11.40. Передача приложения на удаленный репозиторий Heroku

Мы также имеем возможность создать суперпользователя, снова применяя одноразовый динамический режим. Для этого в окне терминала выполните команду:

```
heroku run python manage.py createsuperuser
```

Как только этот процесс будет завершен, мы сможем посмотреть на наш сайт. Он должен работать, хотя в нем еще нет книг и другой информации в БД. Чтобы открыть браузер на новом удаленном веб-сайте, выполните в окне терминала команду:

```
heroku open
```

Сайт можно также открыть по ссылке с именем приложения. Для нашего примера это ссылка: <https://my-world-book.herokuapp.com/>.

11.4. Другие ресурсы для публикации веб-сайта

Кроме Heroku, в Интернете существует много других платных и бесплатных интернет-ресурсов (хостингов) для публикации веб-сайтов, написанных на Python и Django.

Например, сайт pythonanywhere.com. Фактически PythonAnywhere — это облачная платформа, предназначенная для запуска приложений, написанных на Python. Она дает возможность запускать и редактировать веб-приложения на основе распространенных веб-фреймворков Python, задействовать базы данных MySQL и PostgreSQL, онлайн-консоль Bash, веб-редактор кода. Платформа использует сервер веб-приложений, построенный на основе связки Nginx — мощного инструмента для развертывания веб-сервера и uWSGI — бинарного веб-сервера и сервера веб-приложений. На этом ресурсе можно не только опубликовать готовый сайт, но и выполнять редактирование его кода с использованием виртуального окружения, созданного под конкретную задачу.

Свой сайт можно развернуть и на хостинге reg.ru — для этого нужно следовать инструкциям по следующей ссылке: <https://www.reg.ru/support/hosting-i-servery/yazyki-programmirovaniya-i-skripty/kak-ustanovit-django-na-hosting>. Это российский хостинг, и все достаточно простые и понятные инструкции представлены здесь на русском языке.

Можно также развернуть свой и сайт на российском хостинге 1gb.ru. Соответствующие инструкции по публикации сайта доступны на его страницах.

11.5. Краткие итоги

В этой главе были рассмотрены вопросы публикации веб-сайта в сети Интернет. Эта процедура весьма сложная, но необходимая. Сложность ее заключается в том, что приходится вносить много изменений в настройки уже готового веб-приложения. Кроме того, требуется создать на удаленном сервере необходимое окружение, подгрузить в это окружение достаточно большое количество библиотек и дополнительных модулей, перенести базу данных. Хотя Django и берет на себя автоматизацию некоторых из этих процессов, все равно разработчик должен ряд настроек выполнить вручную, что требует определенных знаний и квалификации. Особенно много трудностей возникает у новичков при первой попытке публикации сайта. Поэтому рекомендуется получить навыки публикации сайта на достаточно простом приложении с минимальным количеством страниц и простейшей базой данных.

Послесловие

В этой книге мы рассмотрели только базовые приемы разработки веб-приложений, обеспечивающие создание основных элементов сайтов. Приводимые в ней примеры и шаблоны HTML-страниц были в целях лучшего их понимания сознательно максимально упрощены. Основная цель книги — познакомить ее читателей со структурой приложений на Django, принципами взаимодействий между элементами этого фреймворка, дать вам возможность понять сущность методов доступа к базам данных. Ведь Django позволяет обращаться к таблицам СУБД не напрямую через SQL-запросы, а в терминах класса. Работая с Django, можно создать класс со своим набором переменных и методов, которые потом будут переписаны в SQL автоматически. При этом необходимость самому писать SQL-код отпадает, хотя и не исключена полностью. Если после знакомства с приведенными в книге материалами вы почувствовали, что Python и Django прекрасно работают в среде PyCharm и что применение этого набора инструментов упрощает программистам разработку веб-приложений, то основная цель книги достигнута.

Следующим шагом в освоении веб-технологий является более детальное изучение инструментария для разработки и дизайна HTML-страниц и применения CSS-стилей, служащих основой для создания привлекательного пользовательского интерфейса. Кроме того, в книге охвачены далеко не все тонкости разработки веб-приложений. Достаточно актуальными являются вопросы удаленной загрузки и хранения изображений, создания географических веб-приложений. У Django есть прекрасное расширение — GeoDjango, а также библиотека Mapnik и геопространственное расширение PostGIS для СУБД PostgreSQL. Набор этих модулей позволяет максимально упростить создание географических веб-приложений и служб, основанных на местоположении. Если это издание окажется востребованным, то следующим шагом автора будет подготовка книги по технологическим приемам программирования геопространственных веб-приложений.

А в завершение остается пожелать вам удачного применения в своей практической деятельности тех навыков и знаний, которые вы приобрели, прочитав эту книгу.

