



Узнайте Django 2, создав 4 проекта с нуля

Если вы хотите узнать о полном цикле разработки профессионального веб-приложения с Django, эта книга – для вас.

Вы реализуете блог, социальную сеть, интернет-магазин и платформу для онлайн-обучения. Научитесь применять AJAX и создадите RESTful API, а также узнаете, как настроить Django-проект на работу в боевом режиме.

Эта книга проведет вас по пути создания полезных приложений и познакомит с лучшими практиками решения типовых проблем при разработке. После ее прочтения вы будете понимать, как работает фреймворк Django и как с его помощью создавать продвинутые веб-приложения.

Вы научитесь:

- создавать Django-приложения, пригодные для многократного использования;
- добавлению продвинутых функций, оптимизации кода, использованию кеширования;
- интернационализации и реализации переводов в Django-проектах;
- улучшению пользовательского опыта при работе с сайтом с помощью JavaScript и AJAX;
- интеграции со сторонними социальными сетями;
- интегрировать в Django-проекты возможности других приложений, таких как Redis и Celery;
- работать с RESTful API.

Для максимально эффективного изучения вам необходимо иметь базовые знания Python, HTML и JavaScript, но вы можете быть новичком в Django и даже ни разу до этого момента не сталкиваться с этим фреймворком.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@aliants-kniga.ru

Packt
ДМК
ИЗДАТЕЛЬСТВО
www.dmk.pф

ISBN 978-5-97060-746-6



9 785970 607466 >

Django 2 в примерах

Антонио Меле

Django 2 в примерах

Создавайте мощные и надежные
веб-приложения на Python с нуля

ДМК
ИЗДАТЕЛЬСТВО

Антонио Меле

Django 2 в примерах

Antonio Melé

Django 2 by Example

Build powerful and reliable Python web applications from scratch

Packt
BIRMINGHAM - MUMBAI

Антонио Меле

Django 2 в примерах

Создавайте мощные и надежные веб-приложения Python с нуля



Москва, 2019

УДК 004.42
ББК 32.972
М47

Меле А.

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

ISBN 978-5-97060-746-6

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

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

Издание будет полезно всем разработчикам приложений.

УДК 004.42
ББК 32.972

Authorized Russian translation of the English edition of Django 2 by Example ISBN 9781788472487 © 2018 Packt Publishing.

This translation is published and sold by permission of Packt Publishing, which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

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

© 2018 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2019

Посвящается моей сестре

Содержание

| | |
|--|----|
| Об авторе | 12 |
| О рецензентах | 13 |
| Предисловие | 14 |
| Глава 1. Создание приложения блога | 19 |
| Установка Django | 19 |
| Создание изолированного Python-окружения | 20 |
| Установка Django через pip | 21 |
| Создание первого проекта | 21 |
| Запуск сервера для разработки | 23 |
| Настройки проекта | 24 |
| Проекты и приложения | 25 |
| Создание приложения | 25 |
| Проектирование схемы данных для блога | 26 |
| Активация приложения | 28 |
| Создание и применение миграций | 28 |
| Создание сайта администрирования | 30 |
| Сайт администрирования Django | 30 |
| Добавление собственных моделей на сайт администрирования | 31 |
| Настройка отображения моделей | 33 |
| Работа с QuerySet и менеджерами | 35 |
| Создание объектов | 35 |
| Изменение объектов | 36 |
| Получение объектов | 36 |
| Удаление объектов | 37 |
| Когда выполняются запросы QuerySet'ов | 37 |
| Создание менеджера модели | 38 |
| Обработчики списка статей и страницы подробностей | 38 |
| Создание обработчиков списка и страницы подробностей | 39 |
| Добавление шаблонов URL'ов для обработчиков | 40 |
| Канонические URL'ы для моделей | 41 |
| Создание HTML-шаблонов для обработчиков | 41 |
| Добавление постраничного отображения | 44 |
| Использование обработчиков-классов | 46 |
| Резюме | 47 |
| Глава 2. Добавление продвинутых функций в блог | 48 |
| Функция «Поделиться статьей через e-mail» | 48 |
| Создание Django-форм | 48 |

| | |
|---|----|
| Обработка данных формы..... | 49 |
| Отправка электронной почты с Django | 51 |
| Отображение форм в HTML-шаблонах..... | 53 |
| Добавление подсистемы комментариев..... | 56 |
| Создание модельных форм | 58 |
| Обработка модельных форм | 58 |
| Добавление комментариев в шаблон статьи | 60 |
| Добавление подсистемы тегов | 63 |
| Формирование списка рекомендованных статей..... | 68 |
| Резюме..... | 70 |

Глава 3. Расширение приложения блога 71

| | |
|--|----|
| Создание шаблонных тегов и фильтров | 71 |
| Создание собственных тегов..... | 71 |
| Создание собственных фильтров..... | 76 |
| Добавление карты сайта | 78 |
| Добавление RSS для статей | 81 |
| Добавление полнотекстового поиска..... | 83 |
| Установка PostgreSQL..... | 83 |
| Простые поисковые запросы | 84 |
| Поиск по нескольким полям | 85 |
| Обработчик поиска | 85 |
| Стемминг и ранжирование результатов | 87 |
| Взвешенные запросы..... | 88 |
| Поиск с помощью триграмм | 89 |
| Другие инструменты полнотекстового поиска..... | 90 |
| Резюме..... | 90 |

Глава 4. Создание социальной сети 91

| | |
|--|-----|
| Создание проекта для социальной сети | 91 |
| Запуск проекта | 91 |
| Использование системы аутентификации Django..... | 92 |
| Создание обработчика авторизации | 93 |
| Использование обработчиков аутентификации Django | 98 |
| Обработчики входа и выхода | 98 |
| Обработчики смены пароля | 103 |
| Обработчики восстановления пароля | 105 |
| Регистрация и профили пользователей..... | 109 |
| Регистрация пользователей | 109 |
| Расширение модели пользователя | 113 |
| Подключение системы уведомлений | 118 |
| Реализация бэкэнда аутентификации | 120 |
| Подключение аутентификации через соцсети..... | 122 |
| Аутентификация Facebook..... | 123 |
| Аутентификация Twitter..... | 128 |
| Аутентификация Google | 130 |
| Резюме..... | 134 |

| | |
|---|-----|
| Глава 5. Совместное использование содержимого сайта | 135 |
| Сохранение изображений в закладки на сайте | 135 |
| Создание модели изображения | 136 |
| Добавление отношения «многие ко многим» | 137 |
| Регистрация модели изображения на сайте администрирования | 138 |
| Использование изображений с других сайтов | 138 |
| Валидация полей формы | 139 |
| Переопределение метода save() модельной формы | 140 |
| Букмарклет на jQuery | 143 |
| Создание обработчика для картинки | 151 |
| Добавление превью для изображений | 153 |
| Реализация AJAX-запросов с jQuery | 154 |
| Подключение jQuery | 156 |
| Защита от межсайтовых запросов в AJAX | 156 |
| Выполнение AJAX-запросов с jQuery | 158 |
| Создание собственных декораторов | 160 |
| Постраничный вывод с помощью AJAX | 161 |
| Резюме | 165 |
| Глава 6. Отслеживание действий пользователей | 166 |
| Реализация системы подписок | 166 |
| Отношение «многие ко многим» с промежуточной моделью | 166 |
| Создание обработчиков списка пользователей и подробностей профиля | 169 |
| AJAX-обработчик для создания подписчика | 173 |
| Добавление новостной ленты | 175 |
| Использование подсистемы типов содержимого | 176 |
| Добавление обобщенных отношений | 177 |
| Устранение дублирования новостей в ленте | 179 |
| Добавление активности в новостную ленту | 180 |
| Отображение ленты новостей | 181 |
| Оптимизация QuerySet'a со связанными объектами | 181 |
| Создание шаблонов для новостной ленты | 182 |
| Использование сигналов Django | 184 |
| Работа с сигналами | 184 |
| Конфигурационные классы приложений | 186 |
| Использование Redis для хранения представлений объектов | 188 |
| Установка Redis | 188 |
| Использование Redis в Python-коде | 190 |
| Сохранение представлений объектов в Redis | 191 |
| Хранение рейтинга объектов в Redis | 192 |
| Следующие шаги с Redis | 194 |
| Резюме | 195 |
| Глава 7. Создание онлайн-магазина | 196 |
| Создание проекта | 196 |
| Добавление моделей каталога товаров | 197 |

| | |
|---|------------|
| Регистрация моделей каталога на сайте администрирования..... | 199 |
| Реализация обработчиков для каталога | 200 |
| Добавление шаблонов для отображения страниц каталога..... | 202 |
| Добавление корзины покупок | 206 |
| Использование сессий Django | 206 |
| Настройки сессий..... | 207 |
| Время жизни сессии..... | 207 |
| Хранение данных корзины в сессии..... | 208 |
| Обработка действий с корзиной покупок | 211 |
| Создание контекстного процессора для корзины | 217 |
| Регистрация заказов..... | 220 |
| Создание моделей заказа | 220 |
| Добавление моделей на сайт администрирования | 221 |
| Обработка заказов покупателей | 222 |
| Выполнение асинхронных задач с Celery | 226 |
| Установка Celery | 227 |
| Установка RabbitMQ..... | 227 |
| Подключение Celery к Django-проекту..... | 228 |
| Добавление асинхронных задач | 228 |
| Мониторинг Celery..... | 230 |
| Резюме..... | 231 |
| Глава 8. Управление заказами и платежами | 232 |
| Подключение платежного шлюза..... | 232 |
| Создание аккаунта Braintree | 233 |
| Установка Python-приложения Braintree | 234 |
| Интеграция платежного шлюза в проект | 235 |
| Тестирование платежей..... | 241 |
| Запуск в боевом режиме | 243 |
| Экспорт заказов в CSV-файл | 244 |
| Добавление собственных действий на сайте администрирования..... | 244 |
| Расширение сайта администрирования | 246 |
| Генерация PDF-счетов | 250 |
| Установка WeasyPrint | 251 |
| Создание PDF-шаблона | 251 |
| Формирование PDF-файлов | 252 |
| Отправка PDF-файла на электронную почту | 255 |
| Резюме..... | 256 |
| Глава 9. Расширение онлайн-магазина | 257 |
| Реализация системы купонов..... | 257 |
| Создание моделей | 258 |
| Добавление оплаты купонами | 259 |
| Обработка покупок по купонам..... | 265 |
| Добавление интернационализации и локализации сайта | 267 |
| Интернационализация Django | 267 |
| Подготовка проекта к интернационализации | 269 |

| | |
|---|------------|
| Добавление переводов в Python-код | 271 |
| Перевод в HTML-шаблонах..... | 275 |
| Подключение Rosetta для перевода через сайт администрирования | 279 |
| Грязный перевод | 281 |
| Шаблоны URL'ов для интернационализации..... | 281 |
| Добавление возможности сменить язык сайта..... | 283 |
| Перевод данных в моделях с django-parler | 285 |
| Настройка формата локализации | 292 |
| Валидация форм с django-localflavor | 293 |
| Реализация системы рекомендаций товаров..... | 294 |
| Добавление рекомендаций товаров на основе совершенных заказов | 294 |
| Резюме..... | 301 |
| Глава 10. Создание платформы для онлайн-обучения | 302 |
| Создание проекта | 302 |
| Определение моделей для курсов обучения..... | 303 |
| Регистрация моделей на сайте администрирования | 305 |
| Задание начальных данных с помощью фикстур..... | 306 |
| Создание моделей для содержимого курсов | 308 |
| Виды наследования моделей Django | 309 |
| Создание моделей содержимого курса..... | 311 |
| Создание собственных типов полей для модели | 312 |
| Добавление собственного поля сортировки в модели | 314 |
| Создание системы управления содержимым (CMS) | 318 |
| Добавление системы аутентификации | 318 |
| Создание шаблонов аутентификации | 318 |
| Определение обработчиков-классов | 321 |
| Использование примесей для обработчиков | 321 |
| Работа с группами и правами | 323 |
| Управление модулями курсов и их содержимым..... | 330 |
| Использование наборов форм для модулей курсов..... | 331 |
| Добавление содержимого в модуль | 334 |
| Управление модулями и их содержимым | 339 |
| Изменения порядка модулей и их содержимого | 343 |
| Резюме..... | 346 |
| Глава 11. Отображение и кеширование содержимого курсов | 347 |
| Отображение курсов..... | 347 |
| Добавление регистрации обучающихся..... | 352 |
| Обработка регистрации обучающихся в системе | 352 |
| Реализация записи на курсы | 354 |
| Доступ к содержимому курсов..... | 357 |
| Отображение различного типа содержимого | 360 |
| Использование фреймворка для кеширования..... | 363 |
| Доступные бэкэнды кеширования..... | 363 |
| Установка Memcached | 364 |
| Настройки кеширования | 364 |

| | |
|---|------------|
| Добавление Memcached в проект | 365 |
| Уровни кеширования | 366 |
| Использование низкоуровневого API кеширования | 366 |
| Кеширование фрагментов шаблонов | 370 |
| Кеширование результатов работы обработчиков | 371 |
| Резюме | 372 |
| Глава 12. Реализация API | 373 |
| Создание RESTful API | 373 |
| Установка Django REST Framework | 374 |
| Определение сериализаторов | 374 |
| Принцип работы парсеров и рендереров | 375 |
| Создание обработчиков списка и подробностей | 376 |
| Создание вложенных сериализаторов | 379 |
| Реализация собственных обработчиков | 380 |
| Обработка аутентификации пользователей | 381 |
| Ограничение доступа к обработчикам с помощью разрешений | 382 |
| Создание блоков обработчиков и их маршрутизаторов | 383 |
| Добавление собственных обработчиков в набор | 385 |
| Создание собственных разрешений | 385 |
| Сериализация содержимого курсов | 386 |
| Резюме | 388 |
| Глава 13. Запуск в боевом режиме | 389 |
| Создание окружения для запуска | 389 |
| Управление настройками для нескольких окружений | 389 |
| Настройка PostgreSQL | 391 |
| Проверка проекта | 392 |
| Запуск Django в режиме WSGI-приложения | 393 |
| Установка uWSGI | 393 |
| Конфигурация uWSGI | 393 |
| Установка NGINX | 395 |
| Боевое окружение | 396 |
| Конфигурация NGINX | 396 |
| Настройка отдачи статических и медиафайлов | 397 |
| Защита подключений с помощью SSL | 398 |
| Создание собственного промежуточного слоя | 401 |
| Создание промежуточного слоя для доступа через поддомен | 402 |
| Настройка NGINX на работу с несколькими поддоменами | 403 |
| Добавление собственных команд управления | 404 |
| Резюме | 406 |
| Предметный указатель | 407 |

Об авторе

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

О рецензентах

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

Я бы хотел поблагодарить свою жену за ее поддержку.

Предисловие

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

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

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

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

Для кого эта книга

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

О чем эта книга

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Чтобы получить максимальную пользу от этой книги

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

Принятые обозначения

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

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

Листинг выглядит следующим образом:

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

admin.site.register(Post)
```

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

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

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

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

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

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



Так будут оформляться предупреждения и важные примечания.



Так будут оформляться советы или рекомендации.

ОТЗЫВЫ И ПОЖЕЛАНИЯ

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

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

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

СКАЧИВАНИЕ ИСХОДНОГО КОДА ПРИМЕРОВ

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

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

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

СПИСОК ОПЕЧАТОК

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

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

НАРУШЕНИЕ АВТОРСКИХ ПРАВ

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

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

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

Глава 1

Создание приложения блога

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

В этой главе мы изучим такие темы:

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

УСТАНОВКА DJANGO

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

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

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

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

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

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

Создание изолированного Python-окружения

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

```
pip install virtualenv
```

После установки создайте изолированное окружение с помощью команды:

```
virtualenv my_env
```

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

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

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

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

Для активации виртуального окружения выполним команду:

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

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

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

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

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

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

Установка Django через pip

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

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


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

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

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

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

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


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

СОЗДАНИЕ ПЕРВОГО ПРОЕКТА

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

```
django-admin startproject mysite
```

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

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

Давайте посмотрим на структуру сгенерированного проекта:

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

Каждый из этих файлов имеет свое значение:

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

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

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

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

Вы увидите вывод:

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

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

Запуск сервера для разработки

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

Запустите сервер разработки, выполнив команду из корневого каталога проекта:

```
python manage.py runserver
```

Вы должны будете увидеть что-то подобное:

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

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

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

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

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

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

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



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

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

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

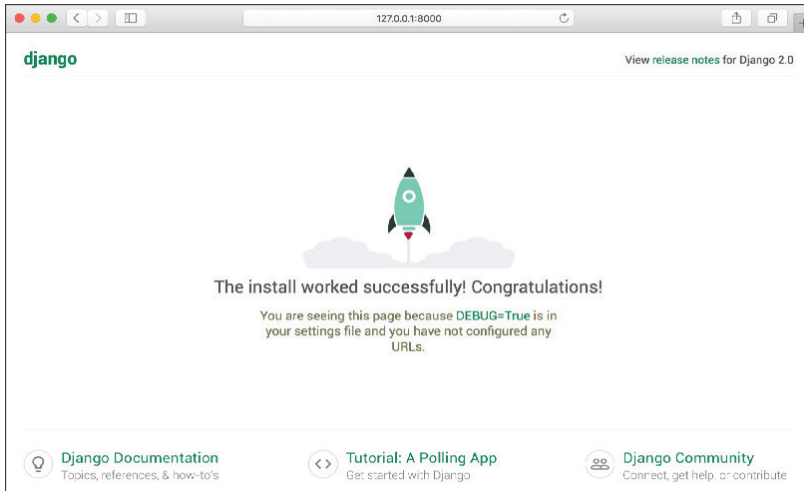


Рис. 1.1 ❖ Страница приветствия Django

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

Настройки проекта

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

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

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

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

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

Проекты и приложения

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

Создание приложения

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

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

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

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

Давайте рассмотрим назначение этих модулей:

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

ПРОЕКТИРОВАНИЕ СХЕМЫ ДАННЫХ ДЛЯ БЛОГА

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

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

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

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

```

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

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

def __str__(self):
    return self.title

```

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

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

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

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

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

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

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

Активация приложения

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

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

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

Создание и применение миграций

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

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

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

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

Вы должны увидеть такой вывод:

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

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

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

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

Вывод должен выглядеть следующим образом:

```
BEGIN;
--
-- Create model Post
--

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

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

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

```
python manage.py migrate
```

Вывод в консоли закончится такой строкой:

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

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

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

СОЗДАНИЕ САЙТА АДМИНИСТРИРОВАНИЯ

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

```
python manage.py createsuperuser
```

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

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

Введите данные, и пользователь будет создан.

Сайт администрирования Django

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

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

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

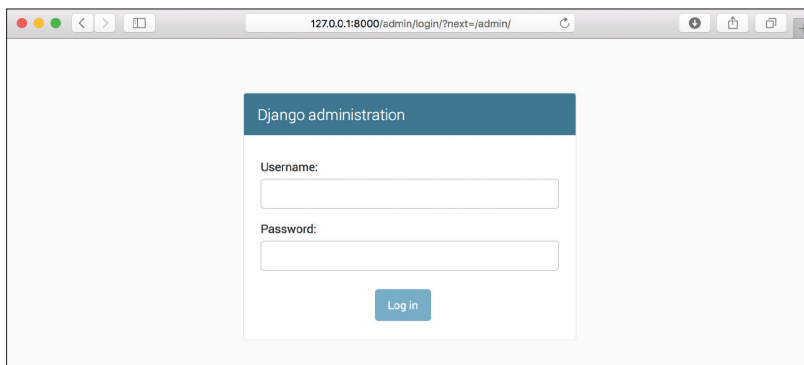


Рис. 1.2 ❖ Страница авторизации на сайте администрирования

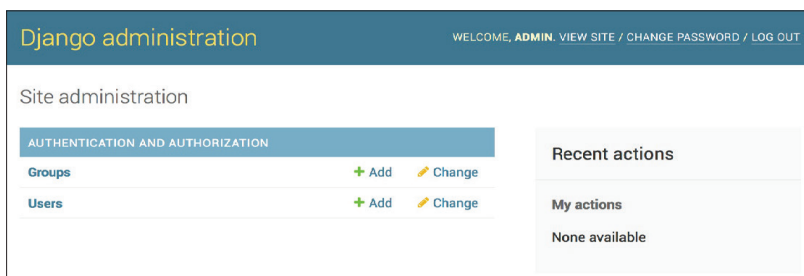


Рис. 1.3 ❖ Главная страница сайта администрирования

Добавление собственных моделей на сайт администрирования

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

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

admin.site.register(Post)
```

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

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

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

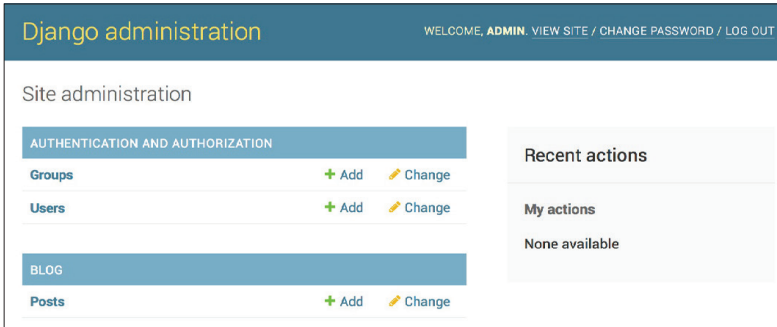


Рис. 1.4 ❖ Собственные модели на сайте администрирования

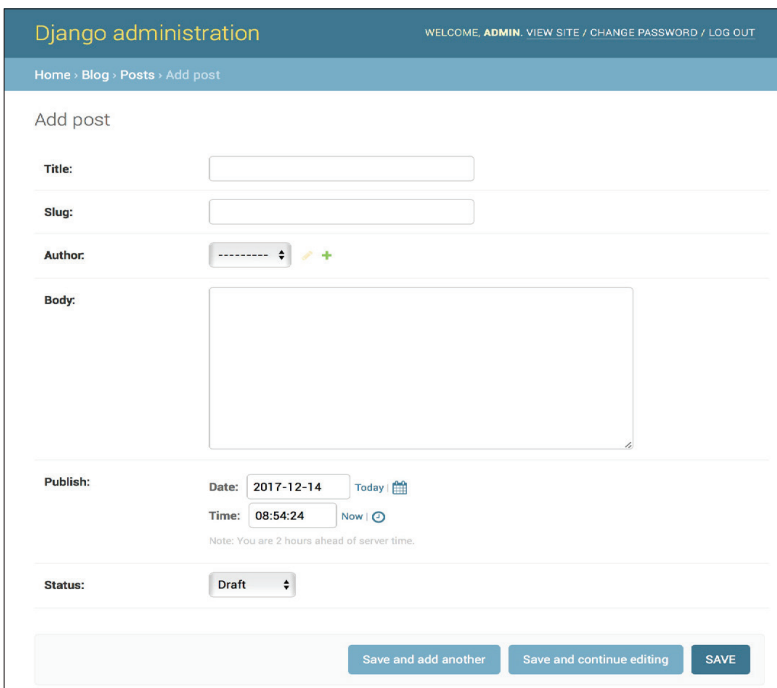


Рис. 1.5 ❖ Форма, созданная автоматически для редактирования статьи

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

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

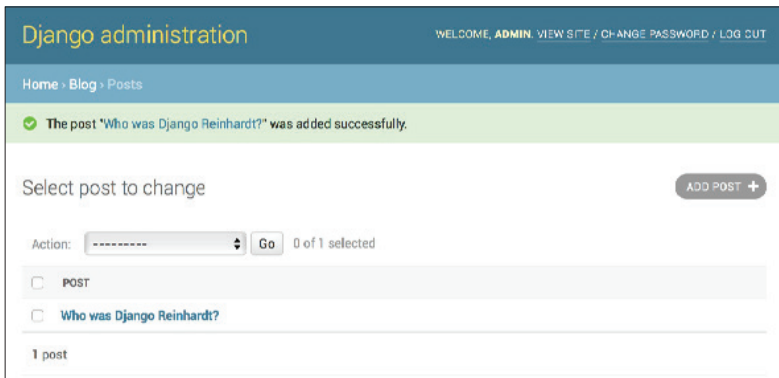


Рис. 1.6 ❖ Список статей

Настройка отображения моделей

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

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

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

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

Давайте настроим отображение модели, добавив несколько других опций:

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

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

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

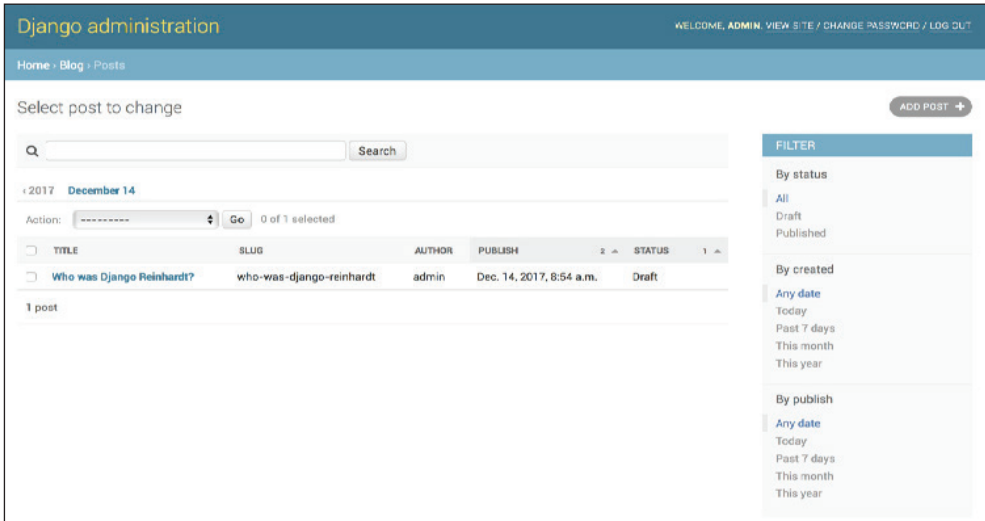


Рис. 1.7 ❖ Настроенная страница списка статей

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

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



Рис. 1.8 ❖ Поле поиска автора в форме редактирования статьи

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

РАБОТА С QUERYSET И МЕНЕДЖЕРАМИ

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

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

Создание объектов

Откройте терминал и выполните следующую команду в консоли Python:

```
python manage.py shell
```

Затем введите следующие строки кода:

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

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

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

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

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

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



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

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

```
post.save()
```

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


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

Изменение объектов

Давайте попробуем изменить заголовок статьи и сохранить ее:

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

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

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

Получение объектов

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

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

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

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

Использование метода `filter()`

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

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

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

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

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

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

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

Использование метода `exclude()`

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

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

Использование метода `order_by()`

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

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

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

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

Удаление объектов

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

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

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

Когда выполняются запросы QuerySet'ов

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

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

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

Создание менеджера модели

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

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

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

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

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

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

Запустите сервер разработки и выполните команду:

```
python manage.py shell
```

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

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

ОБРАБОТЧИКИ СПИСКА СТАТЕЙ И СТРАНИЦЫ ПОДРОБНОСТЕЙ

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

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

Создание обработчиков списка и страницы подробностей

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

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

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

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

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

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

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

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

Добавление шаблонов URL'ов для обработчиков

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

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

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

app_name = 'blog'

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

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

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

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

Если использование `path()` и конвертеров не подходит, можно задействовать `re_path()`. Эта функция позволяет задавать шаблоны URL'ов в виде регулярных выражений. Более подробно об этом написано на странице <https://docs.djangoproject.com/en/2.0/topics/http/urls/#re-path-converters>.

[project.com/en/2.0/ref/urls/#django.urls.re_path](https://docs.python.org/3/howto/regex.html). Если раньше вы не работали с регулярными выражениями, рекомендуем познакомиться с ними на <https://docs.python.org/3/howto/regex.html>.

☑ Создание отдельного файла `urls.py` для каждого приложения – хороший способ сделать приложение доступным для многократного использования в других проектах.

Теперь необходимо добавить определенные выше шаблоны в конфигурацию URL'ов проекта. Отредактируйте файл `urls.py`, который находится в каталоге `mysite`, чтобы он выглядел следующим образом:

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

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

Новый шаблон, добавленный с помощью `include`, подключит конфигурацию приложения блога. Все его адреса будут начинаться с `blog/`. Используемое пространство имен, `blog`, должно быть уникально по всему проекту. Мы будем обращаться к шаблонам приложения по пространству имен, например `blog:post_list`, `blog:post_detail`. Подробнее об этом вы можете прочитать на <https://docs.djangoproject.com/en/2.0/topics/http/urls/#url-namespaces>.

Канонические URL'ы для моделей

Мы можем использовать URL `post_detail`, о котором речь шла в предыдущем разделе, для построения канонического URL'а для объектов `Post`. В Django есть соглашение о том, что метод модели `get_absolute_url()` должен возвращать канонический URL объекта. Для реализации этого поведения мы будем использовать функцию `reverse()`, которая дает возможность получать URL, указав имя шаблона и параметры. Добавьте следующий фрагмент в файл `models.py`:

```
from django.urls import reverse

class Post(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('blog:post_detail', args=[self.publish.year,
                                                self.publish.month, self.publish.day, self.slug])
```

Мы будем использовать метод `get_absolute_url()` в HTML-шаблонах, чтобы получать ссылку на статью.

СОЗДАНИЕ HTML-ШАБЛОНОВ ДЛЯ ОБРАБОТЧИКОВ

Мы создали обработчики и связали их с шаблонами URL'ов. Самое время добавить *HTML-шаблоны* для отображения статей в приемлемом виде.

Создайте следующие папки и файлы в каталоге приложения `blog`:

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

Это будет структура наших HTML-шаблонов. Файл `base.html` содержит каркас верстки сайта и разделяет содержимое на два блока: основной и боковую панель. `list.html` и `detail.html` наследуются от `base.html` и отображают список статей и подробности статьи соответственно.

В Django встроен мощный язык шаблонов, который позволяет задать отображение данных. Он основывается на таких понятиях, как *шаблонные теги*, *переменные* и *фильтры*:

- шаблонные теги управляют процессом генерации HTML и выглядят так: `{% tag %}`;
- переменные шаблона заменяются переданными в контекст значениями в процессе формирования HTML и выглядят так: `{{ variable }}`;
- шаблонные фильтры позволяют изменять переменные контекста и выглядят так: `{{ variable|filter }}`.

Список всех встроенных в Django тегов и фильтров вы можете найти на <https://docs.djangoproject.com/en/2.0/ref/templates/builtins/>.

Давайте отредактируем `base.html` и добавим следующий фрагмент:

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

Строка `{% loadstatic %}` говорит Django о том, что нужно импортировать шаблонный тег `static`, который объявлен в приложении `django.contrib.staticfiles`, указанном в `INSTALLED_APPS`. После его импортирования мы можем использовать запись `{%static %}`. С помощью этого тега можно подключать файлы статики (например, `blog.css`), которые мы увидим дальше в этом примере в папке

static/ приложения. Скопируйте эту папку в ваш проект из кода-примера для первой главы, чтобы применить CSS-стили.

В шаблоне есть два тега `{% block %}`. Они говорят Django, что мы хотим определить блок в этом месте. Шаблоны, унаследованные от `base.html`, смогут заполнять эти блоки собственным содержимым. Мы определили два блока: `title` и `content`.

Давайте отредактируем `post/list.html`, чтобы он выглядел следующим образом:

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

С помощью тега `{% extends %}` указываем, что `list.html` унаследован от базового шаблона `blog/base.html`. Затем заполняем блоки `title` и `content` (проходим по списку статей и отображаем для каждой из них заголовки, дату публикации, автора, тело и каноническую ссылку). В теле статьи мы применяем два фильтра: `truncatewords`, обрезающий текст после указанного количества слов, и `linebreaks`, преобразующий вывод в HTML с переносами строки. Мы можем применять фильтры в цепочке, тогда каждый последующий будет действовать на результат предыдущего.

Откройте консоль и запустите сервер командой `python manage.py runserver`. Не забудьте создать несколько статей через сайт администрирования, чтобы список не был пустым. Теперь перейдите на `http://127.0.0.1:8000/blog/` в браузере, и вы увидите страницу, которую мы только что создали:

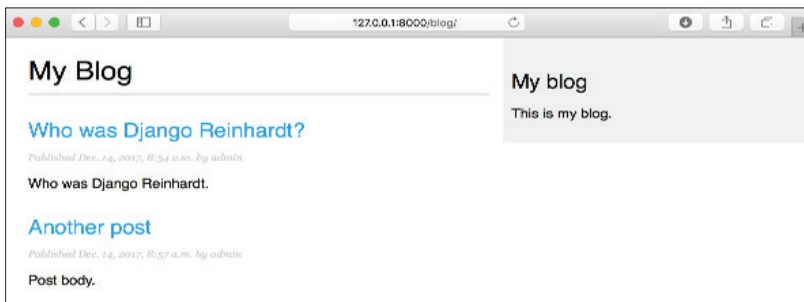


Рис. 1.9 ❖ Рабочий стол пользователя блога

После этого давайте отредактируем файл `post/detail.html`, добавив отображение содержимого статьи:

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

Теперь вы можете открыть страницу статьи, для чего нужно вернуться в браузер и кликнуть на заголовок одной из них. Вы увидите что-то подобное:



Рис. 1.10 ❖ Страница статьи

Обратите внимание на URL: `/blog/2017/12/14/who-was-django-reinhardt/`. Мы создали для статей хорошие, с точки зрения поисковой оптимизации, URL'ы.

ДОБАВЛЕНИЕ ПОСТРАНИЧНОГО ОТОБРАЖЕНИЯ

Когда в наш блог добавят большое количество статей, появится необходимость отображать их более удобно для пользователя, для чего следует разделить весь список на страницы. Django имеет встроенный класс для постраничного отображения, который позволит нам легко управлять им.

Отредактируйте `views.py`, импортируйте классы-пагинаторы из Django и добавьте их в обработчик `post_list`:

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

def post_list(request):
    object_list = Post.published.all()
    paginator = Paginator(object_list, 3) # По 3 статьи на каждой странице.
    page = request.GET.get('page')
    try:
        posts = paginator.page(page)
    except PageNotAnInteger:
        # Если страница не является целым числом, возвращаем первую страницу.
        posts = paginator.page(1)
    except EmptyPage:
        # Если номер страницы больше, чем общее количество страниц, возвращаем последнюю.
        posts = paginator.page(paginator.num_pages)
    return render(request, 'blog/post/list.html', {'page': page, 'posts': posts})
```

Постраничное отображение работает следующим образом:

- 1) мы инициализируем объект класса `Paginator`, указав количество объектов на одной странице;
- 2) извлекаем из запроса GET-параметр `page`, который указывает текущую страницу;
- 3) получаем список объектов на нужной странице с помощью метода `page()` класса `Paginator`;
- 4) если указанный параметр `page` не является целым числом, обращаемся к первой странице. Если `page` больше, чем общее количество страниц, то возвращаем последнюю;
- 5) передаем номер страницы и полученные объекты в шаблон.

Теперь нам нужно создать шаблон, который будет отображать номера страниц для выбора. В папке `templates/` приложения `blog` создайте новый файл `pagination.html`. Добавьте в него следующий фрагмент кода:

```
<div class="pagination">
  <span class="step-links">
    {% if page.has_previous %}
      <a href="?page={{page.previous_page_number}}">Previous</a>
    {% endif %}
    <span class="current">
      Page {{ page.number }} of {{ page.paginator.num_pages }}.
    </span>
    {% if page.has_next %}
      <a href="?page={{ page.next_page_number }}">Next</a>
    {% endif %}
  </span>
</div>
```

В этот шаблон необходимо передать объект `Page` для отображения ссылок на предыдущую, текущую и следующую страницы, а также общее количество объектов. Давайте вернемся в шаблон `blog/post/list.html` и подключим `pagination.html` в самом низу блока `{% content %}`:

```
{% block content %}
...
{% include "pagination.html" with page=posts %}
{% endblock %}
```

Так как страница `Page` передается в шаблон статей под именем `posts`, подключаем шаблон постраничного отображения, указав, чему будет равен параметр `page`. Можно применять такой способ для повторного использования блока постраничного отображения на различных страницах со списками разных объектов.

Теперь откройте в браузере <http://127.0.0.1:8000/blog/>. Внизу списка статей вы увидите блок выбора страницы. Попробуйте перейти на другую страницу.

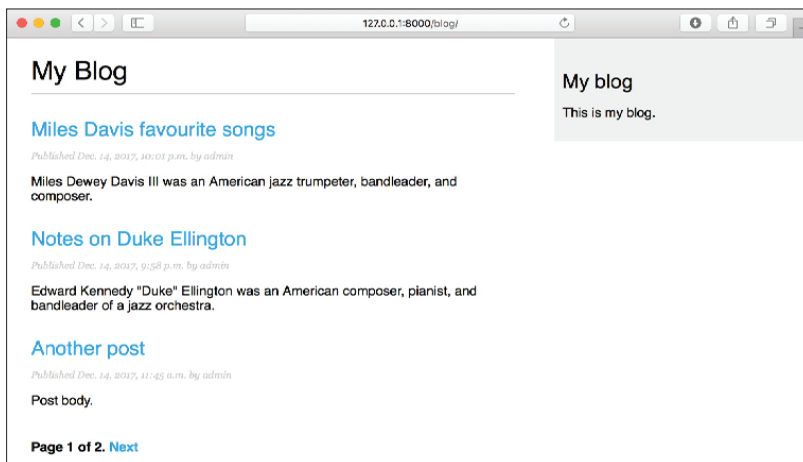


Рис. 1.11 ❖ Постраничное отображение списка статей

ИСПОЛЬЗОВАНИЕ ОБРАБОТЧИКОВ-КЛАССОВ

Использование класса – это альтернативный способ реализации обработчиков. Так как обработчик – это вызываемая функция, которая принимает запрос и возвращает ответ, мы можем реализовать его в виде метода класса. Django предоставляет для этого базовый класс обработчиков. Все они должны быть унаследованы от класса `View`, который управляет вызовом нужного метода в зависимости от HTTP-запроса и некоторыми другими функциями.

Обработчики-классы в некоторых случаях могут быть более полезными, чем функции-обработчики. Их преимущества заключаются в следующем:

- группируют код в несколько функций в зависимости от HTTP-методов запроса, таких как GET, POST, PUT;
- позволяют задействовать множественное наследование для создания многократно используемых обработчиков (их часто называют *примесями*, или *миксинами*).

Вы можете найти полный список и введение в использование обработчиков-классов на <https://docs.djangoproject.com/en/2.0/topics/class-based-views/intro/>.

Мы заменим наш `post_list` на класс-наследник `ListView` Django. Этот базовый класс обработчика списков позволяет отображать несколько объектов любого типа.

Отредактируйте файл `views.py` приложения `blog` и добавьте следующий код:

```
from django.views.generic import ListView

class PostListView(ListView):
    queryset = Post.published.all()
    context_object_name = 'posts'
    paginate_by = 3
    template_name = 'blog/post/list.html'
```

Обработчик `PostListView` является аналогом функции `post_list`. В этом фрагменте кода мы настроили `ListView` на выполнение следующих шагов:

- использовать переопределенный `QuerySet` модели вместо получения всех объектов. Вместо задания атрибута `QuerySet` мы могли бы указать модель `model=Post`, и тогда Django, используя стандартный менеджер модели, получал бы объекты как `Post.objects.all()`;
- использовать `posts` в качестве переменной контекста HTML-шаблона, в которой будет храниться список объектов. Если не указать атрибут `context_object_name`, по умолчанию используется переменная `object_list`;
- использовать постраничное отображение по три объекта на странице;
- использовать указанный шаблон для формирования страницы. Если бы мы не указали `template_name`, то базовый класс `ListView` использовал бы шаблон `blog/post_list.html`.

Теперь откройте файл `urls.py` приложения `blog`, прокомментируйте строки, касающиеся `post_list`, и добавьте новый шаблон для `PostListView`:

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

Для поддержки постраничного вывода мы должны передавать объект страницы, содержащий список статей, в HTML-шаблон. Базовый обработчик Django `ListView` передает этот объект в качестве переменной с именем `page_obj`, поэтому нужно немного откорректировать `post/list.html` и, подключая шаблон постраничного вывода, указать эту переменную:

```
{% include "pagination.html" with page=page_obj %}
```

Откройте `http://127.0.0.1:8000/blog/` в браузере и убедитесь, что все работает так же, как когда мы использовали обработчик-функцию `post_list`. Это простой пример реализации обработчика с помощью базовых классов Django. Вы узнаете о них больше в главе 10 «Реализация платформы для электронного обучения» и последующих главах.

РЕЗЮМЕ

В этой главе мы познакомились с основами фреймворка Django, создав простое приложение блога. Мы описали модели для проекта и применили миграции, а также реализовали обработчики запросов, в том числе постраничное отображение, шаблоны, добавили конфигурацию URL'ов.

В следующей главе мы узнаем, как улучшить наш блог: добавить в него системы комментариев и тегов, настроить возможность делиться статьями по электронной почте.

Глава 2

Добавление продвинутых функций в блог

В предыдущей главе мы создали базовое приложение блога. Теперь преобразуем этот простой проект в полнофункциональный блог с продвинутыми функциями, такими как возможность поделиться статьей, добавление комментариев, тегов для статей, получение подобных статей. В этой главе мы изучим следующие темы:

- отправка электронной почты с Django;
- создание форм и их обработка;
- создание форм для моделей;
- подключение сторонних приложений;
- формирование сложных QuerySet'ов.