Анатолий Постолит

Список источников и литературы

1. CSS — сделай это красиво. <https://tutorial.djangogirls.org/ru/css/>.
2. CSS шаблоны сайтов с использованием бестабличной верстки.
<http://ruseller.com/adds.php?rub=36>.
3. DjangoBook. <https://djbook.ru/index.html>.
4. Python. Урок 1. Установка. <https://devpractice.ru/python-lesson-1-install/>.
5. Python. Урок 16. Установка пакетов в Python.
<https://devpractice.ru/python-lesson-16-install-packages/>.
6. Python. Загрузите последнюю версию для Windows.
<https://www.python.org/downloads/>.
7. Web-программирование. Конспект лекций.
<http://edu.cassiopeia.com.ua/lib/conspectHTML.pdf>
8. Антонио Меле. Django 2 в примерах. Москва: издательство ДМК, 2019.
9. Бесплатный django хостинг. <https://uzverss.livejournal.com/69130.html>.
10. Ваша первая модель. <https://djbook.ru/ch05s06.html>.
11. Веб-платформа Django (Python).
<https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django>.
12. Васильева И. Н., Федоров Д. Ю. WEB-ТЕХНОЛОГИИ. Издательство Санкт-Петербургского Государственного Экономического Университета, 2014. — 67 с.
13. Дронов В. А. Django 3.0. Практика создания веб-сайтов на Python. Санкт-Петербург: издательство БХВ, 2021. — 704 с.
14. Документация Django 1.9. Поля формы.
https://ejudge.lksh.ru/lang_docs/djbook.ru/rel1.9/ref/forms/fields.html.
15. Документация Django 1.9. Справочник по полям модели.
<https://djbook.ru/rel1.9/ref/models/fields.html>.
16. Документация Django 3.0. https://djbook.ru/rel3.0/_
17. Документация Django. <https://docs.djangoproject.com/en/3.0/>.
18. Как сделать бегущую строку на сайте. <https://zarabotat-na-sajte.ru/uroki-html/kak-sdelat-begushhuyu-stroku-na-sajte.html>.

19. Как стать веб-разработчиком.
<https://eternalhost.net/blog/sozdanie-saytov/kak-stat-veb-razrabitchikom>.
20. Катастрофический дефицит. Цифровому прорыву предрекли острую нехватку IT-специалистов. https://www.dp.ru/a/2020/01/24/Katastroficheskij_deficit.
21. Легко настраивайте базу данных SQLite и управляйте ею.
<https://www.filehorse.com/download-sqlitestudio/>.
22. Модели. Django ORM.
https://ru.hexlet.io/courses/python-django-basics/lessons/models/theory_unit.
23. Основы JavaScript. <https://html5book.ru/osnovy-javascript/>.
24. Отношения между моделями в Django.
<http://sheregeda.github.io/blog/2015/01/30/models-in-django/>.
25. Постолит А. django_world_book на сайте [github.com](https://github.com/APostolit/django_world_book).
https://github.com/APostolit/django_world_book.
26. Прохоренок, Н. А. Python 3 и PyQt 5. Разработка приложений / Н. А. Прохоренок, В. А. Дронов. — СПб.: БХВ-Петербург, 2016. — 832 с.
27. Публикация сайта на PythonAnywhere.
<https://tomekgancarczyktyko.gitbooks.io/django/content/ru/deploy/>.
28. Публикация сайта на Python и Django на ресурсе [beget.com](https://beget.com/ru/articles/webapp_python?link=webapp_python#5).
https://beget.com/ru/articles/webapp_python?link=webapp_python#5.
29. Работа программистом Python: требования, вакансии и зарплаты.
<https://pythonru.com/baza-znanij/python-vakansii>.
30. Руководство по веб-фреймворку Django. <https://metanit.com/python/django/>.
31. Сайт для тех, кто изучает веб-технологии и создает сайты.
<https://html5book.ru/>.
32. Свойство CSS background.
<https://zarabotat-na-sajte.ru/uroki-html/kak-sdelat-fon-dlya-sajta.html>.
33. Создаем страницы для сайта на Django и публикуем на Heroku — Урок № 3.
<https://python-scripts.com/django-simple-site-heroku>.
34. Создание блога на Django 3 для начинающих.
<https://python-scripts.com/create-blog-django>.
35. Создание сайта на Python/Django: установка Django и создание проекта.
<https://igorosa.com/sozdanie-sajta-na-pythondjango-ustanovka-django-i-sozdanie-proekta/>.
36. Скачать PyCharm. <https://www.jetbrains.com/pycharm/download/>.
37. Таблица цветов HTML. <https://zarabotat-na-sajte.ru/uroki-html/html-cveta-i-kodi-cvetov.html>.
38. Тег <style> в HTML. <https://zarabotat-na-sajte.ru/uroki-html/style-v-html.html>.
39. Шаблоны Django. Наследование. <https://habr.com/ru/post/23132/>.

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив, сопровождающий книгу, выложен на FTP-сервер издательства «БХВ» по интернет-адресу: <ftp://ftp.bhv.ru/9785977567794.zip>. Ссылка на него доступна и со страницы книги на сайте <https://bhv.ru/>.

Архив содержит папку Listingi с исходными кодами (листингами программ), приведенными в главах 2–11 книги. Листинги программ находятся в папках, соответствующих номеру главы:

Глава 2

Листинг 2.1. Присвоение значений переменным

Листинг 2.2. Арифметические действия с переменными

Листинг 2.3–2.9. Пример описания и использования функции пользователя

Листинг 2.10–2.13. Примеры использования цикла if и while

Листинг 2.14–2.15. Пример использования цикла for для работы с массивами

Листинг 2.16–2.20. Пример создания класса Cat

Листинг 2.21–2.23. Пример создания класса и описания методов класса Car

Глава 3

Листинг 3.1. Пример создания представления (view) для приложения

Листинг 3.2. Пример создания функции index для приложения

Глава 4

Листинг 4.1. Пример использования файла views.py приложения firstapp

Листинг 4.2. Пример использования файла urls.py приложения firstapp

Листинг 4.3–4.5. Пример маршрутизации запросов пользователей в приложении firstapp

Листинг 4.6. Пример создания дополнительных функций в приложении firstapp

Листинг 4.7. Пример сопоставления функций с запросами пользователей в приложении firstapp

Листинг 4.8. Пример определения значений параметра по умолчанию в функциях приложения firstapp

Листинг 4.9. Пример определения маршрутов в файле urls.py

Листинг 4.10. Пример описания функций в файле views.py

Листинг 4.11. Пример определения параметров в файле urls.py с помощью функции path()

Листинг 4.12. Пример модификации файла views.py

Листинг 4.13–4.14. Пример модификации файла urls.py

Листинг 4.15. Пример определения в файле views.py функции products() и users()

Листинг 4.16. Пример изменения маршрутов в файле urls.py

Листинг 4.17. Пример переадресации в файле views.py

Листинг 4.18. Пример изменения маршрутов в файле urls.py

Глава 5

Листинг 5.1. Программный код пустого шаблона страницы index.html

Листинг 5.2. Пример модификации программного кода шаблона страницы index.html

Листинг 5.3. Пример программного кода функции для обработки запроса пользователя

Листинг 5.4. Пример сопоставления функции index с запросом пользователя

Листинг 5.5. Пример внесения изменений в содержание файла home.html

Листинг 5.6. Пример внесения изменений в содержание функции index()

Листинг 5.7. Пример использования класса TemplateResponse

Листинг 5.8. Пример использования функции render()

Листинг 5.9. Пример шаблона страницы index_app1.html

Листинг 5.10. Пример изменения содержания функции defindex()

Листинг 5.11. Пример передачи пользователю сложных данных через шаблон

Листинг 5.12. Программный код шаблона templates\index.html

Листинг 5.13. Пример изменения содержания функции defindex()

Листинг 5.14–5.18. Пример использования стилей в шаблонах HTML-страниц

Листинг 5.19. Пример содержания файла стилей styles.css

Листинг 5.20. Пример использования стилей в шаблонах HTML-страниц

Листинг 5.21. Описание пути к статичным файлам в модуле settings.py

Листинг 5.22. Пример размещения рисунка в шаблоне HTML-страницы

Листинг 5.23. Пример содержания файла стилей styles.css

Листинг 5.24. Описание простейших шаблонов в папке hello\templates\firstapp\

Листинг 5.25. Содержание файла-шаблона contact.html

Листинг 5.26. Содержание изменения файла urls.py

Листинг 5.27. Содержание изменения файла urls.py

Листинг 5.28. Содержание файла-шаблона contact.html

Листинг 5.29–5.31. Внесение изменений в файл setting.py

Листинг 5.32. Пример создания базового шаблона base.html

Листинг 5.33. Пример шаблона главной страницы сайта index.html

Листинг 5.34. Пример изменения код функции defindex()

Листинг 5.35. Пример шаблона страницы сайта about.html

Листинг 5.36. Пример использования тегов в шаблоне страницы сайта about.html

Листинг 5.37–5.38. Пример использования тегов

Листинг 5.39–5.40. Пример использования тегов для вывода информации в цикле

Листинг 5.41. Пример использования тегов

Листинг 5.42. Пример использования тегов для задания значений переменным

Глава 6

Листинг 6.1. Пример создания формы

Листинг 6.2. Пример обращения к форме в функции index()

Листинг 6.3. Пример изменений в шаблон index.html для обращения к форме

Листинг 6.4. Пример изменений содержимого файла forms.py

Листинг 6.5. Пример изменений содержимого шаблона index.html

Листинг 6.6. Пример изменений кода функции index()

Листинг 6.7. Пример кода файла шаблона для главной страницы firstapp/index.html

Листинг 6.8. Пример изменений кода функции index()