Функция «ПОДЕЛИТЬСЯ СТАТЬЕЙ ЧЕРЕЗ E-MAIL»

Для начала мы добавим возможность делиться статьями, отправляя их на электронную почту. Возьмите небольшую паузу и подумайте, как вы можете использовать то, что мы уже изучили для реализации этой функции. Теперь можете свериться со списком шагов, которые нужно проделать для добавления функции «поделиться» в ваш проект:

- создать форму для заполнения имени и e-mail пользователя, e-mail получателя статьи и необязательного комментария;
- создать обработчик в `views.py`, который будет обрабатывать указанные данные и отправлять их на почту;
- добавить URL-шаблон для созданного обработчика в `urls.py` приложения блога;
- создать HTML-шаблон для отображения формы.

Создание Django-форм

Давайте начнем с построения формы для отправки статьи на e-mail. Django имеет встроенную *подсистему форм*, которая позволяет легко управлять ими.

Эта подсистема дает возможность описать поля формы, указать, как они будут отображаться и как будет проверяться корректность данных.

В Django встроены два базовых класса форм:

- Form – позволяет создавать стандартные формы;
- ModelForm – дает возможность создавать формы по объектам моделей.

Создайте файл `forms.py` в папке приложения `blog` и добавьте в него следующий фрагмент кода:

```
from django import forms

class EmailPostForm(forms.Form):
    name = forms.CharField(max_length=25)
    email = forms.EmailField()
    to = forms.EmailField()
    comments = forms.CharField(required=False, widget=forms.Textarea)
```

Это наша первая Django-форма. Обратите внимание на код. Мы создали класс формы, унаследованный от `Form`. Также мы используем различные типы полей Django для соответствующей валидации значений.

i Формы могут быть описаны в любом месте вашего проекта, но есть общее соглашение, чтобы они находились в файле `forms.py` каждого приложения.

Поле `name` имеет тип `CharField`. Этот тип полей будет отображаться как элемент `<input type="text">`. Каждый тип по умолчанию имеет виджет для отображения в HTML. Виджет может быть изменен с помощью параметра `widget`. В поле `comments` мы используем виджет `Textarea` для отображения HTML-элемента `<textarea>` вместо стандартного `<input>`.

Валидация поля также зависит от его типа. Например, поля `email` и `to` имеют тип `EmailField`. Оба могут получать только корректные e-mail-адреса, иначе при валидации формы будет выброшено исключение `forms.ValidationError`. При проверке учитываются и другие параметры. Мы определили максимальную длину в 25 символов для поля `name` и сделали поле `comments` необязательным, указав `required=False`. Все это учитывается при валидации формы. Используемые в этом примере поля являются лишь малой частью подсистемы форм Django. Для того чтобы увидеть полный список и описания, можете перейти на страницу <https://docs.djangoproject.com/en/2.0/ref/forms/fields/>.

Обработка данных формы

Теперь нам необходимо создать новый обработчик для получения данных формы и отправки их на почту, если они корректны. Отредактируйте файл `views.py` приложения `blog` и добавьте фрагмент кода:

```
from .forms import EmailPostForm

def post_share(request, post_id):
    # Получение статьи по идентификатору.
    post = get_object_or_404(Post, id=post_id, status='published')
    if request.method == 'POST':
```

```

# Форма была отправлена на сохранение.
form = EmailPostForm(request.POST)
if form.is_valid():
    # Все поля формы прошли валидацию.
    cd = form.cleaned_data
    # ... Отправка электронной почты.
else:
    form = EmailPostForm()
    return render(request, 'blog/post/share.html',
                  {'post': post, 'form': form})

```

В этом фрагменте мы выполняем следующие действия:

- определяем функцию `post_share`, которая принимает объект запроса `request` и параметр `post_id`;
- вызываем функцию `get_object_or_404()` для получения статьи с указанным идентификатором и убеждаемся, что статья опубликована;
- используем один и тот же обработчик для отображения пустой формы и обработки введенных данных. Для разделения логики отображения формы или ее обработки используется запрос `request`. Заполненная форма отправляется методом `POST`. Если метод запроса – `GET`, необходимо отобразить пустую форму; если приходит запрос `POST`, обрабатываем данные формы и отправляем их на почту.

Для отображения и обработки формы выполняются описанные ниже шаги.

1. Когда обработчик выполняется первый раз с `GET`-запросом, мы создаем объект `form`, который будет отображен в шаблоне как пустая форма:

```
form = EmailPostForm()
```

2. Пользователь заполняет форму и отправляет `POST`-запросом. Мы создаем объект формы, используя полученные из `request.POST` данные:

```

if request.method == 'POST':
    # Форма была отправлена на сохранение.
    form = EmailPostForm(request.POST)

```

3. После этого выполняется проверка введенных данных с помощью метода формы `is_valid()`. Он валидирует все описанные поля формы и возвращает `True`, если ошибок не найдено. Если хотя бы одно поле содержит неверное значение, возвращается `False`. Список полей с ошибками можно посмотреть в `form.errors`.
4. Если форма некорректна, мы возвращаем ее с введенными пользователем данными и сообщениями об ошибках, которые мы добавим в `HTML`-шаблон чуть позже.
5. Если форма валидна, мы получаем введенные данные с помощью `form.cleaned_data`. Этот атрибут является словарем с полями формы и их значениями.



Если форма не проходит валидацию, то в атрибут `cleaned_data` попадут только корректные поля.

Теперь давайте посмотрим, каким образом Django может отправлять сообщения на электронную почту, чтобы полностью реализовать обработчик.

Отправка электронной почты с Django

Отправка сообщений может быть реализована средствами Django. Для начала необходимо установить локальный SMTP-сервер или сконфигурировать доступ к внешнему SMTP-серверу, добавив следующие настройки в `settings.py` проекта:

- `EMAIL_HOST` – хост SMTP-сервера; по умолчанию `localhost`;
- `EMAIL_PORT` – порт SMTP-сервера; по умолчанию `25`;
- `EMAIL_HOST_USER` – логин пользователя для SMTP-сервера;
- `EMAIL_HOST_PASSWORD` – пароль пользователя для SMTP-сервера;
- `EMAIL_USE_TLS` – использовать ли защищенное TLS-подключение;
- `EMAIL_USE_SSL` – использовать ли скрытое TLS-подключение.

Если нельзя использовать SMTP-сервер, можно дать Django указание записывать адреса в консоль, добавив такую настройку в `settings.py`:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

При задании этой настройки Django будет выводить e-mail-сообщения в консоль. Это может быть полезно при тестировании приложения без подключения к SMTP-серверу.

Если вы хотите отправлять сообщения, но на вашем компьютере не установлен локальный почтовый сервер, можете использовать SMTP-сервер вашего почтового провайдера. Следующая конфигурация позволяет отправлять e-mail-сообщения, используя Gmail-сервер и ваш аккаунт Google:

```
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = 'your_password'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Выполните команду `python manage.py shell` для запуска Django-консоли. С помощью следующей команды вы можете протестировать отправку письма:

```
>>> from django.core.mail import send_mail
>>> send_mail('Django mail', 'This e-mail was sent with Django.',
'your_account@gmail.com', ['your_account@gmail.com'], fail_silently=False)
```

Функция `send_mail()` принимает в качестве обязательных аргументов тему, сообщение, отправителя и список получателей. Указав дополнительный параметр `fail_silently=False`, мы говорим, чтобы при сбое в отправке сообщения было сгенерировано исключение. Если в результате выполнения вы увидите `1`, ваше письмо успешно отправлено.

Если вы используете Gmail, нужно также разрешить доступ на странице <https://myaccount.google.com/lesssecureapps>:

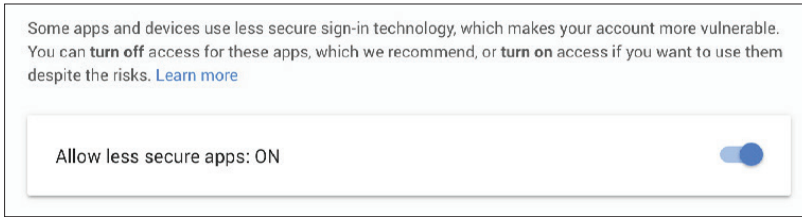


Рис. 2.1 ❖ Разрешение доступа сторонним приложениям к Gmail-аккаунту

Теперь давайте добавим эти функции в обработчик. Отредактируйте `post_share` в `views.py` приложения `blog` следующим образом:

```
from .forms import EmailPostForm

def post_share(request, post_id):
    # Получение статьи по идентификатору.
    post = get_object_or_404(Post, id=post_id, status='published')
    sent = False
    if request.method == 'POST':
        # Форма была отправлена на сохранение.
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Все поля формы прошли валидацию.
            cd = form.cleaned_data
            # Отправка электронной почты.
            post_url = request.build_absolute_uri(post.get_absolute_url())
            subject = '{} ({} recommends you reading "'
                '{}"' .format(cd['name'], cd['email'], post.title)
            message = 'Read "{}" at {}\n{}\n{}'s comments:
                {}'.format(post.title, post_url, cd['name'], cd['comments'])
            send_mail(subject, message, 'admin@myblog.com', [cd['to']])
            sent = True
        else:
            form = EmailPostForm()
            return render(request, 'blog/post/share.html',
                {'post': post, 'form': form, 'sent': sent})
```

Мы объявили переменную `sent`, она будет установлена в `True` после отправки сообщения. Будем использовать эту переменную позже для отображения сообщения об успешной отправке в HTML-шаблоне. Так как нам нужно добавить в сообщение абсолютную ссылку на статью, мы используем метод объекта запроса `request.build_absolute_uri()` и передаем в него результат выполнения `get_absolute_url()` статьи. Полученная абсолютная ссылка будет содержать HTTP-схему и имя хоста. Мы сформировали текст сообщения, используя данные формы, и, наконец, отправили e-mail по адресам, указанным в поле `to` формы.

Теперь, когда обработчик полностью готов, не забудьте добавить для него URL-шаблон. Откройте `urls.py` приложения `blog` и добавьте шаблон для `post_share`:

```
urlpatterns = [
    # ...
    path('<int:post_id>/share/', views.post_share, name='post_share'),
]
```

Отображение форм в HTML-шаблонах

После создания формы, программирования обработчика и добавления URL-шаблона нам остается только создать HTML-шаблон. Добавьте новый файл `share.html` в папке `blog/templates/blog/post/`; вставьте в него следующий код:

```
{% extends "blog/base.html" %}
{% block title %}Share a post{% endblock %}
{% block content %}
  {% if sent %}
    <h1>E-mail successfully sent</h1>
    <p>"{{ post.title }}" was successfully sent to {{ form.cleaned_data.to }}.</p>
  {% else %}
    <h1>Share "{{ post.title }}" by e-mail</h1>
    <form action="." method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <input type="submit" value="Send e-mail">
    </form>
  {% endif %}
{% endblock %}
```

Это шаблон для отображения формы или информации об успешной отправке сообщения. Как вы могли заметить, мы создали HTML-элемент формы, который будет отправляться на сервер методом `POST`:

```
<form action="." method="post">
```

В него добавили объект формы, дав Django указание сгенерировать ее поля как HTML-элементы параграфа `<p>` с помощью метода `as_p`. Кроме этого способа, мы можем отображать форму как нумерованный список или HTML-таблицу с помощью методов `as_ul` или `as_table` соответственно. Если мы хотим выводить каждое поле по отдельности, можно итерировать по ним таким образом:

```
{% for field in form %}
  <div>
    {{ field.errors }}
    {{ field.label_tag }} {{ field }}
  </div>
{% endfor %}
```

Тег `{% csrf_token %}` представляет собой скрытое поле с автоматически сгенерированным токеном для защиты от *подделки межсайтовых запросов* (Cross Site Request Forgery – CSRF-атак). Эти атаки исходят от вредоносных сайтов, которые выполняют нежелательные действия от имени наших пользователей. Больше информации об этой угрозе можно найти на [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)).

Рассмотренный тег генерирует на странице скрытый элемент, который выглядит подобным образом:

```
<input type='hidden' name='csrfmiddlewaretoken'
value='26JjKo2lcEtYkGoV9z4XmJIEHLXN5LDR' />
```

i По умолчанию Django проверяет CSRF-токен всех POST-запросов. Не забывайте добавлять `csrf_token` тег во все формы, которые отправляются методом POST.

Отредактируйте шаблон `blog/post/detail.html` и добавьте после переменной `{{ post.body|linebreaks }}` следующую ссылку для доступа к странице формы:

```
<p>
<a href="{% url 'blog:post_share' post.id %}">Share this post</a>
</p>
```

Помните, что для динамического формирования ссылок используется шаблонный тег `{% url %}`. Мы задаем пространство имен `blog` и URL с названием `post_share`, добавляем ID статьи в качестве параметра и благодаря этому получаем абсолютный URL.

Теперь запустите сервер разработки командой `python manage.py runserver` и откройте в браузере `http://127.0.0.1:8000/blog/`. Кликните на заголовок любой статьи, чтобы перейти на ее страницу. Под основным содержанием вы увидите ссылку, которую мы только что добавили, как показано на скриншоте:

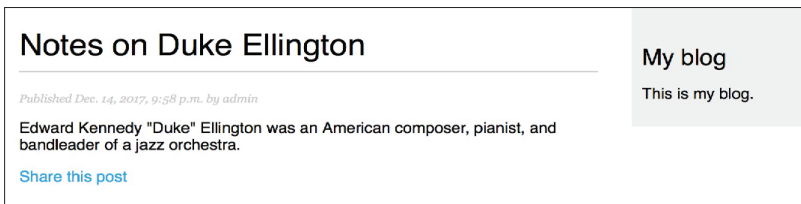


Рис. 2.2 ❖ Ссылка «поделиться» на странице статьи

Кликните на **Share this post**, и вы перейдете на страницу формы для отправки статьи по электронной почте:

Рис. 2.3 ❖ Форма отправки статьи на электронную почту

CSS-стили для формы добавлены в файле `static/css/blog.css` кода примера. Когда вы кликаете на кнопку **SEND E-MAIL**, форма отправляется и валидируется на сервере. Если все поля содержат корректные данные, вам вернется сообщение:

Рис. 2.4 ❖ Сообщение об успешной отправке статьи по указанному e-mail

Если были введены неправильные значения, то вернется страница формы с указанием ошибок рядом с каждым невалидным полем (рис. 2.5).

Стоит отметить, что некоторые современные браузеры могут предотвращать отправку формы с пустыми или некорректными данными. Это происходит потому, что браузер также проводит валидацию полей ввода, учитывая их тип. Если браузер находит неправильно заполненные поля, он выделяет их и не отправляет форму.

Наша форма для отправки статей по электронной почте полностью готова. Теперь давайте добавим в блог систему комментирования статей.

Рис. 2.5 ❖ Форма отправки статьи с указанием полей, заполненных с ошибками

ДОБАВЛЕНИЕ ПОДСИСТЕМЫ КОММЕНТАРИЕВ

Мы реализуем систему комментирования для блога, благодаря чему пользователи смогут оставлять комментарии для статей. Для этого нужно проделать следующие шаги:

- 1) создать модель для сохранения комментариев;
- 2) создать форму для отправки и валидации комментариев;
- 3) добавить обработчик, который будет проверять форму и сохранять комментарии;
- 4) добавить на страницу статьи список комментариев.

Для начала давайте создадим модель комментария. Откройте `models.py` приложения `blog` и добавьте следующий код:

```
class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='comments')
    name = models.CharField(max_length=80)
    email = models.EmailField()
    body = models.TextField()
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    active = models.BooleanField(default=True)

    class Meta:
        ordering = ('created',)

    def __str__(self):
        return 'Comment by {} on {}'.format(self.name, self.post)
```

Модель `Comment` содержит `ForeignKey` для привязки к определенной статье. Это отношение определено как «один ко многим», т. к. одна статья может иметь множество комментариев, но каждый комментарий может быть оставлен только для одной статьи. Атрибут `related_name` позволяет получить доступ к комментариям конкретной статьи. Теперь мы сможем обращаться к статье из комментария, используя запись `comment.post`, и к комментариям статьи при помощи `post.comments.all()`. Если бы мы не определили `related_name`, Django использовал бы имя связанной модели с постфиксом `_set` (например, `comment_set`).

Вы можете найти более подробную информацию об отношении «один ко многим» на странице https://docs.djangoproject.com/en/2.0/topics/db/examples/many_to_one/.

Мы добавили булево поле `active`, для того чтобы была возможность скрыть некоторые комментарии (например, содержащие оскорбления), и определили поле `created` для сортировки комментариев в хронологическом порядке.

Теперь нам нужно синхронизировать новую модель с базой данных. Выполните следующую команду для создания миграции:

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

Вы должны будете увидеть такой вывод в консоли:

```
Migrations for 'blog':
  blog/migrations/0002_comment.py
  - Create model Comment
```

Django сгенерировал файл `0002_comment.py` в папке `migrations` приложения `blog`. Теперь нужно выполнить миграцию, чтобы в базе данных была создана таблица. Выполните команду:

```
python manage.py migrate
```

В результате вы увидите вывод, содержащий строку:

```
Applying blog.0002_comment... OK
```

Миграция только что была применена, благодаря чему в базе данных была создана новая таблица `blog_comment`.

Теперь мы можем добавить новую модель на сайт администрирования для управления комментариями через интерфейс. Откройте файл `admin.py` приложения `blog`, импортируйте модель `Comment` и добавьте класс `ModelAdmin`:

```
from .models import Post, Comment

@admin.register(Comment)
class CommentAdmin(admin.ModelAdmin):
    list_display = ('name', 'email', 'post', 'created', 'active')
    list_filter = ('active', 'created', 'updated')
    search_fields = ('name', 'email', 'body')
```

Запустите сервер разработки командой `python manage.py runserver` и откройте в браузере `http://127.0.0.1:8000/admin/`. Вы должны будете увидеть новую модель в разделе **BLOG**:

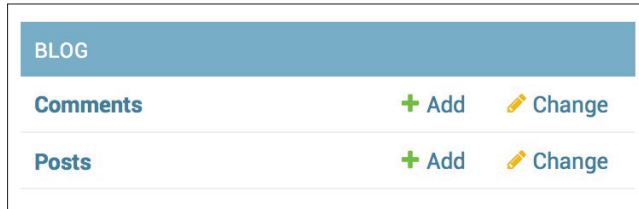


Рис. 2.6 ❖ Раздел **Comments** (Комментарии) на сайте администрирования

Эта модель теперь зарегистрирована на сайте администрирования, и мы можем легко управлять комментариями.

Создание модельных форм

Нам все еще необходимо создать форму, для того чтобы пользователи могли оставлять комментарии. Помните, что Django предоставляет два базовых класса для форм: `Form` и `ModelForm`. Первый мы уже использовали в предыдущем примере. Сейчас пришло время задействовать `ModelForm`, т. к. нам нужно динамически генерировать форму по модели `Comment`. Отредактируйте файл `forms.py` приложения `blog` и добавьте следующие строки:

```
from .models import Comment

class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ('name', 'email', 'body')
```

Все, что нужно для создания формы из модели, – указать, какую модель использовать в опциях класса `Meta`. Django найдет нужную модель и автоматически построит форму. Каждое поле модели будет сопоставлено полю формы соответствующего типа. По умолчанию Django использует все поля модели. Но мы можем явно указать, какие использовать, а какие – нет. Для этого достаточно определить списки `fields` или `exclude` соответственно. Для нашей формы `CommentForm` мы будем использовать `name`, `email` и `body` – только эти поля должны быть доступны пользователям, для того чтобы оставить комментарий.

Обработка модельных форм

Теперь давайте доработаем обработчик страницы статьи, для того чтобы он мог отображать форму и сохранять комментарий. Отредактируйте файл `views.py`, добавьте импорт модели `Comment` и ее формы `CommentForm`, отредактируйте обработчик `post_detail` таким образом:

```
from .models import Post, Comment
from .forms import EmailPostForm, CommentForm

def post_detail(request, year, month, day, post):
```

```

post = get_object_or_404(Post, slug=post,status='published',publish__year=year,
                        publish__month=month,publish__day=day)
# Список активных комментариев для этой статьи.
comments = post.comments.filter(active=True)
new_comment = None
if request.method == 'POST':
    # Пользователь отправил комментарий.
    comment_form = CommentForm(data=request.POST)
    if comment_form.is_valid():
        # Создаем комментарий, но пока не сохраняем в базе данных.
        new_comment = comment_form.save(commit=False)
        # Привязываем комментарий к текущей статье.
        new_comment.post = post
        # Сохраняем комментарий в базе данных.
        new_comment.save()
    else:
        comment_form = CommentForm()
return render(request, 'blog/post/detail.html',{'post': post,
                                              'comments': comments,
                                              'new_comment': new_comment,
                                              'comment_form': comment_form})

```

Давайте подробнее рассмотрим, что мы добавили в обработчик. Мы использовали функцию `post_detail` для отображения статьи и добавили `QuerySet` для получения всех активных комментариев:

```
comments = post.comments.filter(active=True)
```

Мы создали объект `QuerySet`, используя объект статьи `post` и менеджер связанных объектов `comments`, определенный в модели `Comment` в аргументе `related_name`.

Для сохранения комментария используем этот же обработчик. Мы задаем переменную `new_comment` как `None`. Позже она используется, когда новый комментарий будет успешно создан. Для инициализации формы при `GET`-запросе используем запись `comment = CommentForm()`. Если же получаем `POST`-запрос, то заполняем форму данными из запроса и валидируем ее методом `is_valid()`. Если форма заполнена некорректно, отображаем HTML-шаблон с сообщениями об ошибках. Если все поля успешно прошли валидацию, выполняем следующие шаги:

- 1) создаем новый объект `Comment`, вызвав метод `save()` и записав комментарий в переменную `new_comment`:

```
new_comment = comment_form.save(commit=False)
```

Метод `save()` создает объект модели, с которой связана форма, и сохраняет его в базу данных. Если в качестве аргумента метода передать `commit=False`, то объект будет создан, но не будет сохранен в базу данных. Это может быть полезным, когда перед сохранением объекта вам нужно каким-либо образом его изменить.

i Метод `save()` доступен для `ModelForm`, но не для `Form`, т. к. последние не привязываются к моделям.

2) указываем в комментарии ссылку на объект статьи:

```
new_comment.post = post
```

Благодаря этому мы связываем созданный комментарий с текущей статьей;

3) наконец, сохраняем комментарий в базу данных вызовом метода `save()`:

```
new_comment.save()
```

Наш обработчик теперь готов отображать и сохранять комментарии.

Добавление комментариев в шаблон статьи

Мы реализовали функциональность для управления комментариями статьи. Теперь необходимо доработать шаблон `post/detail.html` для отображения:

- общего количества комментариев для статьи;
- списка комментариев;
- формы для создания нового комментария.

Для начала добавим общее число комментариев на страницу. Откройте шаблон `post/detail.html` и добавьте следующий фрагмент в блок `content`:

```
{% with comments.count as total_comments %}
<h2>{{ total_comments }} comment{{ total_comments|pluralize }}</h2>
{% endwith %}
```

Мы используем ORM Django в шаблоне, выполняя `comments.count()`. Обратите внимание, что для вызова функций в шаблонах не нужно указывать круглые скобки. Тег `{% with %}` позволяет назначить переменной новое имя, которое можно использовать внутри блока до ближайшего тега `{% endwith %}`.

i Тег `{% with %}` полезен в случаях, когда в шаблоне нам нужно несколько раз обращаться к функциям, выполняющим запросы в базу данных или сложные вычисления.

Мы используем шаблонный фильтр `pluralize` для отображения слова `comment` во множественном числе, если это будет необходимо. Этот фильтр принимает значение, по которому определяется количество объектов, и возвращает соответствующий результат. Мы обсудим шаблонные фильтры подробнее в главе 3 «Расширение приложения блога».

Фильтр `pluralize` возвращает строку с постфиксом «s», если значение больше, чем 1. Указанный в примере текст будет отображаться как «0 comments», «1 comment» или «N comments». В Django включено множество шаблонных тегов, которые помогут нам отображать информацию более доступно.

Теперь давайте добавим список комментариев. Допишите следующий фрагмент в шаблон `post/detail.html`:

```
{% for comment in comments %}
<div class="comment">
  <p class="info">
    Comment {{ forloop.counter }} by {{ comment.name }}
    {{ comment.created }}
  </p>
  {{ comment.body|linebreaks }}
</div>
{% empty %}
<p>There are no comments yet.</p>
{% endfor %}
```

Здесь используется тег `{% for %}` для итерации по комментариям. Мы отображаем стандартное сообщение, если список `comments` пуст, и информируем пользователей, что комментариев для этой статьи пока нет. Каждый комментарий пронумерован с помощью переменной `{{ forloop.counter }}`, которая содержит номер текущей итерации цикла. Затем мы отображаем имя автора, дату создания и текст комментария.

Наконец, необходимо отобразить форму или сообщение об успешно созданном комментарии. Добавьте следующий фрагмент в шаблон:

```
{% if new_comment %}
<h2>Your comment has been added.</h2>
{% else %}
<h2>Add a new comment</h2>
<form action="." method="post">
  {{ comment_form.as_p }}
  {% csrf_token %}
  <p><input type="submit" value="Add comment"></p>
</form>
{% endif %}
```

Этот код максимально прост: если `new_comment` не существует, мы показываем поля формы создания комментария, в противном случае отображаем сообщение о его успешном сохранении. Откройте в браузере `http://127.0.0.1:8000/blog/` и перейдите на страницу любой статьи. Вы увидите что-то подобное:

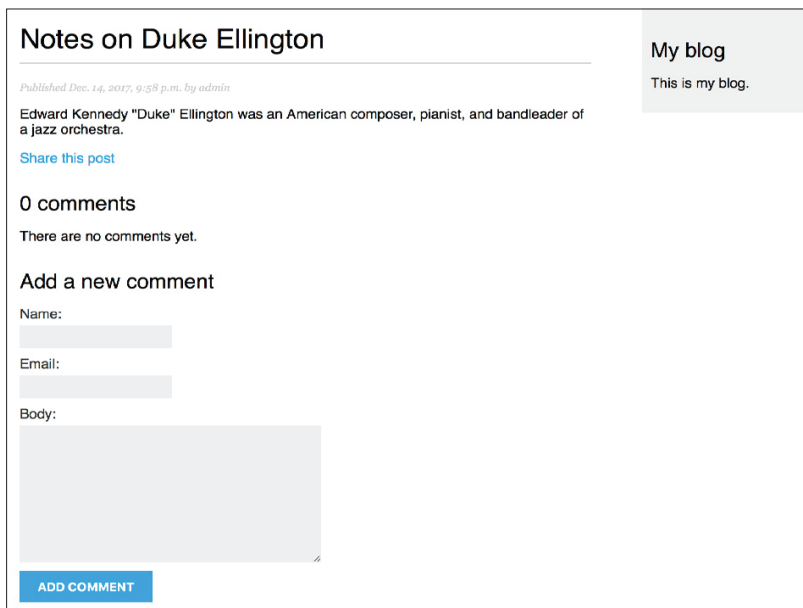


Рис. 2.7 ❖ Страница статьи с информацией о количестве комментариев и формой для создания комментария

Добавьте пару комментариев с помощью формы. Они должны будут отобразиться в хронологическом порядке (от новых к более старым):

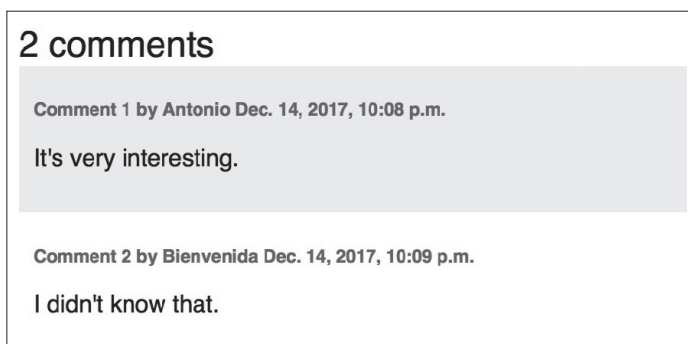


Рис. 2.8 ❖ Комментарии к статье

Откройте <http://127.0.0.1:8000/admin/blog/comment/> в браузере. Вы увидите страницу администрирования комментариев и несколько созданных объектов. Кликните на любой из них. Откроется страница редактирования. Снимите на ней чекбокс **Active** и сохраните запись. Вы будете перенаправлены на

страницу списка статей, и в столбце **Active** будет отображена соответствующая иконка. Например, на скриншоте первый комментарий не является активным:

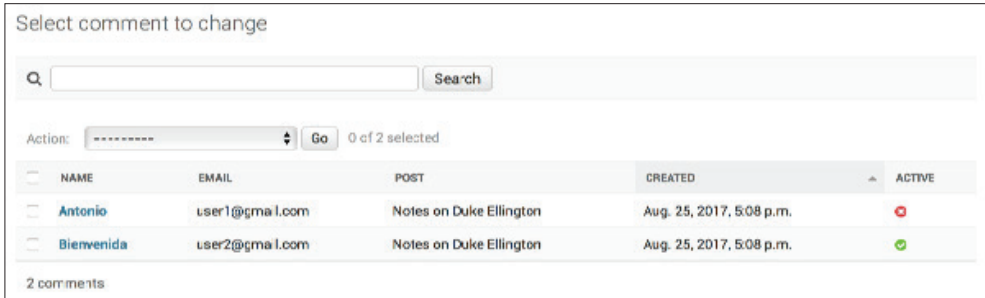


Рис. 2.9 ❖ Список комментариев на сайте администрирования

Если вы вернетесь на страницу статьи, то увидите, что отредактированный вами комментарий исчез. Более того, он не засчитывается в общем количестве комментариев. Благодаря полю `active` вы можете скрывать от показа некоторые комментарии.

ДОБАВЛЕНИЕ ПОДСИСТЕМЫ ТЕГОВ

После реализации системы комментирования настало время создать механизм для добавления тегов статей. Для этой цели мы будем использовать стороннее Django-приложение `django-taggit` – это приложение, которое добавляет модель `Tag` и менеджер для легкого тегирования любого объекта. Вы можете обратиться к его исходному коду на <https://github.com/alex/django-taggit>.

Для начала необходимо установить `django-taggit` с помощью `pip`:

```
pip install django_taggit==0.22.2
```

Затем откройте файл `settings.py` нашего проекта `mysite` и добавьте `taggit` в настройку `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    # ...
    'blog.apps.BlogConfig',
    'taggit',
]
```

Откройте `models.py` приложения `blog` и добавьте менеджер `TaggableManager` к вашей модели `Post`:

```
from taggit.managers import TaggableManager

class Post(models.Model):
    # ...
    tags = TaggableManager()
```

Менеджер `tags` позволит нам добавлять, получать список и удалять теги для объектов статей.

Выполните следующую команду, для того чтобы создать миграцию с этим изменением в модели:

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

Вы увидите в консоли такой вывод:

```
Migrations for 'blog':
  blog/migrations/0003_post_tags.py
  - Addfieldtagstopost
```

Теперь выполните команду для создания необходимых таблиц приложения `django-taggit` и синхронизации изменений модели `Post`:

```
python manage.py migrate
```

Вы увидите, что миграции успешно применены:

```
Applying taggit.0001_initial... OK
Applying taggit.0002_auto_20150616_2121... OK
Applying blog.0003_post_tags... OK
```

Теперь наша база данных готова к использованию моделей приложения `django-taggit`. Давайте посмотрим, как использовать менеджер `tags`. Откройте терминал и выполните команду `python manage.py shell`, затем введите указанные ниже строки. Для начала мы получим одну статью (по ID, равному 1):

```
>>> from blog.models import Post
>>> post = Post.objects.get(id=1)
```

Затем добавим несколько тегов к статье и получим их список, чтобы убедиться, что они успешно сохранены:

```
>>> post.tags.add('music', 'jazz', 'django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>, <Tag: django>]>
```

Наконец, удалим какой-нибудь тег и снова обратимся к списку тегов:

```
>>> post.tags.remove('django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>]>
```

Это задание не должно вызвать затруднений. Запустите сервер разработки командой `python manage.py runserver` и откройте в браузере `http://127.0.0.1:8000/admin/taggit/tag/`. Вы увидите страницу администрирования и список объектов модели `Tag`, тегов, приложения `taggit`:

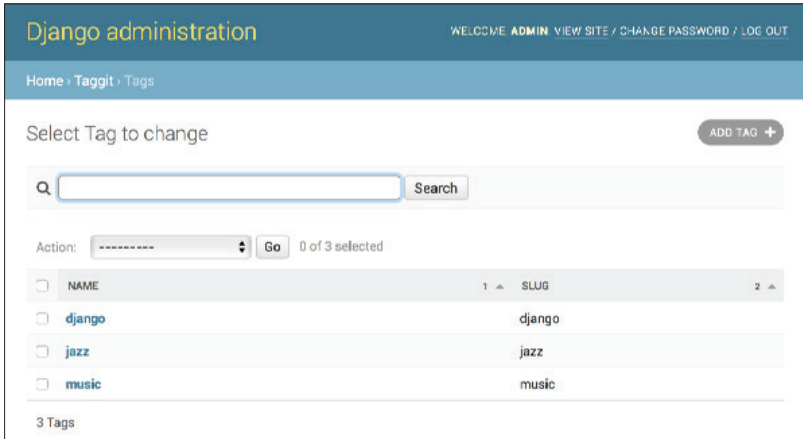


Рис. 2.10 ❖ Список тегов на сайте администрирования

Перейдите на страницу списка статей <http://127.0.0.1:8000/admin/blog/post/> и откройте любую из статей для редактирования. Вы увидите, что добавилось поле Tags. Теперь вы можете с легкостью управлять тегами статьи:

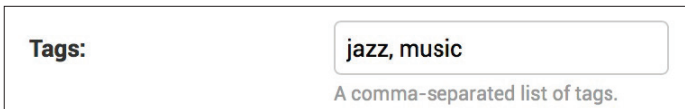


Рис. 2.11 ❖ Поле для добавления тегов на странице редактирования статьи

Давайте добавим отображение тегов на странице статей. Откройте шаблон `blog/post/list.html` и добавьте следующий HTML-код после заголовка статьи:

```
<p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
```

Шаблонный фильтр `join` работает так же, как Python-функция `join()`: соединяет элементы в строку, указывая в промежутках заданную строку. Откройте страницу <http://127.0.0.1:8000/blog/>. Под каждым заголовком статьи вы должны увидеть связанные с ней теги:

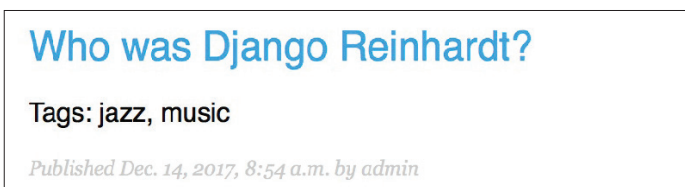


Рис. 2.12 ❖ Отображение тегов на странице статьи

Теперь нужно добавить возможность фильтровать список статей по определенному тегу. Откройте файл `views.py` приложения `blog`, импортируйте модель `Tag` из `django-taggit` и измените обработчик `post_list` следующим образом:

```
from taggit.models import Tag

def post_list(request, tag_slug=None):
    object_list = Post.published.all()
    tag = None

    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        object_list = object_list.filter(tags__in=[tag])

    paginator = Paginator(object_list, 3) # По 3 статьи на каждой странице.
    # ...
```

После этого изменения в обработчике выполняем описанные ниже действия.

1. Принимаем необязательный аргумент `tag_slug`, который по умолчанию равен `None`. Этот параметр будет задаваться в URL'e.
2. Внутри обработчика формируем начальный `QuerySet`, находим все опубликованные статьи и, если указан слаг тега, получаем соответствующий объект модели `Tag` с помощью метода `get_object_or_404()`.
3. Наконец, фильтруем изначальный список статей и оставляем только те, которые связаны с полученным тегом. Так как это связь «многие ко многим», необходимо фильтровать статьи по вхождению тегов в список тегов, который в нашем случае состоит из единственного элемента.

Помните, что `QuerySet`'ы ленивы. `QuerySet` для получения списка статей будет выполнен только при итерации по списку статей во время формирования HTML-шаблона.

Наконец, отредактируйте вызов функции `render()` в конце обработчика и передайте дополнительную переменную контекста. В итоге обработчик должен выглядеть так:

```
def post_list(request, tag_slug=None):
    object_list = Post.published.all()
    tag = None

    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        object_list = object_list.filter(tags__in=[tag])

    paginator = Paginator(object_list, 3) # По 3 статьи на каждой странице.
    page = request.GET.get('page')
    try:
        posts = paginator.page(page)
    except PageNotAnInteger:
        # Если указанная страница не является целым числом.
        posts = paginator.page(1)
    except EmptyPage:
        # Если указанный номер больше, чем всего страниц, возвращаем последнюю.
        posts = paginator.page(paginator.num_pages)
```

```
return render(request, 'blog/post/list.html',{'page': page,'posts': posts, 'tag': tag})
```

Откройте файл `urls.py` приложения `blog`, прокомментируйте шаблон для обработчика `PostListView` и раскомментируйте шаблон для `post_list`:

```
path('', views.post_list, name='post_list'),
# path('', views.PostListView.as_view(), name='post_list').
```

Добавьте дополнительный URL-шаблон, чтобы была возможность обратиться к списку статей, связанных с определенным тегом:

```
path('tag/<slug:tag_slug>/',views.post_list, name='post_list_by_tag').
```

Как вы можете видеть, оба шаблона обращаются к одному и тому же обработчику, но у них разные названия. Первый URL-шаблон вызовет `post_list` без дополнительных аргументов, в то время как второй будет передавать аргумент `tag_slug`. Мы используем преобразователь `slug`, для того чтобы ограничить возможные символы URL'a в качестве тега (могут быть использованы только прописные буквы, числа, нижние подчеркивания и дефисы).

Так как теперь мы используем функцию `post_list` в качестве обработчика, отредактируйте шаблон `blog/post/list.html` и добавьте постраничное отображение для статей `posts`:

```
{% include "pagination.html" with page=posts %}
```

Добавьте следующий фрагмент выше цикла `{% for %}`:

```
{% if tag %}
<h2>Posts tagged with "{{ tag.name }}"</h2>
{% endif %}
```

Если пользователь первый раз заходит в блог, он увидит все статьи. Если он отфильтрует их по тегу, то увидит сообщение об используемом теге и соответствующий список. Теперь добавим теги для каждой статьи:

```
<p class="tags">
  Tags:
  {% for tag in post.tags.all %}
    <a href="{% url "blog:post_list_by_tag" tag.slug %}">
      {{ tag.name }}
    </a>
    {% if not forloop.last %}, {% endif %}
  {% endfor %}
</p>
```

В этом фрагменте мы проходим по всем тегам статьи и отображаем ссылку, перейдя по которой, можно отфильтровать статьи по тегу. Для формирования URL'a используется запись `{% url "blog:post_list_by_tag" tag.slug %}` с указанием имени URL-шаблона и передачей слага тега как дополнительного параметра.

Откройте страницу <http://127.0.0.1:8000/blog/> и кликните на любую ссылку тега. Вы увидите список статей, отфильтрованных по этому тегу:

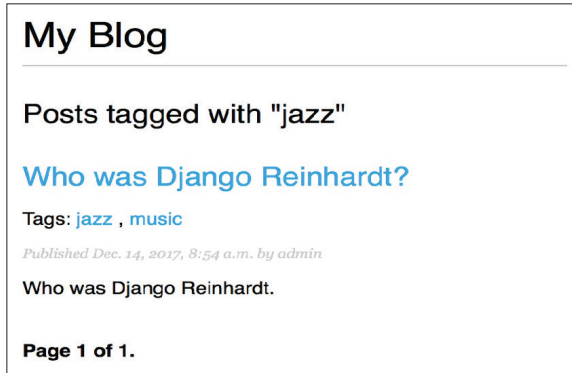


Рис. 2.13 ❖ Список статей, связанных с определенным тегом

ФОРМИРОВАНИЕ СПИСКА РЕКОМЕНДОВАННЫХ СТАТЕЙ

После создания системы тегирования статьи можно реализовать множество дополнительных возможностей. Используя теги, мы можем легко классифицировать статьи. Например, статьи с одинаковой тематикой будут иметь несколько совпадающих тегов. В этом разделе мы добавим возможность отображать статьи одной тематики (статьи, рекомендуемые пользователю).

Для того чтобы получить статьи, похожие на текущую, необходимо выполнить следующие шаги:

- 1) получить все теги для текущей статьи;
- 2) получить все статьи, которые связаны хотя бы с одним тегом;
- 3) исключить текущую статью из списка похожих, чтобы не дублировать ее;
- 4) отсортировать результат по количеству совпадений тегов;
- 5) в случае, если две и более статьи имеют одинаковый набор тегов, выбрать ту из них, которая является самой новой;
- 6) ограничить выборку тем количеством статей, которое мы хотим отображать в списке рекомендуемых.

Эти шаги могут быть преобразованы в сложный QuerySet, который мы добавим в обработчик `post_detail`. Откройте файл `views.py` приложения блога и добавьте в начало следующий импорт:

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

Это функция агрегации `Count` из Django. Она позволяет выполнять агрегирующий запрос для подсчета количества тегов на уровне базы данных. Пакет `django.db.models` содержит еще несколько агрегирующих функций:

- `Avg` – среднее значение;
- `Max` – максимальное значение;

- Min – минимальное значение;
- Count – количество объектов.

Подробнее про агрегацию вы можете прочитать на странице документации по Django: <https://docs.djangoproject.com/en/2.0/topics/db/aggregation/>.