Листинг 6.9. Пример изменений кода файла forms.py

Листинг 6.10. Пример создания поля forms.BooleanField

Листинг 6.11. Пример использования метода forms.BooleanField

Листинг 6.12. Пример использования метода forms.NullBooleanField

Листинг 6.13. Пример использования метода forms.CharField

Листинг 6.14. Пример использования метода forms.EmailField

Листинг 6.15. Пример использования метода forms.GenericIPAddressField

Листинг 6.16. Пример использования метода forms.RegexField

Листинг 6.17. Пример использования метода forms.SlugField

Листинг 6.18. Пример использования метода forms.URLField

Листинг 6.19. Пример использования метода forms.UUIDField

Листинг 6.20. Пример использования метода forms.ComboField

Листинг 6.21. Пример использования метода forms.FilePathField

Листинг 6.22. Пример использования метода forms.FilePathField

Листинг 6.23. Пример использования метода forms.FileField

Листинг 6.24. Пример использования метода forms.ImageField

Листинг 6.25. Пример использования метода forms.DateField

Листинг 6.26. Пример использования метода forms.TimeField

Листинг 6.27. Пример использования метода forms.DateTimeField

Листинг 6.28. Пример использования метода forms.DurationField

Листинг 6.29. Пример использования метода `forms.SplitDateTimeField`

Листинг 6.30. Пример использования метода `forms.IntegerField`

Листинг 6.31. Пример использования метода `forms.DecimalField`

Листинг 6.32. Пример использования метода `forms.DecimalField`

Листинг 6.33. Пример использования метода `forms.FloatField`

Листинг 6.34. Пример использования метода `forms.ChoiceField`

Листинг 6.35. Пример использования метода `forms.TypedChoiceField`

Листинг 6.36. Пример использования метода `forms.MultipleChoiceField`

Листинг 6.37. Пример использования метода `forms.TypedMultipleChoiceField`

Листинг 6.38. Пример изменения внешнего вида поля с помощью `widget`

Листинг 6.39. Пример изменения внешнего вида поля с помощью `widget`

Листинг 6.40. Пример задания начальных значений полей с помощью свойства `initial`

Листинг 6.41. Пример задания порядка следования полей на форме

Листинг 6.42. Пример задания порядка следования полей на форме

Листинг 6.43. Пример задания порядка следования полей на форме

Листинг 6.44. Пример задания подсказок к полям формы

Листинг 6.45. Пример задания подсказок к полям формы

Листинг 6.46. Пример настройки вида формы

Листинг 6.47–6.51. Пример проверки (валидации) данных

Листинг 6.52–6.54. Пример детальной настройки полей формы

Листинг 6.55–6.60. Пример стилизации полей формы

Глава 7

Листинг 7.1. Создания модели данных на простом примере

Листинг 7.2–7.5. Пример работы с объектами модели данных
(чтение и запись информации в БД)

Листинг 7.6–7.10. Пример работы с объектами модели данных
(редактирование и удаление информации из БД)

Листинг 7.11–7.12. Пример организации связей между таблицами в модели данных

Листинг 7.13. Пример организации связей между таблицами «многие-ко-многим»

Листинг 7.14. Пример организации связей между таблицами «один-к-одному»

Глава 8

Листинг 8.1. Программный код создания главной страницы сайта «Мир книг»

Листинг 8.2. Программный код файла `urls.py` сайта «Мир книг»

Листинг 8.3–8.4. Программный код модели для хранения языков книг

Листинг 8.5. Программный код модели для хранения авторов книг

Листинг 8.6. Программный код модели для хранения книг

Листинг 8.7–8.8. Программный код модели для хранения экземпляров книг

Листинг 8.9–8.11. Программный код настройки отображения списков в административной панели сайта

Листинг 8.12. Программный код добавления фильтров к спискам в административной панели сайта

Листинг 8.13. Программный код формирования макета с подробным представлением элемента списка в административной панели сайта

Листинг 8.14. Программный код разделения страницы на секции с отображением связанной информации в административной панели сайта

Листинг 8.15. Пример встроенного редактирования связанных записей в административной панели сайта

Глава 9

Листинг 9.1–9.4. Пример создания URL-преобразования

Листинг 9.5–9.8. Пример создания базового шаблона сайта и шаблона для главной страницы сайта «Мир книг»

Листинг 9.9. Пример отображения списков и детальной информации об элементе списка

Листинг 9.10–9.21. Пример отображения списков и детальной информации об элементе списка

Глава 10

Листинг 10.1–10.2. Пример организации сессий

Листинг 10.3–10.6. Пример аутентификации и авторизации пользователей в Django

Листинг 10.7–10.12. Пример создания страницы для сброса пароля пользователя

Листинг 10.13. Пример создания страницы для проверки подлинности входа пользователя в систему

Листинг 10.14–10.21. Пример формирования страниц сайта для создания заказов на книги

Листинг 10.22. Пример создания формы

Листинг 10.23–10.32. Пример создания формы для ввода и обновления информации об авторах книг на основе класса `Form`