Добавьте следующие строки в обработчик `post_detail` перед вызовом функции `render()`:

```
# Формирование списка похожих статей.
post_tags_ids = post.tags.values_list('id', flat=True)
similar_posts = Post.published.filter(tags__in=post_tags_ids)\
    .exclude(id=post.id)
similar_posts = similar_posts.annotate(same_tags=Count('tags'))\
    .order_by('-same_tags', '-publish')[:4]
```

Этот код работает следующим образом:

- 1) получает все ID тегов текущей статьи. Метод `QuerySet`'а `values_list()` возвращает кортежи со значениями заданного поля. Мы указали `flat=True`, чтобы получить «плоский» список вида `[1, 2, 3, ...]`;
- 2) получает все статьи, содержащие хоть один тег из полученных ранее, исключая текущую статью;
- 3) использует функцию агрегации `Count` для формирования вычисляемого поля `same_tags`, которое содержит определенное количество совпадающих тегов;
- 4) сортирует список опубликованных статей в убывающем порядке по количеству совпадающих тегов для отображения первыми максимально похожих статей и делает срез результата для отображения только четырех статей.

Добавьте объект `similar_posts` в контекст шаблона для функции `render()`:

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

Теперь отредактируйте HTML-шаблон `blog/post/detail.html` и добавьте следующий фрагмент для отображения рекомендуемых статей перед списком комментариев (рис. 2.14).

Сейчас мы можем отображать рекомендуемые статьи для наших пользователей. Django-taggit также включает менеджер `similar_objects()`, который можно использовать для поиска подобных объектов. Ознакомьтесь с полным описанием менеджеров `django-taggit` вы можете на странице <https://django-taggit.readthedocs.io/en/latest/api.html>.

Попробуйте добавить список тегов на страницу статьи по аналогии с тем, как мы это сделали в шаблоне `blog/post/list.html`.

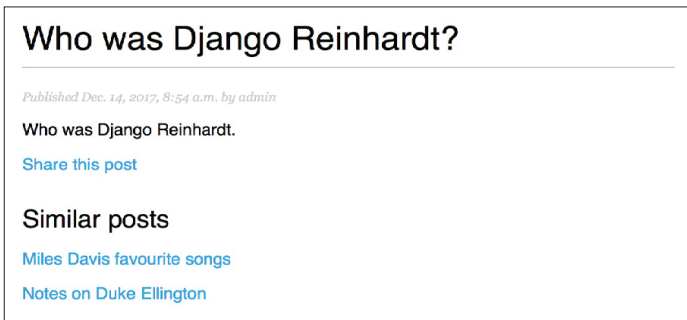


Рис. 2.14 ❖ Отображение списка похожих статей

РЕЗЮМЕ

В этой главе мы узнали, как работать с формами и модельными формами Django. Мы добавили возможность делиться статьями по электронной почте и систему комментирования статей. Также мы реализовали тегирование статей с помощью стороннего приложения и создали свой первый сложный QuerySet для получения подобных объектов.

В следующей главе мы научимся создавать свои шаблонные теги и фильтры, реализуем карту сайта и полнотекстовый поиск для статей.

Глава 3

Расширение приложения блога

В предыдущей главе мы рассмотрели работу с формами и узнали, как интегрировать в проект сторонние приложения. В этой главе мы познакомимся со следующими темами:

- создание собственных шаблонных тегов и фильтров;
- добавление карты сайта и RSS для статей;
- реализация полнотекстового поиска с PostgreSQL.

СОЗДАНИЕ ШАБЛОННЫХ ТЕГОВ И ФИЛЬТРОВ

Django предоставляет множество встроенных тегов, например `{% if %}` или `{% block %}`. Мы уже использовали некоторые из них в шаблонах. Список всех тегов Django приведен на странице <https://docs.djangoproject.com/en/2.0/ref/templates/builtins/>.

Несмотря на их многообразие, Django позволяет разработчику создать собственные шаблонные теги для выполнения необходимых действий. Это может быть очень полезно, когда в шаблонах необходимо выполнять специфические действия, не реализованные во встроенных тегах Django.

Создание собственных тегов

Django предоставляет несколько функций для простой реализации собственного тега:

- `simple_tag` – обрабатывает данные и возвращает строку;
- `inclusion_tag` – обрабатывает данные и возвращает сформированный фрагмент шаблона.

Шаблонные теги должны быть реализованы в рамках Django-приложения.

Давайте определим собственный шаблонный тег. В каталоге приложения `blog` создайте новую папку `templatetags` и добавьте пустой файл `__init__.py`. Создайте еще один файл внутри этого каталога и назовите его `blog_tags.py`. У нас должна получиться такая структура каталогов и файлов:

```
blog/
  __init__.py
  models.py
  ...
  templatetags/
    __init__.py
    blog_tags.py
```

То, как называется файл для тегов, имеет значение. Мы будем использовать его для загрузки тегов в HTML-шаблонах.

Мы начнем с реализации простого тега, который будет выводить количество опубликованных статей. Отредактируйте `blog_tags.py` и добавьте следующий код:

```
from django import template
from ..models import Post

register = template.Library()

@register.simple_tag
def total_posts():
    return Post.published.count()
```

Только что мы создали простой шаблонный тег, который возвращает количество опубликованных в блоге статей. Для того чтобы зарегистрировать наши теги, каждый модуль с функциями тегов должен определять переменную `register`. Эта переменная является объектом класса `template.Library` и используется для регистрации пользовательских тегов и фильтров в системе. В файле мы определяем тег `total_posts`, реализованный в виде функции, и оборачиваем его в декоратор `@register.simple_tag` для регистрации нового тега. Django будет использовать название функции в качестве названия тега. Однако можно указать явно, как обращаться к тегу из шаблонов. Для этого достаточно передать в декоратор аргумент `name` – `@register.simple_tag(name='my_tag')`.

i После добавления нового модуля с тегами необходимо перезагрузить сервер для разработки, чтобы Django зарегистрировал определенные теги и фильтры и мы могли использовать их в HTML-шаблонах.

Перед тем как использовать собственные шаблонные теги, необходимо подключить их в шаблоне с помощью `{% load %}`, используя имя модуля, в котором описаны теги и фильтры. Откройте файл `blog/templates/base.html` и добавьте вверху файла строку `{% load blog_tags %}`. Теперь можно использовать тег для показа количества опубликованных статей, добавив `{% total_posts %}` в HTML-шаблон таким образом:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
  <head>
```

```

<title>{% block title %}{% endblock %}</title>
<link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}{% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
    <p>This is my blog. I've written {% total_posts %} posts so far.</p>
  </div>
</body>
</html>

```

Необходимо перезапустить сервер разработки, чтобы Django смог использовать созданные файлы. Остановите проект нажатием **Ctrl+C** и запустите снова командой:

```
python manage.py runserver
```

Откройте в браузере <http://127.0.0.1:8000/blog/>. Вы увидите количество статей в боковой панели, как на скриншоте:

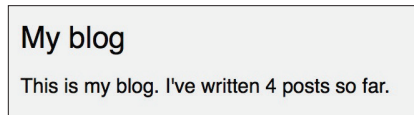


Рис. 3.1 ❖ Отображение количества статей в блоге на главной странице

Мощь собственных шаблонных тегов заключается в том, что мы можем обрабатывать любые данные системы и добавлять их в шаблон при его формировании. Можно выполнять запросы в базу данных или вычислять какие-то значения и отображать это в наших шаблонах.

Теперь создадим тег для добавления последних статей блога на боковую панель. На этот раз мы будем использовать инклюзивный тег, с помощью которого сможем задействовать переменные контекста, возвращаемые тегом, для формирования шаблона. Отредактируйте `blog_tags.py` и добавьте следующий фрагмент:

```

@register.inclusion_tag('blog/post/latest_posts.html')
def show_latest_posts(count=5):
    latest_posts = Post.published.order_by('-publish')[:count]
    return {'latest_posts': latest_posts}

```

Здесь мы регистрируем тег с помощью декоратора `@register.inclusion_tag` и указываем шаблон, который будет использоваться для формирования HTML. Функция будет принимать один дополнительный аргумент – `count`, по умол-

чанию равный 5. С его помощью мы сможем из шаблона указать количество статей для отображения. Используем `count` для ограничения результата запроса `Post.published.order_by('-publish')[:count]`. Обратите внимание, что функция тега возвращает словарь переменных вместо простого значения. Инклюзивные теги должны возвращать только словари контекста, который затем будет использован для формирования HTML-шаблона. Чтобы задать любое другое количество статей, используйте такую запись: `{% show_latest_posts 3 %}`.

Теперь создайте новый шаблон в каталоге `blog/post/` и назовите его `latest_posts.html`. Добавьте в него следующий фрагмент:

```
<ul>
  {% for post in latest_posts %}
    <li><a href="{{ post.get_absolute_url }}">{{ post.title }}</a></li>
  {% endfor %}
</ul>
```

Мы отображаем нумерованный список статей, используя переменную контекста `latest_posts`, которую получаем из тега. Отредактируйте `blog/base.html` и добавьте новый шаблонный тег для показа трех последних статей. Код боковой панели должен выглядеть следующим образом:

```
<div id="sidebar">
  <h2>Myblog</h2>
  <p>This is my blog. I've written {% total_posts %} posts so far.</p>

  <h3>Latest posts</h3>
  {% show_latest_posts 3 %}
</div>
```

Когда вызывается шаблонный тег, в него передается аргумент `count`, указывающий на количество статей. В результате выполнения функции `show_latest_posts` сформированный HTML-фрагмент появится в том месте, где тег был вставлен в родительский шаблон.

Теперь обновите страницу в браузере. Боковая панель должна выглядеть следующим образом:



Рис. 3.2 ❖ Последние статьи в боковой панели

Наконец, мы создадим простой шаблонный тег для отображения статей с наибольшим количеством комментариев. Результат его действий будет сохраняться в переменную, поэтому может быть использован многократно и без повторных вычислений. Отредактируйте `blog_tags.py` и добавьте следующие импорт и функцию тега:

```
from django.db.models import Count

@register.simple_tag
def get_most_commented_posts(count=5):
    return Post.published.annotate(total_comments=Count('comments'))
        .order_by('-total_comments')[ :count]
```

В этом фрагменте кода мы формируем `QuerySet`, используя метод `annotate()` для добавления к каждой статье количества ее комментариев. `Count` используется в качестве функции агрегации, которая вычисляет количество комментариев `total_comments` для каждого объекта `Post`. Также мы сортируем `QuerySet` по этому полю в порядке убывания. Как и в предыдущем примере, тег принимает дополнительный аргумент `count`, чтобы ограничить количество выводимых статей.

В дополнение к `Count` Django предоставляет другие функции агрегации: `Avg`, `Max`, `Min` и `Sum`. Найти более подробное описание и примеры их использования вы можете на <https://docs.djangoproject.com/en/2.0/topics/db/aggregation/>.

Отредактируйте `blog/base.html` и добавьте следующий код в `<div>` элемент боковой панели:

```
<h3>Most commented posts</h3>
{% get_most_commented_posts as most_commented_posts %}
<ul>
  {% for post in most_commented_posts %}
    <li><a href="{{ post.get_absolute_url }}">{{ post.title }}</a></li>
  {% endfor %}
</ul>
```

Мы записываем результат в переменную с помощью ключевого слова `as` и указываем после него имя переменной. Для того чтобы наиболее комментируемые статьи были доступны в шаблоне через переменную `most_commented_posts`, используем запись вида `{% get_most_commented_posts as most_commented_posts %}`. Затем отображаем их в нумерованном списке.

Откройте браузер и обновите страницу, чтобы увидеть окончательный результат. У вас должно получиться что-то похожее:

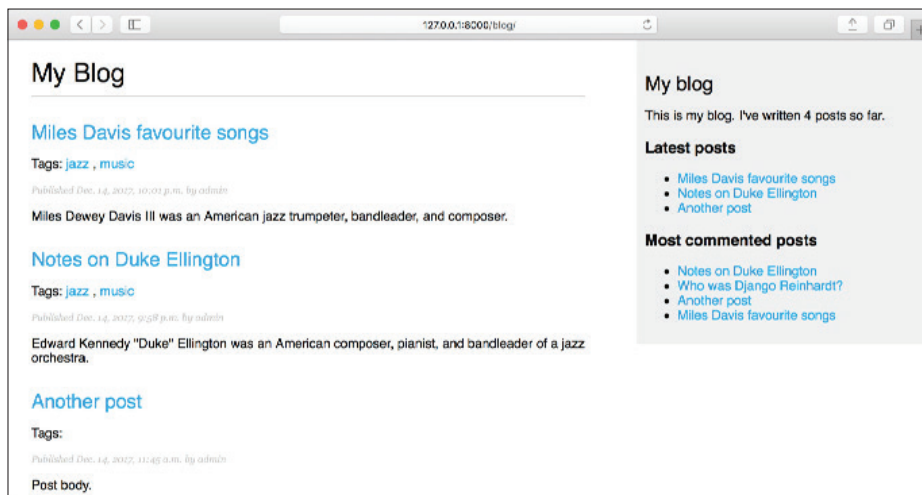


Рис. 3.3 ❖ Наиболее комментируемые статьи в боковой панели

Теперь у вас есть представление, как создавать собственные шаблонные теги. Более подробную информацию об этом вы можете найти на странице <https://docs.djangoproject.com/en/2.0/howto/custom-template-tags/>.

Создание собственных фильтров

В Django реализовано множество стандартных шаблонных фильтров, которые позволяют изменять переменные в шаблонах. Фильтр – это Python-функция, которая принимает один или два аргумента. В первый передается изменяемая переменная контекста, во второй – любая дополнительная переменная. Второй аргумент не является обязательным. Фильтр выглядит так: `{{ variable|my_filter:"foo" }}`. Мы можем использовать последовательно сколько угодно фильтров, например `{{ variable|filter1|filter2 }}`. Каждый из них будет применяться к результату предыдущего в цепочке фильтра.

Мы создадим собственный фильтр, чтобы добавить возможность заполнять тело статьи с помощью форматирования Markdown, которое будет формировать корректный HTML при отображении статьи. *Markdown* – это синтаксис форматирования, который легко использовать прямо в тексте, к тому же он может быть конвертирован в HTML. Более подробно об этой разметке вы можете прочитать на странице <https://daringfireball.net/projects/markdown/basics>.

Сначала установите Python-пакет с помощью `pip`:

```
pip install Markdown==2.6.11
```

Затем отредактируйте `blog_tags.py` и добавьте следующий код:

```
from django.utils.safestring import mark_safe
import markdown
```

```
@register.filter(name='markdown')
def markdown_format(text):
    return mark_safe(markdown.markdown(text))
```

Мы зарегистрировали фильтр так же, как регистрировали теги. Во избежание коллизий имен нашей функции и установленного Markdown-пакета, мы назвали функцию `markdown_format`, а при регистрации фильтра указали ее имя, которое будет использоваться в шаблонах: `{{ variable|markdown }}`. При формировании шаблона Django обращаем внимание на HTML, генерируемый фильтрами. Мы используем функцию `mark_safe`, чтобы пометить результат работы фильтра как HTML-код, который нужно учитывать при построении шаблона. По умолчанию Django не доверяет любому HTML, получаемому из переменных контекста или фильтров. Единственное исключение – фрагменты, помеченные с помощью `mark_safe`. Это условие предотвращает отображение потенциально опасного HTML, но в то же время позволяет обработать код, которому вы доверяете.

Теперь загрузите модуль с фильтром и тегами в шаблон списка статей. Добавьте следующую строку вверху файлов `blog/post/list.html` и `blog/post/detail.html` после выражения с `{% extends %}`:

```
{% loadblog_tags %}
```

Обратите внимание на строку в шаблоне `post/detail.html`:

```
{{ post.body|linebreaks }}
```

Замените ее на следующую:

```
{{ post.body|markdown }}
```

Затем в шаблоне `post/list.html` замените строку

```
{{ post.body|truncatewords:30|linebreaks }}
```

на строку с использованием нашего фильтра:

```
{{ post.body|markdown|truncatewords_html:30 }}
```

Фильтр `truncatewords_html` обрезает строку после указанного количества слов, не считая незакрытые HTML-теги.

Откройте браузер на странице `http://127.0.0.1:8000/admin/blog/post/add/` и добавьте статью с таким текстом:

```
This is a post formatted with markdown
```

```
-----
```

```
*This is emphasized* and **this is more emphasized**.
```

```
Here is a list:
```

```
* One
* Two
* Three
```

```
And a [link to the Django site](https://www.djangoproject.com/)
```

Откройте браузер и посмотрите, как отформатировано тело статьи. Вы должны будете видеть что-то подобное:



Рис. 3.4 ❖ Статья, отформатированная с помощью Markdown

Как вы можете видеть из этого примера, собственные фильтры очень полезны, когда нужно переопределить форматирование. Больше информации о реализации своих шаблонных фильтров можно найти на <https://docs.djangoproject.com/en/2.0/howto/custom-template-tags/#writing-custom-template-filters>.

ДОБАВЛЕНИЕ КАРТЫ САЙТА

Django предоставляет подсистему для динамического формирования карты сайта. *Карта сайта* – это XML-файл, который говорит поисковым роботам о страницах сайта, их структуре и частоте их обновления. С ее помощью мы помогаем поисковым роботам индексировать сайт.

Подсистема карты сайта Django реализована в пакете `django.contrib.sites`, который позволяет настроить связь объектов приложения и конкретных сайтов, которые запускаются вместе с нашим проектом. Это может быть полезным, когда вы хотите запустить несколько Django-сайтов в одном проекте. Для добавления подсистемы карты сайта необходимо активировать два приложения. Отредактируйте `settings.py` проекта и добавьте в список `INSTALLED_APPS` приложения `django.contrib.sites` и `django.contrib.sitemaps`. Также добавьте идентификатор сайта:

```
SITE_ID = 1

# Application definition
INSTALLED_APPS = [
    # ...
    'django.contrib.sites',
    'django.contrib.sitemaps',
]
```

Теперь выполните команду запуска миграции, чтобы Django создал необходимые таблицы этих приложений в базе данных:

```
python manage.py migrate
```

В результате вы увидите вывод в консоли:

```
Applying sites.0001_initial... OK
Applying sites.0002_alter_domain_unique... OK
```

Теперь приложение `sites` синхронизировано с базой данных. Создайте новый файл в каталоге приложения `blog` и назовите его `sitemaps.py`. Откройте его и добавьте содержимое:

```
from django.contrib.sitemaps import Sitemap
from .models import Post

class PostSitemap(Sitemap):
    changefreq = 'weekly'
    priority = 0.9

    def items(self):
        return Post.published.all()

    def lastmod(self, obj):
        return obj.updated
```

Мы создали собственный объект карты сайта, унаследовав его от `Sitemap` модуля `sitemaps`. Атрибуты `changefreq` и `priority` показывают частоту обновления страниц статей и степень их совпадения с тематикой сайта (максимальное значение – 1). Метод `items()` возвращает `QuerySet` объектов, которые будут отображаться в карте сайта. По умолчанию Django использует метод `get_absolute_url()` объектов списка, чтобы получать их URL'ы. Ранее мы реализовали этот метод для класса `Post` в главе 1 «Создание приложения блога» для получения канонических URL'ов статей. Если вы хотите указать URL для каждого объекта, добавьте метод `location` в класс карты сайта. Метод `lastmod` принимает каждый объект из результата вызова `items()` и возвращает время последней модификации статьи. Подробности о том, как работает карта сайта в Django, можно найти на странице <https://docs.djangoproject.com/en/2.0/ref/contrib/sitemaps/>.

Наконец, необходимо добавить URL карты сайта. Отредактируйте файл `urls.py` следующим образом:

```
from django.urls import path, include
from django.contrib import admin
from django.contrib.sitemaps.views import sitemap
from blog.sitemaps import PostSitemap

sitemaps = {'posts': PostSitemap,}

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls', namespace='blog')),
    path('sitemap.xml', sitemap, {'sitemaps': sitemaps},
         name='django.contrib.sitemaps.views.sitemap')
]
```

В этом коде мы подключили необходимые модули и определили словарь карт сайта. Также мы добавили шаблон URL'a, который соответствует адресу sitemap.xml и обработчику sitemap. Далее запустите сервер разработки и откройте страницу <http://127.0.0.1:8000/sitemap.xml>. Вы увидите следующий XML:

```
<?xml version="1.0" encoding="utf-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://example.com/blog/2017/12/15/markdown-post/</loc>
    <lastmod>2017-12-15</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2017/12/14/who-was-django-reinhardt/</loc>
    <lastmod>2017-12-14</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
</urlset>
```

URL для каждой статьи был сформирован с помощью метода `get_absolute_url()`. Атрибут `lastmod` соответствует полю `updated` статьи, как мы указали в объекте карты сайта, значения `changefreq` и `priority` также взяты из класса `PostSitemap`. Как вы могли заметить, в качестве домена используется `example.com`. Этот домен берется из объекта `Site`, сохраненного в базе данных. При синхронизации приложения `sites` с базой данных был создан объект по умолчанию. Откройте в браузере <http://127.0.0.1:8000/admin/sites/site/>. Вы должны увидеть:

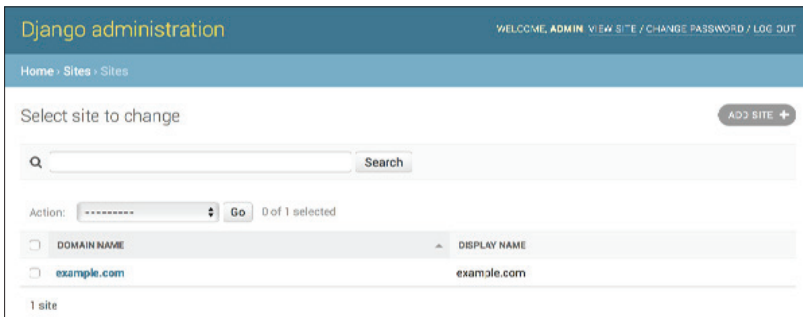


Рис. 3.5 ❖ Раздел сайтов на сайте администрирования

Это страница администрирования карты сайта. Вы можете увидеть домен или хост, который используется подсистемой карт сайтов по умолчанию, и приложения, которые зависят от него. Чтобы сгенерировать URL'ы, которые существуют в вашем локальном окружении, замените домен на `localhost:8000`, как показано на следующем скриншоте, и сохраните:

Change site

Domain name: localhost:8000

Display name: localhost:8000

Рис. 3.6 ❖ Настройки текущего сайта

Теперь URL'ы в XML-файле карты сайта будут сформированы с учетом нового домена. В боевом окружении необходимо заменить имя домена на фактическое, где будет размещено приложение.

ДОБАВЛЕНИЕ RSS ДЛЯ СТАТЕЙ

Django имеет встроенную систему для автоматической генерации RSS или Atom-фидов. *Фид* (feed) – это форма данных (чаще всего XML), которая предоставляет пользователям часто обновляемый контент. Пользователи смогут подписаться на обновление записей, используя агрегаторы – программное обеспечение для чтения новостей и получения уведомлений о новых фидах.

Создайте новый файл в папке приложения `blog` и назовите его `feeds.py`. Добавьте следующие строки кода:

```
from django.contrib.syndication.views import Feed
from django.template.defaultfilters import truncatewords
from .models import Post

class LatestPostsFeed(Feed):
    title = 'My blog'
    link = '/blog/'
    description = 'New posts of my blog.'

    def items(self):
        return Post.published.all()[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return truncatewords(item.body, 30)
```

Мы унаследовали наш класс от `Feed` – класса подсистемы фидов Django. Атрибуты `title`, `link` и `description` будут представлены в RSS элементами `<title>`, `<link>` и `<description>` соответственно.

Метод `items()` получает объекты, которые будут включены в рассылку. Мы берем только последние 5 опубликованных статей для этого фида. Методы `item_title()` и `item_description()` получают для каждого объекта из результата `items()` заголовки и описание. Также мы используем встроенный шаблонный фильтр `truncatewords`, чтобы ограничить описание статей тридцатью словами.

Теперь отредактируйте файл `blog/urls.py`, импортируйте класс `LatestPostsFeed`, который мы только что определили, и добавьте новый URL-шаблон.

```
from .feeds import LatestPostsFeed

urlpatterns = [
    # ...
    path('feed/', LatestPostsFeed(), name='post_feed'),
]
```

Перейдите на страницу `http://127.0.0.1:8000/blog/feed/`. Вы должны будете увидеть RSS-фид, который включает последние пять статей:

```
<?xml version="1.0" encoding="utf-8"?>
<rss xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">
  <channel>
    <title>My blog</title>
    <link>http://localhost:8000/blog/</link>
    <description>New posts of my blog.</description>
    <atom:link href="http://localhost:8000/blog/feed/" rel="self"/>
    <language>en-us</language>
    <lastBuildDate>Fri, 15 Dec 2017 09:56:40 +0000</lastBuildDate>
    <item>
      <title>Who was Django Reinhardt?</title>
      <link>http://localhost:8000/blog/2017/12/14/who-was-django-reinhardt/</link>
      <description>Who was Django Reinhardt.</description>
      <guid>http://localhost:8000/blog/2017/12/14/who-was-django-reinhardt/</guid>
    </item>
    ...
  </channel>
</rss>
```

Если вы откроете этот же URL в RSS-клиенте, то увидите статьи в удобном человеку интерфейсе.

Последний шаг – добавить ссылку на RSS-подписку на боковую панель. Откройте шаблон `blog/base.html` и добавьте следующую строку в `<div>` боковой панели под строкой с общим количеством статей:

```
<p><a href="{% url 'blog:post_feed' %}">Subscribe to my RSS feed</a></p>
```

Перейдите на `http://127.0.0.1:8000/blog/` в браузере и посмотрите на боковую панель. Новая ссылка подписки на RSS должна вести вас на фид статей.



Рис. 3.7 ❖ Ссылка на активацию RSS-подписки

ДОБАВЛЕНИЕ ПОЛНОТЕКСТОВОГО ПОИСКА

Давайте добавим полнотекстовый поиск в блог. ORM Django позволяет нам выполнять простые операции поиска совпадений (например, фильтр `contains` или его регистронезависимая версия `icontains`). С его помощью мы можем выполнить запрос для поиска статей, которые содержат слово `framework` в теле:

```
from blog.models import Post
Post.objects.filter(body__contains='framework')
```

Когда вы хотите использовать более сложные поисковые запросы, получать выборки по подобию или придавать разные веса разным условиям, необходимо использовать приложения для полнотекстового поиска.

Django предоставляет мощный поисковый механизм, рассчитанный на использование PostgreSQL и его возможностей полнотекстового поиска. Модуль `django.contrib.postgres` предоставляет специфичную для PostgreSQL функциональность, которая недоступна при использовании Django с другими СУБД. Подробнее про полнотекстовый поиск в PostgreSQL вы можете найти на странице <https://www.postgresql.org/docs/10/static/textsearch.html>.

i Хотя Django может работать с различными СУБД, в него встроен пакет, который реализует более широкий функционал при работе с PostgreSQL и недоступен для других СУБД.

Установка PostgreSQL

До этого момента мы использовали в проекте SQLite. Этой СУБД достаточно для разработки. Но для применения в реальных проектах рекомендуется использовать более мощные базы данных, такие как PostgreSQL, MySQL, Oracle. Мы заменим СУБД проекта на PostgreSQL для использования встроенных в нее возможностей полнотекстового поиска.

Если вы работаете под Linux, необходимо установить пакеты для работы PostgreSQL с Python:

```
sudo apt-get install libpq-dev python-dev
```

Затем установите PostgreSQL следующей командой:

```
sudo apt-get install postgresql postgresql-contrib
```

Если вы используете macOS или Windows, скачайте PostgreSQL с сайта <https://www.postgresql.org/download/> и установите ее.

Также необходимо установить Psycopg2 – адаптер PostgreSQL для Python. Выполните следующую команду:

```
pip install psycopg2==2.7.4
```

Давайте создадим пользователя для управления базами данных в PostgreSQL. Откройте терминал и выполните команду:

```
su postgres
createuser -dP blog
```


Вам необходимо будет ввести пароль для нового пользователя. После этого создайте базу данных `blog` и назначьте нового пользователя `blog` ее владельцем с помощью команды

```
createdb -E utf8 -U blog blog
```

Отредактируйте файл `settings.py` проекта и замените настройку `DATABASES` на использование PostgreSQL:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'blog',
        'USER': 'blog',
        'PASSWORD': '*****',
    }
}
```

Замените пароль из примера на ваш пароль для пользователя `blog`. Новая база данных пуста, поэтому нужно выполнить команду для применения всех миграций:

```
python manage.py migrate
```

Также создайте суперпользователя:

```
python manage.py createsuperuser
```

После этого вы можете запустить сервер разработки и авторизоваться на сайте администрирования `http://127.0.0.1:8000/admin/` по данным нового пользователя.

Так как вы сменили базу данных, в блоге нет статей. Заполните новую базу несколькими статьями, чтобы мы могли проверить работу поиска.

Простые поисковые запросы

Отредактируйте файл `settings.py` проекта и добавьте `django.contrib.postgres` в список установленных приложений `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    # ...
    'django.contrib.postgres',
]
```

Теперь мы можем выполнять поиск по одному полю с помощью `search` в `QuerySet`'е:

```
from blog.models import Post
Post.objects.filter(body__search='django')
```

Этот запрос использует PostgreSQL для создания вектора по полю `body` и фразы поиска `django`. В результат попадают записи, соответствующие поисковому вектору.

Поиск по нескольким полям

Кроме этого, вам может понадобиться поиск по нескольким полям. В этом случае необходимо определить `SearchVector`. Давайте построим вектор, который позволит осуществлять поиск по полям `title` и `body` модели `Post`:

```
from django.contrib.postgres.search import SearchVector
from blog.models import Post

Post.objects.annotate(search=SearchVector('title', 'body')).filter(search='django')
```

Используя аннотацию и определив `SearchVector` для обоих полей, мы добавляем функциональность поиска совпадений по заголовку и телу статей.

i Полнотекстовый поиск – это дорогая операция. Если вы применяете его для нескольких сотен записей и более, в базе данных необходимо определить индекс, состоящий из столбцов, участвующих в поиске. Django предоставляет класс `SearchVectorField` для такого поля модели. Более подробно об этом вы можете прочитать на странице <https://docs.djangoproject.com/en/2.0/ref/contrib/postgres/search/#performance>.

Обработчик поиска

Обработчик нужно добавить, для того чтобы пользователи могли пользоваться поиском. Для начала нужно создать форму. Отредактируйте файл `forms.py` приложения `blog` и добавьте в него следующий класс:

```
class SearchForm(forms.Form):
    query = forms.CharField()
```

Пользователи будут использовать поле формы `query` для задания поискового запроса. Отредактируйте файл `views.py` приложения `blog` и добавьте следующий код:

```
from django.contrib.postgres.search import SearchVector
from .forms import EmailPostForm, CommentForm, SearchForm

def post_search(request):
    form = SearchForm()
    query = None
    results = []
    if 'query' in request.GET:
        form = SearchForm(request.GET)
    if form.is_valid():
        query = form.cleaned_data['query']
        results = Post.objects.annotate(
            search=SearchVector('title', 'body'),
        ).filter(search=query)
    return render(request, 'blog/post/search.html', {'form': form,
                                                    'query': query,
                                                    'results': results})
```

В добавленном коде мы создаем объект формы `SearchForm`. Поисковый запрос будет отправляться методом `GET`, чтобы результирующий URL содержал в себе

фразу поиска в параметре `query`. Для того чтобы определить, отправлена ли форма для поиска, обращаемся к параметру запроса `query` из словаря `request.GET`. Когда запрос отправлен, мы иницилируем объект формы с параметрами из `request.GET`, проверяем корректность введенных данных. Если форма валидна, формируем запрос на поиск статей с использованием объекта `SearchVector` по двум полям: `title` и `body`.

Обработчик поиска готов. Теперь необходимо добавить шаблон для отображения формы и результатов поиска. Создайте новый файл `search.html` в каталоге шаблонов `/blog/post/` и вставьте в него следующий код:

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

{% block title %}Search{% endblock %}

{% block content %}
  {% if query %}
    <h1>Posts containing "{{ query }}"</h1>
    <h3>
      {% with results.count as total_results %}
        Found {{ total_results }} result {{ total_results|pluralize }}
      {% endwith %}
    </h3>
    {% for post in results %}
      <h4><a href="{{ post.get_absolute_url }}">{{ post.title }}</a></h4>
      {{ post.body|truncatewords:5 }}
    {% empty %}
      <p>There are no results for your query.</p>
    {% endfor %}
    <p><a href="{% url "blog:post_search" %}">Search again</a></p>
  {% else %}
    <h1>Search for posts</h1>
    <form action="." method="get">
      {{ form.as_p }}
      <input type="submit" value="Search">
    </form>
  {% endif %}
{% endblock %}
```

Так же как и в обработчике поиска, мы проверяем, была ли форма отправлена с параметром `query`. Перед отправкой формы отображаем ее и кнопку поиска. После того как пользователь ввел поисковую фразу и нажал на кнопку, показываем результат – количество найденных статей и фразу, по которой осуществлялся поиск.

Отредактируйте файл `urls.py` приложения `blog` и добавьте в него следующий URL:

```
path('search/', views.post_search, name='post_search'),
```

Откройте браузер на `http://127.0.0.1:8000/blog/search/`, чтобы увидеть форму (рис. 3.8).

Рис. 3.8 ❖ Форма поиска статей

Введите фразу и нажмите кнопку **Search**. Вы увидите статьи, подходящие под запрос:

Рис. 3.9 ❖ Результат поиска статей

Поздравляем! Только что вы реализовали полнотекстовый поиск на сайте.

Стемминг и ранжирование результатов

Django предоставляет класс `SearchQuery` для преобразования фраз в объект поискового запроса. По умолчанию все фразы пропускаются через алгоритм *стемминга*, который помогает получить больше совпадений. Кроме этого, вы можете отсортировать результаты по релевантности. PostgreSQL предоставляет функцию *ранжирования*, которая сортирует результаты на основе того, как часто встречаются фразы поиска и как близко друг к другу они находятся. Давайте воспользуемся этой особенностью СУБД. Отредактируйте файл `views.py` приложения `blog` и импортируйте классы:

```
from django.contrib.postgres.search import SearchVector, SearchQuery, SearchRank
```

Обратите внимание на следующие строки:

```
results = Post.objects.annotate(
    search=SearchVector('title', 'body'),
).filter(search=query)
```

Замените их с использованием классов, которые только что импортировали:

```
search_vector = SearchVector('title', 'body')
search_query = SearchQuery(query)
results = Post.objects.annotate(
    search=search_vector,
    rank=SearchRank(search_vector, search_query)
).filter(search=search_query).order_by('-rank')
```

В этом фрагменте мы создаем объект `SearchQuery`, фильтруем с его помощью результаты и используем `SearchRank` для ранжирования статей. Вы можете открыть <http://127.0.0.1:8000/blog/search/> и протестировать поиск и ранжирование по различным поисковым фразам. На скриншоте приведен пример поиска статей, отсортированных по количеству употребления слова `django` в заголовке и содержимом статей:

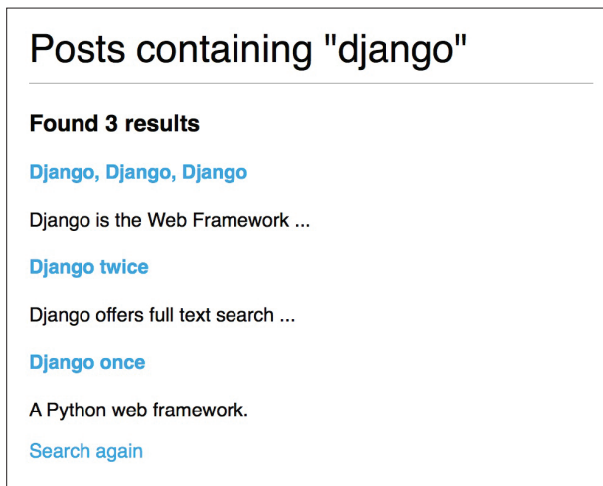


Рис. 3.10 ❖ Вывод отфильтрованного списка статей с учетом ранжирования

Взвешенные запросы

Мы можем повысить значимость некоторых векторов, чтобы совпадения по ним считались более релевантными, чем по остальным. Например, можно настроить поиск так, чтобы статьи с совпадениями в заголовке были в большем приоритете перед статьями с совпадениями в содержимом. Для этого отредактируйте `views.py` приложения `blog`:

```
search_vector = SearchVector('title', weight='A') + SearchVector('body', weight='B')
search_query = SearchQuery(query)
results = Post.objects.annotate(
    rank=SearchRank(search_vector, search_query)
).filter(rank__gte=0.3).order_by('-rank')
```

В этом примере мы применяем векторы по полям `title` и `body` с разным весом. По умолчанию используются веса D, C, B и A, которые соответствуют числам 0.1, 0.2, 0.4 и 1. Мы применили вес 1.0 для вектора по полю `title` и 0.4 – для вектора по полю `body`. В конце отбрасываем статьи с низким рангом и показываем только те, чей ранг выше 0.3.

Поиск с помощью триграмм

Еще одна возможность поиска – по сходству *триграмм*. Триграмма – это последовательность из трех символов. Вы можете измерить подобие двух строк, подсчитав количество совпадений триграмм. Такая метрика на практике является очень эффективной для определения подобия строк или слов во многих языках.

Чтобы использовать триграммы в PostgreSQL, необходимо подключить расширение `pg_trgm`. Выполните команду для входа в консоль PostgreSQL:

```
psql blog
```

Затем установите расширение `pg_trgm`, выполнив следующую команду:

```
CREATE EXTENSION pg_trgm
```

Давайте отредактируем обработчик, чтобы вместо векторов он использовал триграммы. Откройте файл `view.py` приложения `blog` и добавьте следующий импорт:

```
from django.contrib.postgres.search import TrigramSimilarity
```

После этого замените поисковый запрос модели `Post`:

```
results = Post.objects.annotate(
    similarity=TrigramSimilarity('title', query),
).filter(similarity__gt=0.3).order_by('-similarity')
```

Откройте страницу `http://127.0.0.1:8000/blog/search/` в браузере и убедитесь в эффективности поиска с помощью триграмм. Следующий пример демонстрирует случай типичной опечатки пользователя при вводе поисковой фразы. Несмотря на это, подходящая статья все же была найдена (рис. 3.11).

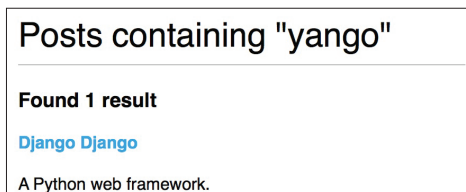


Рис. 3.11 ❖ Результаты поиска по ключевой фразе, содержащей опечатку

Теперь в вашем проекте есть мощный инструмент поиска. Более подробно полнотекстовый поиск рассмотрен в официальной документации Django на странице <https://docs.djangoproject.com/en/2.0/ref/contrib/postgres/search/>.

Другие инструменты полнотекстового поиска

Кроме PostgreSQL, можно использовать и другие приложения для полнотекстового поиска, например Solr или Elasticsearch. Они могут быть интегрированы в проект с помощью Haystack. Haystack – это приложение Django, которое работает как абстрактный уровень между Django и различными приложениями для поиска. Он предоставляет API, очень похожий на API QuerySet'ов Django, с помощью которого вы можете настраивать ваше приложение, не завязываясь на конкретную СУБД. Больше информации о Haystack можно найти на сайте <http://haystacksearch.org/>.

РЕЗЮМЕ

В этой главе мы узнали, как создавать собственные шаблонные теги и фильтры, чтобы расширить функциональность HTML-шаблонов. Также мы создали карту сайта для поисковых роботов и RSS-фиды, чтобы пользователи могли подписываться на рассылку статей нашего блога. В завершение добавили полнотекстовый поиск с помощью PostgreSQL.

В следующей главе мы узнаем, как создать социальную сеть, используя подсистему аутентификации Django, добавим собственные классы для профилей пользователей и интегрируем в проект систему авторизации других социальных сетей.

Глава 4

Создание социальной сети

В предыдущей главе мы узнали, как создать карту сайта и фиды, как добавить полнотекстовый поиск в приложение блога. В этой главе мы реализуем приложение социальной сети с авторизацией пользователей, восстановлением паролей, а также узнаем, как расширить модели профиля пользователей и добавить авторизацию посредством других социальных сетей.

В этой главе мы изучим следующие темы:

- использование системы аутентификации Django;
- создание обработчиков регистрации пользователей;
- расширение модели пользователям;
- добавление авторизации через другие социальные сети.

Давайте начнем с создания нового проекта.

СОЗДАНИЕ ПРОЕКТА ДЛЯ СОЦИАЛЬНОЙ СЕТИ

Мы реализуем приложение, которое даст пользователям возможность делиться фотографиями и картинками, которые они нашли в интернете. Для этого проекта нам понадобится:

- система аутентификации, для того чтобы пользователи могли регистрироваться в соцсети, заходить под своим аккаунтом, редактировать профиль и менять пароль;
- система подписок, чтобы пользователи могли наблюдать за обновлениями, происходящими у их друзей;
- отображение картинок, которыми делятся пользователи;
- лента для каждого пользователя, чтобы каждый мог видеть обновления у тех, на кого подписался.

Запуск проекта

Откройте терминал и создайте виртуальное окружение с помощью следующих команд:

```
mkdir env
virtualenv env/bookmarks
source env/bookmarks/bin/activate
```


Вывод консоли будет содержать название виртуального окружения в скобках, следовательно, оно успешно создано и активировано:

```
(bookmarks)Laptop:~ zenx$
```

Установите Django в виртуальное окружение:

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

Затем создайте новый проект `bookmarks` с помощью команды:

```
django-admin startproject bookmarks
```

После формирования начальной структуры проекта выполните команды, которые создадут приложение `account`:

```
cd bookmarks/  
django-admin startapp account
```

Помните: чтобы Django узнал о нашем приложении, нужно добавить его в список `INSTALLED_APPS` настроек `settings.py`. Разместите `account` перед другими в списке установленных приложений:

```
INSTALLED_APPS = [  
    'account.apps.AccountConfig',  
    # ...  
]
```

Позже мы добавим шаблоны для страниц аутентификации. Django ищет шаблоны в порядке приложений в `INSTALLED_APPS`, поэтому, размещая наше приложение первым, мы гарантируем, что именно его шаблоны будут использоваться по умолчанию вместо шаблонов, объявленных в других приложениях.

Теперь необходимо синхронизировать базу данных с моделями приложений, уже включенных в `INSTALLED_APPS`:

```
python manage.py migrate
```

В выводе консоли вы увидите, что Django применил все начальные миграции приложений. Мы будем создавать систему аутентификации с использованием классов, предоставляемых Django.

ИСПОЛЬЗОВАНИЕ СИСТЕМЫ АУТЕНТИФИКАЦИИ DJANGO

В фреймворк уже встроена система, которая может обрабатывать аутентификацию пользователей, сессии, права, разрешения и группы пользователей. В Django включены обработчики для типичных действий пользователя: входа и выхода из аккаунта, смены и восстановления пароля.

Система аутентификации реализована в пакете `django.contrib.auth` и используется другими пакетами из пакета `django.contrib`. Ранее мы уже использовали систему аутентификации в главе 1, чтобы создать суперпользователя для приложения блога и доступа к сайту администрирования.

Когда вы создаете новый проект Django с помощью команды `startproject`, система аутентификации автоматически добавляется в настройки проекта, поскольку `django.contrib.auth` включен в список установленных приложений. Кроме этого, в настройке `MIDDLEWARE` задаются два промежуточных слоя:

- `AuthenticationMiddleware` – связывает пользователей и запросы с помощью сессий;
- `SessionMiddleware` – обрабатывает сессию запроса.

Промежуточный слой – это класс с методами, которые выполняются при обработке каждого запроса и формировании ответа. Мы будем использовать их еще несколько раз на протяжении этой книги и даже создадим собственный промежуточный слой в главе 13.

Система аутентификации определяет следующие модели:

- `User` – модель пользователя с основными полями `username`, `password`, `email`, `first_name`, `last_name` и `is_active`;
- `Group` – модель группы пользователей;
- `Permission` – разрешение для пользователя или группы пользователей на выполнение определенных действий.

Система также включает обработчики и формы, которые мы применим чуть позже.

Создание обработчика авторизации

Начнем этот раздел с использования системы аутентификации Django, для того чтобы позволить пользователям входить на сайт под их аккаунтом. Обработчик должен выполнить следующие действия:

- 1) получить логин и пароль пользователя из формы;
- 2) сверить данные с теми, что содержатся в базе данных;
- 3) проверить, активен ли пользователь;
- 4) авторизовать пользователя и создать сессию.

Для начала добавим форму авторизации. Создайте файл `forms.py` в приложении `account` и опишите в нем следующий класс:

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

Это форма, которая будет использоваться для авторизации пользователя. Обратите внимание, что мы используем виджет `PasswordInput`, который будет сформирован в HTML как элемент `<input>` с атрибутом `type="password"`, поэтому браузер будет работать с ним как с полем пароля. Отредактируйте файл `views.py` приложения `account` и добавьте в него код:

```
from django.http import HttpResponse
from django.shortcuts import render
from django.contrib.auth import authenticate, login
```

```
from .forms import LoginForm

def user_login(request):
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            user = authenticate(request,
                               username=cd['username'],
                               password=cd['password'])

            if user is not None:
                if user.is_active:
                    login(request, user)
                    return HttpResponse('Authenticated successfully')
                else:
                    return HttpResponse('Disabled account')
            else:
                return HttpResponse('Invalid login')
        else:
            form = LoginForm()
    return render(request, 'account/login.html', {'form': form})
```

Наш обработчик авторизации работает следующим образом. Когда `user_login` вызывается с GET-запросом, мы создаем новую форму логина выражением `form = LoginForm()` и отображаем ее в шаблоне. Когда пользователь отправляет форму POST-запросом, мы обрабатываем ее:

- 1) создаем объект формы с данными `form = LoginForm(request.POST)`;
- 2) проверяем, правильно ли заполнена форма `form.is_valid()`. Если она не валидна, появляется сообщение с ошибками (например, если пользователь не заполнил какое-нибудь поле);
- 3) если данные введены верно, сверяем их с данными в базе с помощью функции `authenticate()`. Она принимает аргументы `request`, `username` и `password` и возвращает объект пользователя `User`, если он успешно аутентифицирован. В противном случае вернется `None`. Если пользователь не был аутентифицирован, возвращаем объект `HttpResponse` с сообщением о некорректном логине или пароле;
- 4) если пользователь был аутентифицирован, проверяем, активен ли он, через атрибут модели пользователя Django, `is_active`. Если пользователь неактивный, возвращаем `HttpResponse` с соответствующим сообщением;
- 5) если пользователь активный, авторизуем его на сайте. Это происходит посредством вызова функции `login()`, которая запоминает пользователя в сессии. Затем возвращаем `HttpResponse` с сообщением об успешной авторизации.



Обратите внимание на различия между `authenticate()` и `login()`. Функция `authenticate()` проверяет идентификационные данные и возвращает объект `User`, если они корректны. Функция `login()` сохраняет текущего пользователя в сессии.

Теперь необходимо описать URL-шаблон для этого обработчика. Создайте `urls.py` в папке приложения `account` и добавьте в него следующий код:

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

urlpatterns = [
    # Обработчики действий со статьями.
    path('login/', views.user_login, name='login'),
]
```

Отредактируйте файл `urls.py` проекта, расположенного в папке `bookmarks`: импортируйте `include` и добавьте путь до приложения `account`:

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

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

Теперь обработчик логина доступен по URL'у. Самое время создать для него HTML-шаблон. Так как у нас нет ни одного шаблона, начнем с определения базового, от которого пойдут остальные. Создайте следующую структуру каталогов и файлов в папке приложения `account`:

```
templates/
  account/
    login.html
  base.html
```

Отредактируйте файл `base.html` и добавьте в него следующий код:

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
  <head>
    <title>{% block title %}{% endblock %}</title>
    <link href="{% static 'css/base.css' %}" rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <span class="logo">Bookmarks</span>
    </div>
    <div id="content">
      {% block content %}{% endblock %}
    </div>
  </body>
</html>
```

Это будет базовый шаблон для нашего веб-сайта. По аналогии с тем, как мы делали в предыдущем проекте, подключите в `base.html` CSS-стили. Вы можете найти файлы стилей в коде – примере для этой главы. Скопируйте папку

static/ приложения account из кода-примера в соответствующую папку вашего проекта.

Базовый шаблон определяет блоки title и content, которые можно будет переопределять в шаблонах, унаследованных от base.html.

Давайте добавим шаблон для формы логина. Откройте файл account/login.html и добавьте следующий фрагмент:

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

{% block title %}Log-in{% endblock %}

{% block content %}
<h1>Log-in</h1>
<p>Please, use the following form to log-in:</p>
<form action="." method="post">
  {{ form.as_p }}
  {% csrf_token %}
  <p><input type="submit" value="Log in"></p>
</form>
{% endblock %}
```

Этот шаблон отображает форму, созданную в обработчике. Так как мы будем посылать форму методом POST, в форму добавлен тег {% csrf_token %} для осуществления защиты от CSRF-атак, о которых мы уже узнали из главы 2.

Сейчас в базе данных нет пользователей, поэтому необходимо создать суперпользователя для доступа к сайту администрирования и управления другими пользователями. Откройте терминал и выполните python manage.py createsuperuser. Заполните желаемые логин, электронную почту и пароль. Затем запустите сервер разработки командой python manage.py runserver и откройте в браузере страницу по адресу http://127.0.0.1:8000/admin/. Зайдите на сайт администрирования под учетной записью суперпользователя. Вы увидите блок системы аутентификации Django с моделями User и Group:

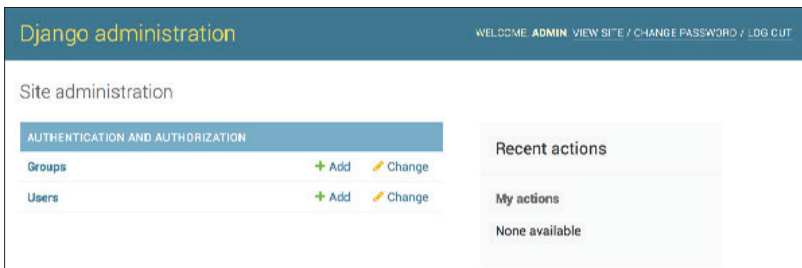


Рис. 4.1 ❖ Разделы **Users** (Пользователи) и **Groups** (Группы) на сайте администрирования

Создайте нового пользователя через сайт администрирования и откройте страницу http://127.0.0.1:8000/account/login/. Вы увидите форму логина:

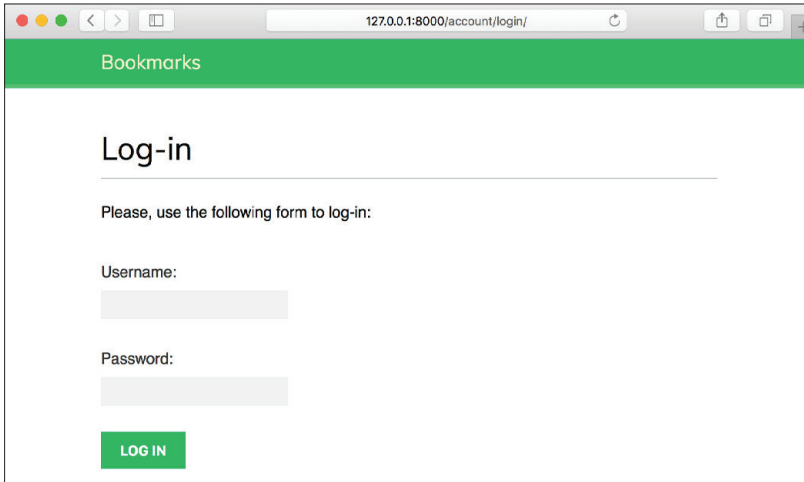


Рис. 4.2 ❖ Страница авторизации

Заполните ее, оставив одно поле пустым, и попытайтесь отправить форму. Вы увидите, что форма не будет обработана, а Django отобразит сообщение об ошибке:

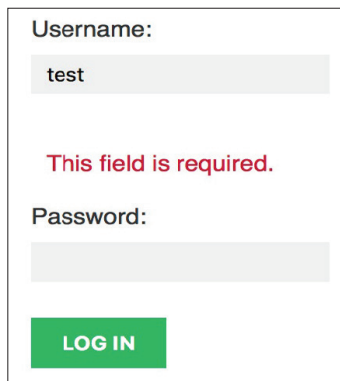


Рис. 4.3 ❖ Форма авторизации с ошибками

Стоит отметить, что некоторые современные браузеры могут предотвратить отправку формы с пустыми обязательными полями. Это происходит потому, что браузеры перед отправкой валидируют форму, основываясь на типах полей. Если валидация не проходит, браузер информирует пользователя о полях, которые необходимо заполнить.

Если вы введете логин несуществующего пользователя или неправильный пароль, то получите соответствующее сообщение.

После того как вы введете корректные идентификационные данные и отправите форму, сервер вернет сообщение об успешной авторизации:

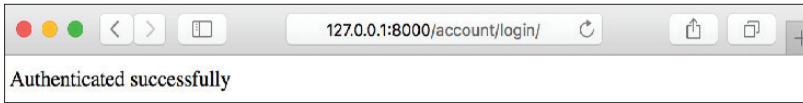


Рис. 4.4 ❖ Сообщение об успешной авторизации

Использование обработчиков аутентификации Django

Django реализует несколько форм и обработчиков в подсистеме аутентификации, которые легко использовать. Обработчик логина, который мы только что создали, – хороший пример, благодаря которому можно понять процесс аутентификации пользователей в Django. Но в большинстве случаев достаточно использовать стандартные средства, уже реализованные в Django.

Для работы с аутентификацией Django предоставляет обработчики-классы. Все они описаны в `django.contrib.auth.views`:

- `LoginView` – обработчик входа пользователя в его аккаунт;
- `LogoutView` – обработчик выхода пользователя из-под его учетной записи.

В Django реализованы обработчики смены пароля пользователем:

- `PasswordChangeView` – обрабатывает форму смены пароля;
- `PasswordChangeDoneView` – обработчик, на который будет перенаправлен пользователь после успешной смены пароля.

Также реализованы обработчики для восстановления пароля пользователя:

- `PasswordResetView` – обработчик восстановления пароля. Он генерирует временную ссылку с токеном и отправляет ее на электронную почту пользователя;
- `PasswordResetDoneView` – отображает страницу с сообщением о том, что ссылка восстановления пароля была отправлена на электронную почту;
- `PasswordResetConfirmView` – позволяет пользователю указать новый пароль;
- `PasswordResetCompleteView` – отображает сообщение об успешной смене пароля.

Вышеописанные обработчики экономят время, когда на сайте нужно реализовать функционал аутентификации. Все они используют классы форм и HTML-шаблоны по умолчанию, но при желании их можно переопределить и воспользоваться собственными.

Больше информации о стандартных обработчиках системы аутентификации можно найти на странице <https://docs.djangoproject.com/en/2.0/topics/auth/default/#all-authentication-views>.

Обработчики входа и выхода

Отредактируйте файл `urls.py` приложения `account` таким образом:

```
from django.urls import path
from django.contrib.auth import views as auth_views
```

```

from . import views

urlpatterns = [
    # Определенные ранее обработчики.
    # path('login/', views.user_login, name='login'),
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]

```

Мы закомментировали шаблон для обработчика `user_login`, чтобы использовать `LoginView`, предоставляемый Django. Также мы добавили шаблон для обработчика выхода пользователя из-под учетной записи – `LogoutView`.

Создайте новую папку внутри `templates/` приложения `account`, назовите ее `registration`. Относительно этого каталога Django будет искать шаблоны страниц аутентификации.

Пакет `django.contrib.admin` включает несколько шаблонов системы аутентификации, которые используются для сайта администрирования. Так как мы разместили приложение `account` в начале списка `INSTALLED_APPS`, при совпадении путей Django будет использовать наши шаблоны вместо тех, которые определены в других приложениях.

Давайте определим шаблон страницы с формой ввода логина и пароля. Создайте новый файл `login.html` в папке `templates/registration` и добавьте в него следующий код:

```

{% extends "base.html" %}

{% block title %}Log-in{% endblock %}

{% block content %}
<h1>Log-in</h1>
{% if form.errors %}
  <p>Your username and password didn't match.Please try again.</p>
{% else %}
  <p>Please, use the following form to log-in:</p>
{% endif %}
<div class="login-form">
  <form action="{% url 'login' %}" method="post">
    {{ form.as_p }}
    {% csrf_token %}
    <input type="hidden" name="next" value="{{ next }}" />
    <p><input type="submit" value="Log-in"></p>
  </form>
</div>
{% endblock %}

```

Этот шаблон страницы логина похож на тот, который мы создали ранее. По умолчанию Django использует форму Authentication Form из модуля `django.contrib.auth.forms`. Эта форма пытается аутентифицировать пользователя и генерирует исключение валидации, если попытка не удалась. В этом случае мы можем использовать условие `{% ifform.errors %}`, чтобы убедиться, верно ли указаны данные пользователя. Обратите внимание, что мы добавили скрытый

элемент `<input>`, чтобы отправить данные под именем `next` (куда перенаправить пользователя после авторизации).

Параметр `next` должен быть корректным URL'ом. Если он указан, обработчик логина Django перенаправит пользователя по этому URL'у после успешной авторизации.

Теперь создайте файл `logged_out.html` в каталоге `registration` и добавьте в него следующий фрагмент кода:

```
{% extends "base.html" %}
{% block title %}Logged out{% endblock %}
{% block content %}
<h1>Logged out</h1>
<p>You have been successfully logged out.
  You can <a href="{% url "login" %}">log-in again</a>.
</p>
{% endblock %}
```

Этот шаблон будет отображаться после того, как пользователь выйдет из своего аккаунта.

Мы добавили шаблоны URL'ов и HTML-шаблоны для обработчиков входа и выхода. Наш сайт готов к авторизации пользователей посредством системы аутентификации Django.

Теперь мы создадим обработчик для отображения рабочего стола, который пользователь увидит при входе в свой аккаунт. Откройте файл `views.py` приложения `account` и добавьте в него:

```
from django.contrib.auth.decorators import login_required
@login_required
def dashboard(request):
    return render(request, 'account/dashboard.html', {'section': 'dashboard'})
```

Наш обработчик обернут в декоратор `login_required`. Он проверяет, авторизован ли пользователь. Если пользователь авторизован, Django выполняет обработку. В противном случае пользователь перенаправляется на страницу логина. При этом в GET-параметре задается `next` -адрес запрашиваемой страницы. Таким образом, после успешного прохождения авторизации пользователь будет перенаправлен на страницу, куда он пытался попасть. Именно для этих целей мы вставили скрытое поле `next` в форму логина.

Также мы добавили переменную контекста `section`, с помощью которой сможем узнать, какой раздел сайта сейчас просматривает пользователь.

Теперь нам нужно добавить шаблон для отображения рабочего стола. Создайте новый файл `dashboard.html` внутри каталога `templates/account/`. Он должен выглядеть таким образом:

```
{% extends "base.html" %}
{% block title %}Dashboard{% endblock %}
{% block content %}
```

```
<h1>Dashboard</h1>
<p>Welcome to your dashboard.</p>
{% endblock %}
```

Затем добавьте шаблон обработчика в файл `urls.py` приложения `account`:

```
urlpatterns = [
    # ...
    path('', views.dashboard, name='dashboard'),
]
```

Отредактируйте файл `settings.py` проекта и добавьте в него строки:

```
LOGIN_REDIRECT_URL = 'dashboard'
LOGIN_URL = 'login'
LOGOUT_URL = 'logout'
```

Эти настройки отвечают за следующий функционал:

- `LOGIN_REDIRECT_URL` – указывает адрес, куда Django будет перенаправлять пользователя при успешной авторизации, если не указан `GET`-параметр `next`;
- `LOGIN_URL` – адрес, куда нужно перенаправлять пользователя для входа в систему, например из обработчиков с декоратором `login_required`;
- `LOGOUT_URL` – адрес, перейдя по которому, пользователь выйдет из своего аккаунта.

Давайте соединим все вместе, добавив ссылки на вход и выход в базовый шаблон. Чтобы отображать нужную ссылку, следует определить, выполнил ли текущий пользователь вход в свой аккаунт или он является анонимным. Текущий пользователь задается в объекте `HttpRequest` промежуточным слоем. Мы можем в любой момент обратиться к нему через `request.user` и получим объект типа `User`. Неавторизованный пользователь является объектом типа `AnonymousUser`. Чтобы определить, с каким типом пользователя мы имеем дело, достаточно обратиться к атрибуту `is_authenticated`.

Отредактируйте файл `base.html` и измените элемент `<div>` с идентификатором `header`:

```
<div id="header">
  <span class="logo">Bookmarks</span>
  {% if request.user.is_authenticated %}
    <ul class="menu">
      <li {% if section == "dashboard" %} class="selected"{% endif %}>
        <a href="{% url "dashboard" %}">My dashboard</a>
      </li>
      <li {% if section == "images" %} class="selected"{% endif %}>
        <a href="#">Images</a>
      </li>
      <li {% if section == "people" %} class="selected"{% endif %}>
        <a href="#">People</a>
      </li>
    </ul>
  {% endif %}
  <span class="user">
```

```

{% if request.user.is_authenticated %}
  Hello {{ request.user.first_name }},
  <a href="{% url 'logout' %}">Logout</a>
{% else %}
  <a href="{% url 'login' %}">Log-in</a>
{% endif %}
</span>
</div>

```

Как вы можете видеть в предыдущем фрагменте, мы отображаем меню сайта только в том случае, если пользователь авторизован. Также мы проверяем текущий раздел, чтобы добавить CSS-класс `selected` в соответствующий ``-элемент для выделения цветом. Если пользователь авторизован, мы отображаем его имя в шапке. В противном случае даем ссылку на страницу логина.

Откройте страницу `http://127.0.0.1:8000/account/login/`. Вы должны будете увидеть страницу входа. Введите корректные логин и пароль и нажмите кнопку **Login**. Вы видите рабочий стол:

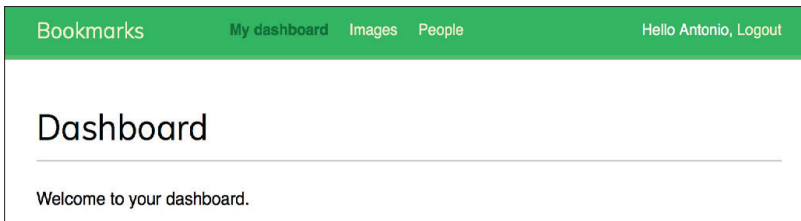


Рис. 4.5 ❖ Рабочий стол авторизованного пользователя

Обратите внимание, что раздел `Mydashboard` выделен CSS-стилями, т. к. у него есть дополнительный класс `selected`. Так как наш пользователь авторизован, его имя отображается в правой части шапки. Кликните на ссылку **Logout**. Вы увидите такую страницу:

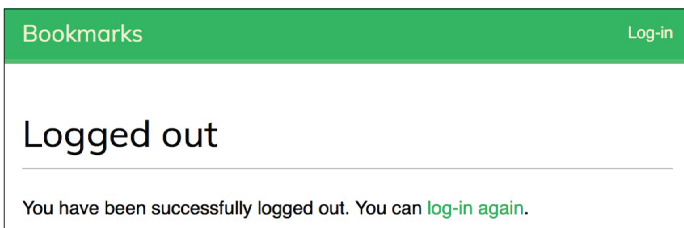


Рис. 4.6 ❖ Сообщение об успешном выходе из аккаунта

На скриншоте вы можете заметить, что пользователь вышел из своего аккаунта (меню сайта больше не отображается). Вместо этого в правой части шапки указана ссылка на страницу логина.

i Если вместо нашей страницы вы видите страницу административного сайта Django, проверьте настройку `INSTALLED_APPS` и убедитесь, что приложение `django.contrib.admin` указано после приложения `account`. В обоих этих приложениях шаблоны могут быть получены одним способом, причем Django будет использовать первый шаблон, который найдет.

Обработчики смены пароля

Для удобства пользователя необходимо добавить возможность менять пароль, после того как он войдет в свой аккаунт. Для этой цели мы воспользуемся обработчиками Django. Нужно будет только описать шаблоны URL'ов и HTML-страницы. Откройте файл `urls.py` приложения `account` и добавьте следующие шаблоны URL'ов:

```
# Шаблоны для доступа к обработчикам смены пароля.
path('password_change/',
     auth_views.PasswordChangeView.as_view(),
     name='password_change'),
path('password_change/done/',
     auth_views.PasswordChangeDoneView.as_view(),
     name='password_change_done').
```

Обработчик `PasswordChangeView` будет проверять форму смены пароля, а `PasswordChangeDoneView` – отображать сообщение о том, что операция выполнена успешно. Давайте создадим шаблоны для них.

Добавьте новый файл `password_change_form.html` в каталог `templates/registration/` приложения `account`. Скопируйте в него следующий код:

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

{% block title %}Change you password{% endblock %}

{% block content %}
<h1>Change you password</h1>
<p>Use the form below to change your password.</p>
<form action="." method="post">
  {{ form.as_p }}
  <p><input type="submit" value="Change"></p>
  {% csrf_token %}
</form>
{% endblock %}
```

Шаблон `password_change_form.html` отображает форму для смены пароля. Теперь создайте файл `password_change_done.html` в этой же папке и добавьте в него следующий код:

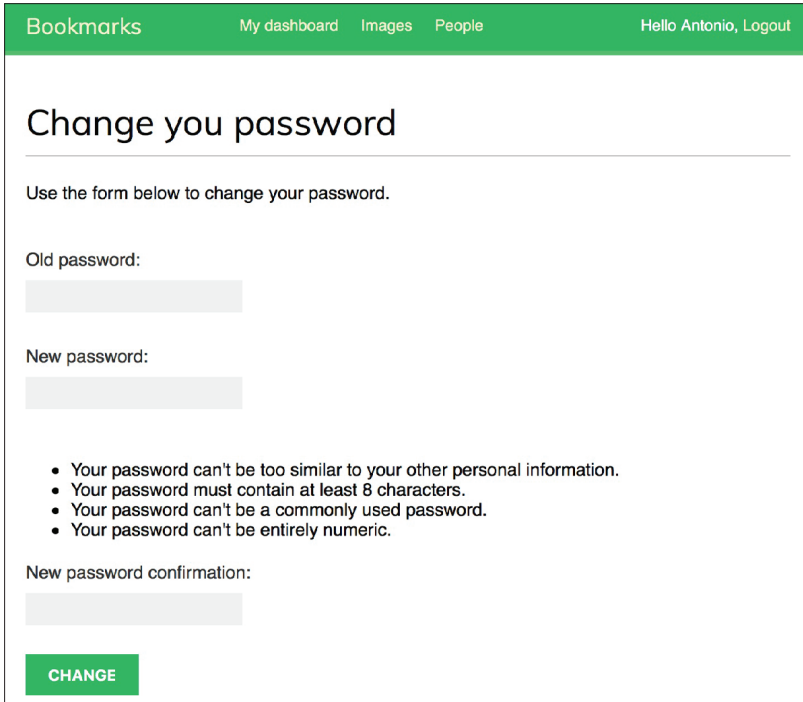
```
{% extends "base.html" %}

{% block title %}Password changed{% endblock %}

{% block content %}
<h1>Password changed</h1>
<p>Your password has been successfully changed.</p>
{% endblock %}
```

Шаблон `password_change_done.html` содержит простое сообщение, которое говорит об успешной смене пароля.

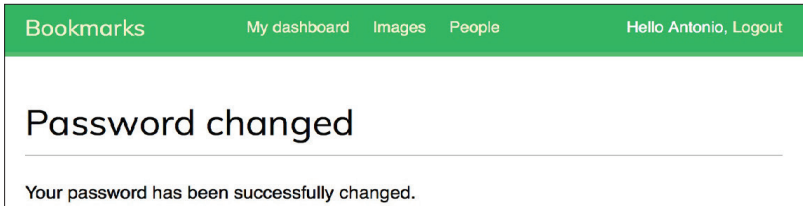
Откройте страницу `http://127.0.0.1:8000/account/password_change/`. Если ваш текущий пользователь не авторизован, то Django перенаправит вас на страницу логина. После процедуры авторизации вы увидите форму смены пароля:



The screenshot shows a web browser window with a green header bar. The header contains the text "Bookmarks" on the left, "My dashboard Images People" in the center, and "Hello Antonio, Logout" on the right. The main content area has a heading "Change you password" followed by a horizontal line. Below the heading is the text "Use the form below to change your password." There are three input fields: "Old password:", "New password:", and "New password confirmation:". Below the "New password:" field is a bulleted list of password requirements: "Your password can't be too similar to your other personal information.", "Your password must contain at least 8 characters.", "Your password can't be a commonly used password.", and "Your password can't be entirely numeric." At the bottom left of the form is a green button with the text "CHANGE".

Рис. 4.7 ❖ Форма смены пароля пользователя

Заполните ее, указав ваши текущий и новый пароли, нажмите кнопку **CHANGE**. В результате вы увидите такую страницу:



The screenshot shows a web browser window with a green header bar. The header contains the text "Bookmarks" on the left, "My dashboard Images People" in the center, and "Hello Antonio, Logout" on the right. The main content area has a heading "Password changed" followed by a horizontal line. Below the heading is the text "Your password has been successfully changed."

Рис. 4.8 ❖ Сообщение об успешной смене пароля

Выйдите из профиля и попытайтесь авторизоваться с новым паролем, чтобы убедиться, что все корректно работает.

Обработчики восстановления пароля

Добавьте следующие шаблоны URL'ов в файл `urls.py` приложения `account`:

```
# Обработчики восстановления пароля.
path('password_reset/',
     auth_views.PasswordResetView.as_view(),
     name='password_reset'),
path('password_reset/done/',
     auth_views.PasswordResetDoneView.as_view(),
     name='password_reset_done'),
path('reset/<uidb64>/<token>/',
     auth_views.PasswordResetConfirmView.as_view(),
     name='password_reset_confirm'),
path('reset/done/',
     auth_views.PasswordResetCompleteView.as_view(),
     name='password_reset_complete').
```

Добавьте в каталог `templates/registration/` приложения `account` новый файл `password_reset_form.html` с формой восстановления пароля:

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

{% block title %}Reset your password{% endblock %}

{% block content %}
<h1>Forgotten your password?</h1>
<p>Enter your e-mail address to obtain a new password.</p>
<form action="." method="post">
  {{ form.as_p }}
  <p><input type="submit" value="Send e-mail"></p>
  {% csrf_token %}
</form>
{% endblock %}
```

Теперь добавьте еще один файл в эту же папку с названием `password_reset_email.html` и содержимым:

```
Someone asked for password reset for email {{ email }}. Follow the link
below:
{{ protocol }}://{{ domain }}{% url "password_reset_confirm" uidb64=uid
token=token %}
Your username, in case you've forgotten: {{ user.get_username }}
```

Этот шаблон будет использоваться для формирования сообщения, отправляемого пользователю на электронную почту при восстановлении пароля.

Добавьте в эту же папку файл `password_reset_done.html`, содержащий сообщение о смене пароля:

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

{% block title %}Reset your password{% endblock %}
```

```
{% block content %}
<h1>Reset your password</h1>
<p>We've emailed you instructions for setting your password.</p>
<p>If you don't receive an email, please make sure you've
    entered the address you registered with.</p>
{% endblock %}
```

Создайте еще один файл, `password_reset_confirm.html`, с содержимым:

```
{% extends "base.html" %}
{% block title %}Reset your password{% endblock %}
{% block content %}
<h1>Reset your password</h1>
{% if validlink %}
<p>Please enter your new password twice:</p>
<form action="." method="post">
    {{ form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Change my password" /></p>
</form>
{% else %}
<p>The password reset link was invalid, possibly because it
    has already been used. Please request a new password reset.</p>
{% endif %}
{% endblock %}
```

Мы проверяем, является ли ссылка на восстановление пароля, по которой перешел пользователь, корректной. Обработчик `PasswordResetConfirmView` добавляет ее в контекст шаблона `password_reset_confirm.html`. Если ссылка правильная и действительно была сгенерирована Django, мы отображаем форму восстановления пароля.

Создайте еще один шаблон с названием `password_reset_complete.html`. Он будет содержать сообщение об успешной смене пароля:

```
{% extends "base.html" %}
{% block title %}Password reset{% endblock %}
{% block content %}
<h1>Password set</h1>
<p>Your password has been set. You can
<a href="{% url "login" %}">log in now</a></p>
{% endblock %}
```

Наконец, отредактируйте файл `registration/login.html` приложения `account` и добавьте ссылку после элемента `<form>` формы логина:

```
<p><a href="{% url "password_reset" %}">Forgotten your password?</a></p>
```

Откройте в браузере `http://127.0.0.1:8000/account/login/` и кликните на ссылку **Forgotten your password?**. Вы увидите страницу восстановления пароля:

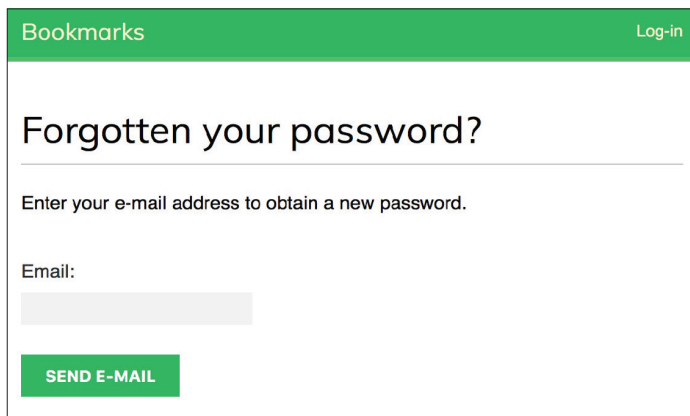
A screenshot of a web page with a green header bar containing the text 'Bookmarks' on the left and 'Log-in' on the right. The main content area has a white background. At the top, the heading 'Forgotten your password?' is displayed in a large, black, sans-serif font. Below the heading is a horizontal line. Underneath the line, the text 'Enter your e-mail address to obtain a new password.' is centered. Below this text is the label 'Email:' followed by a light gray rectangular input field. At the bottom of the form is a green rectangular button with the white text 'SEND E-MAIL'.

Рис. 4.9 ❖ Страница ввода адреса электронной почты для восстановления пароля

На этом этапе важным шагом является добавление конфигурации SMTP-сервера в `settings.py`, чтобы наше приложение могло отправлять электронные письма. Мы уже познакомились с тем, как это делать, в главе 2. Но во время разработки и тестирования достаточно настроить Django на отправку сообщений в консоль вместо использования SMTP-сервера. Для этих целей Django предоставляет специальный бэкэнд, который необходимо подключить в файле настроек проекта, `settings.py`:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Настройка `EMAIL_BACKEND` определяет класс, который будет использоваться для отправки электронной почты.

Вернитесь в браузер, введите e-mail существующего пользователя и нажмите кнопку **SEND E-MAIL**. Django отобразит такую страницу:

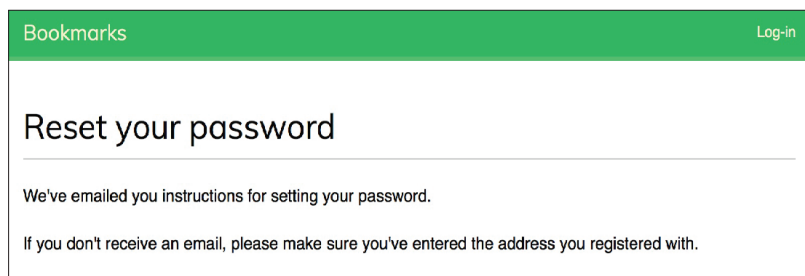
A screenshot of a web page with a green header bar containing the text 'Bookmarks' on the left and 'Log-in' on the right. The main content area has a white background. At the top, the heading 'Reset your password' is displayed in a large, black, sans-serif font. Below the heading is a horizontal line. Underneath the line, the text 'We've emailed you instructions for setting your password.' is centered. Below this text is another line of centered text: 'If you don't receive an email, please make sure you've entered the address you registered with.'

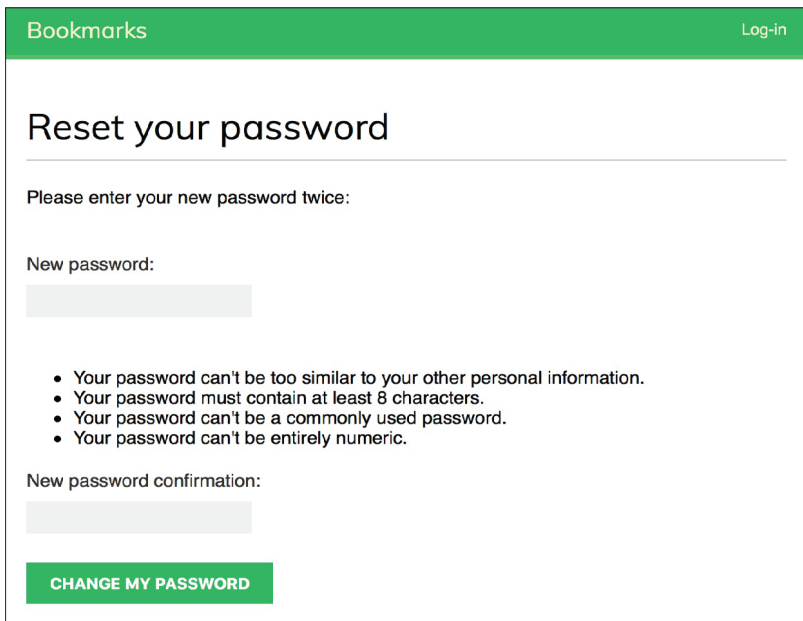
Рис. 4.10 ❖ Сообщение о том, что инструкция по восстановлению пароля отправлена на электронную почту

Посмотрите в терминал, где запущен сервер для разработки. Вы увидите сгенерированное электронное сообщение:

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: user@domain.com
Date: Fri, 15 Dec 2017 14:35:08 -0000
Message-ID: <20150924143508.62996.55653@zenx.local>
```

Someone asked for password reset for email user@domain.com. Follow the link below:
<http://127.0.0.1:8000/account/reset/MQ/45f-9c3f30caafd523055fcc/>
Your username, in case you've forgotten: zenx

Оно сформировано по шаблону `password_reset_email.html`, который мы добавили ранее. URL для восстановления пароля содержит токен, который автоматически генерирует Django. Скопируйте этот адрес и откройте страницу в браузере. Вы должны будете увидеть страницу с формой восстановления пароля:



Bookmarks Log-in

Reset your password

Please enter your new password twice:

New password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

New password confirmation:

CHANGE MY PASSWORD

Рис. 4.11 ❖ Форма восстановления пароля

Эта страница формируется из шаблона `password_reset_confirm.html`. Заполните форму и отправьте ее, нажав кнопку **CHANGE MY PASSWORD**. Django зашифрует новый пароль, сохранит его в базе данных и отобразит страницу об успешной смене пароля:

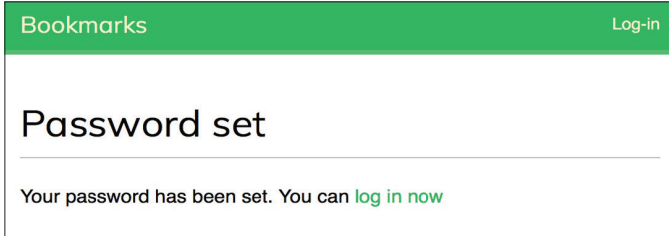


Рис. 4.12 ❖ Сообщение об успешной смене пароля

Теперь вы можете войти в свою учетную запись с помощью нового пароля.

Токен, который генерирует Django для смены пароля, может быть использован только один раз. Если вы повторно откроете ссылку, отправленную на почту, то увидите сообщение о том, что используется устаревший токен.

Мы добавили в проект обработчики системы аутентификации Django, подходящие в большинстве случаев. Но если вы хотите реализовать другое поведение, то всегда можно создать собственные обработчики.

Django также предоставляет шаблоны URL'ов для обработчиков аутентификации. Вы можете закомментировать то, что мы указывали в файле `urls.py` для приложения `account`, и вместо этого подключить пути приложения `django.contrib.auth.urls`:

```
from django.urls import path, include
# ...

urlpatterns = [
    # ...
    path('', include('django.contrib.auth.urls')),
]
```

Найти список всех predefined в Django шаблонов URL'ов можно на странице <https://github.com/django/django/blob/stable/2.0.x/django/contrib/auth/urls.py>.

РЕГИСТРАЦИЯ И ПРОФИЛИ ПОЛЬЗОВАТЕЛЕЙ

Существующие в системе пользователи могут входить, выходить из аккаунта и восстанавливать пароль. Теперь мы добавим обработчик, чтобы позволить им регистрироваться в системе и давать дополнительные сведения в профиле.

Регистрация пользователей

Давайте создадим простой обработчик регистрации пользователей на сайте. Для начала добавим форму, чтобы новый пользователь мог ввести свои логин, имя и пароль. Отредактируйте файл `forms.py` приложения `account`, дописав в него следующие строки:

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

class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password', widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password', widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ('username', 'first_name', 'email')

    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Passwords don\'t match.')
        return cd['password2']
```

Мы создали модельную форму для пользователя, включив в нее только поля `username`, `first_name` и `email`. Они будут валидироваться в соответствии с типом полей модели. Например, если пользователь введет логин, который уже используется, ему вернется сообщение с указанием на ошибку, из-за того что в модели поле `username` определено как `unique=True`. Также мы добавили два поля: `password` и `password2` – для задания и подтверждения пароля. В методе `clean_password2()` мы проверяем, совпадают ли оба пароля. Если они отличаются, то будет возвращена ошибка. Мы можем добавлять методы с названием вида `clean_<fieldname>()` для любого поля формы, чтобы Django проверял соответствующее поле и в случае некорректных данных привязывал ошибку к нему. Кроме того, у форм реализован метод `clean()`, который проверяет корректность всей формы. Он применяется, когда необходимо выполнить проверку взаимосвязанных полей.

Django предоставляет класс `UserCreationForm`, который расположен в модуле `django.contrib.auth.forms`. Он очень похож на класс формы, созданный нами.

Отредактируйте файл `views.py` в приложении `account` и добавьте следующий код:

```
from .forms import LoginForm, UserRegistrationForm

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Создаем нового пользователя, но пока не сохраняем в базу данных.
            new_user = user_form.save(commit=False)
            # Задаем пользователю зашифрованный пароль.
            new_user.set_password(user_form.cleaned_data['password'])
```

```

# Сохраняем пользователя в базе данных.
new_user.save()
return render(request,
              'account/register_done.html',
              {'new_user': new_user})

else:
    user_form = UserRegistrationForm()
    return render(request, 'account/register.html', {'user_form': user_form})

```

Обработчик регистрации нового пользователя предельно простой. Вместо сохранения пароля пользователя «как есть», мы используем метод `set_password()` модели `User`. Он сохранит пароль в зашифрованном виде.