Листинг 10.33–10.40. Пример создания формы для ввода и обновления информации об авторах книг на основе класса `ModelForm`

Глава 11

Листинг 11.1–11.3. Пример подготовки веб-сайта к публикации

Листинг 11.4. Пример изменения кода при создании репозитория приложения на `Github`

Листинг 11.5. Пример изменения кода при подключении веб-сервера `Gunicorn`

Листинг 11.6–11.8. Пример использования пакетов для обеспечения доступа к базе данных на `Heroku`

Предметный указатель

C

Cross Site Request Forgery, CSRF 392, 418
CSS-класс 227

D

Django ORM 232
DOU (Direct Observation Unit,
Блок прямого наблюдения за рейтингами
языков программирования) 56

G

GET-запрос 402
GitHub 88, 422
Git-репозиторий 421

H

HTML-документы 16, 19
HTML-форма 283, 392, 394, 396
HTML-шаблон 89
HTTP-запрос 89
HTTP-сервер 432

I

IaaS, Infrastructure as a Service
(Инфраструктура как Сервис) 416
IDE, Integrated Development Environment 14
IP-адрес 189
ISBN (International Standard Book Number,
международный стандартный книжный
номер) 281

O

Object-Relational Mapping (ORM) 232

P

PaaS, Platform as a Service (Платформа
как Сервис) 417
POST-запрос 221, 392, 402

S

SQL-запрос 231, 245, 414

U

URL-mapper 89
URL-преобразование 330, 346, 349, 356
URL-преобразования (маршруты) 329

A

- Аббревиатура CRUD 240
- Автоматическое тестирование приложения 414
- Авторизация 390
 - ◊ пользователей 270, 363, 364
- Административная панель 317
 - ◊ Django 269, 304, 314, 328, 329, 392
 - ◊ Django Admin 303
 - ◊ сайта 364, 365
- Административный терминал Git 441, 446
- Аккаунт суперпользователя 365
- Аргументы функции 65
- Архитектура
 - ◊ Model-View-Template (MVT) 89
 - ◊ Model-View-Controller (MVC) 89
- Атака CSRF (Cross-Site Request Forgery) 177
- Атрибут
 - ◊ exclude 322
 - ◊ fields 321, 322
 - ◊ fieldsets 322
 - ◊ list_filter 317
 - ◊ style 143
- Атрибуты (параметры) тегов 20
- Аутентификация 357, 358, 365
 - ◊ пользователя 363

Б

- База данных
 - ◊ MySQL 449
 - ◊ Postgres 433, 435
 - ◊ PostgreSQL 449
 - ◊ SQLite 233, 248, 263, 275, 433–435
- Базовые блоки веб-приложения на Django 90
- Базовый
 - ◊ класс models.Model 289
 - ◊ шаблон 160–163, 350, 371, 398
- Библиотека
 - ◊ Django 51, 52, 270
 - ◊ SQLite 98
 - ◊ WhiteNoise 437
- Боковая навигационная панель (главное меню) 330

В

- Варианты отображения полей на форме 216
- Ввод
 - ◊ IP-адреса 189

- ◊ временного промежутка 201
- ◊ даты 198
- ◊ даты и времени 201
- ◊ десятичных чисел 204
- ◊ значений времени 200
- ◊ текста 186
- ◊ текста с проверкой соответствия заданным форматам 193
- ◊ текста типа «slug» 190
- ◊ текста, соответствующего регулярному выражению 189
- ◊ универсального указателя ресурса (URL) 191
- ◊ универсального уникального идентификатора UUID 192
- ◊ целых чисел 203
- ◊ чисел с плавающей точкой 206
- ◊ электронного адреса 187
- Веб-сайт 15, 16
- Веб-сервер Django 415
- Веб-сервер Gunicorn 432, 433
- Веб-страница 15, 21
- Веб-формы 17
- Веб-фреймворк Django 15, 19, 49, 51–53, 55, 86, 88, 270
- Виджет 181, 184
 - ◊ Textarea 212
 - ◊ TextInput 212
 - ◊ Календарь 397
 - ◊ по умолчанию 181
 - ◊ формы 284
- Виджеты 171, 181, 321, 392
- Виды стилей CSS 143
- Внешний ключ 316
- Внешняя таблица стилей 147
- Внутренние стили 143, 145, 146
- Восстановление доступа к аккаунту пользователя 375
- Встроенные стили 143, 145, 146
- Встроенный календарь 404
- Вход пользователя в систему 382
- Выбор
 - ◊ вложенных каталогов 195
 - ◊ данных из списка 206, 208
 - ◊ и загрузка файлов 196
 - ◊ и загрузка файлов изображений 198
 - ◊ файла из предложенного списка 194, 195
- Выход изображения 151