Теперь отредактируйте файл `urls.py` приложения `account`, добавив следующий шаблон:

```
path('register/', views.register, name='register').
```

Наконец, давайте создадим HTML-шаблоны. Добавьте файл `register.html` в каталог приложения `account/` с таким содержимым:

```

{% extends "base.html" %}

{% block title %}Create an account{% endblock %}

{% block content %}
<h1>Create an account</h1>
<p>Please, sign up using the following form:</p>
<form action="." method="post">
  {{ user_form.as_p }}
  {% csrf_token %}
  <p><input type="submit" value="Create my account"></p>
</form>
{% endblock %}

```

И еще один шаблон `register_done.html`:

```

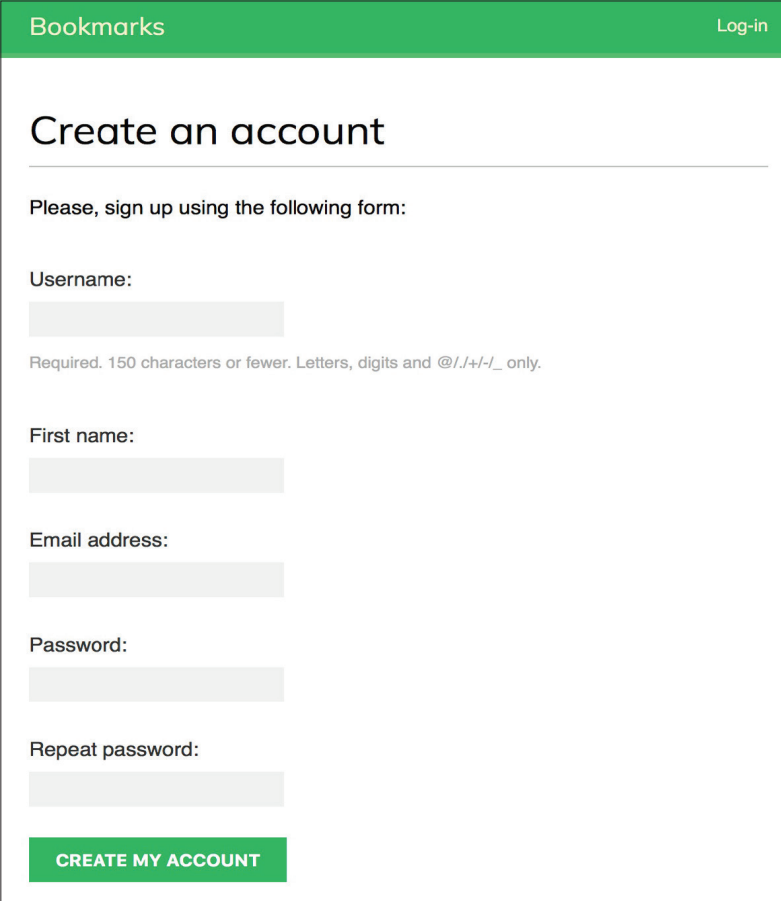
{% extends "base.html" %}

{% block title %}Welcome{% endblock %}

{% block content %}
<h1>Welcome {{ new_user.first_name }}!</h1>
<p>Your account has been successfully created.
  Now you can <a href="{% url "login" %}">log in</a>.</p>
{% endblock %}

```

Откройте в браузере `http://127.0.0.1:8000/account/register/`. Вы увидите страницу регистрации пользователя:



Bookmarks Log-in

Create an account

Please, sign up using the following form:

Username:

Required. 150 characters or fewer. Letters, digits and @/./+/_ only.

First name:

Email address:

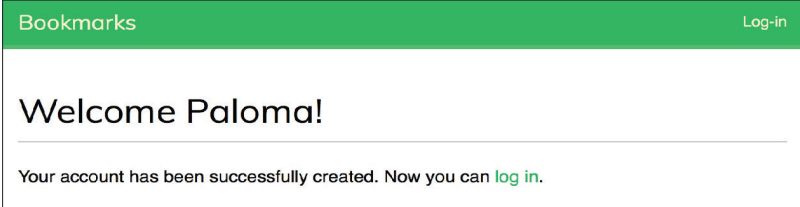
Password:

Repeat password:

CREATE MY ACCOUNT

Рис. 4.13 ❖ Форма регистрации нового пользователя

Заполните форму и нажмите на кнопку **CREATE MY ACCOUNT**. Если все поля заполнены правильно, пользователь будет создан, а вы увидите соответствующее сообщение:



Bookmarks Log-in

Welcome Paloma!

Your account has been successfully created. Now you can [log in](#).

Рис. 4.14 ❖ Сообщение об успешном создании аккаунта

Кликните на ссылку **log in** и введите логин и пароль, чтобы убедиться, что вы имеете доступ к своему аккаунту.

Теперь мы можем добавить ссылку на страницу регистрации в шаблон логина. Отредактируйте файл `registration/login.html`; найдите строку:

```
<p>Please, use the following form to log-in:</p>
```

Замените ее на:

```
<p>Please, use the following form to log-in. If you don't have an account <a href="{% url 'register' %}">register here</a></p>
```

Только что мы сделали регистрацию доступной со страницы логина.

Расширение модели пользователя

В модели пользователя Django описан минимальный набор полей для хранения сведений о пользователях. В большинстве случаев при работе с аккаунтами этих полей достаточно. Но в некоторых проектах могут понадобиться дополнительные данные пользователей. Чтобы сохранять эти сведения, нужно создать свою модель профиля, которая будет содержать все дополнительные поля и ссылку «один к одному» на модель Django.

Давайте создадим собственную модель для профиля. Отредактируйте `models.py` приложения `account` и добавьте следующий код:

```
from django.db import models
from django.conf import settings

class Profile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
                               on_delete=models.CASCADE)
    date_of_birth = models.DateField(blank=True, null=True)
    photo = models.ImageField(upload_to='users/%Y/%m/%d/', blank=True)

    def __str__(self):
        return 'Profile for user {}'.format(self.user.username)
```

i Чтобы наш код не зависел от конкретной модели пользователя, мы используем функцию `get_user_model()`. Она возвращает модель, указанную в настройке `AUTH_USER_MODEL`, и мы можем легко заменять класс пользователя, т. к. не обращались в коде напрямую к конкретной модели.

Поле «один к одному» `user` позволит нам связать дополнительные данные с конкретным пользователем. Мы передаем `CASCADE` в качестве параметра `on_delete`, поэтому связанные с пользователем данные будут удалены при удалении основного объекта `User`. Поле `photo` имеет тип `ImageField`. Для работы с ним нам необходимо установить библиотеку изображений `Pillow` с помощью команды:

```
pip install Pillow==5.1.0
```

Для того чтобы Django знал, где хранить медиафайлы, загруженные пользователями, добавьте следующие строки в `settings.py`:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

MEDIA_URL – это базовый URL, от которого будут формироваться адреса файлов. MEDIA_ROOT – путь в файловой системе, где эти файлы будут храниться. Мы не задаем этот путь явно, а используем BASE_DIR, чтобы наш код был универсальным.

Откройте файл `urls.py` проекта `bookmark` и отредактируйте код таким образом:

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

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

```
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Теперь сервер разработки Django сможет возвращать медиафайлы при обращении к ним по URL'у.

i Функция `static()` подходит только для локальной разработки, но не для применения на боевых серверах. Никогда не используйте Django в качестве поставщика статических и медиафайлов.

Откройте терминал и выполните команду для создания миграций:

```
python manage.py makemigrations
```

Вы увидите такой вывод:

```
Migrations for 'account':
  account/migrations/0001_initial.py
  - CreateModelProfile
```

Теперь нужно синхронизировать изменения в моделях с базой данных:

```
python manage.py migrate
```

Вы увидите вывод, содержащий такую строку:

```
Applying account.0001_initial... OK
```

Отредактируйте файл `admin.py` приложения `account` и зарегистрируйте модель `Profile` на сайте администрирования:

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

@admin.register(Profile)
class ProfileAdmin(admin.ModelAdmin):
    list_display = ['user', 'date_of_birth', 'photo']
```

Запустите сервер разработки командой `python manage.py runserver` и откройте в браузере `http://127.0.0.1:8000/admin/`. Вы должны увидеть раздел Profiles:

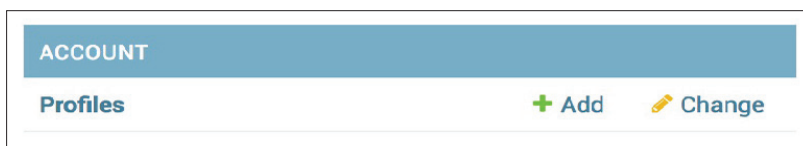


Рис. 4.15 ❖ Раздел профилей пользователей на сайте администрирования

Теперь добавим возможность пользователя редактировать профили через сайт. Импортируйте новую модель в файл `forms.py` приложения `account` и добавьте формы:

```
from .models import Profile

class UserEditForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ('first_name', 'last_name', 'email')

class ProfileEditForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ('date_of_birth', 'photo')
```

Эти формы предназначены для:

- `UserEditForm` – позволит пользователям менять имя, фамилию, e-mail (поля встроенной в Django модели);
- `ProfileEditForm` – позволит модифицировать дополнительные сведения, которые мы сохраняем в модели `Profile` (дату рождения и аватар).

Импортируйте в файле `views.py` приложения `account` модель `Profile`:

```
from .models import Profile
```

Затем добавьте эти строки в обработчик `register` перед строкой `new_user.save()`:

```
# Создание профиля пользователя.
Profile.objects.create(user=new_user)
```

Когда пользователь регистрируется на сайте, мы создаем пустой профиль, ассоциированный с ним. Для тех пользователей, которые были созданы ранее, необходимо вручную добавить объекты `Profile` через сайт администрирования.

Теперь давайте создадим обработчик для сохранения изменений в профиле. Добавьте следующий фрагмент кода в тот же файл:

```
from .forms import LoginForm, UserRegistrationForm, UserEditForm, ProfileEditForm

@login_required
def edit(request):
```



```

if request.method == 'POST':
    user_form = UserEditForm(instance=request.user,data=request.POST)
    profile_form = ProfileEditForm(instance=request.user.profile,
                                   data=request.POST,
                                   files=request.FILES)
    if user_form.is_valid() and profile_form.is_valid():
        user_form.save()
        profile_form.save()
else:
    user_form = UserEditForm(instance=request.user)
    profile_form = ProfileEditForm(instance=request.user.profile)
return render(request,'account/edit.html',
              {'user_form': user_form,'profile_form': profile_form})

```

Мы обернули функцию в декоратор `login_required`, потому что для изменения профиля пользователь должен быть авторизован. В этом обработчике мы используем две формы: `UserEditForm` (для базовых сведений о пользователе) и `ProfileEditForm` (для дополнительной, расширенной информации). Для валидации данных вызываем метод `is_valid()` каждой из форм. Если обе формы заполнены корректно, сохраняем их с помощью метода `save()`.

Добавьте эту строку в `urls.py` приложения `account`:

```
path('edit/', views.edit, name='edit'),
```

Наконец, создадим для этого обработчика шаблон `edit.html` в `templates/account/`. Добавьте в него код формы редактирования профиля:

```

{% extends "base.html" %}

{% block title %}Edit your account{% endblock %}

{% block content %}
<h1>Edit your account</h1>
<p>You can edit your account using the following form:</p>
<form action="." method="post" enctype="multipart/form-data">
  {{ user_form.as_p }}
  {{ profile_form.as_p }}
  {% csrf_token %}
  <p><input type="submit" value="Save changes"></p>
</form>
{% endblock %}

```

Мы добавили в форму `enctype="multipart/form-data"`, чтобы с ее помощью можно было загружать файлы. Мы используем одну HTML-форму для отправки обеих Django-форм: и `user_form`, и `profile_form`.

Зарегистрируйтесь под новым пользователем и откройте страницу `http://127.0.0.1:8000/account/edit/`. Вы увидите такую форму:

The screenshot shows a web interface with a green header bar containing 'Bookmarks', 'My dashboard', 'Images', 'People', and 'Hello Paloma, Logout'. Below the header, the main content area is titled 'Edit your account'. Underneath the title, there is a message: 'You can edit your account using the following form:'. The form consists of several input fields: 'First name:' with 'Paloma', 'Last name:' with 'Melé', 'Email address:' with 'paloma@zenxit.com', 'Date of birth:' with '1981-04-14', and 'Photo:' with a 'Choose File' button and the text 'no file selected'. At the bottom of the form is a green button labeled 'SAVE CHANGES'.

Рис. 4.16 ❖ Форма редактирования профиля данных пользователя

Теперь мы можем добавить ссылки на страницы редактирования профиля и смены пароля на рабочий стол. Откройте шаблон `account/dashboard.html` и замените эту строку:

```
<p>Welcome to your dashboard.</p>
```

на новую со ссылками на страницы:

```
<p>Welcome to your dashboard.  
  You can <a href="{% url "edit" %}">edit your profile</a> or  
  <a href="{% url "password_change" %}">change your password</a>.  
</p>
```

Теперь пользователям удобно переходить на страницу редактирования профиля прямо с рабочего стола. Откройте `http://127.0.0.1:8000/account/` и убедитесь, что ссылки доступны:

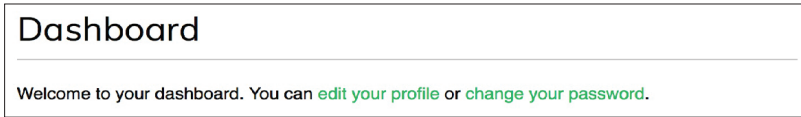


Рис. 4.17 ❖ Ссылки на страницы редактирования профиля на рабочем столе

Использование собственной модели пользователя

Кроме использования ссылки типа «один к одному», Django позволяет полностью заменить модель пользователя. Для этого наш класс должен быть наследником `AbstractUser`, который реализует базовые методы для пользователя. Более подробно о такой замене можно прочитать на странице <https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#substituting-a-custom-user-model>.

Использование собственной модели пользователя сделает наш код более гибким, но может стать причиной чуть более сложной интеграции со сторонними приложениями, которые взаимодействуют с моделью пользователя Django.

Подключение системы уведомлений

Теперь пользователь может активно взаимодействовать с нашей системой. При этом в некоторых случаях было бы неплохо информировать его о результатах действий. В Django встроена *система сообщений*, которая позволяет отображать одноразовые уведомления.

Она расположена в пакете `django.contrib.messages` и по умолчанию добавлена в список `INSTALLED_APPS` настроек, если проект был создан с помощью команды `python manage.py startproject`. Обратите внимание, что файл настроек содержит в настройке `MIDDLEWARE` промежуточный слой `django.contrib.messages.middleware.MessageMiddleware`.

Система сообщений предоставляет простой механизм отправлять уведомления пользователям. По умолчанию они хранятся в cookie и отображаются при последующем запросе пользователя. Чтобы использовать эту систему, достаточно импортировать модуль `messages` и добавить новое уведомление:

```
from django.contrib import messages
messages.error(request, 'Something went wrong')
```

Мы можем создать уведомление с помощью функции `add_message()` или любой из перечисленных ниже:

- `success()` – сообщение об успешном завершении действия;
- `info()` – информационное сообщение;
- `warning()` – что-то пошло не так, но пока ошибка не критична;
- `error()` – действие не завершилось или произошла ошибка;
- `debug()` – отладочное сообщение, которое не будет отображаться в боевом окружении.

Давайте добавим уведомления на наш сайт. Так как система сообщений настраивается для всего проекта, мы можем отображать уведомления в базовом

шаблоне. Откройте файл `base.html` приложения `account` и добавьте следующий фрагмент кода между `<div>`-элементом с ID `header` и `<div>`-элементом с ID `content`:

```
{% if messages %}
<ul class="messages">
  {% for message in messages %}
    <li class="{{ message.tags }}">
      {{ message|safe }}
      <a href="#" class="close">x</a>
    </li>
  {% endfor %}
</ul>
{% endif %}
```

Система сообщений автоматически подключает контекстный процессор `django.contrib.messages.context_processors.messages`, который добавляет переменную `message` в контекст. Вы можете найти список активированных контекстных процессоров в настройке `TEMPLATES`, в атрибуте `context_processors`. Переменная `message` может быть использована в шаблонах для отображения всех существующих уведомлений для конкретного пользователя.

Теперь давайте добавим в наш обработчик уведомления. Откройте файл `views.py` приложения `account`, импортируйте модуль `messages` и измените обработчик `edit` таким образом:

```
from django.contrib import messages

@login_required
def edit(request):
    if request.method == 'POST':
        # ...
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
            messages.success(request, 'Profile updated successfully')
        else:
            messages.error(request, 'Error updating your profile')
    else:
        user_form = UserEditForm(instance=request.user)
        # ...
```

Мы добавили уведомление об успешном изменении данных профиля. Если какая-то из форм содержит ошибки, отобразится уведомление с ошибкой.

Откройте `http://127.0.0.1:8000/account/edit/` и попробуйте отредактировать профиль. Если вы ввели корректные данные, то увидите такое уведомление:

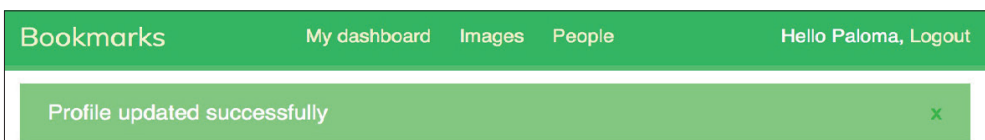


Рис. 4.18 ❖ Уведомление об успешном изменении профиля

Попробуйте ввести невалидные данные (например, задайте дату рождения для поля **Date of birth** в неправильном формате), и вы увидите такое уведомление:

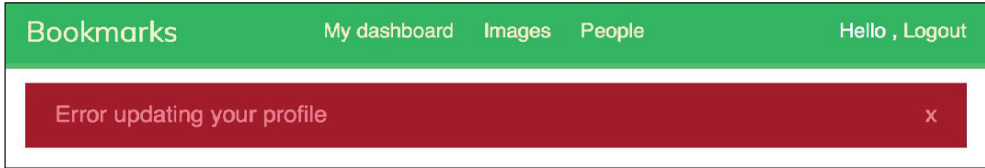


Рис. 4.19 ❖ Уведомление о том, что изменить профиль не удалось

Более подробную информацию о системе сообщений вы можете найти на странице <https://docs.djangoproject.com/en/2.0/ref/contrib/messages/>.

РЕАЛИЗАЦИЯ БЭКЭНДА АУТЕНТИФИКАЦИИ

Django позволяет разработчику аутентифицировать пользователей, применяя различные ресурсы. Настройка `AUTHENTICATION_BACKENDS` содержит список бэкэндов аутентификации вашего проекта. По умолчанию он выглядит так:

```
['django.contrib.auth.backends.ModelBackend']
```

Бэкэнд по умолчанию `ModelBackend` аутентифицирует пользователей, используя модель из `django.contrib.auth`. Это подходит для большинства проектов. Но мы можем создать собственный бэкэнд, чтобы добавить новый способ аутентификации (например, через LDAP или любую другую систему). Более подробно про переопределение аутентификации можно прочитать на странице <https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#other-authentication-sources>.

Каждый раз, когда мы пытаемся аутентифицировать пользователя функцией `authenticate()`, Django пробует применить каждый из бэкэндов, указанных в `AUTHENTICATION_BACKENDS`, по очереди, пока не дойдет до того, который успешно аутентифицирует пользователя. Если ни один из бэкэндов не сможет этого сделать, пользователь не будет аутентифицирован в нашей системе.

Django предоставляет простой способ создания собственного *бэкэнда аутентификации*. Достаточно описать класс, в котором есть два метода:

- `authenticate()` – принимает в качестве параметров объект запроса `request` и идентификационные данные пользователя. Он должен возвращать объект пользователя, если данные корректны; в противном случае – `None`. Аргумент `request` имеет тип `HttpRequest`, но может быть и `None`;
- `get_user()` – принимает ID и должен вернуть соответствующий объект пользователя.

Создать свой бэкэнд – это значит создать Python-класс, который реализует эти два метода. Мы добавим бэкэнд, который даст возможность пользователям использовать e-mail вместо логина для входа на сайт.

Создайте новый файл `authentication.py` в приложении `account` и добавьте:

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

class EmailAuthBackend(object):
    """Выполняет аутентификацию пользователя по e-mail."""
    def authenticate(self, request, username=None, password=None):
        try:
            user = User.objects.get(email=username)
            if user.check_password(password):
                return user
            return None
        except User.DoesNotExist:
            return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

Этот код представляет простейший бэкэнд аутентификации. Метод `authenticate()` получает объект `request`, `username` и `password`. Мы можем использовать другие названия для аргументов, но выбираем именно `username` и `password`, чтобы без проблем работать с системой аутентификации Django, где могут быть использованы именованные аргументы. Этот класс выполняет следующие действия:

- `authenticate()` – пытается получить пользователя, соответствующего указанному электронной почте и паролю, с помощью метода `check_password()` модели пользователя. Этот метод выполняет шифрование пароля и сравнивает результат с тем, который хранится в базе данных;
- `get_user()` – получает пользователя по ID, который задается как аргумент `user_id`. Django использует бэкэнд, который аутентифицировал пользователя, чтобы получать объект `User` на протяжении всей сессии.

Отредактируйте файл `settings.py` проекта и добавьте следующую настройку:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
]
```

В предыдущем фрагменте кода к основному бэкэнду `ModelBackend`, который использует логин и пароль пользователя в качестве идентификационных данных, мы добавили собственный бэкэнд, который проверяет e-mail вместо логина. Помните, что Django будет использовать бэкэнды по порядку, поэтому теперь пользователь сможет аутентифицироваться и с помощью электронной почты. Идентификационные данные сначала будут проверены `ModelBackend`. Если этот бэкэнд не вернет объект пользователя, Django попытается аутентифицировать его с помощью нашего класса, `EmailAuthBackend`.

i Порядок, в котором бэкэнды указаны в настройке `AUTHENTICATION_BACKENDS`, имеет значение. Если одни и те же идентификационные данные окажутся корректными для нескольких бэкэндов, Django остановит проверку, как только первый из них вернет объект пользователя.

ПОДКЛЮЧЕНИЕ АУТЕНТИФИКАЦИИ ЧЕРЕЗ СОЦСЕТИ

Мы можем добавить авторизацию с помощью распространенных соцсетей, таких как Facebook, Twitter, Google. Python Social Auth – это Python-приложение, которое дает возможность пользователям использовать аккаунты сторонних соцсетей для входа на наш сайт. Исходный код и документация Python Social Auth доступны по ссылке <https://github.com/python-social-auth>.

В этом приложении реализованы бэкэнды для некоторых Python-фреймворков, в том числе и для Django. Для установки с помощью `pip` откройте консоль и выполните команду:

```
pip install social-auth-app-django==2.1.0
```

Добавьте приложение `social_django`, `python-social-auth` для Django-проектов, в список `INSTALLED_APPS` файла `settings.py`:

```
INSTALLED_APPS = [
    #...
    'social_django',
]
```

Теперь выполните синхронизацию моделей с базой данных:

```
python manage.py migrate
```

Вы должны увидеть миграции добавленного нами приложения:

```
Applying social_django.0001_initial... OK
Applying social_django.0002_add_related_name... OK
...
Applying social_django.0008_partial_timestamp... OK
```

Python-social-auth содержит бэкэнды для множества сервисов. Полный список поддерживаемых социальных сетей можно найти на странице <https://python-social-auth.readthedocs.io/en/latest/backends/index.html#supported-backends>.

Мы подключим бэкэнды для Facebook, Twitter и Google.

Для этого необходимо добавить шаблон URL'a авторизации в проект. Откройте файл `urls.py` проекта `bookmark` и подключите шаблоны `social_django`:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/', include('social_django.urls', namespace='social')),
]
```

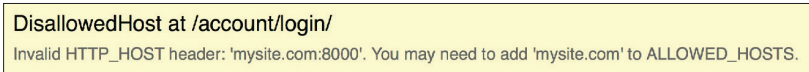
Некоторые социальные сервисы не позволяют выполнять перенаправление на `127.0.0.1` или `localhost` после успешной аутентификации пользователя. Для того чтобы работать с ними локально, нам нужен домен. Для его созда-

ния в том случае, если вы пользуетесь Linux или macOSX, отредактируйте файл `/etc/hosts` и добавьте следующую строку:

```
127.0.0.1 mysite.com
```

Это укажет вашему компьютеру, что при обращении к `mysite.com` необходимо использовать адрес `127.0.0.1`. Если вы пользуетесь Windows, сделайте то же самое. Файл находится в `C:\Windows\System32\Drivers\etc\hosts`.

Для того чтобы проверить, что домен настроен на нужный адрес, запустите сервер командой `pythonmanage.pyrunserver` и откройте страницу `http://mysite.com:8000/account/login/`. Вы должны будете увидеть ошибку:



```
DisallowedHost at /account/login/
Invalid HTTP_HOST header: 'mysite.com:8000'. You may need to add 'mysite.com' to ALLOWED_HOSTS.
```

Рис. 4.20 ❖ Ошибка `DisallowedHost` при неправильно настроенном хосте

Django проверяет домены, перечисленные в настройке `ALLOWED_HOSTS`. Это мера предосторожности для предотвращения атак подмены HTTP-заголовков. Django разрешает использовать только те домены, которые указаны в `ALLOWED_HOSTS`. Более подробно об этой настройке можно прочитать на <https://docs.djangoproject.com/en/2.0/ref/settings/#allowed-hosts>.

Отредактируйте файл `settings.py` проекта таким образом:

```
ALLOWED_HOSTS = ['mysite.com', 'localhost', '127.0.0.1']
```

Кроме `mysite.com`, мы явно задали `localhost` и `127.0.0.1`. Это сделано, чтобы иметь возможность обращаться к сайту через `localhost`, который используется Django по умолчанию при настройке `DEBUG`, равной `True`, и пустом списке `ALLOWED_HOSTS`. Теперь вы сможете открыть страницу авторизации `http://mysite.com:8000/account/login/`.

Аутентификация Facebook

Чтобы пользователь мог использовать аккаунт Facebook для входа на сайт, добавьте строку в настройку `AUTHENTICATION_BACKENDS` файла `settings.py`:

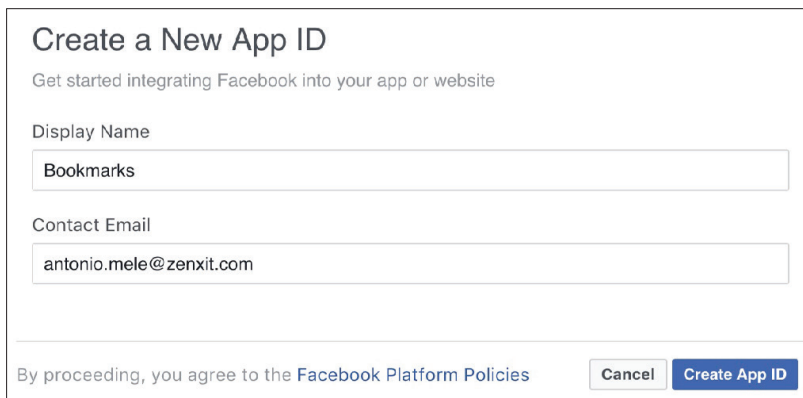
```
'social_core.backends.facebook.FacebookOAuth2',
```

Кроме этого, нам необходимо создать аккаунт разработчика в Facebook и новое Facebook-приложение. Откройте <https://developers.facebook.com/apps/>, и вы увидите личный кабинет разработчика:



Рис. 4.21 ❖ Личный кабинет разработчика Facebook-приложений

Кликните на кнопку **Add a New App**. Вам отобразится форма для создания ID нового приложения:



Create a New App ID

Get started integrating Facebook into your app or website

Display Name

Bookmarks

Contact Email

antonio.mele@zenxit.com

By proceeding, you agree to the Facebook Platform Policies

Cancel Create App ID

Рис. 4.22 ❖ Форма создания нового Facebook-приложения

Введите Bookmarks в поле **Display Name**, вашу электронную почту и кликните **Create App ID**. Вы увидите доску нашего нового приложения и различные функции, которые можно добавить к нему. Обратите внимание на следующий блок **Facebook Login** и кликните **Set Up**:

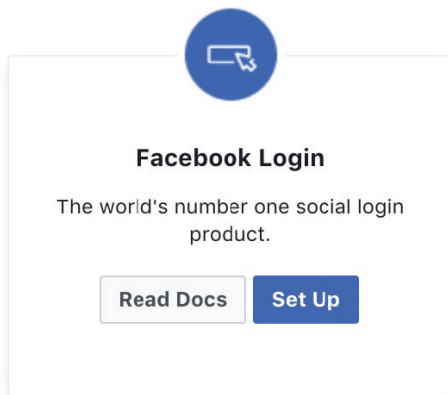


Рис. 4.23 ❖ Начало создания Facebook-приложения

Вам будет предложено выбрать платформу, с которой будет работать приложение:

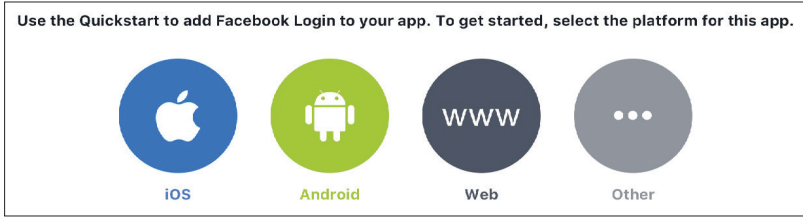


Рис. 4.24 ❖ Выбор платформы для приложения

Выберите **Web**, и вам откроется форма с полем для указания адреса сайта:

Рис. 4.25 ❖ Форма ввода адреса сайта

Введите в поле **Site URL** значение `http://mysite.com:8000/` и нажмите кнопку **Save**. Остальные шаги из вводной инструкции можно пропустить. Кликните на **Dashboard** в левом меню. Вы увидите что-то похожее на этот скриншот:

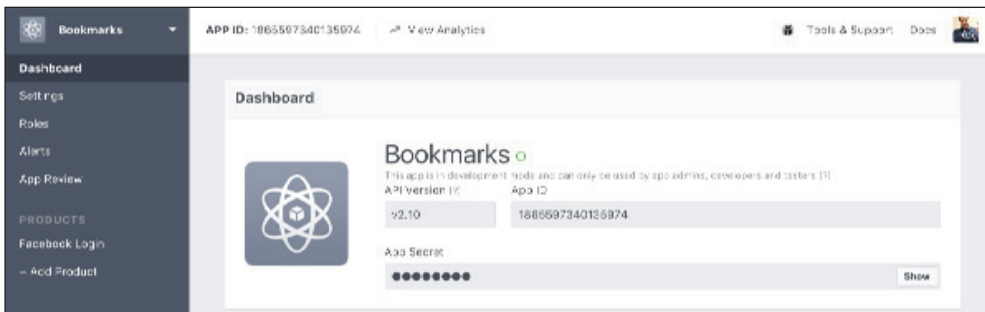


Рис. 4.26 ❖ Данные Facebook-приложения

Скопируйте ключи **App ID** и **App Secret** и добавьте их в настройки settings.py проекта:

```
SOCIAL_AUTH_FACEBOOK_KEY = 'XXX' # Facebook App ID
SOCIAL_AUTH_FACEBOOK_SECRET = 'XXX' # Facebook App Secret
```

Мы можем указать, какие данные хотим запрашивать из Facebook-аккаунта. Для этого нужно задать настройку SOCIAL_AUTH_FACEBOOK_SCOPE с дополнительными правами, которые будут запрошены у пользователя при попытке авторизации:

```
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email']
```

Теперь вернитесь в Facebook и кликните **Settings**. Вы увидите форму со множеством настроек нашего приложения. Добавьте **mysite.com** в поле **App Domains**:



Рис. 4.27 ❖ Добавление домена нашего приложения в список разрешений для Facebook-приложения

Кликните на кнопку **Save Changes**. Затем в левом меню кликните на **Facebook Login**. Убедитесь, что следующие настройки активны:

- Client OAuth Login;
- Web OAuth Login;
- Embedded Browser OAuth Login.

Добавьте `http://mysite.com:8000/social-auth/complete/facebook/` в поле **Valid OAuth redirect URLs**. Теперь это поле должно выглядеть так, как показано на рис. 4.28.

Откройте шаблон `registration/login.html` приложения `account` и добавьте ссылку на авторизацию через Facebook в конец блока `content`:

```
<div class="social">
  <ul>
    <li class="facebook"><a href="{% url 'social:begin' 'facebook' %}">
      Signin with Facebook</a></li>
    </ul>
</div>
```

Откройте `http://mysite.com:8000/account/login/`. Теперь страница логина содержит ссылку на авторизацию через Facebook (рис. 4.29).

Client OAuth Settings

Client OAuth Login
Enables the standard OAuth client-to-client flow. Secure your application and prevent abuse by locking down which token redirect URIs are allowed with the options below. Disable globally if not used. [?]

Web OAuth Login
Enables web based OAuth client login for building custom login flows. [?]

Force Web OAuth Reauthentication
When on, prompts people to enter their Facebook password in order to log in on the web. [?]

Embedded Browser OAuth Login
Enables browser control redirect uri for OAuth client login. [?]

Use Strict Mode for Redirect URIs
Only allow redirects that use the Facebook SDK or that exactly match the Valid OAuth Redirect URIs. Strongly recommended. [?]

Valid OAuth redirect URIs

Login from Devices
Enables the OAuth client login flow for devices like a smart TV [?]

Рис. 4.28 ❖ Настройки OAuth для Facebook-приложения

Bookmarks
Log-in

Log-in

Please, use the following form to log-in. If you don't have an account [register here](#)

Username:

Password:

[Sign in with Facebook](#)

[LOG-IN](#)

[Forgotten your password?](#)

Рис. 4.29 ❖ Страницы входа на сайт с ссылкой авторизации через Facebook

Кликните кнопку **Signin with Facebook**. Вы будете перенаправлены в Facebook и увидите окно, запрашивающее права на доступ к публичным данным вашего аккаунта приложением Bookmarks:

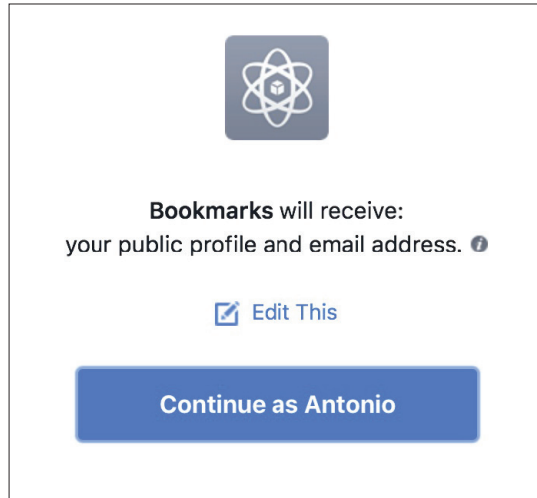


Рис. 4.30 ❖ Приложение Bookmarks запрашивает доступ к Facebook-аккаунту пользователя

Кликните кнопку **Continue as**. Вы будете авторизованы и перенаправлены на рабочий стол. Обратите внимание, что нужно указать URL перенаправления в настройке `LOGIN_REDIRECT_URL`. Мы только что убедились, что добавить аутентификацию с помощью другой соцсети предельно просто.

Аутентификация Twitter

Для использования Twitter отредактируйте файл `settings.py`, добавив следующую строку в настройку `AUTHENTICATION_BACKENDS`:

```
'social_core.backends.twitter.TwitterOAuth',
```

Нам необходимо создать новое приложение в вашем Twitter-аккаунте. Откройте <https://apps.twitter.com/app/new>. Вы увидите страницу создания нового Twitter-приложения (рис. 4.31).

Заполните данные о приложении, включая настройки:

- Website: `http://mysite.com:8000/`;
- Callback URL: `http://mysite.com:8000/social-auth/complete/twitter/`.

Затем откройте настройки приложения, кликнув на **Create your Twitter application**. Перейдите на вкладку **Keys and Access Tokens**. Вы увидите информацию, похожую на ту, что представлена на рис. 4.32:

Скопируйте **Consumer Key** и **Consumer Secret** в настройки `settings.py` проекта:

```
SOCIAL_AUTH_TWITTER_KEY = 'XXX' # Twitter Consumer Key
SOCIAL_AUTH_TWITTER_SECRET = 'XXX' # Twitter Consumer Secret
```

Application Details

Name *

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? [OAuth 1.0a](#) applications should explicitly specify their `oauth_callback` URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Рис. 4.31 ❖ Регистрация нового Twitter-приложения

Bookmarks

Details
Settings
Keys and Access Tokens
Permissions

Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

| | |
|------------------------------|---|
| Consumer Key (API Key) | eJJU1AzzEQFJ6PAgqLjc18TH1 |
| Consumer Secret (API Secret) | ***** |
| Access Level | Read and write (modify app permissions) |

Рис. 4.32 ❖ Данные созданного Twitter-приложения

Теперь отредактируйте шаблон `registration/login.html`, добавив ссылку на кнопку аутентификации в Twitter в ``-элемент:

```
<li class="twitter">
  <a href="{% url "social:begin" "twitter" %}">Login with Twitter</a>
</li>
```

Откройте страницу `http://mysite.com:8000/account/login/` и кликните на ссылку **Login with Twitter**. Вы будете перенаправлены в Twitter. Приложение запросит доступ к публичным данным (рис. 4.33).

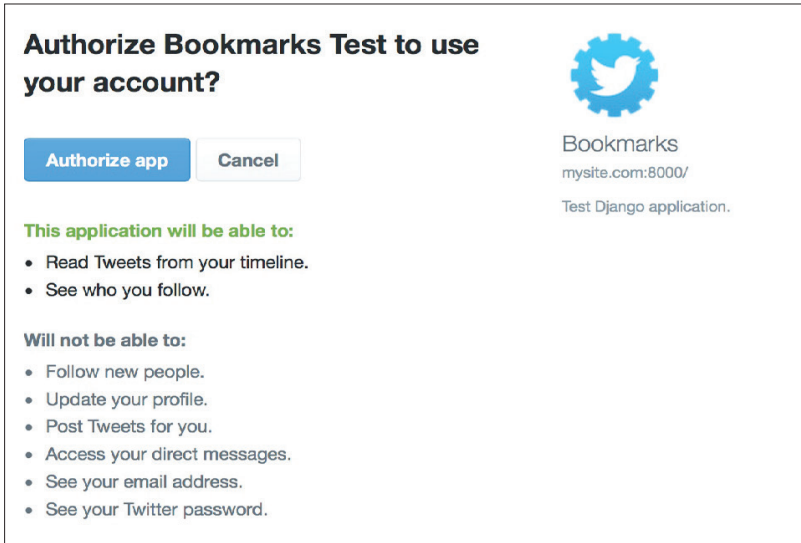


Рис. 4.33 ❖ Приложение Bookmarks запрашивает доступ к Twitter-аккаунту пользователя

Кликните **Authorize app**. Пользователь будет авторизован и перенаправлен на рабочий стол нашего сайта.

Аутентификация Google

Google использует аутентификацию с помощью OAuth2. Подробности об OAuth2 приведены на странице <https://developers.google.com/identity/protocols/OAuth2>.

Для подключения аутентификации Google добавьте соответствующий бэк-энд в настройку AUTHENTICATION_BACKENDS файла settings.py:

```
'social_core.backends.google.GoogleOAuth2'
```

Сначала необходимо создать ключ для доступа к API в консоли разработчика Google. Откройте <https://console.developers.google.com/apis/credentials>, кликните на ссылку **Select a project** и создайте новый (рис. 4.34).

После создания проекта на странице **Credentials** в выпадающем списке **Create credentials** выберите **OAuth client ID** (рис. 4.35).

Google попросит вас сконфигурировать страницу согласия (рис. 4.36).

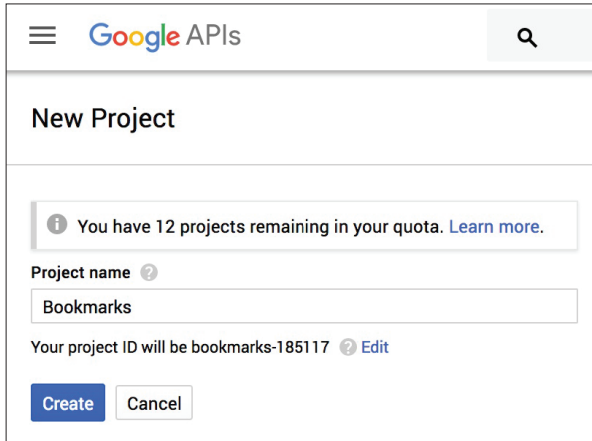


Рис. 4.34 ❖ Создание нового Google-проекта

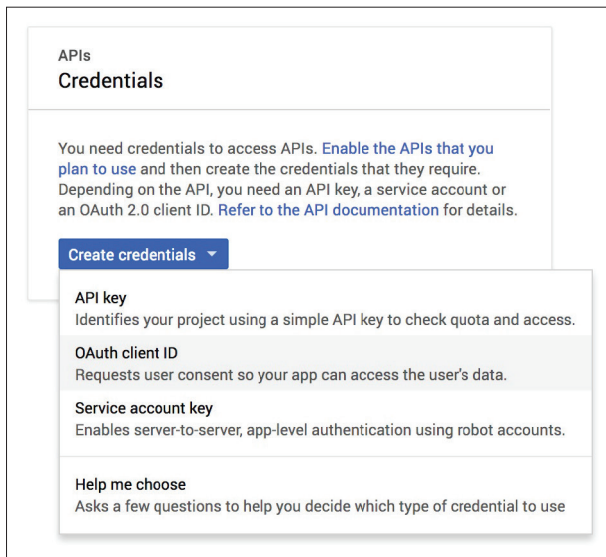


Рис. 4.35 ❖ Меню настроек Google-проекта

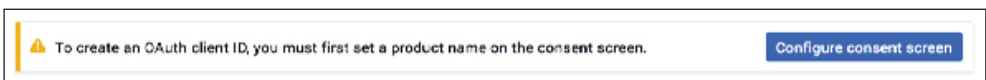


Рис. 4.36 ❖ Запрос на настройку страницы согласия пользователя

Это та страница, которую пользователь увидит, когда попытается авторизоваться через Google. Кликните на кнопку **Configure consent screen**. Заполните ваш электронный адрес, **Product name** – Bookmarks, и сохраните, нажав на кнопку **Save**. Страница согласия будет создана, а вы перенаправлены на последний шаг создания ID-приложения.

Заполните в форме настроек приложения следующие сведения:

- Application type – выберите веб-приложение;
- Name – введите Bookmarks;
- Authorized redirect URIs – добавьте ссылку `http://mysite.com:8000/social-auth/complete/google-oauth2/`.

Эта форма будет выглядеть следующим образом:

Application type

- Web application
- Android [Learn more](#)
- Chrome App [Learn more](#)
- iOS [Learn more](#)
- PlayStation 4
- Other

Name

Restrictions
Enter JavaScript origins, redirect URIs or both

Authorised JavaScript origins
For use with requests from a browser. This is the origin URI of the client application. It cannot contain a wildcard (`https://*.example.com`) or a path (`https://example.com/subdir`). If you're using a non-standard port, you must include it in the origin URI.

Authorised redirect URIs
For use with requests from a web server. This is the path in your application that users are redirected to after they have authenticated with Google. The path will be appended with the authorisation code for access. Must have a protocol. Cannot contain URL fragments or relative paths. Cannot be a public IP address.

`http://mysite.com:8000/social-auth/complete/google-oauth2/` ×

Рис. 4.37 ❖ Форма настроек приложения

Кликните на кнопку **Create**, и вы получите ключи **Client ID** и **Client Secret**. Добавьте их в настройки проекта `settings.py`:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = 'XXX' # Google Consumer Key
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = 'XXX' # Google Consumer Secret
```

В левом меню консоли разработчика Google в разделе **APIs & Services** нажмите на ссылку **Library**. Вы увидите список всех интерфейсов GoogleAPI. Выберите **Google+ API** и активируйте его, нажав на кнопку **ENABLE**:

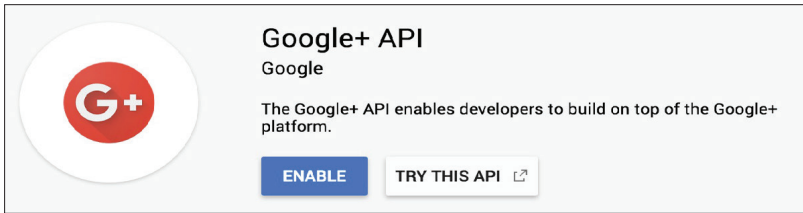


Рис. 4.38 ❖ Активация Google+ API

Отредактируйте шаблон `login.html`, добавив еще один элемент в ``-список – ссылку на авторизацию через Google:

```
<li class="google">
  <a href="{% url "social:begin" "google-oauth2" %}">
    Login with Google
  </a>
</li>
```

Откройте страницу авторизации `http://mysite.com:8000/account/login/`. Теперь на ней должны быть три ссылки для авторизации через социальные сети, которые мы подключили:

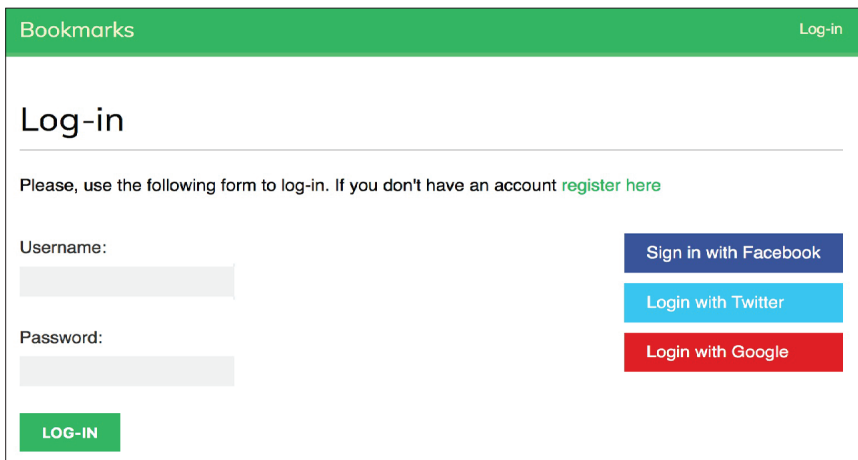


Рис. 4.39 ❖ Страница входа на сайт со ссылками на авторизацию Facebook, Twitter и Google

Нажмите на кнопку **Login with Google**. Вы будете авторизованы с использованием Google-аккаунта и перенаправлены на рабочий стол нашего сайта.

Так мы добавили аутентификацию через сторонние соцсети в проект. Мы легко можем подключить и другие сервисы, используя Python Social Auth.

РЕЗЮМЕ

В этой главе мы узнали, как реализовать систему аутентификации, и создали собственную модель пользователя. Также мы добавили возможность использовать аккаунты других соцсетей для входа на наш сайт.

В следующей главе мы узнаем, как создать букмарклет – систему сохранения изображений в виде закладок в браузере, генерировать превью картинок и выполнять AJAX-запросы.

Глава 5

Совместное использование содержимого сайта

В предыдущей главе мы добавили регистрацию и аутентификацию пользователей. Мы узнали, как сделать собственную модель для профилей пользователей, и добавили аутентификацию через основные социальные сети.

В этой главе мы создадим JavaScript-букмарклет для доступа через наш сайт к содержимому других сайтов, добавим AJAX-запросы с помощью jQuery и Django и рассмотрим следующие темы:

- создание отношений «многие ко многим»;
- переопределение поведения форм;
- использование jQuery с Django;
- реализация букмарклета на jQuery;
- создание превью изображений с помощью `src-thumbnaill`;
- реализация обработчиков для AJAX-запросов;
- добавление собственных декораторов для обработчиков;
- страничный вывод с помощью AJAX.

СОХРАНЕНИЕ ИЗОБРАЖЕНИЙ В ЗАКЛАДКИ НА САЙТЕ

Мы добавим пользователям возможность сохранять в закладки картинки, найденные на других сайтах, и делиться ими на нашем сайте. Для этого необходимо выполнить следующие шаги:

- 1) определить модель для сохранения картинок и связанной с ними информации;
- 2) создать форму и обработчик для загрузки картинок;
- 3) реализовать систему, которая позволит публиковать на нашем сайте изображения, найденные во внешних источниках.

Для начала создайте новое приложение в проекте `bookmarks` командой:

```
django-admin startapp images
```

Добавьте его в список `INSTALLED_APPS` файла настроек `settings.py`:

```
INSTALLED_APPS = [
    # ...
    'images.apps.ImagesConfig',
]
```

Теперь приложение `images` активировано, давайте добавим модель для сохранения сведений о картинках.

Создание модели изображения

Отредактируйте файл `models.py` приложения `images` и добавьте следующий код:

```
from django.db import models
from django.conf import settings

class Image(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL,
                             related_name='images_created', on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, blank=True)
    url = models.URLField()
    image = models.ImageField(upload_to='images/%Y/%m/%d/')
    description = models.TextField(blank=True)
    created = models.DateField(auto_now_add=True, db_index=True)

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

Эта модель будет использована для сохранения изображений, добавленных в закладки. Давайте разберем ее атрибуты подробнее:

- `user` – указывает пользователя, который добавляет изображение в закладки. Это поле является внешним ключом и использует связь «один ко многим». Пользователь может сохранять много изображений, но каждая картинка может быть сохранена только одним пользователем. Также мы добавили аргумент `on_delete`, равный `CASCADE`, поэтому объекты картинок будут удаляться при удалении их владельца;
- `title` – заголовок картинки;
- `slug` – краткое наименование картинки, слаг. Может содержать только буквы, цифры, нижние подчеркивания, дефисы, используется для создания семантических URL'ов;
- `url` – ссылка на оригинальную картинку;
- `image` – файл изображения;
- `description` – необязательное поле описания;
- `created` – дата и время создания объекта в базе данных. Так как мы указали `auto_now_add`, текущие время и дата будут подставлены автоматически. Аргумент `db_index=True` говорит Django о необходимости создать индекс по этому полю.



Индексы баз данных улучшают производительность. Рассмотрите возможность добавления `db_index=True` для полей, которые часто используются в `filter()`, `exclude()`, `or-`

`der_by()`. Для полей с `unique=True` и `ForeignKey` индексы создаются автоматически. Для определения составного индекса можно использовать `Meta.index_together`.

Мы переопределим метод `save()` модели `Image`, для того чтобы автоматически формировать слаг, основываясь на указанном заголовке `title`. Импортируйте функцию `slugify()` и добавьте в модель метод `save()`:

```
from django.utils.text import slugify

class Image(models.Model):
    # ...
    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.title)
        super(Image, self).save(*args, **kwargs)
```

Если в этом фрагменте кода у изображения нет слага, функция `slugify()` автоматически формирует его из переданного заголовка, после чего мы сохраняем объект картинку. Таким образом, слаг будет генерироваться автоматически, пользователю не нужно будет вводить его вручную.

Добавление отношения «многие ко многим»

Мы добавим еще одно поле в модель `Image` для запоминания пользователей, которым понравилось изображение. В этом случае мы будем использовать отношение «многие ко многим», т. к. одному пользователю может понравиться множество картинок, а одна картинка может понравиться многим пользователям.

Добавьте поле отношения «многие ко многим» в модель `Image`:

```
users_like = models.ManyToManyField(settings.AUTH_USER_MODEL,
                                   related_name='images_liked',
                                   blank=True)
```

Когда мы определяем поле `ManyToManyField`, Django создает промежуточную таблицу, содержащую первичные ключи объектов связанных моделей. Поле `ManyToManyField` может быть определено в любой из них.

Так же как и `ForeignKey`, `ManyToManyField` позволяет указать название атрибута, по которому будут доступны связанные объекты. Этот тип поля предоставляет менеджер отношения «многие ко многим», с помощью которого можно обращаться к связанным объектам в виде `image.users_like.all()` или из объекта пользователя `user` как `user.images_likes.all()`.

Откройте терминал и создайте начальную миграцию приложения:

```
python manage.py makemigrations images
```

Вы увидите такой вывод:

```
Migrations for 'images':
  images/migrations/0001_initial.py
  - Create model Image
```

Теперь выполните команду синхронизации моделей с базой данных:

```
python manage.py migrate images
```

Вы увидите сообщение об успешном применении миграции:

```
Applying images.0001_initial... OK
```

Наша модель `Image` теперь синхронизирована с базой данных.

Регистрация модели изображения на сайте администрирования

Чтобы зарегистрировать новую модель, отредактируйте файл `admin.py` приложения `images`:

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

@admin.register(Image)
class ImageAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'image', 'created']
    list_filter = ['created']
```

Запустите сервер для разработки командой `python manage.py runserver`. Откройте терминал на `http://127.0.0.1:8000/admin/`, и вы увидите раздел для модели `Image`:



Рис. 5.1 ❖ Раздел **Images** для изображений на сайте администрирования

ИСПОЛЬЗОВАНИЕ ИЗОБРАЖЕНИЙ С ДРУГИХ САЙТОВ

Мы добавим возможность сохранять изображения с других сайтов в закладки. Пользователю нужно будет указать URL картинки, ее заголовок и необязательное описание. Наше приложение скачает изображение и создаст объект `Image` в базе данных.

Давайте начнем с реализации формы для сохранения объекта картинки. Создайте файл `forms.py` в каталоге приложения `images` и добавьте в него следующий код:

```
from django import forms
from .models import Image

class ImageCreateForm(forms.ModelForm):
    class Meta:
```

```

model = Image
fields = ('title', 'url', 'description')
widgets = {'url': forms.HiddenInput,}

```

Как вы можете заметить, мы используем модельную форму `ModelForm` для модели `Image`, которая включает поля `title`, `url`, `description`. Пользователи не будут вручную заполнять адрес. Вместо этого мы добавим JavaScript-инструмент для выбора картинки на любом постороннем сайте, а наша форма будет получать URL изображения в качестве параметра. Мы заменили виджет по умолчанию для поля `url` и используем `HiddenInput`. Этот виджет формируется как `input`-элемент с атрибутом `type="hidden"`. Мы сделали это для того, чтобы пользователи не видели поле `url`.

Валидация полей формы

Для проверки того, что введенный URL является корректным, убедимся, что имя файла заканчивается на `.jpg` или `.jpeg`, и разрешим использовать только JPEG картинки. Как мы уже узнали благодаря предыдущей главе, Django дает возможность проверять каждое поле формы по отдельности с помощью методов вида `clean_<fieldname>()`. Эти методы вызываются, когда мы обращаемся к методу `is_valid()` формы. Внутри функции валидации мы можем подменять значение или генерировать ошибки для конкретного поля. Добавьте следующие строки в форму `ImageCreateForm`:

```

def clean_url(self):
    url = self.cleaned_data['url']
    valid_extensions = ['.jpg', '.jpeg']
    extension = url.rsplit('.', 1)[1].lower()
    if extension not in valid_extensions:
        raise forms.ValidationError('The given URL does not ' \
                                   'match valid image extensions.')
    return url

```

В этом фрагменте мы определили метод `clean_url()` для валидации поля `url`. Он работает таким образом:

- 1) получает значение поля `url`, обращаясь к атрибуту формы `cleaned_data`;
- 2) разделяет URL, чтобы получить расширение файла и проверить, является ли оно корректным. Если это не так, форма генерирует исключение `ValidationError`. В этом методе мы выполняем очень простую проверку URL, но можно добавить более продвинутые механизмы валидации ссылки на изображение.

В дополнение к валидации URL необходимо загрузить файл и сохранить его. Для этого, например, мы могли бы реализовать обработчик, который будет заполнять форму и скачивать изображение. Но мы пойдем по более известному пути и переопределим метод `save()` модельной формы, чтобы выполнять это действие при любом сохранении формы, а не только в конкретном обработчике.

Переопределение метода `save()` модельной формы

Как мы знаем, класс `ModelForm` предоставляет метод `save()` для сохранения объекта модели в базу данных. Этот метод принимает необязательный аргумент `commit`, который позволяет настроить поведение (действительно ли нужно сохранять объект в базу данных). Если `commit` равен `False`, метод вернет объект модели, но не сохранит его. Мы переопределим `save()` формы, чтобы скачивать файл картинки и сохранять его.

Добавьте следующие строки в начало файла `forms.py`:

```
from urllib import request
from django.core.files.base import ContentFile
from django.utils.text import slugify
```

Затем определите метод `save()` в форме `ImageCreateForm`:

```
def save(self, force_insert=False, force_update=False, commit=True):
    image = super(ImageCreateForm, self).save(commit=False)
    image_url = self.cleaned_data['url']
    image_name = '{}.{}'.format(
        slugify(image.title),
        image_url.rsplit('.', 1)[1].lower())
    # Скачиваем изображение по указанному адресу.
    response = request.urlopen(image_url)
    image.image.save(image_name, ContentFile(response.read()), save=False)
    if commit:
        image.save()
    return image
```

Мы переопределили метод `save()`, оставив параметры оригинального метода класса `ModelForm`. Добавленная функция выполняет следующие шаги:

- 1) создает объект `image`, вызвав метод `save()` с аргументом `commit=False`;
- 2) получает URL из атрибута `cleaned_data` формы;
- 3) генерирует название изображения, совмещая слаг и расширение картинки;
- 4) использует Python-пакет `urllib`, чтобы скачать файл картинки, и вызывает метод `save()` поля изображения, передавая в него объект скачанного файла, `ContentFile`. Также используется аргумент `commit=False`, чтобы пока не сохранять объект в базу данных;
- 5) при переопределении метода важно оставить стандартное поведение, поэтому сохраняем объект изображения в базу данных только в том случае, если `commit` равен `True`.

Теперь нам нужен обработчик для взаимодействия с этой формой. Откройте файл `views.py` приложения `images` и добавьте следующий фрагмент:

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
from django.contrib import messages
from .forms import ImageCreateForm
```

```

@login_required
def image_create(request):
    if request.method == 'POST':
        # Форма отправлена.
        form = ImageCreateForm(data=request.POST)
        if form.is_valid():
            # Данные формы валидны.
            cd = form.cleaned_data

            new_item = form.save(commit=False)
            # Добавляем пользователя к созданному объекту.
            new_item.user = request.user
            new_item.save()
            messages.success(request, 'Image added successfully')
            # Перенаправляем пользователя на страницу сохраненного изображения.
            return redirect(new_item.get_absolute_url())
        else:
            # Заполняем форму данными из GET-запроса.
            form = ImageCreateForm(data=request.GET)
    return render(request,
                  'images/image/create.html',
                  {'section': 'images', 'form': form})

```

Мы добавили декоратор `login_required` к обработчику `image_create`, чтобы доступ к нему имели только авторизованные пользователи. Этот обработчик выполняет следующие действия:

- 1) получает начальные данные и создает объект формы. Эти данные содержат `url` и `title` картинки со стороннего сайта, они будут переданы в качестве аргументов GET-запроса JavaScript-инструментом, который мы добавим чуть позже. Пока подразумеваем, что данные будут;
- 2) если форма отправлена POST-запросом, проверяет ее корректность. Если данные валидны, создает новый объект `Image`, но пока не сохраняет его в базу данных, передавая аргумент `commit=False`;
- 3) привязывает текущего пользователя к картинке. Так мы узнаем, кто загрузил это изображение;
- 4) сохраняет объект `image` в базу данных;
- 5) наконец, создает уведомление и перенаправляет пользователя на канонический URL новой картинки. Мы пока не реализовали метод `get_absolute_url()` для модели `Image`, но добавим его чуть позже.

Создайте новый файл `urls.py` в приложении `images` и добавьте следующий код:

```

from django.urls import path
from . import views

app_name = 'images'

urlpatterns = [
    path('create/', views.image_create, name='create'),
]

```

Для того чтобы добавить URL-шаблоны приложения `images`, отредактируйте `urls.py` проекта `bookmarks`:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/',
         include('social_django.urls', namespace='social')),
    path('images/', include('images.urls', namespace='images')),
]
```

Наконец, нужно добавить шаблон для отображения формы. Создайте следующую структуру папок и файлов в приложении `images`:

```
templates/
  images/
    image/
      create.html
```

Внесите в шаблон `create.html` код формы добавления картинки:

```
{% extends "base.html" %}
{% block title %}Bookmark an image{% endblock %}
{% block content %}
<h1>Bookmark an image</h1>

<form action="." method="post">
  {{ form.as_p }}
  {% csrf_token %}
  <input type="submit" value="Bookmark it!">
</form>
{% endblock %}
```

Теперь откройте в браузере страницу `http://127.0.0.1:8000/images/create/?title=...&url=...`, подставив в GET-параметры `title` и `url` соответствующие значения, заголовок и URL, для JPEG-картинки с любого сайта.

Например, вы можете использовать такой URL:

```
http://127.0.0.1:8000/images/create/?title=%20Django%20and%20Duke&url=http://upload.wikimedia.org/wikipedia/commons/8/85/Django_Reinhardt_and_Duke_Ellington_%28Gottlieb%29.jpg
```

Вы увидите форму и превью изображения (рис. 5.2).

Добавьте описание и кликните на кнопку **BOOKMARK IT!**. Новый объект `Image` будет сохранен в базу данных. При этом вы увидите сообщение о том, что у модели `Image` нет метода `get_absolute_url()` (рис. 5.3).

Не беспокойтесь об этом. Ошибка появляется в связи с тем, что мы пока не реализовали метод `get_absolute_url()` для модели `Image`. Но мы добавим его чуть позже. Откройте `http://127.0.0.1:8000/admin/images/image/` и убедитесь, что объект изображения сохранен (рис. 5.4).

Рис. 5.2 ❖ Форма добавления картинки

```
AttributeError at /images/create/
'Image' object has no attribute 'get_absolute_url'
```

Рис. 5.3 ❖ Ошибка Attribute Error после сохранения изображения

| TITLE | SLUG | IMAGE | CREATED |
|--|-----------------|---------------------------------------|---------------|
| <input type="checkbox"/> Django and Duke | django-and-duke | images/2017/11/05/django-and-duke.jpg | Dec. 16, 2017 |

Рис. 5.4 ❖ Сохраненное изображение на сайте администрирования

Букмарклет на jQuery

Букмарклет – это сохраненные в браузере закладки с изображениями. Он реализуется на языке JavaScript и расширяет функциональность браузера. Когда пользователь кликает на закладку, выполняется наш JavaScript-код, что позволяет работать с содержимым любого сайта. Это очень полезная возможность для создания инструментов, взаимодействующих с другими сайтами.

Некоторые онлайн-сервисы, например Pinterest, реализуют собственный букмарклет, чтобы их пользователи могли делиться контентом на других плат-

формах. Мы создадим букмарклет аналогичным образом, чтобы наши пользователи могли делиться на сайте картинками из других источников.

Для реализации букмарклета будем использовать jQuery. Это популярная библиотека JavaScript, которая позволяет быстрее разрабатывать функциональность для браузеров. Более подробно о ней вы можете прочесть на сайте <https://jquery.com/>.

Давайте подумаем, как пользователи будут использовать букмарклет:

- 1) пользователь один раз сохраняет ссылку с нашего сайта в закладках браузера. Ссылка содержит JavaScript-код в атрибуте href. Этот код будет сохранен в закладке;
- 2) пользователь переходит на любой сайт и кликает на закладку. Сохраненный JavaScript-код начинает выполняться.

Так как JavaScript-код сохраняется в виде закладки, мы не сможем изменить его после сохранения. Это ограничение можно обойти, добавив функцию, которая будет обновлять выполняемый код букмарклета. Наши пользователи будут сохранять именно его, и мы сможем модифицировать код букмарклета в любое время. При выполнении кода из закладки будет скачиваться актуальная версия букмарклета.

Создайте шаблон `bookmarklet_launcher.js` в папке `images/templates/`. Это будет код для обновления кода букмарклета:

```
(function(){
  if (window.myBookmarklet !== undefined){
    myBookmarklet();
  }
  else {
    document.body.appendChild(
      document.createElement('script')
    ).src='http://127.0.0.1:8000/static/js/bookmarklet.js?r=' +
      Math.floor(Math.random()*999999999999999999);
  }
})();
```

Этот фрагмент проверяет, был ли уже загружен код букмарклета, который хранится в переменной `myBookmarklet`. Так мы избегаем лишней загрузки кода в случае, когда пользователь повторно кликает на букмарклет. Если переменная `myBookmarklet` не содержит значения, код загружает другой JavaScript-файл, добавляя `<script>`-элемент в документ. Тег `<script>` загружает `bookmarklet.js`, добавляя к названию случайное число. Это необходимо для предотвращения кеширования файла браузером.

Актуальный код букмарклета будет находиться в файле `bookmarklet.js`. Это позволит обновлять выполняемый код без необходимости для пользователей обновлять закладку, которую они добавили ранее. Давайте вставим ссылку для загрузки букмарклета на рабочий стол, чтобы пользователи смогли сохранить его в закладки.

Отредактируйте шаблон `account/dashboard.html` приложения `account`. Он должен выглядеть таким образом:

```

{% extends "base.html" %}
{% block title %}Dashboard{% endblock %}
{% block content %}
<h1>Dashboard</h1>

{% with total_images_created=request.user.images_created.count %}
  <p>Welcome to your dashboard. You have bookmarked
  {{total_images_created}} image{{ total_images_created|pluralize }}.</p>
{% endwith %}

<p>Drag the following button to your bookmarks toolbar to bookmark images
from other websites
  <a href="javascript:{% include "bookmarklet_launcher.js" %}" class="button">
  Bookmark it</a>
<p>

<p>You can also <a href="{% url "edit" %}">edit your profile</a> or
<a href="{% url "password_change" %}">change your password</a>.<p>
{% endblock %}

```

Теперь рабочий стол отображает общее количество картинок, сохраненных пользователем. Мы используем тег `{% with %}`, чтобы задать это число. Также мы добавили ссылку с атрибутом `href`, содержащим адрес скрипта для загрузки букмарклета, код из шаблона `bookmarklet_launcher.js`. Откройте <http://127.0.0.1:8000/account/>. Вы увидите измененную страницу:

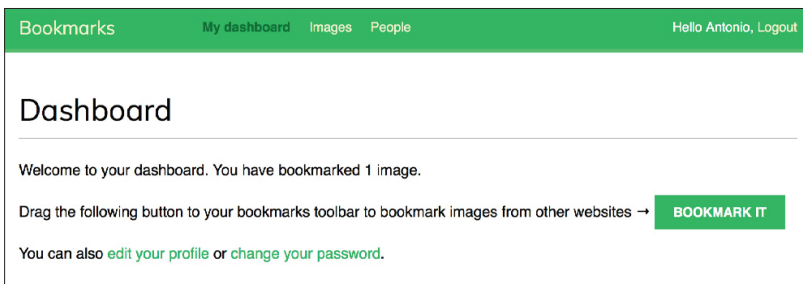


Рис. 5.5 ❖ Ссылка на добавление букмарклета в закладки на рабочем столе

Создайте аналогичную структуру каталогов и файлов внутри приложения `images`:

```

static/
  js/
    bookmarklet.js

```

Папку `static/css/` можно найти в папке приложения `images` в примерах кода к этой главе. Скопируйте каталог `css/` в созданный нами каталог `static/`. Файл `css/bookmarklet.css` определяет стили для нашего JavaScript-букмарклета.

Отредактируйте файл `bookmarklet.js` и добавьте в него следующий фрагмент:

```

(function(){
  var jquery_version = '3.3.1';
  var site_url = 'http://127.0.0.1:8000/';
  var static_url = site_url + 'static/';
  var min_width = 100;
  var min_height = 100;

  function bookmarklet(msg) {
    // Здесь мы добавим код самого букмарклета.
  };

  // Проверка, подключена ли jQuery.
  if(typeof window.jQuery != 'undefined') {
    bookmarklet();
  } else {
    // Проверка, что атрибут $ окна не занят другим объектом.
    var conflict = typeof window.$ != 'undefined';
    // Создание тега <script> с загрузкой jQuery.
    var script = document.createElement('script');
    script.src = '//ajax.googleapis.com/ajax/libs/jquery/' +
      jquery_version + '/jquery.min.js';
    // Добавление тега в блок <head> документа.
    document.head.appendChild(script);
    // Добавление возможности использовать несколько попыток для загрузки jQuery.
    var attempts = 15;
    (function(){
      // Проверка, подключена ли jQuery
      if(typeof window.jQuery == 'undefined') {
        if(--attempts > 0) {
          // Если не подключена, пытаемся снова загрузить
          window.setTimeout(arguments.callee, 250)
        } else {
          // Превышено число попыток загрузки jQuery, выводим сообщение.
          alert('An error occurred while loading jQuery')
        }
      } else {
        bookmarklet();
      }
    })();
  }
})();

```

Это основной код, который будет загружать букмарклет. Он отслеживает, была ли загружена jQuery на текущем сайте, и, если не была, загружает ее. Если библиотека уже была подключена, код выполняет функцию `bookmarklet()`. Также в начале кода определены некоторые переменные:

- `jquery_version` – необходимая версия jQuery;
- `site_url` и `static_url` – URL'ы нашего сайта и статических файлов;
- `min_width` и `min_height` – минимальные ширина и высота в пикселях для картинок, которые будет загружать букмарклет.

Теперь давайте реализуем функцию `bookmarklet`. Отредактируйте ее, чтобы она выглядела таким образом:

```
function bookmarklet(msg) {
  // Загрузка CSS-стилей.
  var css = jQuery('<link>');
  css.attr({
    rel: 'stylesheet',
    type: 'text/css',
    href: static_url + 'css/bookmarklet.css?r=' +
    Math.floor(Math.random()*9999999999999999999)
  });
  jQuery('head').append(css);

  // Загрузка HTML.
  box_html = '<div id="bookmarklet"><a href="#"id="close">&times;</a>
    <h1>Select an image to bookmark:</h1><divclass="images"></div></div>';
  jQuery('body').append(box_html);

  // Добавление скрытия букмарклета при нажатии на крестик.
  jQuery('#bookmarklet #close').click(function(){
    jQuery('#bookmarklet').remove();
  });
};
```

В этом коде мы выполняем следующие действия:

- 1) загружаем стили `bookmarklet.css`, добавляя случайное число для предотвращения кеширования стилей браузером;
- 2) добавляем HTML-элемент в `<body>` текущего сайта. Этот элемент содержит `<div>` с изображениями, найденными на сайте;
- 3) добавляем событие, которое удаляет наш HTML из документа сайта, когда пользователь кликает на кнопку закрытия блока. Используем селектор `#bookmarklet#close`, чтобы найти элемент с ID `close`, у которого есть родительский элемент с ID `bookmarklet`. Селекторы jQuery позволяют нам находить HTML-элементы. Они возвращают все подходящие объекты. Более подробную информацию о селекторах jQuery можно найти на странице <https://api.jquery.com/category/selectors/>.

После загрузки CSS-стилей и HTML-кода для букмарклета необходимо найти все изображения на текущем сайте. Добавьте этот фрагмент кода в конец функции `bookmarklet()`:

```
// Находим картинки на текущем сайте и вставляем их в окно букмарклета.
jQuery.each(jQuery('img[src$=".jpg"]'), function(index, image) {
  if (jQuery(image).width() >= min_width && jQuery(image).height() >= min_height){
    image_url = jQuery(image).attr('src');
    jQuery('#bookmarklet .images').append(
      '<a href="#"></a>'
    );
  }
});
```

Здесь используется селектор `img[src$=".jpg"]`, чтобы найти все ``-элементы, у которых значение атрибута `src` заканчивается на `jpg`. Так мы найдем все JPEG-изображения на текущем сайте. Для итерации по ним обращаемся к методу

jQueryeach(). Все изображения, большие по ширине и высоте, чем заданные `min_width` и `min_height`, добавляем в наш контейнер `<divclass="images">`.

Вы сможете загрузить букмарклет на любой сайт, включая те, которые используют протокол HTTPS. В настоящее время SSL применяется на большинстве сайтов. Из соображений безопасности браузер предотвратит загрузку букмарклета по протоколу HTTP на сайте, который использует HTTPS.

Сервер Django предназначен только для разработки и не поддерживает HTTPS. Чтобы протестировать букмарклет с HTTPS, нам необходим Ngrok. Ngrok – это инструмент для создания туннеля и открытия вашего локального хоста в общую сеть по HTTP или HTTPS. Давайте установим его.

Скачайте Ngrok для вашей операционной системы на сайте <https://ngrok.com/download> и запустите его, выполнив команду:

```
./ngrok http 8000
```

С помощью этой команды мы велит Ngrok создать туннель к вашему локальному хосту на порте 8000 и присвоить ему доступное извне имя хоста. Вы увидите подобный вывод:

```
Session Status   online
Version          2.2.8
Region           United States (us)
Web Interface    http://127.0.0.1:4040
Forwarding       http://3f6ad53c.ngrok.io -> localhost:8000
Forwarding       https://3f6ad53c.ngrok.io -> localhost:8000

Connections      ttl    opn    rt1    rt5    p50    p90
0                0      0      0.00  0.00  0.00  0.00
```

Ngrok вывел информацию о том, что наш сайт, запущенный Django локально на 8000 порте, теперь доступен извне по адресам: `http://3f6ad53c.ngrok.io` и `https://3f6ad53c.ngrok.io`. Эти URL'ы используют протоколы HTTP и HTTPS. Также вы можете увидеть выполняемые к серверу запросы, если перейдете на `http://127.0.0.1:4040`.

Чтобы Django мог обрабатывать запросы, отредактируйте `settings.py` проекта и добавьте новый адрес в список `ALLOWED_HOSTS`:

```
ALLOWED_HOSTS = [
    'mysite.com',
    'localhost',
    '127.0.0.1',
    '3f6ad53c.ngrok.io'
]
```

Так мы сможем обращаться к нашему приложению через внешний адрес. Откройте `https://3f6ad53c.ngrok.io/account/login/`, заменив домен на тот, который для вас сформировал Ngrok. Вы увидите страницу входа.

Отредактируйте шаблон `bookmarklet_launcher.js` и замените `http://127.0.0.1:8000/` на тот домен, который вам назначил Ngrok, указав протокол HTTPS, например:

```
(function(){
  if (window.myBookmarklet !== undefined){
    myBookmarklet();
  }
  else {
    document.body.appendChild(
      document.createElement('script')
    ).src='https://3f6ad53c.ngrok.io/static/js/bookmarklet.js?r='
      + Math.floor(Math.random()*999999999999999999);
  }
})();
```

В файле `js/bookmarklet.js` обратите внимание на строку:

```
var site_url = 'http://127.0.0.1:8000/';
```

Замените URL на новый адрес с протоколом HTTPS:

```
var site_url = 'https://3f6ad53c.ngrok.io/';
```

Откройте `https://3f6ad53c.ngrok.io/account/`, заменив ваш домен Ngrok. Войдите в аккаунт и перетащите кнопку **BOOKMARK IT** в закладки браузера:

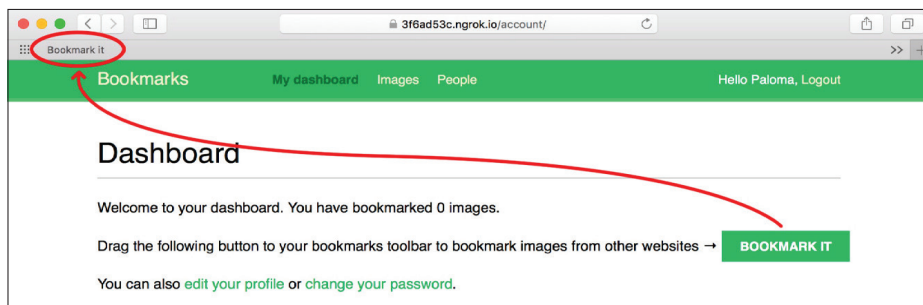


Рис. 5.6 ❖ Добавление букмарклета в закладки

Перейдите на любой сайт и кликните на букмарклет. Вы увидите, как справа появился белый блок, содержащий все JPEG-картинки текущего сайта с размером больше, чем 100×100 пикселей (рис. 5.6).

Этот HTML-контейнер содержит все картинки сайта, которые можно добавить в закладки. Для этого нужно просто кликнуть на изображение. Отредактируйте файл `js/bookmarklet.js` и добавьте этот фрагмент в конец функции `bookmarklet()`:

```
// Когда изображение выбрано, добавляем его в список сохраненных картинок на нашем сайте.
jQuery('#bookmarklet .images a').click(function(e){
  selected_image = jQuery(this).children('img').attr('src');
  // Скрываем букмарклет.
  jQuery('#bookmarklet').hide();
  // Открываем новое окно с формой сохранения изображения.
```

```

window.open(site_url + 'images/create/?url='
+ encodeURIComponent(selected_image)
+ '&title=' + encodeURIComponent(jQuery('title').text()),
'_blank');
});

```

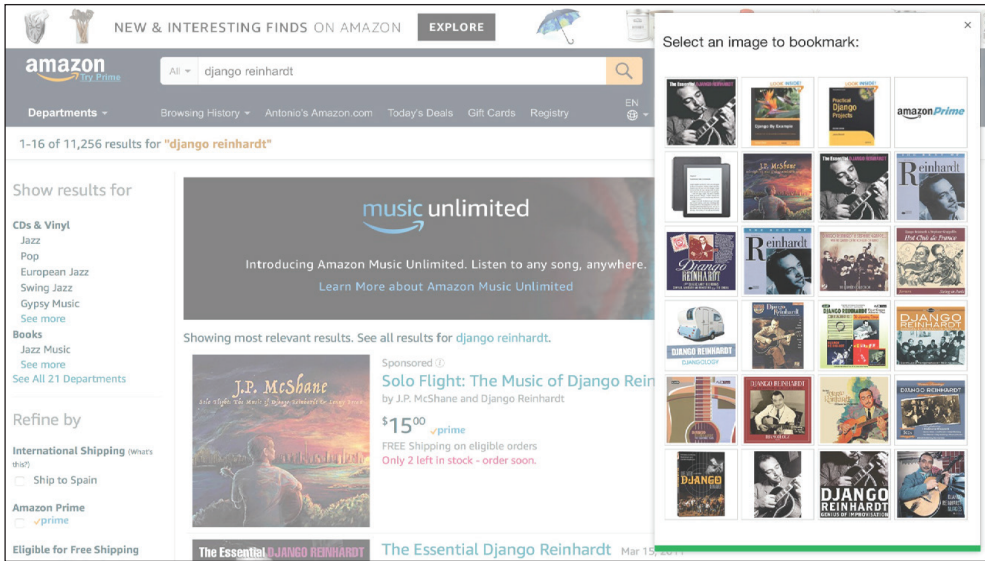


Рис. 5.7 ❖ Окно букмарклета с изображениями с текущего сайта

Этот код выполняет следующие действия:

- 1) привязывает обработчик события `click` на ссылку изображения;
- 2) когда пользователь кликает на изображение, сохраняет адрес картинки в переменную `selected_image`;
- 3) скрывает букмарклет и открывает новую вкладку браузера с GET-параметрами (передает заголовок страницы и URL картинки).

Откройте новый URL в браузере и кликните на букмарклет, чтобы открылся блок выбора картинок. Если вы кликнете на одну из них, то будете перенаправлены на новую страницу с уже заполненными заголовком и URL'ом изображения (рис. 5.8).

Поздравляем! Только что мы создали свой первый букмарклет и интегрировали его в Django-проект.

Рис. 5.8 ❖ Форма сохранения изображения в закладки

СОЗДАНИЕ ОБРАБОТЧИКА ДЛЯ КАРТИНКИ

Теперь добавим обработчик для редактирования сведений об уже сохраненных изображениях. Откройте `views.py` приложения `images` и добавьте эту функцию:

```
from django.shortcuts import get_object_or_404
from .models import Image

def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images', 'image': image})
```

Это простой обработчик, который показывает сведения об изображении. Добавьте URL-шаблон для него в файл `urls.py` приложения `images`:

```
path('detail/<int:id>/<slug:slug>/', views.image_detail, name='detail'),
```

Теперь давайте определим метод `get_absolute_url()` для модели `Image`. Для этого вставьте в файл `models.py` приложения `images` следующие строки:

```
from django.urls import reverse

class Image(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('images:detail', args=[self.id, self.slug])
```

Помните, что общепринятый способ задания канонического URL'a – определение для модели метода `get_absolute_url()`.

Наконец, создадим шаблон для показа сохраненной картинки, `detail.html`, в папке `/images/image/` приложения `images`:

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

{% block title %}{{ image.title }}{% endblock %}

{% block content %}
<h1>{{ image.title }}</h1>

{% with total_likes=image.users_like.count %}
  <div class="image-info">
    <div>
      <span class="count">{{ total_likes }} like{{ total_likes|pluralize }}</span>
    </div>
    {{ image.description|linebreaks }}
  </div>
  <div class="image-likes">
    {% for user in image.users_like.all %}
      <div>
        
        <p>{{ user.first_name }}</p>
      </div>
    {% empty %}
      Nobody likes this image yet.
    {% endfor %}
  </div>
{% endwith %}
{% endblock %}
```

Этот шаблон показывает подробные сведения о картинке, добавленной в закладки. Мы используем тег `{% with %}`, для того чтобы сохранить в переменную `total_likes` результат выполнения `QuerySet`'а, вычисляющего количество лайков текущей картинки. Так мы избегаем выполнения этого запроса дважды. Также мы отображаем описание изображения и проходим по списку `image.users_like.all` для отображения списка пользователей, которым понравилась текущая картинка.



Использование тега `{% with %}` в шаблонах – хороший способ избежать многократного вычисления `QuerySet`'ов.

Теперь добавьте через букмарклет одну картинку в закладки. Вы будете перенаправлены на страницу подробностей этой картинки, которая будет выглядеть аналогичным образом:

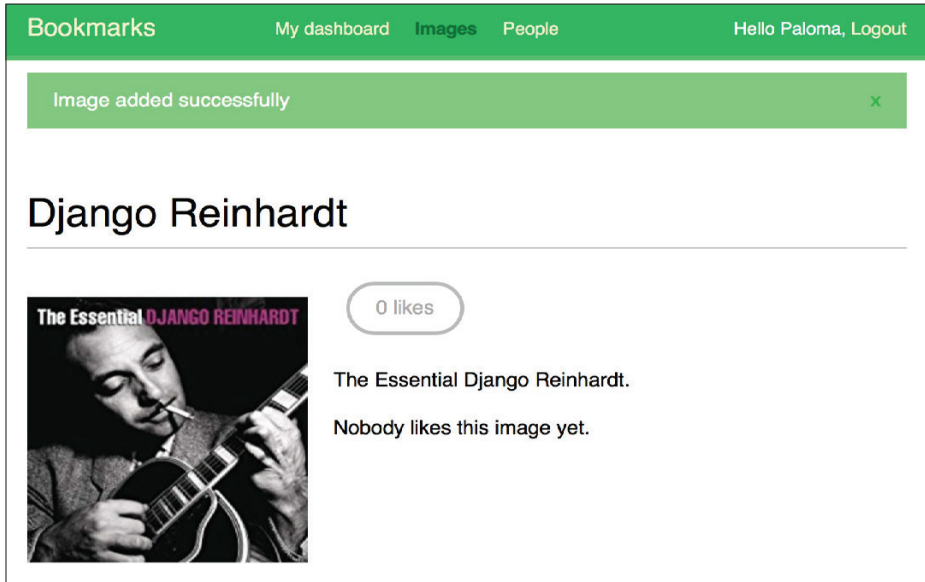


Рис. 5.9 ❖ Страница сохраненной на нашем сайте картинки

ДОБАВЛЕНИЕ ПРЕВЬЮ ДЛЯ ИЗОБРАЖЕНИЙ

На странице подробностей об изображении мы показываем картинку в ее оригинальном размере. Но размеры могут сильно различаться для разных изображений. Некоторые из них могут оказаться настолько большими, что их загрузка будет занимать значительное время. Лучший способ показывать картинки, приведенные к единому размеру, – генерировать их превью. Для этого мы будем использовать Django-приложение `sorl-thumbnail`.