Г

Гвидо Ван Россум 14

Гиперссылки 29

Главная

◊ модель 256, 260

◊ страница сайта 382, 404

◊ таблица 292, 293

◊ таблица БД 256, 258

Главное меню 335

◊ сайта 330, 410

Д

Движок БД SQLite 98, 276

Действия с переменными 62

Декоратор @register 315

Динамический сайт 16, 17

Диспетчер URL-адресов (URLs) 89

Дистрибутив Python 34

Добавление данных в БД 241

Домашняя страница сайта 372

Дочерняя таблица 292, 293

◊ в БД 258

Дружественные URL-адреса («стабы»)

349

З

Зависимая модель 256, 260

Зависимая таблица БД 256

Заголовок документа 19

Запрос

◊ GET 250

◊ get_queryset() 389

◊ HttpRequest 334

◊ POST 246, 250

Защита

◊ от межсайтовой подделки запроса 171

◊ приложения от CSRF-атак 176

И

Идентификаторы полей 173

Идентификация пользователя 371

Изображения 151

Имя поля 283

Индекс TIOBE 56

Инструментальное средство PyCharm 57

Интегрированная среда разработки 14

◊ Python 35

Интегрирующий объект QuerySet 288

Интерактивная среда разработки IDE

PyCharm 39

Интерактивная среда разработки

программного кода PyCharm 55

Интернет-браузер 16, 21

Интерпретатор языка Python 15, 34, 38, 49,

51, 55, 58, 83

Интерфейс PyCharm 433

Инфраструктура как Сервис (IaaS) 416

Использование сессий 361

К

Календарь 307

Каскадное удаление 256, 258

Каскадные таблицы стилей (Cascading Style Sheets, CSS) 15, 16, 30, 142, 147

Класс 75, 102

◊ forms.Form 172

◊ HttpResponse (ответ HTTP) 106

◊ HttpResponseRedirect() 278

◊ inlines 324

◊ ListView 343

◊ LoginRequiredMixin 388

◊ Meta 286, 407

◊ ModelAdmin 315

◊ reverse 289

◊ HttpResponseRedirect 127, 136, 141

◊ TemplateView 153

◊ формы Form() 396, 406, 409

◊ формы ModelForm() 406, 408, 409

Классы 31

Клиент Heroku 442

Ключевое поле (id) 284

Ключевые поля БД 258

Код

◊ CSS 339

◊ JavaScript 33, 337, 339, 393

Комментарии 63

Конструктор

◊ models.ForeignKey 256

◊ models.ManyToManyField() 262

◊ models.OneToOneField() 266

Концепция Model-View-Controller (MVC) 87

Кортеж list_display 316

Л

Логика проверки ввода 181

M

- Маркер 24
- Маркированные списки 24
- Маршрутизация запросов 105, 126
- Маршруты 107, 111, 121
- Массивы 57, 69, 74
- Менеджер
 - ◊ SQLiteStudio 53, 54, 248, 303, 311
 - ◊ баз данных SQLite 53
 - ◊ пакетов pip 15, 45, 46
 - ◊ работы с базами данных SQLiteStudio 55
- Метаданные 285, 286
- Метка
 - ◊ для поля 175
 - ◊ поля 283
- Метки (placeholders) 334
- ◊ полей 173
- Метод
 - ◊ admin.site.register() 315
 - ◊ as_view() 343, 410
 - ◊ EmailValidator 188
 - ◊ filter() 288
 - ◊ GET 402
 - ◊ get_absolute_url() 295
 - ◊ is_valid() 220
 - ◊ objects.all() 288
 - ◊ objects.all().count() 334
 - ◊ POST 399, 401, 402
 - ◊ RegexValidator 190
 - ◊ save() 287, 401
 - ◊ URLValidator 192
 - ◊ validate_slug 191
 - ◊ validate_unicode_slug 191
 - ◊ ValidationError 186
- Методы 76, 286
 - ◊ объекта 82
- Механизм
 - ◊ валидации 217
 - ◊ переадресации 123
- Миграция 235, 246, 257, 258, 267, 274, 363
 - ◊ данных 297, 317
- Множественный выбор данных из списка 209, 210
- Модели (Models) 90, 280, 306
- Модели
 - ◊ авторизации 363
 - ◊ данных 231, 257, 274, 303, 314, 328, 329, 342, 407

Модель 232

- ◊ данных 283, 294, 295, 318
- ◊ данных Django 233
- Модули 82
- Модуль 103
 - ◊ psycopg2 435, 436, 439
 - ◊ whitenoise 438

Н

- Наименование класса 77
- Настройка
 - ◊ DEBUG 419
 - ◊ SECRET_KEY 418
 - ◊ шаблонов 128
- Несвязанная форма 395
- Нумерованные списки 24