Откройте терминал и установите это приложение командой:

```
pip install sorl-thumbnail==12.4.1
```

Подключите его к проекту, добавив в список `INSTALLED_APPS` настроек `settings.py`:

```
INSTALLED_APPS = [
    # ...
    'sorl.thumbnail',
]
```

Затем синхронизируйте модели с базой данных:

```
python manage.py migrate
```

Вы увидите вывод, который содержит такую строку, следовательно, теперь наше приложение может использовать возможности `sorl-thumbnails`:

```
Applying thumbnail.0001_initial... OK
```

Приложение `sorl-thumbnail` предоставляет несколько способов создания превью изображений. Например, шаблонный тег `{% thumbnail %}` предназначен для генерации превью в шаблонах, поле `ImageField` подходит для определения превью в модели. Мы будем использовать первый способ. Отредактируйте шаблон `images/image/detail.html`, заменив строку

```

```

на фрагмент с использованием тега `{% thumbnail %}`:

```
{% load thumbnail %}
{% thumbnail image.image "300" as im %}
  <a href="{{ image.image.url }}">
    
  </a>
{% endthumbnail %}
```

В этом фрагменте мы создаем превью с фиксированным размером 300 пикселей. Когда пользователь первый раз загрузит эту страницу, создастся превью. Оно же будет использоваться при всех последующих обращениях к странице. Запустите сервер разработки командой `python manage.py runserver` и перейдите на страницу любого изображения. Будет сформировано превью, и вы увидите именно его.

Приложение `sorl-thumbnails` поддерживает различные настройки генерации превью (например, можно выбирать алгоритмы обрезки изображений, разные эффекты). Если у вас возникли трудности с генерацией превью, добавьте в настройки `settings.py` атрибут `THUMBNAIL_DEBUG=True`, с помощью чего сможете отладить этот процесс. Полную документацию приложения `sorl-thumbnail` можно найти на сайте <https://sorl-thumbnail.readthedocs.io/>.

РЕАЛИЗАЦИЯ AJAX-ЗАПРОСОВ С JQUERY

В этом разделе мы добавим в приложение AJAX-запросы. *AJAX* – сокращение от *асинхронный, JavaScript и XML*. Этот термин охватывает механизмы, позволяющие выполнять асинхронные HTTP-запросы, т. е. отправку данных на сервер и получение результатов без перезагрузки всей страницы. Несмотря на расшифровку сокращения, XML не является обязательным форматом. Мы можем отправлять и получать данные и в других форматах (например, JSON, HTML или текстом).

Мы добавим ссылку на страницу изображения, чтобы пользователи могли лайкнуть картинку, кликнув на эту ссылку. Запрос на лайк будет выполняться асинхронно, через AJAX. Для начала нужно создать обработчик, чтобы пользо-

ватели могли отмечать картинки как понравившиеся и снимать эту отметку. Отредактируйте `views.py` приложения `images`, добавьте эту функцию:

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST

@login_required
@require_POST
def image_like(request):
    image_id = request.POST.get('id')
    action = request.POST.get('action')
    if image_id and action:
        try:
            image = Image.objects.get(id=image_id)
            if action == 'like':
                image.users_like.add(request.user)
            else:
                image.users_like.remove(request.user)
            return JsonResponse({'status': 'ok'})
        except:
            pass
    return JsonResponse({'status': 'ok'})
```

Здесь мы используем два декоратора для функции. Декоратор `login_required` не даст неавторизованным пользователям доступ к этому обработчику. Декоратор `require_POST` возвращает ошибку `HttpResponseNotAllowed` (статус ответа 405), если запрос отправлен не методом `POST`. Таким образом, обработчик будет выполняться только при `POST`-запросах. В Django также реализованы декораторы `required_GET`, запрещающий любые методы, кроме `GET`, и декоратор `require_http_methods`, принимающий список разрешенных методов в качестве аргумента.

В этом обработчике мы используем два `POST`-параметра:

- `image_id` – ID изображения, для которого выполняется действие;
- `action` – действие, которое хочет выполнить пользователь (строковое значение `like` или `unlike`).

Мы используем менеджер отношения «многие ко многим» `users_like` модели `Image`, чтобы добавлять и удалять пользователей с помощью методов `add()` и `remove()` соответственно. Если вы вызываете `add()` и передаете в него пользователя, который уже связан с текущей картинкой, дубликат не будет создан. Аналогично при вызове `remove()` и попытке удалить пользователя, который не связан с изображением, ошибка не появится. Еще один полезный метод менеджера «многие ко многим» – `clear()`. Он удаляет все отношения.

В конце обработчика используем объект `JsonResponse`, который возвращает HTTP-ответ с типом `application/json` и преобразует объекты в JSON.

Добавьте URL-шаблон для этого обработчика в файл `urls.py` приложения `images`:

```
path('like/', views.image_like, name='like'),
```


Подключение jQuery

Для обращения к серверу через AJAX нужно подключить jQuery в шаблоне `base.html` приложения `account`. Откройте этот файл и добавьте следующие строки перед закрывающим тегом `</body>`:

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
<script>
  $(document).ready(function(){
    {% block domready %}{% endblock %}
  });
</script>
```

Мы подключили библиотеку jQuery из CDN Google. Есть второй способ ее подключения: скачать jQuery с сайта <https://jquery.com/> в папку `static` и подключить код оттуда.

Мы добавили тег `<script>` для использования JavaScript-кода. Метод `$(document).ready()` – это функция jQuery, она получает в качестве аргумента код, который будет выполнен, когда удастся выстроить *иерархию документа* (Document Object Model – DOM). Браузер формирует DOM в виде древовидной структуры после загрузки страницы. Передавая выполняемый код в метод `ready()`, гарантируем, что элементы, с которыми мы будем взаимодействовать, будут загружены и доступны в DOM (наш код выполнится только после того, как браузер сформирует его).

Внутри обработчика мы добавили блочный тег Django и назвали его `domready`. Так дочерние шаблоны смогут добавлять собственный JavaScript-код, который должен выполняться после формирования DOM.

Не пугайтесь, что мы смешиваем JavaScript-код и шаблонный синтаксис Django. Шаблоны Django обрабатываются на сервере, формируя готовый HTML, в то время как JavaScript выполняется в браузере. Но в некоторых случаях бывает необходимо обрабатывать JavaScript-код средствами Django.

В примерах этой главы мы прописываем JavaScript код в шаблонах Django. Но предпочтительный способ использования JavaScript (особенно в том случае, если код содержит множество строк) – это описывать его в `.js`-файлах и подключать через тег `{% static %}`.

Защита от межсайтовых запросов в AJAX

Из главы 2 мы узнали, что существуют межсайтовые атаки. Django реализует защиту от них с помощью CSRF-токена во всех POST-запросах. Когда необходимо отправлять форму, мы можем включать в нее тег `{% csrf_token %}`. Но для AJAX-запросов обычно не используют добавление CSRF-токена в данные каждого запроса. Вместо этого Django позволяет нам указать собственный заголовок `X-CSRFToken` со значением токена. Таким образом, при использовании jQuery или любой другой JavaScript-библиотеки для асинхронных запросов за-

головок X-CSRFToken будет содержать токен, вследствие чего Django сможет обработать этот запрос.

Для того чтобы добавить токен в каждый асинхронный запрос, необходимо:

- 1) получить CSRF-токен из csrfToken (задается, когда защита от CSRF активирована в настройках проекта);
- 2) отправить токен в заголовке X-CSRFToken.

Более подробно о CSRF-защите при AJAX-запросах можно прочитать на странице [https:// docs.djangoproject.com/en/2.0/ref/csrf/#ajax](https://docs.djangoproject.com/en/2.0/ref/csrf/#ajax).

Отредактируйте фрагмент, который мы последний раз добавили в base.html, чтобы он выглядел таким образом:

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/js-cookie@2/src/js.cookie.min.js"></script>
<script>
var csrfToken = Cookies.get('csrfToken');
functioncsrfSafeMethod(method) {
    // Для этих методов токен не будет подставляться в заголовок.
    return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
}
$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!csrfSafeMethod(settings.type) && !this.crossDomain) {
            xhr.setRequestHeader("X-CSRFToken", csrfToken);
        }
    }
});
$(document).ready(function(){
    {% block domready %}{% endblock %}
});
</script>
```

Новый код выполняет следующие действия:

- 1) подключает плагин JSCookie из публичного CDN. JSCookie – это библиотека, облегчающая работу с куками. Познакомиться с ней можно на сайте <https://github.com/js-cookie/js-cookie>;
- 2) получает значение csrfToken с помощью Cookies.get();
- 3) добавляет функцию csrfSafeMethod(), которая определяет, нужно ли проверять CSRF-токен для текущего HTTP-метода. Методы, которые не требуют проверки, – GET, HEAD, OPTIONS, TRACE;
- 4) настраивает AJAX-запросы методом \$.ajaxSetup(). Перед отправкой каждого AJAX-запроса проверяет, нужно ли выставлять CSRF-токен. Если нужно, то задает в заголовке X-CSRFToken запроса значение токена, полученное из куков.

В нашем случае CSRF-токен будет добавляться в заголовки запросов методами POST и PUT.

Выполнение AJAX-запросов с jQuery

Давайте выведем на страницу картинку с ссылкой для возможности лайкнуть ее. Отредактируйте шаблон `images/image/detail.html` приложения `images`, найдите строку

```
{% with total_likes=image.users_like.count %}
```

и замените ее на новую:

```
{% with total_likes=image.users_like.count users_like=image.users_like.all %}
```

Затем отредактируйте `<div>` элемент с `css-классом` `image-info` таким образом:

```
<div class="image-info">
  <div>
    <span class="count">
      <span class="total">{{ total_likes }}</span>
      like{{ total_likes|pluralize }}
    </span>
    <a href="#" data-id="{{ image.id }}" data-action="{% if
request.user in users_like %}un{% endif %}like"
class="like button">
      {% if request.user not in users_like %}
        Like
      {% else %}
        Unlike
      {% endif %}
    </a>
  </div>
  {{ image.description|linebreaks }}
</div>
```

Здесь мы добавили еще одну переменную с помощью тега `{% with %}`, для того чтобы запомнить результат выполнения `QuerySet'a` `image.users_like.all` и избежать его повторного вычисления. Мы показываем число пользователей, которым понравилась текущая картинка, и ссылку, чтобы лайкнуть или отменить лайк. Чтобы понять, какую именно ссылку показывать, проверяем, находится ли текущий пользователь в списке `users_like` (тех, кому картинка понравилась). Также мы добавили атрибуты в тег `<a>`:

- `data-id` – ID изображения;
- `data-action` – действие, которое нужно выполнять после клика пользователя на ссылку (`like` или `unlike`).

Оба этих атрибута будут отправляться на сервер через AJAX в обработчик `image_like`. Когда пользователь кликает на ссылку `like/unlike` на стороне браузера, необходимо выполнить следующие действия:

- 1) отправить AJAX-запрос, передав в качестве параметров ID изображения и действие;
- 2) если получен успешный результат, следует обновить атрибут `data-action` тега `<a>` на противоположное значение (`like/unlike`) и заменить текст соответствующим образом;
- 3) обновить общее количество лайков `likes`.

Добавьте блок `domready` в конец шаблона `images/image/detail.html`, включив в него следующий код:

```
{% block domready %}
$( 'a.like' ).click( function( e ) {
    e.preventDefault();
    $.post(
        '{% url "images:like" %}',
        { id: $( this ).data( 'id' ), action: $( this ).data( 'action' ) },
        function( data ) {
            if ( data[ 'status' ] == 'ok' ) {
                var previous_action = $( 'a.like' ).data( 'action' );

                // Изменяем переменную действия.
                $( 'a.like' ).data( 'action', previous_action == 'like' ? 'unlike' : 'like' );
                // Изменяем текст ссылки.
                $( 'a.like' ).text( previous_action == 'like' ? 'Unlike' : 'Like' );
                // Обновляем общее количество лайков.
                var previous_likes = parseInt( $( 'span.count .total' ).text() );
                $( 'span.count .total' ).text(
                    previous_action == 'like' ? previous_likes + 1 : previous_likes - 1
                );
            }
        }
    );
});
{% endblock %}
```

В этом скрипте мы выполняем следующие действия:

- 1) используем селектор `$('a.like')`, чтобы найти все элементы `<a>`, у которых есть `css-класс like`;
- 2) определяем функцию – обработчик события `click`. Она будет вызываться при каждом клике на ссылке `like/unlike`;
- 3) внутри обработчика используем `e.preventDefault()`, чтобы отменить поведение по умолчанию для ссылки `<a>` (при клике пользователь не будет перенаправлен на страницу по ссылке);
- 4) используем функцию `$.post()` для выполнения асинхронного `POST`-запроса. `jQuery` также предоставляет функции `$.get()` для отправки `GET`-запроса и `$.ajax()` (в нем мы можем указать метод запроса);
- 5) мы используем шаблонный тег `Django{% url %}`, чтобы получить URL обработчика `AJAX`-запроса на сервере;
- 6) формируем `POST`-параметры для обработчика на сервере (`id` и `action`). Получаем их значения из атрибутов `data-id` и `data-action` элемента `<a>`;
- 7) определяем функцию, которая будет вызвана при успешном выполнении запроса. Она получает в качестве аргумента `data` данные из тела ответа;
- 8) проверяем атрибут `status` полученных в ответе данных – равен ли он значению `ok`. Если равен, заменяем атрибут `data-action` и текст ссылки на противоположные. Так пользователь сможет отменить свое действие;
- 9) увеличиваем или уменьшаем общее количество лайков в зависимости от выполненного действия.

Откройте страницу добавленной в закладки картинки. Вы увидите блок с общим количеством лайков и кнопку **LIKE**:

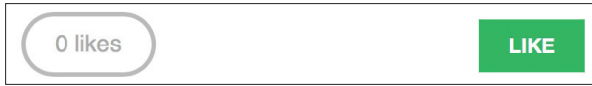


Рис. 5.10 ❖ Отображение количества пользователей, которым понравилась текущая картинка

Кликните на кнопку **LIKE**, и вы увидите, как количество лайков увеличилось на один, а текст кнопки сменился на **UNLIKE**:



Рис. 5.11 ❖ Количество пользователей, которым понравилось изображение, увеличилось

Если вы кликнете на кнопку снова, ваш лайк отменится, а текст на кнопке и общее количество лайков изменятся соответствующим образом.

При использовании JavaScript, особенно для выполнения AJAX, рекомендуем применять инструменты для отладки JavaScript и HTTP-запросов. Все современные браузеры имеют встроенные инструменты разработчика для отладки JavaScript. Обычно для доступа к нему вы можете кликнуть правой кнопкой мыши на любом элементе на сайте и в контекстном меню выбрать **Посмотреть исходный код элемента (Inspectelement)**.

СОЗДАНИЕ СОБСТВЕННЫХ ДЕКОРАТОРОВ

Давайте добавим ограничения для AJAX-обработчиков, чтобы они могли принимать только AJAX-запросы. У объекта запроса Django есть метод `is_ajax()`, который проверяет, сделан ли запрос с помощью XMLHttpRequest. Это означает, что он является асинхронным. Значение задается в HTTP-заголовке `HTTP_X_REQUESTED_WITH`, который поддерживается большинством современных AJAX-библиотек.

Мы создадим декоратор, проверяющий заголовок `HTTP_X_REQUESTED_WITH`. *Декоратор* – это функция, которая принимает в качестве аргумента другую функцию и расширяет ее поведение, не изменяя исходное. Если вы незнакомы с понятием декоратор, то вам стоит познакомиться с ним на странице <https://www.python.org/dev/peps/pep-0318/>.

Так как наш декоратор будет применим к любому обработчику, независимо от того, в каком приложении он используется, давайте создадим в проекте новый пакет `common`. Добавьте такую структуру каталогов и файлов в проект `bookmarks`:

```
common/
  __init__.py
  decorators.py
```

Откройте файл `decorators.py` и добавьте в него следующий код:

```
from django.http import HttpResponseRedirect

def ajax_required(f):
    def wrap(request, *args, **kwargs):
        if not request.is_ajax():
            return HttpResponseRedirect()
        return f(request, *args, **kwargs)
    wrap.__doc__=f.__doc__
    wrap.__name__=f.__name__
    return wrap
```

Это и есть наш декоратор `ajax_required`. В нем определена функция, которая возвращает объект `HttpResponseBadRequest` (код ошибки – 400), если запрос не является AJAX-запросом. В противном случае возвращается результат выполнения декорируемой функции.

Теперь можно обернуть в декоратор наш AJAX-обработчик. Откройте файл `views.py` приложения `images` и отредактируйте его таким образом:

```
from common.decorators import ajax_required

@ajax_required
@login_required
@require_POST
def image_like(request):
    # ...
```

Если вы обратитесь к URL'у `http://127.0.0.1:8000/images/like/` напрямую через браузер, то получите ошибку 400.



Если вы замечаете, что выполняете одни и те же действия в нескольких обработчиках, скорее всего, вам нужно реализовать декоратор или вынести повторяющийся фрагмент кода в новую функцию во избежание дублирования кода.

ПОСТРАНИЧНЫЙ ВЫВОД С ПОМОЩЬЮ AJAX

Кроме страницы подробностей о картинке, необходимо отображать список всех изображений, сохраненных в закладки. Для реализации постраничного отображения будем использовать AJAX-запросы. При прокрутке списка до конца в него будет подгружаться следующая страница посредством AJAX.

Для начала необходимо создать обработчик для постраничного показа картинок. Он должен обрабатывать оба типа запросов: и синхронные, и асинхронные. Когда пользователь первый раз зайдет на страницу, мы будем генерировать список из первых восьми картинок. Когда пользователь прокрутит список до конца, мы выполним AJAX-запрос и добавим следующую страницу изображений в конец.

Этот обработчик будет обрабатывать и стандартную загрузку страницы, и AJAX-запрос для получения следующих 8 картинок. Добавьте в файл `views.py` приложения `images` такую функцию:

```
from django.http import HttpResponse
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

@login_required
def image_list(request):
    images = Image.objects.all()
    paginator = Paginator(images, 8)
    page = request.GET.get('page')
    try:
        images = paginator.page(page)
    except PageNotAnInteger:
        # Если переданная страница не является числом, возвращаем первую.
        images = paginator.page(1)
    except EmptyPage:
        if request.is_ajax():
            # Если получили AJAX-запрос с номером страницы, большим, чем их количество,
            # возвращаем пустую страницу.
            return HttpResponse('')
        # Если номер страницы больше, чем их количество, возвращаем последнюю.
        images = paginator.page(paginator.num_pages)
    if request.is_ajax():
        return render(request, 'images/image/list_ajax.html',
                      {'section': 'images', 'images': images})
    return render(request, 'images/image/list.html',
                  {'section': 'images', 'images': images})
```

В этом обработчике мы формируем `QuerySet` для получения всех изображений, сохраненных в закладки. Затем создаем объект `Paginator` и получаем постраничный список картинок. Если желаемой страницы не существует, обрабатываем исключение `EmptyPage`. В случае AJAX-запроса возвращаем пустое значение, чтобы остановить дальнейшую прокрутку списка картинок. Передаем контекст в два HTML-шаблона:

- для AJAX-запросов используем `list_ajax.html`. Он содержит только HTML для показа картинок;
- для стандартных запросов используем `list.html`. Этот шаблон наследуется от `base.html` и показывает полноценную страницу, на которую добавлен список картинок из `list_ajax.html`.

Добавьте URL-шаблон для этого обработчика в файл `urls.py` приложения `images`:

```
path('', views.image_list, name='list'),
```

Наконец, нужно создать упомянутые выше HTML-шаблоны. Внутри каталога `images/image/` создайте файл `list_ajax.html` с таким содержимым:

```
{% load thumbnail %}
{% for image in images %}
<div class="image">
```

```

<a href="{{ image.get_absolute_url }}">
  {% thumbnail image.image "300x300" crop="100%" as im %}
    <a href="{{ image.get_absolute_url }}">
      
    </a>
  {% endthumbnail %}
</a>
<div class="info">
  <a href="{{ image.get_absolute_url }}" class="title">{{ image.title }}</a>
</div>
</div>
{% endfor %}

```

Этот фрагмент будет показывать только список изображений. Мы используем его для формирования ответа на AJAX-запрос следующей страницы. Теперь добавьте еще один файл, `list.html`, который содержит этот код:

```

{% extends "base.html" %}

{% block title %}Images bookmarked{% endblock %}

{% block content %}
<h1>Images bookmarked</h1>
<div id="image-list">
  {% include "images/image/list_ajax.html" %}
</div>
{% endblock %}

```

Этот шаблон наследуется от `base.html`. Во избежание дублирования кода мы подключили к этому шаблону список картинок из `list_ajax.html`. Также файл `list.html` должен содержать код JavaScript для загрузки страниц списка.

Добавьте этот фрагмент в конец шаблона `list.html`:

```

{% block domready %}
var page = 1;
var empty_page = false;
var block_request = false;

$(window).scroll(function() {
  var margin = $(document).height() - $(window).height()-200;
  if ($(window).scrollTop() > margin && empty_page == false &&
    block_request == false) {
    block_request = true;
    page += 1;
    $.get('?page=' + page, function(data) {
      if(data == '') {empty_page = true;}
      else {
        block_request = false;
        $('#image-list').append(data);
      }
    });
  }
});
{% endblock %}

```


Этот код реализует подгрузку следующей страницы списка при прокрутке до конца. Мы добавили JavaScript-код в блок `domready`, который определили раньше в базовом шаблоне `base.html`. Этот скрипт выполняет действия, перечисленные ниже.

1. Определяет переменные:
 - `page` (хранит текущую страницу);
 - `empty_page` (является ли последняя полученная страница пустой. Таким образом, мы сможем остановить дальнейшие запросы при получении пустого ответа);
 - `block_request` (блокирует отправку запросов, если уже есть выполняемый запрос).
2. Задает обработчик, который будет выполняться при прокручивании страницы, с помощью `$(window).scroll()`.
3. Вычисляет разницу между высотой всей страницы и высотой видимой части, сохраняя в переменную `margin`. Это будет количество пикселей, оставшихся до конца страницы. Мы вычитаем 200 пикселей, чтобы загрузка следующей страницы началась немного раньше, чем пользователь промотает до конца.
4. Обращается к серверу для получения следующей страницы, проверяя, не является ли последней текущая показанная страница и не был ли послан другой запрос.
5. Выставляет переменную `block_request` в `true`, чтобы избежать дополнительных запросов, и увеличивает счетчик страниц `page`.
6. Выполняет GET-запрос с помощью `$.get()` и получает фрагмент HTML в переменной `data`. Далее возможны два сценария:
 - ответ не содержит данных, следовательно, мы получили пустой список (достигли конца списка). Выставляем переменную `empty_page` в `true` и больше не обращаемся к серверу при прокрутке;
 - ответ содержит данные. В таком случае добавляем полученные данные в конец списка – HTML-элемента с идентификатором `image-list`.

Откройте в браузере `http://127.0.0.1:8000/images/`. Вы увидите список картинок, которые уже добавили в закладки (рис. 5.12).

Прокрутите страницу до конца, и следующие 8 картинок загрузятся с сервера. Не забудьте, что в ваших закладках должно быть больше 8 изображений, только тогда постраничный вывод начнет работать. Для просмотра запросов и ответов и их отладки можете воспользоваться встроенными инструментами браузера.

Наконец, отредактируйте `base.html` приложения `account` и добавьте ссылку на список сохраненных картинок:

```
<li {% if section == "images" %}class="selected"{% endif %}>
<a href="{% url "images:list" %}">Images</a>
</li>
```

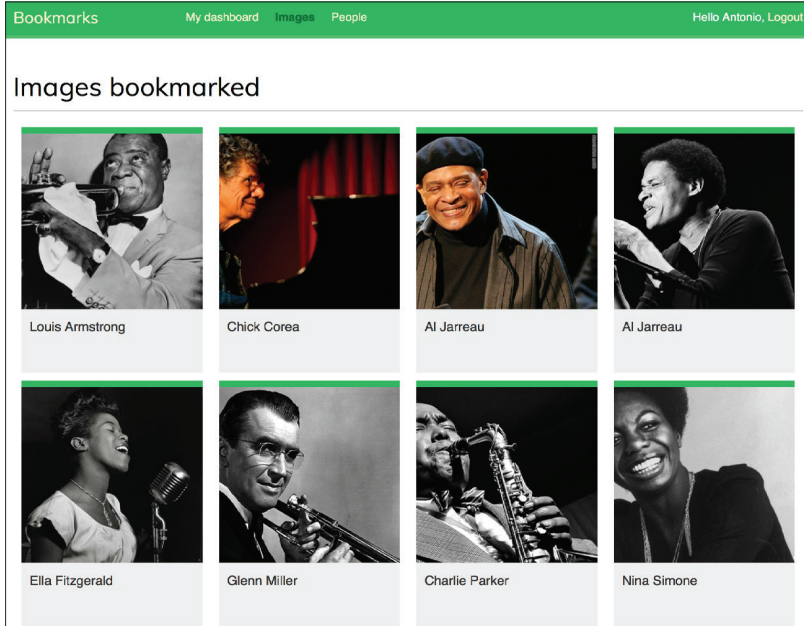


Рис. 5.12 ❖ Список добавленных в закладки картинок

Теперь ваши пользователи смогут переходить к закладкам прямо с рабочего стола.

РЕЗЮМЕ

В этой главе вы реализовали букмарклет для сохранения картинок с других сайтов, использовали обработчики AJAX, jQuery и добавили постраничный вывод с помощью AJAX.

В следующей главе мы познакомим вас с тем, как создать систему подписок пользователей и ленту активности. Вы поработаете с сигналами, обобщенными отношениями и денормализацией отношений, а также узнаете, как использовать Redis с Django.

Глава 6

Отслеживание действий пользователей

В предыдущей главе мы реализовали обработчики для AJAX-запросов, использовали jQuery и добавили на сайт букмарклет для сохранения в закладках картинок с других сайтов.

В этой главе мы узнаем, как реализовать систему подписок и ленту новостей. Мы будем использовать сигналы Django и подключим к проекту хранилище Redis. Нам предстоит познакомиться со следующими темами:

- реализация отношения «многие ко многим» с промежуточной моделью;
- создание системы подписок;
- добавление приложения новостной ленты;
- обобщенные отношения моделей;
- оптимизация запросов для связанных объектов;
- использование сигналов Django;
- хранение данных в Redis.

РЕАЛИЗАЦИЯ СИСТЕМЫ ПОДПИСОК

Давайте добавим систему подписок в наш проект. Пользователи смогут подписываться друг на друга и просматривать активность своих друзей. Такое отношение между пользователями можно задать с помощью связи «многие ко многим». В прямом направлении один пользователь может подписаться на несколько других, в обратном – у пользователя может быть несколько подписчиков.

Отношение «многие ко многим» с промежуточной моделью

В предыдущей главе мы добавили отношение «многие ко многим» с помощью поля модели ManyToManyField, и Django создал для него таблицу. Это подходит для большинства случаев, но иногда необходимо явно задать *промежуточную модель*. Например, это может быть полезно, когда необходимо сохранить дополнительную информацию об отношении (например, когда оно было созда-

но) или некоторые поля, поясняющие суть отношения.

Мы добавим такую промежуточную модель. Есть две причины, объясняющие необходимость этого в нашем проекте:

- мы используем стандартную модель пользователя `User` из Django и не можем ее модифицировать;
- мы хотим сохранить время создания отношения.

Отредактируйте файл `models.py` приложения `account` и добавьте в него модель:

```
class Contact(models.Model):
    user_from = models.ForeignKey('auth.User',
                                  related_name='rel_from_set',
                                  on_delete=models.CASCADE)
    user_to = models.ForeignKey('auth.User', related_name='rel_to_set',
                                on_delete=models.CASCADE)
    created = models.DateTimeField(auto_now_add=True, db_index=True)

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

    def __str__(self):
        return '{} follows {}'.format(self.user_from, self.user_to)
```

Этот фрагмент описывает модель `Contact`, которая будет использоваться для связи пользователей. Она содержит следующие поля:

- `user_from` – `ForeignKey` на пользователя-подписчика;
- `user_to` – `ForeignKey` на пользователя, на которого подписались;
- `created` – поле `DateTimeField` с `auto_now_add=True` для сохранения времени создания связи.

В базе данных индекс автоматически создается для полей `ForeignKey`. Мы также создадим индекс по полю `created` с помощью атрибута `db_index=True`. Это может ускорить запросы с фильтрацией и сортировкой по нему.

С помощью ORM Django можно добавить пользователя `user1` в подписчики пользователя `user2` таким образом:

```
user1 = User.objects.get(id=1)
user2 = User.objects.get(id=2)
Contact.objects.create(user_from=user1, user_to=user2)
```

Менеджеры `rel_from_set` и `rel_to_set` будут возвращать `QuerySet`'ы модели `Contact`. Чтобы связать пользователя и его подписчиков, в модели `User` нужно добавить поле типа `ManyToManyField`:

```
following = models.ManyToManyField('self',
                                    through=Contact,
                                    related_name='followers',
                                    symmetrical=False)
```

В этом фрагменте мы говорим Django, что необходимо использовать заданную явно промежуточную модель с помощью параметра `through=Contact`, переданного в `ManyToManyField`. Это отношение «многие ко многим» между объектами `User`: мы указываем ссылку на `self`, чтобы она указывала на текущую модель.

i Когда вам понадобятся дополнительные поля для описания отношения «многие ко многим», создайте дополнительную модель, которая будет содержать ForeignKey на обе связанные модели. Укажите ее в одной из связанных моделей как поле ManyToManyField с аргументом through.

Если модель User определена в нашем приложении, мы можем добавить в нее это поле напрямую. Но в нашем случае это невозможно, т. к. используется модель пользователя из `django.contrib.auth`. Поэтому мы добавим поле динамически. Отредактируйте файл `models.py` приложения `account` и вставьте следующий фрагмент:

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

# Динамическое добавление поля following в модель User
User.add_to_class('following',
                 models.ManyToManyField('self',
                                       through=Contact,
                                       related_name='followers',
                                       symmetrical=False))
```

Здесь мы обращаемся к методу модели `add_to_class()`, чтобы с легкостью изменить ее. Стоит отметить, что такой способ добавления атрибута рекомендуется использовать только в особенных случаях. Здесь это оправданно, т. к. мы:

- упростили доступ к связанным объектам с помощью Django ORM – `user.followers.all()` и `user.following.all()`. Добавили промежуточную модель `Contact` и избежали лишних объединений таблиц в базе данных, как это было бы, если бы мы определили связь в модели `Profile`;
- создали таблицу для связи из модели `Contact`. Таким образом, динамически добавленное в модель `User` поле `ManyToManyField` никак не повлияет на таблицу пользователей;
- избежали создания собственной модели пользователя, оставив стандартную модель `User` и все ее возможности.

Не забудьте, что в большинстве случаев предпочтительно использовать обычный способ добавления полей, а не динамический, как мы сделали это для модели `User`. Django позволяет определить и собственную модель пользователя. Более подробно это описано в документации на странице <https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#specifying-a-custom-user-model>.

Вы могли заметить, что отношение определено с параметром `symmetrical=False`. Когда мы создаем поле `ManyToManyField` на ту же самую модель, Django воспроизводит симметричные отношения. Поэтому мы явно указали `symmetrical=False`, чтобы оно было несимметричным. Так, если вы подпишетесь на меня, я не буду автоматически добавлен в список ваших подписчиков.

i Когда вы используете промежуточную модель для отношений «многие ко многим», некоторые методы менеджеров перестают работать, например `add()`, `create()`, `remove()`. Теперь вам будет нужно явно добавлять или удалять объекты промежуточной модели.

Выполните команду создания миграций для приложения `account`:

```
python manage.py makemigrations account
```

Вы увидите в консоли такой вывод:

```
Migrations for 'account':
  account/migrations/0002_contact.py
  - Create model Contact
```

Теперь синхронизируйте модели и таблицы в базе данных:

```
python manage.py migrate account
```

Вы увидите сообщение об успешном применении миграций:

```
Applying account.0002_contact... OK
```

Модель `Contact` теперь синхронизирована с базой данных, и мы можем создавать отношения между пользователями. До сих пор наш сайт не умеет показывать список пользователей и страницу профиля. Давайте создадим соответствующие обработчики для модели `User`.

Создание обработчиков списка пользователей и подробностей профиля

Откройте файл `views.py` приложения `account` и добавьте в него следующий код:

```
from django.shortcuts import get_object_or_404
from django.contrib.auth.models import User

@login_required
def user_list(request):
    users = User.objects.filter(is_active=True)
    return render(request,
                  'account/user/list.html',
                  {'section': 'people',
                  'users': users})

@login_required
def user_detail(request, username):
    user = get_object_or_404(User, username=username, is_active=True)
    return render(request,
                  'account/user/detail.html',
                  {'section': 'people',
                  'user': user})
```

Это простые обработчики. Функция `user_list` получает список всех активных пользователей. В модели `User` есть поле `is_active` для выставления активности пользователя. Мы фильтруем записи по условию `is_active=True` и получаем только активных. Обработчик возвращает всех подходящих пользователей, но вы можете доработать его и реализовать постраничное отображение, как в функции `image_list`.

Обработчик `user_detail` использует функцию `get_object_or_404()` для получения активного пользователя по его логину. Если такого пользователя не существует, будет возвращен HTTP-ответ со статусом 404.

Добавьте URL-шаблоны для новых обработчиков в файл `urls.py` приложения `account`:

```
urlpatterns = [
    # ...
    path('users/', views.user_list, name='user_list'),
    path('users/<username>/', views.user_detail, name='user_detail'),
]
```

Мы будем использовать шаблон `user_detail` для генерации канонических URL'ов пользователей с помощью метода `get_absolute_url()`. Другой способ задать URL для модели – задействовать настройку `ABSOLUTE_URL_OVERRIDES`.

Отредактируйте файл `settings.py` проекта и добавьте такой фрагмент:

```
from django.urls import reverse_lazy

ABSOLUTE_URL_OVERRIDES = {
    'auth.user': lambda u: reverse_lazy('user_detail', args=[u.username])
}
```

Django динамически добавляет метод `get_absolute_url()` для каждой модели, перечисленной в настройке `ABSOLUTE_URL_OVERRIDES`. В этом случае из настройки будет возвращаться соответствующий модели URL. В примере для пользователя мы возвращаем URL по имени `user_detail`. Теперь мы можем вызвать метод `get_absolute_url()` для объекта `User`, чтобы получить его канонический адрес.

Откройте Python-консоль с помощью команды `python manage.py shell` и выполните эти строки для тестирования метода:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.latest('id')
>>> str(user.get_absolute_url())
'/account/users/ellington/'
```

Возвращается ожидаемый адрес. Теперь необходимо создать шаблоны для этих обработчиков. Добавьте в каталог `templates/account/` приложения `account` одну папку и два файла:

```
/user/
  detail.html
  list.html
```

Вставьте в шаблон `account/user/list.html` следующий фрагмент:

```
{% extends "base.html" %}
{% load thumbnail %}

{% block title %}People{% endblock %}

{% block content %}
<h1>People</h1>
```

```

<div id="people-list">
  {% for user in users %}
    <div class="user">
      <a href="{{ user.get_absolute_url }}">
        {% thumbnail user.profile.photo "180x180" crop="100%" as im %}
          
        {% endthumbnail %}
      </a>
      <div class="info">
        <a href="{{ user.get_absolute_url }}" class="title">
          {{ user.get_full_name }}
        </a>
      </div>
    </div>
  {% endfor %}
</div>
{% endblock %}

```

Этот шаблон отображает список всех активных пользователей на сайте. Мы проходим по списку и используем тег `{% thumbnail %}` приложения `sorl-thumbnail` для отображения аватара пользователя.

Откройте базовый шаблон проекта `base.html` и добавьте ссылку на список пользователей:

```

<li {% if section == "people" %}class="selected"{% endif %}>
  <a href="{% url "user_list" %}">People</a>
</li>

```

Запустите сервер разработки командой `python manage.py runserver` и откройте в браузере `http://127.0.0.1:8000/account/users/`. Вы должны будете увидеть, что в шапке появилась новая ссылка:

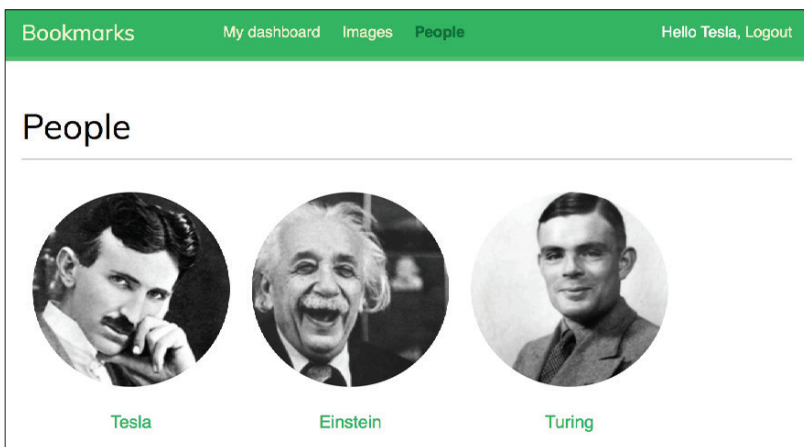


Рис. 6.1 ❖ Ссылка на пользователей сайта в шапке

Если у вас есть проблемы с формированием превью картинок, можно включить режим отладки, указав `THUMBNAIL_DEBUG = True` в файле `settings.py` проекта.

Отредактируйте шаблон `account/user/detail.html` приложения `account` и добавьте в него следующий код:

```
{% extends "base.html" %}
{% load thumbnail %}

{% block title %}{ user.get_full_name }}{% endblock %}
{% block content %}
<h1>{{ user.get_full_name }}</h1>
<div class="profile-info">
  {%thumbnail user.profile.photo "180x180" crop="100%" as im%}
  
  {% endthumbnail %}
</div>
{% with total_followers=user.followers.count %}
  <span class="count">
    <span class="total">{{ total_followers }}</span>
    follower{{ total_followers|pluralize }}
  </span>
  <a href="#" data-id="{{ user.id }}"
    data-action="{% if request.user in user.followers.all %}un{% endif %}follow"
    class="follow button">
    {% if request.user not in user.followers.all %}
      Follow
    {% else %}
      Unfollow
    {% endif %}
  </a>
  <div id="image-list" class="image-container">
    {% include "images/image/list_ajax.html" with
      images=user.images_created.all %}
  </div>
{% endwith %}
{% endblock %}
```

Это шаблон подробностей о пользователе. Мы отображаем профиль и используем тег `{% thumbnail %}` для аватара, а также показываем количество подписчиков и ссылку, для того чтобы подписаться или отписаться от пользователя. Для сохранения сведений в базе данных чуть позже добавим AJAX-запрос и соответствующий обработчик. В HTML-элемент `<a>` добавлены атрибуты `data-id` и `data-action`, которые содержат ID открытого профиля и действие `follow` или `unfollow`, доступное текущему пользователю. Также на странице показаны картинки, которые пользователь добавил в закладки. Для этого используется шаблон `images/image/list_ajax.html`.

Откройте браузер снова и кликните на профиль пользователя, который поделился с подписчиками картинками. Вы увидите похожую страницу:

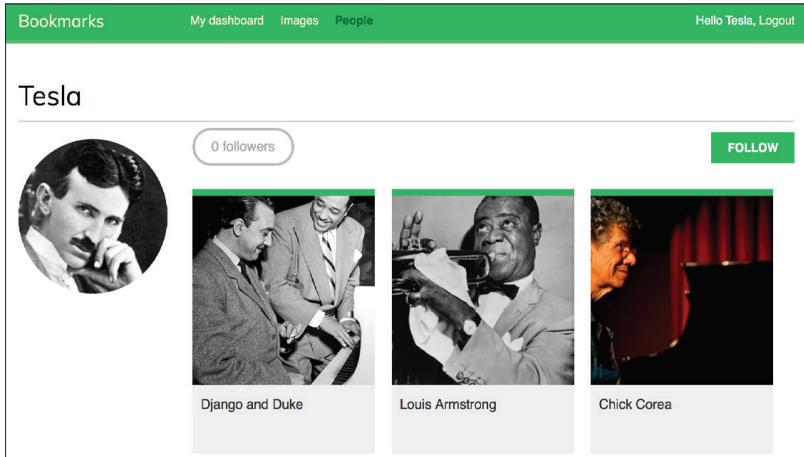


Рис 6.2 ❖ Страница пользователя сайта

АJAX-обработчик для создания подписчика

Давайте определим AJAX-обработчик, который будет добавлять и удалять пользователей из подписчиков. Допишите в файл `views.py` приложения `account` следующий фрагмент:

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST
from common.decorators import ajax_required
from .models import Contact

@ajax_required
@require_POST
@login_required
def user_follow(request):
    user_id = request.POST.get('id')
    action = request.POST.get('action')
    if user_id and action:
        try:
            user = User.objects.get(id=user_id)
            if action == 'follow':
                Contact.objects.get_or_create(user_from=request.user, user_to=user)
            else:
                Contact.objects.filter(user_from=request.user, user_to=user).delete()
            return JsonResponse({'status': 'ok'})
        except User.DoesNotExist:
            return JsonResponse({'status': 'ok'})
    return JsonResponse({'status': 'ok'})
```

Обработчик `user_follow` очень похож на `image_like`, который мы создали ранее. Так как мы используем промежуточную модель для связи пользователя

и его подписчиков, стандартные методы `add()` и `remove()` менеджера недоступны. Поэтому мы напрямую создаем и удаляем, используя модель `Contact`.