О

- Облачные решения 416
- Облачный сервис Heroku 416, 420, 421, 433, 435, 437, 442
- Обновление объекта в БД 243
- Обобщенные базовые классы визуализации данных 342
- Обобщенные классы 392, 407
 - ◊ показа списка 342
 - ◊ редактирования форм 414
 - ◊ создания страниц 407
- Обобщенный класс
 - ◊ отображения списка 343
 - ◊ отображения списков 352
- Общие аргументы полей моделей 284
- Объект 78, 281
 - ◊ HttpResponseRedirect (ответ HTTP) 107, 178
 - ◊ QuerySet 241, 243, 288
 - ◊ request (запрос) 106, 107
 - ◊ UploadedFile 196, 198
 - ◊ запроса пользователя 102
 - ◊ класса 75
 - ◊ формы 173
- Объектно-ориентированный язык программирования
 - ◊ JavaScript 32
 - ◊ Python 56
- Объекты 74, 232
 - ◊ Python 90
 - ◊ модели данных 245

Объявление 142
 ◇ стиля 142
 Одноразовый динамический режим 448
 Окно
 ◇ администрирования сайта 306
 ◇ терминала PyCharm 91
 Окружение развертывания 416
 Основные способы доступа к данным
 модели 280
 Основные типы полей таблиц 280
 Ответ (response) 106
 Отключение атрибута required 219
 Отношение
 ◇ «многие-ко-многим» 285, 297
 ◇ «один-ко-многим» 285, 289
 Отношения между моделями 281
 Очистка данных 395

П

Пакетный менеджер pip 35
 Панель
 ◇ администратора сайта 305, 365, 372
 ◇ администрирования Django 322
 ◇ администрирования сайта 306, 312
 ◇ календаря 328
 Параметр 139
 ◇ DATABASES 233
 ◇ ENGINE 233
 ◇ label 175
 ◇ NAME 233
 ◇ page_obj 354
 ◇ request 359
 ◇ STATIC_URL 339
 ◇ success_url 408
 ◇ widget 211
 Параметры
 ◇ корректности вводимых данных 218
 ◇ строки запроса 121
 ◇ функции 64
 ◇ передаваемые через
 ▫ интернет-адрес 120
 ▫ строку запроса 120
 Пароли пользователей 380
 Пароль суперпользователя 386
 Первичный ключ 261, 266, 294, 295,
 297, 410
 ◇ в таблице базы данных 236
 ◇ для модели 284

Переадресация 123, 124
 ◇ временная 123
 ◇ постоянная 123
 Передача в шаблон
 ◇ простых данных 137
 ◇ сложных данных 140
 Переменная
 ◇ collectstatic 436, 437
 ◇ context 362
 ◇ session 359
 ◇ STATIC_URL 436
 ◇ TEMPLATES 158
 ◇ urlpatterns (шаблон URL) 107
 Переменные 62– 64, 66, 67, 74
 Платформа как Сервис (PaaS) 417
 Подключение модуля 84
 Подсказки 216
 Поиск хостинга 415
 Поле
 ◇ exclude 408
 ◇ fields 408
 Пользовательская функция 66
 Пользовательские
 ◇ аккаунты 357
 ◇ функции 64
 Поля формы 172, 181, 408
 Порядок следования полей на форме 213
 Постоянная переадресация 123
 Постраничный вывод (Pagination) 352
 Постраничный
 ◇ вывод длинных списков 357
 ◇ показ данных 342
 Представление (view) 153, 246, 314, 331, 333,
 343, 357, 362, 386, 394, 408
 Представление (View) 89, 90
 Представления (views) 105, 126, 137, 274,
 278, 280, 329, 369
 Предупреждение об ошибке в PyCharm 174
 Приложение
 ◇ «Администратор сайта» 279
 ◇ Django 273
 ◇ Django Admin 303, 383
 ◇ PyCharm 270, 272, 273
 ◇ SQLiteStudio 237, 253
 Приоритет встроенного стиля над
 внутренним 146
 Присваивание 62
 Присвоение стилей полям формы 226, 227,
 230
 Пробные бесплатные тарифы 417

Проверка
 ◇ введенных значений на стороне клиента 220
 ◇ данных 395
 ◇ корректности данных на стороне сервера 220
 Программа SQLiteStudio 236
 Программная оболочка PyCharm 57, 130, 147
 Проект
 ◇ Django 91
 ◇ Open Source 88
 ◇ Python 44
 Протокол HTTP 359
 Публичный веб-сервер 415

P

Разграничение уровней доступа 357, 358
 Регистрация приложения в проекте Django 275
 Регулярное выражение 110, 115, 189, 194, 343, 347
 Регулярные выражения 110, 111, 357
 Редактирование объектов модели 250, 253
 Реляционная база данных SQLite 98
 Репозиторий
 ◇ Git 440
 ◇ GitHub 422
 ◇ Heroku 422, 447
 ◇ Python Package Index (PyPI) 46
 Родительская таблица 292, 293
 ◇ в БД 258, 259

C

Свойство
 ◇ help_text 215
 ◇ initial 213
 ◇ формы error_css_class 227
 ◇ формы required_css_class 227
 Связанная таблица 292, 293
 Связанная форма 395
 Связанные данные 259
 Связующая таблица 264, 293
 Связывание маршрутов с представлениями 247
 Связь
 ◇ «многие-ко-многим» 261, 263, 265, 293, 310, 311
 ◇ «один-к-одному» 265, 266
 ◇ «один-ко-многим» 255, 265, 292, 296, 310, 316, 384
 Селектор 142

Серверная часть системы 394
 Сертификат SSL 417
 Сессии 357, 358, 359
 Синтаксис регулярных выражений 348
 Система
 ◇ CDN (Content Delivery Network) 436
 ◇ аутентификации и авторизации Django 363
 ◇ сброса пароля 375
 ◇ смены пароля 378
 ◇ управления базами данных (СУБД) 232
 Системные функции 64, 65
 Скриптовый язык программирования JavaScript 15, 18
 Скрипты
 ◇ Django 91
 ◇ JavaScript 142, 147
 Смена пароля пользователя 377, 379, 380
 Содержимое таблицы БД 248
 Соединительная таблица 261
 Создание нового проекта PyCharm 57
 Сопоставление URL-адресов 342
 Специальные теги 163
 Списки определений 25
 Среда разработки программного кода PyCharm 15, 49
 Статические ресурсы 339
 Статические файлы 142, 151, 436
 Статический сайт 16
 Статусные коды 125
 Страница
 ◇ администратора сайта 278
 ◇ аутентификации 369
 ◇ выхода из системы 374
 ◇ идентификации пользователя 382
 ◇ профиля пользователя 371
 Структура проекта Django 92
 Суперпользователь 303–306, 328, 364
 Схема Model-View-Controller (MVC) 87
 Сценарии JavaScript 32
 Счетчик посещений домашней страницы 358

T

Таблица базы данных 235, 246, 268, 283
 Таблицы 26
 Таблицы-«источники» 261
 Тег label 393
 Теги 15, 19, 22–25, 32
 ◇ HTML 21
 Текст «slug» 190

Тело документа 19
 Терминал
 ◊ PyCharm 235, 236, 257, 258, 262, 266, 272, 273, 297, 303, 304, 379, 385, 434, 436
 ◊ администратора Git 425–427, 430
 Тестирование приложения 274
 Тип запроса
 ◊ GET 177
 ◊ POST 178
 Типовой базовый шаблон 335
 Типовые модульные сетки 28
 Типы
 ◊ запросов 177
 ◊ отношений в БД 255
 ◊ полей в модели данных Django 237
 ◊ полей в различных СУБД 239, 240
 ◊ полей в формах Django 179

У

Удаление
 ◊ данных из БД 245
 ◊ объектов модели 253
 Универсальный указатель ресурса (URL) 191
 Универсальный уникальный идентификатор UUID 192
 Условия 70
 Установка модуля 83
 Учетная запись superuser (суперпользователь) 304

Ф

Файл миграции 236, 298, 301, 385
 Файлы cookie 87, 359, 360
 Форма 171, 371, 400, 404
 ◊ HTML 392, 393
 Формы 170, 178, 231, 270, 303
 ◊ аутентификации 363
 Фреймворк
 ◊ Bootstrap 337
 ◊ SQLite Studio 15
 Функции 62, 64, 68, 78
 Функции-представления 113
 Функция 102
 ◊ «обратного просмотра» (reverse lookup) 351
 ◊ path 111
 ◊ path() 109, 117, 118, 120
 ◊ re_path 118
 ◊ re_path() 110
 ◊ render (предоставлять) 132

◊ render() 127, 136, 334, 343
 ◊ reverse_lazy() 408
 ◊ url() 343, 347
 ◊ view() 343

Х

Хостинг-провайдер 416, 417
 Хэшированный пароль 87
 Хеш-функция 87

Ц

Цикл 72
 ◊ for 74, 398
 ◊ while 73

Ч

Чтение данных из БД 241

Ш

Шаблон 334
 ◊ HTML-страницы 408
 ◊ URL-адреса 342
 ◊ модель.поле 345
 ◊ регулярного выражения 194
 ◊ страницы 246
 Шаблоны (Templates) 90, 127
 ◊ URL-адресов 410
 ◊ преобразования URL-адресов 357
 ◊ регулярных выражений (URL-шаблоны) 348

Э

Электронный адрес 187
 Элемент input типа type="submit" 393
 Эфемерная файловая система 420, 433

Я

Язык
 ◊ SQL 232, 240
 ◊ программирования Perl 17, 18, 19
 ◊ программирования Python 14, 15, 18, 19, 34, 36, 38, 39, 46, 47, 52, 55, 69, 86
 ◊ разметки гипертекста HTML (Hyper Text Markup Language) 15, 16, 19, 392, 393
 ◊ создания шаблонов Django 335
 ◊ сценариев общего назначения PHP 17, 18, 19