Добавьте URL-шаблон для этого обработчика в файл `urls.py` приложения `account`:

```
path('users/follow/', views.user_follow, name='user_follow'),
```

Убедитесь, что этот шаблон будет находиться перед шаблоном `user_detail`. В противном случае запрос по адресу `/users/follow/` подойдет к регулярному отображению шаблона `user_detail`, при этом будет вызван не тот обработчик, который мы ожидаем. Помните, что Django проверяет каждый URL-шаблон по порядку, пока не найдет первый подходящий.

Добавьте следующий код в шаблон `user/detail.html` приложения `account`:

```
{% block domready %}
$( 'a.follow' ).click( function( e ) {
    e.preventDefault();
    $.post( '{% url "user_follow" %}',
        {
            id: $( this ).data( 'id' ),
            action: $( this ).data( 'action' )
        },
        function( data ) {
            if ( data[ 'status' ] == 'ok' ) {
                var previous_action = $( 'a.follow' ).data( 'action' );

                // Изменяем действие на противоположное.
                $( 'a.follow' ).data( 'action',
                    previous_action == 'follow' ? 'unfollow' : 'follow' );

                // Изменяем текст ссылки.
                $( 'a.follow' ).text( previous_action == 'follow' ? 'Unfollow' : 'Follow' );

                // Обновляем количество подписчиков.
                var previous_followers = parseInt(
                    $( 'span.count .total' ).text() );
                $( 'span.count .total' ).text(
                    previous_action == 'follow' ? previous_followers + 1 :
                    previous_followers - 1
                );
            }
        }
    );
});
{% endblock %}
```

Этот JavaScript-код выполняет AJAX-запрос, чтобы добавить в подписчики или удалить из них текущего пользователя и изменить текст ссылки соответствующим образом. Мы используем jQuery для выполнения запроса и выставляем атрибут `data-action` и текст `<a>`-элемента, учитывая предыдущие значения. При успешном выполнении запроса обновляем количество подписчиков. Откройте профиль любого существующего пользователя и кликните на ссылку **FOLLOW**, чтобы протестировать функционал, который мы только что добавили.



Рис. 6.3 ❖ Количество подписчиков пользователя

ДОБАВЛЕНИЕ НОВОСТНОЙ ЛЕНТЫ

Многие социальные сети отображают ленту активности пользователей, так чтобы они могли наблюдать за действиями друг друга. Такая лента представляет собой список событий пользователя или некоторой группы (например, новостная лента Facebook). Простые примеры активности: «пользователь X добавил в закладки картинку Y» или «пользователь X подписался на обновления пользователя Y». Мы добавим ленту новостей. Так каждый подписчик сможет видеть изменения, сделанные пользователями, на которых он подписан. Чтобы это реализовать, нам нужна модель для сохранения сведений об активности. Давайте опишем ее.

Создайте приложение `actions` внутри папки проекта с помощью команды:

```
python manage.py startapp actions
```

Теперь подключите приложение к проекту, добавив его в список `INSTALLED_APPS` файла `settings.py`:

```
INSTALLED_APPS = [
    # ...
    'actions.actions.ActionsConfig',
]
```

Отредактируйте файл `models.py` приложения `actions` и добавьте новую модель:

```
from django.db import models

class Action(models.Model):
    user = models.ForeignKey('auth.User', related_name='actions',
                             db_index=True, on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    created = models.DateTimeField(auto_now_add=True, db_index=True)

    class Meta:
        ordering = ('-created',)
```

В этом фрагменте мы определяем модель `Action` для хранения информации о действиях пользователя. Она содержит следующие поля:

- `user` – пользователь, который выполнил действие. Внешний ключ `ForeignKey` на стандартную модель Django – `User`;
- `verb` – информация о том, какое действие было выполнено;
- `created` – дата и время, показывающие, когда был создан объект. Мы используем `auto_now_add=True`, чтобы автоматически выставлять текущее время, когда объект сохраняется в базу данных.

С помощью такой модели мы можем сохранять простые действия, такие как «пользователь X сделал что-то». Для того чтобы указывать объект, над которым

было произведено действие (например, «пользователь X добавил в закладки картинку Y» или «пользователь X подписался на обновления пользователя Y»), необходимо добавить поле `target`. Однако, как мы уже знаем, `ForeignKey` может ссылаться только на одну модель, а в нашем случае нужна возможность указывать объект любой модели. Тут на помощь приходит *подсистема типов содержимого Django*.

Использование подсистемы типов содержимого

В Django добавлено приложение, расположенное в `django.contrib.contenttypes`, которое знает про все модели установленных приложений нашего проекта и предоставляет обобщенный интерфейс для обращения к ним.

Приложение `django.contrib.contenttypes` автоматически добавляется в список `INSTALLED_APPS`, если вы создаете проект командой `startproject`. Оно используется другими приложениями из пакета `contrib`, например подсистемой аутентификации и сайтом администрирования.

В приложении `contenttypes` определена модель `ContentType`. Ее объекты содержат сведения о реальных моделях проекта и автоматически создаются при добавлении новых. `ContentType` содержит следующие поля:

- `app_label` – имя приложения системы, к которому относится описываемая модель. Значение автоматически подставляется из атрибута `app_label` опций `Meta` модели. Например, модель `Image` принадлежит приложению `images`;
- `model` – название класса модели;
- `name` – понятное человеку название модели. Значение подставляется из атрибута `verbose_name` опций `Meta` модели.

Давайте посмотрим, как мы можем работать с объектами `ContentType`. Откройте консоль командой `python manage.py shell`. Мы можем получить объект `ContentType`, соответствующий определенной модели, с помощью запроса, содержащего фильтрацию по `app_label` и `model`:

```
>>> from django.contrib.contenttypes.models import ContentType
>>> image_type = ContentType.objects.get(
app_label='images', model='image')
>>> image_type
<ContentType: image>
```

Чтобы получить класс отслеживаемой модели, можно вызвать метод `model_class()`:

```
>>> image_type.model_class()
<class 'images.models.Image'>
```

Также часто необходимо получить объект `ContentType` для определенной модели. Это можно сделать следующим образом:

```
>>> from images.models import Image
>>> ContentType.objects.get_for_model(Image)
<ContentType: image>
```

Это всего несколько примеров использования типов содержимого. Django предоставляет и другие. Подробное описание можно найти на странице: <https://docs.djangoproject.com/en/2.0/ref/contrib/contenttypes/>.

Добавление обобщенных отношений

В *обобщенных связях* ContentType играет роль посредника между моделями. Для создания такой связи нам необходимо добавить в модель три поля:

- поле ForeignKey на модель ContentType. Оно будет указывать на модель, с которой связана текущая;
- поле для хранения ID связанного объекта. Обычно определяется как PositiveIntegerField для идентификаторов, автоматически созданных Django;
- поле для определения связи и управления ей. Обращается к предыдущим двум полям и является объектом типа GenericForeignKey.

Давайте добавим их в модель действия пользователя. Отредактируйте models.py приложения actions, чтобы получить следующую картину:

```
from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey

class Action(models.Model):
    user = models.ForeignKey('auth.User', related_name='actions',
                            db_index=True, on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    target_ct = models.ForeignKey(ContentType, blank=True, null=True,
                                  related_name='target_obj', on_delete=models.CASCADE)
    target_id = models.PositiveIntegerField(null=True, blank=True, db_index=True)
    target = GenericForeignKey('target_ct', 'target_id')
    created = models.DateTimeField(auto_now_add=True, db_index=True)

    class Meta:
        ordering = ('-created',)
```

Мы добавили в модель Action эти поля:

- target_ct – внешний ключ на модель ContentType;
- target_id – PositiveIntegerField для хранения идентификатора на связанный объект;
- target – GenericForeignKey – поле для обращения к связанному объекту на основании его типа и ID.

Django не создает столбец на уровне базы данных для полей GenericForeignKey. Вместо этого сохраняются значения полей target_ct и target_id. Оба определены параметрами blank=True и null=True, поэтому наличие связанного объекта вовсе не обязательно при сохранении объекта Action.



Вы можете сделать свое приложение более гибким с помощью обобщенных отношений, не используя при этом внешние ключи на определенные модели, когда в этом есть необходимость.

Выполните команду создания начальной миграции для приложения actions:

```
python manage.py makemigrations actions
```

Вы увидите такой вывод:

```
Migrations for 'actions':
  actions/migrations/0001_initial.py
    - Create model Action
```

Затем выполните синхронизацию моделей и базы данных с помощью команды:

```
python manage.py migrate
```

В выводе консоли вы увидите, что созданная миграция успешно применилась:

```
Applying actions.0001_initial... OK
```

Давайте добавим модель Action на сайт администрирования. Отредактируйте файл admin.py приложения actions и вставьте следующий фрагмент:

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

@admin.register(Action)
class ActionAdmin(admin.ModelAdmin):
    list_display = ('user', 'verb', 'target', 'created')
    list_filter = ('created',)
    search_fields = ('verb',)
```

Запустите сервер разработки командой python manage.py runserver и перейдите на <http://127.0.0.1:8000/admin/actions/action/add/>. Вы увидите подобную страницу:

The screenshot shows the Django administration interface for adding a new action. The page title is "Django administration" and the user is logged in as "ANTONIO". The breadcrumb trail is "Home > Actions > Actions > Add action". The form is titled "Add action" and contains the following fields:

- User:** A dropdown menu with a search icon and a plus sign.
- Target ct:** A dropdown menu.
- Target id:** A text input field with a search icon.
- Verb:** A text input field.

At the bottom of the form, there are three buttons: "Save and add another", "Save and continue editing", and "SAVE".

Рис. 6.4 ❖ Страница добавления действия на сайте администрирования

Как можно заметить на скриншоте, из трех добавленных полей отображаются только два: `target_ct` и `tager_id`. Именно они сохраняются в базе данных в соответствующих столбцах таблицы. Поле `target_ct` позволяет нам выбрать любую модель проекта. Однако можно добавить ограничение на доступные модели с помощью атрибута `limit_choices_to` поля `target_ct`.

Создайте новый файл `utils.py` в каталоге проекта `actions`. Мы включим функцию, необходимую для большего удобства при создании объекта `Action`. Добавьте в этот файл такой фрагмент:

```
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    action = Action(user=user, verb=verb, target=target)
    action.save()
```

Функция `create_action()` позволяет создать активность и связать ее с некоторым объектом `target`, если он будет передан в качестве аргумента. Мы будем использовать данную функцию для большего удобства при создании записи об активности пользователя.

Устранение дублирования новостей в ленте

Иногда пользователи могут выполнять одно и то же действие много раз. Например, они могут кликать кнопки **LIKE** и **UNLIKE** несколько раз подряд, что приведет к дублированию записей в новостной ленте их подписчиков. Во избежание такого поведения следует немного улучшить функцию `create_action()`.

Отредактируйте файл `utils.py` приложения `actions` следующим образом:

```
import datetime
from django.utils import timezone
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    # Поиск похожего действия, совершенного за последнюю минуту.
    now = timezone.now()
    last_minute = now - datetime.timedelta(seconds=60)
    similar_actions = Action.objects.filter(user_id=user.id, verb= verb,
                                           created__gte=last_minute)

    if target:
        target_ct = ContentType.objects.get_for_model(target)
        similar_actions = similar_actions.filter(target_ct=target_ct,
                                                target_id=target.id)

    if not similar_actions:
        # Похожее действие не найдено, создаем новое.
        action = Action(user=user, verb=verb, target=target)
        action.save()
        return True
    return False
```


Мы добавили в `create_action()` проверку на наличие аналогичной активности пользователя. Теперь функцию возвращает `True` или `False`. Это зависит от того, было ли сохранено событие. Вот как мы проверяем наличие дубликата активности:

- сначала получаем текущую дату и время, используя средства Django – `timezone.now()`. Этот метод работает так же, как и `datetime.datetime.now()`, за исключением того, что возвращаемый объект учитывает часовой пояс. Django использует настройку `USE_TZ`, чтобы управлять поддержкой временных зон. По умолчанию для проектов, созданных командой `startproject` в `settings.py`, добавлено `USE_TZ=True`;
- используем переменную `last_minute` для момента времени, опережающего текущее на минуту, и пытаемся получить активность пользователя, созданную за последнюю минуту;
- если аналогичных активностей за последнюю минуту не найдено, создаем новый объект `Action` и возвращаем `True`. В противном случае мы ничего не создаем и возвращаем `False`.

Добавление активности в новостную ленту

Самое время добавить активности в обработчик, чтобы сформировать новостную ленту для подписчиков. Мы будем создавать объекты `Action` для таких событий:

- пользователь добавил картинку в закладки;
- пользователь лайкнул изображение;
- пользователь создал аккаунт;
- пользователь подписался на обновления.

Отредактируйте файл `views.py` приложения `images` и импортируйте функцию:

```
from actions.utils import create_action
```

В обработчике `image_create` добавьте вызов `create_action()` после сохранения изображения:

```
new_item.save()
create_action(request.user, 'bookmarked image', new_item)
```

В обработчике `image_like` добавьте ее после того, как создается связь `users_like`:

```
image.users_like.add(request.user)
create_action(request.user, 'likes', image)
```

Теперь импортируйте функцию в файл `views.py` приложения `account`:

```
from actions.utils import create_action
```

Добавьте вызов функции `create_action()` после создания объекта `Profile` в обработчике `register`:

```
Profile.objects.create(user=new_user)
create_action(new_user, 'has created an account')
```

В обработчике `user_follow` также добавьте вызов функции:

```
Contact.objects.get_or_create(user_from=request.user, user_to=user)
create_action(request.user, 'is following', user)
```

Как вы могли заметить, во всех предыдущих фрагментах мы легко добавляли создание объектов `Action` с помощью используемой функции `create_action()`.

Отображение ленты новостей

Наконец, пришло время отобразить новостную ленту. Мы будем показывать ее на рабочем столе пользователя. Отредактируйте файл `views.py` приложения `account`. Для этого импортируйте модель `Action` и добавьте в обработчик рабочего стола фрагмент:

```
from actions.models import Action

@login_required
def dashboard(request):
    # По умолчанию отображаем все действия.
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id', flat=True)
    if following_ids:
        # Если текущий пользователь подписался на кого-то,
        # отображаем только действия этих пользователей.
        actions = actions.filter(user_id__in=following_ids)
    actions = actions[:10]

    return render(request, 'account/dashboard.html',
                  {'section': 'dashboard', 'actions': actions})
```

В этом обработчике мы получаем все события, кроме тех, которые выполнил текущий пользователь. По умолчанию отображаем последние события всех пользователей системы. Если пользователь подписался на кого-либо, мы начинаем показывать только действия интересных ему пользователей. Вдобавок мы ограничиваем количество записей и отображаем последние 10. Мы не используем `order_by()` `QuerySet`'а, потому что записи и так будут отсортированы по порядку, указанному в опциях модели `Action` – `ordering = ('-created',)`.

Оптимизация `QuerySet`'а со связанными объектами

Каждый раз, когда мы получаем объект `Action`, нам нужен доступ к связанному пользователю `User` и его профилю `Profile`. Django ORM предоставляет простой способ получать связанные объекты во время выполнения основного запроса, чтобы уменьшить количество дополнительных.

Использование `select_related()`

Django определяет для `QuerySet`'ов метод `select_related()`, который дает возможность получить объекты, связанные отношением «один ко многим». Запрос получится чуть более сложным, но позволит избежать многократного обращения к базе данных для доступа к связанным объектам. Метод `select_re-`

lated() подходит для полей ForeignKey и OneToOne. Он добавит в SQL-запрос инструкцию JOIN и включит поля связанного объекта в инструкцию SELECT.

Для того чтобы воспользоваться select_related(), отредактируйте строку:

```
actions = actions[:10]
```

И добавьте вызов этого метода:

```
actions = actions.select_related('user', 'user__profile')[:10]
```

Мы используем user__profile, чтобы получить данные модели Profile в том же самом запросе. Если бы мы не передали аргументы в select_related(), Django обратился бы к связанным объектам для всех полей ForeignKey. Всегда ограничивайте список необходимых вам связей при использовании select_related().

i Продуманное использование select_related() может значительно сократить время выполнения обработчиков.

Использование prefetch_related()

Метод select_related() помогает нам оптимизировать доступ к объектам, связанным отношением «один ко многим». Но он не работает для отношений «многие ко многим» и «многие к одному» (ManyToMany и обратная связь для ForeignKey). Для этого случая Django предоставляет метод prefetch_related(), который работает аналогично select_related(), но может быть применен и к упомянутым связям. В отличие от select_related(), где поиск связей происходит в базе данных, этот метод связывает объекты уже на уровне Python. Используя prefetch_related(), мы можем обращаться и к полям типов GenericRelation и GenericForeignKey.

Отредактируйте views.py приложения account и добавьте метод prefetch_related():

```
actions = actions.select_related('user', 'user__profile')\
    .prefetch_related('target')[:10]
```

Теперь этот запрос будет работать быстрее, т. к. связанные с активностями объекты будут уже получены из базы данных и не потребуют излишних запросов.

Создание шаблонов для новостной ленты

Теперь давайте создадим шаблон, в котором будем показывать подробности конкретной активности. Создайте новую папку templates внутри каталога приложения actions с такой структурой:

```
actions/
  action/
    detail.html
```

Откройте файл actions/action/detail.html и добавьте в него фрагмент кода:

```
{% load thumbnail %}
{% with user=action.user profile=action.user.profile %}
<div class="action">
```

```

<div class="images">
  {% if profile.photo %}
    {% thumbnail user.profile.photo "80x80" crop="100%" as im %}
      <a href="{{ user.get_absolute_url }}">
        
      </a>
    {% endthumbnail %}
  {% endif %}
  {% if action.target %}
    {% with target=action.target %}
      {% if target.image %}
        {% thumbnail target.image "80x80" crop="100%" as im %}
          <a href="{{ target.get_absolute_url }}">
            
          </a>
        {% endthumbnail %}
      {% endif %}
    {% endwith %}
  {% endif %}
</div>
<div class="info">
  <p>
    <span class="date">{{ action.created|timesince }} ago</span><br />
    <a href="{{ user.get_absolute_url }}">{{ user.first_name }}</a>
    {{ action.verb }}
    {% if action.target %}
      {% with target=action.target %}
        <a href="{{ target.get_absolute_url }}">{{ target }}</a>
      {% endwith %}
    {% endif %}
  </p>
</div>
</div>
{% endwith %}

```

В этом шаблоне мы показываем объект `Action`. Для начала определяем блок `{% with %}`, чтобы получить пользователя, совершившего действие, и его профиль. Затем показываем картинку объекта `target`, если у активности есть связанный объект. Наконец, добавляем ссылку на пользователя, который выполнил это действие, тип активности и связанный объект, если он существует.

Теперь давайте отобразим новостную ленту на рабочем столе. Добавьте в конец блока `content` файла `account/dashboard.html` приложения `account` такой фрагмент:

```

<h2>What's happening</h2>
<div id="action-list">
  {% for action in actions %}
    {% include "actions/action/detail.html" %}
  {% endfor %}
</div>

```

Откройте в браузере `http://127.0.0.1:8000/account/`. Зайдите в аккаунт любого пользователя и выполните несколько действий от его имени. Затем войдите под другим пользователем, подпишитесь на первого, и вы увидите, что на рабочем столе появились действия, которые вы выполнили ранее:

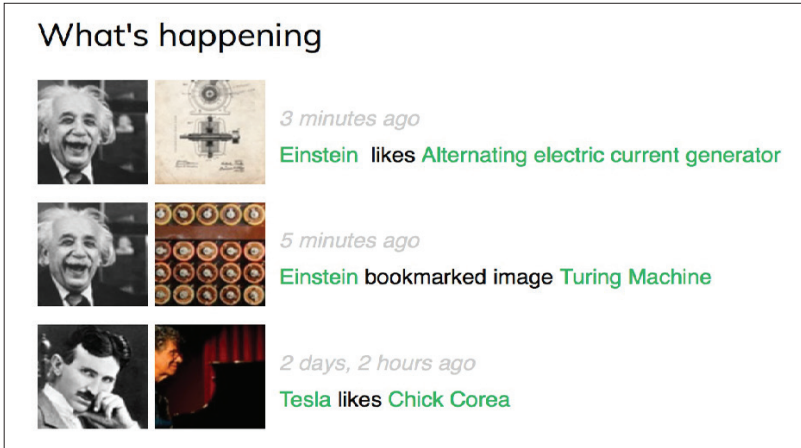


Рис. 6.5 ❖ Новостная лента

Только что мы создали полноценную систему подписок, в которую можем легко добавлять новые типы действий. В качестве тренировки вы можете добавить постраничное отображение новостей с помощью AJAX, как мы уже делали это в обработчике `image_list`.

ИСПОЛЬЗОВАНИЕ СИГНАЛОВ DJANGO

В некоторых случаях нам может понадобиться денормализация данных. *Денормализация* – это процесс, когда мы разделяем данные на несколько логически связанных блоков, так чтобы обращение к ним стало более эффективным с точки зрения доступа к базе данных. Этот прием нужно применять только тогда, когда вы уверены в его необходимости. Одно из неприятных последствий денормализации – увеличение сложности обновления данных и поддержания их целостности.

Давайте посмотрим пример того, как ускорить выполнение запросов к модели изображений с помощью денормализации. При этом нам нужно будет иначе обновлять разделенные данные. Мы денормализуем модель `Image` и применим сигналы Django для поддержания связанных таблиц в соответствии.

Работа с сигналами

В Django встроен диспетчер сигналов, который позволяет функции-подписчику `receiver` быть уведомленной о том, что произошло определенное действие.

Сигналы очень полезны, когда нам нужно выполнить определенную обработку при наступлении определенного события. Кроме того, мы можем создать свои сигналы, так чтобы другие части системы получали уведомление о совершенных нашим кодом действиях.

Django предоставляет несколько *сигналов для моделей*. Они расположены в `django.db.models.signals`. Вот некоторые из них:

- `pre_save` и `post_save` – отправляются перед и после сохранения объекта модели методом `save()`;
- `pre_delete` и `post_delete` – отправляются перед и после удаления объекта модели или объектов `QuerySet`'а с помощью метода `delete()`;
- `m2m_changed` – отправляется при изменении полей `ManyToManyField`.

Это только небольшая часть всех сигналов Django. Полный список и подробное описание можно найти на <https://docs.djangoproject.com/en/2.0/ref/signals/>.

Допустим, мы хотим получить список картинок по популярности. Для этого можно использовать функцию агрегации и отсортировать изображения по количеству лайков. Ранее мы использовали агрегацию в главе 3. Этот фрагмент кода выполнит все необходимые действия:

```
from django.db.models import Count
from images.models import Image

images_by_popularity = Image.objects.annotate(
    total_likes=Count('users_like')).order_by('-total_likes')
```

При этом сортировка картинок по вычисленным значениям `likes` – более дорогая с точки зрения вычислений операция, чем сортировка их по хранящемуся полю. Мы можем добавить поле в модель `Image`, чтобы сохранять информацию о текущем количестве лайков для каждого изображения. Так мы ускорим запросы, которые обращаются к этому значению.

Откройте файл `models.py` приложения `images` и добавьте новое поле `total_likes`:

```
class Image(models.Model):
    # ...
    total_likes = models.PositiveIntegerField(db_index=True, default=0)
```

Поле `total_likes` будет хранить количество лайков для картинки. Выделение некоего количества в отдельное поле бывает полезным, когда вы хотите фильтровать или сортировать `QuerySet` по нему.

Существует несколько других способов улучшения производительности запросов, о которых стоит задуматься, до того как применять денормализацию. Вы можете рассмотреть использование индексов базы данных, оптимизацию запросов, кеширование, а только потом – денормализацию.

Выполните команду для создания миграции с новым полем в модель:

```
python manage.py makemigrations images
```

Вы увидите такой вывод в консоли:

```
Migrations for 'images':
  images/migrations/0002_image_total_likes.py
    - Add field total_likes to image
```

Затем выполните синхронизацию моделей и базы данных командой:

```
python manage.py migrate images
```

В результате вы увидите, что миграция успешно применилась:

```
Applying images.0002_image_total_likes... OK
```

Возникает вопрос: как поддерживать значение этого поля в актуальном состоянии? Для его решения мы добавим функцию-подписчик `receiver` на сигнал `m2m_changed`. Создайте новый файл `signals.py` в каталоге приложения `images` и добавьте в него такой код:

```
from django.db.models.signals import m2m_changed
from django.dispatch import receiver
from .models import Image

@receiver(m2m_changed, sender=Image.users_like.through)
def users_like_changed(sender, instance, **kwargs):
    instance.total_likes = instance.users_like.count()
    instance.save()
```



Мы регистрируем функцию `users_like_changed` как функцию-подписчик с помощью декоратора `receiver` и подписываемся на сигнал `m2m_changed`, отправляемый связями `Image.users_like.through`. Использование декоратора – один из способов добавления функции-подписчика. Кроме этого, можно использовать метод `connect()` объекта `Signal`.

Сигналы Django выполняются синхронно и блокируются, поэтому не нужно обращаться к ним в асинхронных задачах. При этом возможность запускать такие задачи из сигналов остается доступной.

Теперь мы должны связать сигнал и функцию-обработчик, которая будет вызываться каждый раз при срабатывании сигнала. Рекомендуемый способ зарегистрировать сигналы – импортировать их в методе `ready()` конфигурационного класса приложения. Django предоставляет реестр приложений, с помощью которого мы можем настраивать и управлять ими.

Конфигурационные классы приложений

Django позволяет нам определить класс для конфигурации конкретного приложения. Когда вы создаете новое с помощью команды `startapp`, Django добавляет файл `apps.py` в каталоге приложения. В этом файле описан базовый класс конфигурации, который унаследован от `AppConfig`.

Эти классы позволяют нам хранить метаданные приложения и предоставляют интроспекцию. Более подробную информацию о конфигурации приложений вы можете найти на странице: <https://docs.djangoproject.com/en/2.0/ref/applications/>.

Для того чтобы зарегистрировать функции – обработчики сигналов, достаточно импортировать их в методе `ready()` конфигурационного класса. Этот метод вызывается сразу после того, как будет полностью заполнен реестр приложений. Любая логика, связанная с инициализацией нашего приложения, должна быть объяснена в этом методе.

Давайте зарегистрируем нашу функцию-подписчик, чтобы Django вызывал ее при вызове сигнала `m2m_changed`. Отредактируйте файл `apps.py` приложения `images` и добавьте в него такой фрагмент:

```
from django.apps import AppConfig

class ImagesConfig(AppConfig):
    name = 'images'

    def ready(self):
        # Импортируем файл с описанной функцией-подписчиком на сигнал.
        import images.signals
```

Мы импортировали модуль сигналов внутри метода `ready()`. Таким образом, они будут доступны сразу после загрузки приложения `images`.

Выполните команду запуска сервера разработки:

```
python manage.py runserver
```

Откройте браузер на странице изображения и нажмите кнопку **LIKE**. Вернитесь на сайт администрирования и перейдите по адресу `http://127.0.0.1:8000/admin/images/image/1/change/`. Обратите внимание на поле `total_likes`. Вы увидите, что количество лайков увеличилось:

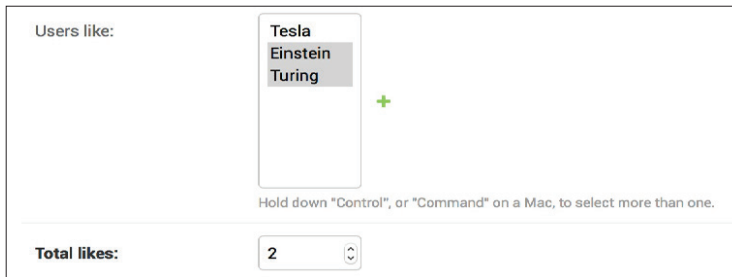


Рис. 6.6 ❖ Отображение поля количества лайков картинки на сайте администрирования

Теперь мы можем использовать поле `total_likes`, чтобы отсортировать изображения по популярности или показать рейтинг картинки и при этом избежать сложного запроса в базу данных. Запись с использованием функции агрегации:

```
from django.db.models import Count
images_by_popularity = Image.objects.annotate(
    likes=Count('users_like')).order_by('-likes')
```


Она преобразуется в более короткое и эффективное с точки зрения вычислений выражение:

```
images_by_popularity = Image.objects.order_by('-total_likes')
```

В результате вы получаете менее «дорогой» SQL-запрос. Это один из примеров того, как можно использовать сигналы Django.

i Применяйте сигналы с осторожностью, т. к. их использование делает порядок выполнения кода менее очевидным. В большинстве случаев сигналы вам не нужны, если вы точно знаете, какую функцию необходимо вызвать.

Следует выставить начальный рейтинг для уже созданных картинок. Запустите консоль командой `python manage.py shell` и выполните код:

```
from images.models import Image
for image in Image.objects.all():
    image.total_likes = image.users_like.count()
    image.save()
```

Теперь количество лайков для каждого сохраненного изображения актуально.

ИСПОЛЬЗОВАНИЕ REDIS ДЛЯ ХРАНЕНИЯ ПРЕДСТАВЛЕНИЙ ОБЪЕКТОВ

Redis – это хранилище данных, которое работает по принципу «ключ – значение» и оптимизировано для быстрого ввода-вывода. Оно позволяет использовать различные структуры данных. Redis хранит все в оперативной памяти, но мы можем настроить копирование блоков данных на диски с определенной периодичностью или при наступлении некоторого действия. Redis очень хорошо расширяется. Если сравнивать его с другими хранилищами «ключ – значение», то можно увидеть, что он предоставляет множество команд управления и поддерживает большой набор типов данных: строки, хеши, списки, кортежи, сортированные кортежи и даже битовые карты или HyperLogLogs.

Хотя SQL прекрасно подходит для долговременного хранения относительно постоянных данных, когда речь заходит о часто изменяющейся информации или той, к которой необходимо иметь быстрый доступ (например, к кешу), Redis будет более эффективным решением. Давайте посмотрим, как можно применить Redis в нашем проекте и дополнить его новой функциональностью.

Установка Redis

Скачайте последнюю версию Redis с сайта <https://redis.io/download>, распакуйте `tar.gz` архив и выполните команду из папки `redis`:

```
cd redis-4.0.9
make
```

После установки запустите сервер Redis командой `src/redis-server`:

```
src/redis-server
```

Вы должны будете увидеть в консоли такие строки:

```
# Server initialized
* Ready to accept connections
```

Это значит, что Redis запущен и готов принимать запросы. По умолчанию Redis запускается на порте 6379. Но вы можете указать другое значение с помощью флага `-port`, например `redis-server -port 6655`.

Откройте еще одно окно консоли и запустите клиента Redis:

```
src/redis-cli
```

Вы увидите, что в консоли появится приглашение на ввод команд:

```
127.0.0.1:6379>
```

Клиент Redis позволяет выполнять команды напрямую через консоль. Давайте попробуем несколько из них. Выполните команду `SET`, чтобы добавить в хранилище значение

```
127.0.0.1:6379> SET name "Peter"
OK
```

Только что введенная команда создает ключ `name` и сохраняет значение `Peter` в хранилище Redis. Сообщение `OK` говорит о том, что все прошло успешно. Для того чтобы получить значение, которое хранится по некоторому ключу, выполните команду `GET`, например:

```
127.0.0.1:6379> GET name
"Peter"
```

Кроме этого, вы можете проверить, существует ли некоторый ключ, с помощью команды `EXISTS`. Она возвращает `1`, если ключ сохранен в хранилище, и `0` в противном случае:

```
127.0.0.1:6379> EXISTS name
(integer) 1
```

Redis позволяет указать срок хранения для любого ключа в секундах с помощью команды `EXPIRE`. В том случае, если вы хотите указать конкретное время, когда ключ должен быть удален, подойдет команда `EXPIREAT`, которая принимает дату в Unix-формате. Ограничение времени полезно, когда Redis используется в качестве кеша или для временных данных. Вот пример использования команд:

```
127.0.0.1:6379> GET name
"Peter"
127.0.0.1:6379>
EXPIRE name 2 (integer) 1
```

Подождите 2 с и попытайтесь получить значение по ключу name:

```
127.0.0.1:6379> GET name
(nil)
```

Ответ (nil) аналогичен нулевому результату и говорит о том, что по указанному ключу данных нет. Еще одна полезная команда – DEL. Она используется для удаления ключа из хранилища:

```
127.0.0.1:6379> SET total 1
OK
127.0.0.1:6379> DEL total (integer) 1
127.0.0.1:6379> GET total
(nil)
```

Redis поддерживает большое количество команд для различных типов данных, таких как хеши, наборы, строки и др. Описание с примерами их использования можно найти на странице <https://redis.io/commands>, а список всех поддерживаемых типов данных – на <https://redis.io/topics/data-types>.

Использование Redis в Python-коде

Чтобы работать с хранилищем из Python-кода, необходимо установить пакет redis-ру с помощью pip:

```
pip install redis==2.10.6
```

Документацию по этому приложению можно найти на странице <https://redis-py.readthedocs.io/>.

Пакет redis-ру предоставляет два класса для доступа к Redis: StrictRedis и Redis. Оба реализуют один и тот же функционал, т. е. взаимодействуют с хранилищем. Отличие в том, что Redis является расширением класса StrictRedis и переопределяет некоторые методы для поддержания обратной совместимости с более ранними версиями. Мы будем использовать StrictRedis, т. к. работа с ним очень похожа на вызов команд Redis. Откройте Python-консоль и выполните следующую команду:

```
>>> import redis
>>> r = redis.StrictRedis(host='localhost', port=6379, db=0)
```

Этот код создает соединение с базой. В Redis базы данных идентифицируются по уникальному индексу вместо имени, как это принято в реляционных СУБД. По умолчанию клиент подключается к базе с идентификатором 0. Количество доступных баз данных равно 16, но это можно изменить в конфигурационном файле Redis, redis.conf.

Теперь давайте сохраним значение из Python-консоли:

```
>>> r.set('foo', 'bar')
True
```

Метод возвращает True. Это значит, что ключ был успешно сохранен. Теперь мы можем обратиться к нему, вызвав соответствующий метод:

```
>>> r.get('foo')
b'bar'
```

Как вы можете заметить, методы класса `StrictRedis` аналогичны командам клиента Redis.

Давайте настроим наш Django-проект на работу с Redis. Откройте файл `settings.py` проекта `bookmarks` и добавьте следующие строки:

```
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 0
```

Это настройки подключения к хранилищу Redis, которое мы будем использовать.

Сохранение представлений объектов в Redis

Давайте попробуем сохранять информацию о количестве просмотров картинок. Если для реализации этой функциональности мы будем использовать ORM Django, то каждый раз при открытии страницы картинки будет происходить запрос `UPDATE` в базу данных. Если же мы будем сохранять количество просмотров в Redis, то при каждом отображении страницы картинки будет происходить инкремент числа, хранящегося в оперативной памяти, что существенно быстрее доступа в базу данных.

Отредактируйте файл `views.py` приложения `images` и добавьте после последнего импорта следующий фрагмент:

```
import redis
from django.conf import settings

# Подключение к Redis.
r = redis.StrictRedis(host=settings.REDIS_HOST,
                     port=settings.REDIS_PORT,
                     db=settings.REDIS_DB)
```

В этом коде мы создаем постоянное соединение с Redis, чтобы использовать его в обработчиках, а не открывать каждый раз. Теперь отредактируйте функцию `image_detail` следующим образом:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # Увеличиваем количество просмотров картинки на 1.
    total_views = r.incr('image:{}'.format(image.id))
    return render(request, 'images/image/detail.html',
                  {'section': 'images', 'image': image, 'total_views': total_views})
```

Мы добавили увеличение количества просмотров на 1 с помощью метода `incr`. Если ключ еще не существует, то он будет создан, а только после этого добавится единица. Метод `incr()` возвращает итоговое значение, которое мы сохраняем в переменную `total_views` и передаем в контекст шаблона. Формируем ключ для хранилища в виде `object-type:id:field`, например `image:33:id`.

Таким образом мы обезопасим себя от дублирования ключей и непредвиденного поведения.

i При работе с Redis есть соглашение о наименованиях ключей: рекомендуется использовать двоеточие как разделитель между частями ключа. Такие названия позволяют понять, какие данные хранятся по определенному ключу, только при условии, что мы просто взглянем на код.

Отредактируйте шаблон `images/image/detail.html` приложения `images` и добавьте следующие строки после существующего элемента ``:

```
<span class="count">
  {{ total_views }} view{{ total_views|pluralize }}
</span>
```

Теперь откройте страницу любой картинки и перезагрузите ее несколько раз. Вы увидите, что счетчик просмотров увеличивается при каждой загрузке:

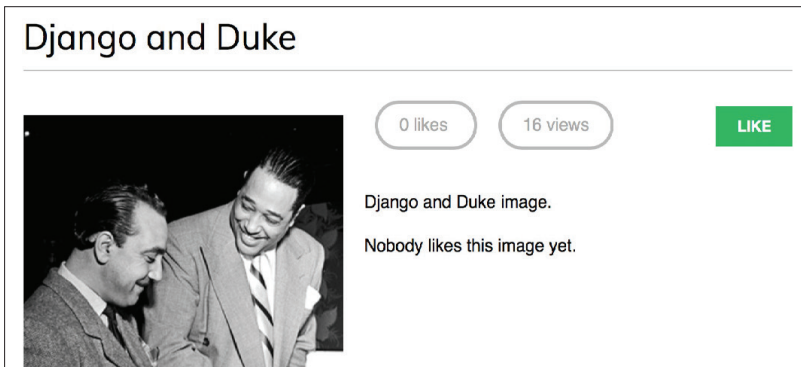


Рис. 6.7 ❖ Счетчик просмотров на странице картинки

Поздравляем! Вы только что подключили Redis к проекту и успешно применили его.

Хранение рейтинга объектов в Redis

Давайте добавим что-то более существенное, например создадим список наиболее просматриваемых картинок. Для реализации этой функциональности можно использовать сортированные наборы Redis. *Набор* (*set*) – это коллекция неповторяющихся строк, каждая из которых имеет рейтинг. Именно по нему мы будем сортировать элементы этой коллекции.

Отредактируйте файл `views.py` приложения `images`, допишите несколько строк в обработчик `image_detail`, как показано ниже:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
```

```
# Увеличиваем общее количество просмотров на 1.
total_views = r.incr('image: {}:views'.format(image.id))
# Увеличиваем рейтинг картинки на 1.
r.zincrby('image_ranking', image.id, 1)
return render(request,
               'images/image/detail.html',
               {'section': 'images',
                'image': image,
                'total_views': total_views})
```

Мы используем метод `zincrby()`, чтобы работать с сортированным набором данных о количестве просмотров каждой картинки. Сохраняем просмотры по ключу вида `image_ranking` и идентификатору картинки `id`. Таким образом, мы будем иметь актуальные сведения о том, сколько раз каждое изображение было просмотрено пользователями сайта.

Теперь давайте создадим обработчик, который добавит на сайт блок с самыми просматриваемыми картинками. Добавьте его код в файл `views.py` приложения `images`:

```
@login_required
def image_ranking(request):
    # Получаем набор рейтинга картинок.
    image_ranking = r.zrange('image_ranking', 0, -1, desc=True)[:10]
    image_ranking_ids = [int(id) for id in image_ranking]
    # Получаем отсортированный список самых популярных картинок.
    most_viewed = list(Image.objects.filter(id__in=image_ranking_ids))
    most_viewed.sort(key=lambda x: image_ranking_ids.index(x.id))
    return render(request,
                  'images/image/ranking.html',
                  {'section': 'images',
                   'most_viewed': most_viewed})
```

В этом обработчике мы выполняем такие действия:

- 1) используем метод `zrange()` для доступа к нескольким элементам сортированного списка. В качестве аргументов можно передать начальный и конечный индексы необходимых элементов. Указав `0` начальным и `1` конечным, мы получим все элементы из хранилища. Также есть возможность задать сортировку в убывающем порядке, передав аргумент `desc=True`. После получения результата из хранилища Redis ограничиваем количество объектов до `10 - [:10]`;
- 2) сохраняем идентификаторы нужных картинок в списке `image_ranking_ids`. Затем обращаемся к модели `Image` и получаем соответствующие этой переменной объекты картинок. При этом мы передаем `QuerySet` изображений в функцию `list()`, чтобы выполнить сортировку методом `sort()`;
- 3) сортируем картинки по их идентификаторам и порядку в списке, полученном из Redis. Теперь самые просматриваемые изображения расположены в нужном порядке, и мы можем показать их в шаблоне.

Создайте файл `ranking.html` внутри директории шаблонов `images/image` приложения `images`. Вставьте в него такой код:

```
{% extends "base.html" %}
{% block title %}Images ranking{% endblock %}
{% block content %}
<h1>Images ranking</h1>
<ol>
  {% for image in most_viewed %}
    <li>
      <a href="{{ image.get_absolute_url }}">{{ image.title }}</a>
    </li>
  {% endfor %}
</ol>
{% endblock %}
```

Код этого шаблона максимально прост. Мы проходим по списку изображений, который передан в контекст в переменной `most_viewed`, и показываем их название со ссылкой на страницу картинки.

Наконец, необходимо добавить шаблон URL'а, чтобы обработчик стал доступен. Отредактируйте файл `urls.py` приложения `images` и вставьте такую строку: `path('ranking/', views.image_ranking, name='create')`,

Запустите сервер для разработки и откройте страницу любого сохраненного изображения. Перезагрузите ее несколько раз. Теперь перейдите на `http://127.0.0.1:8000/images/ranking/`. Вы увидите рейтинг картинок:

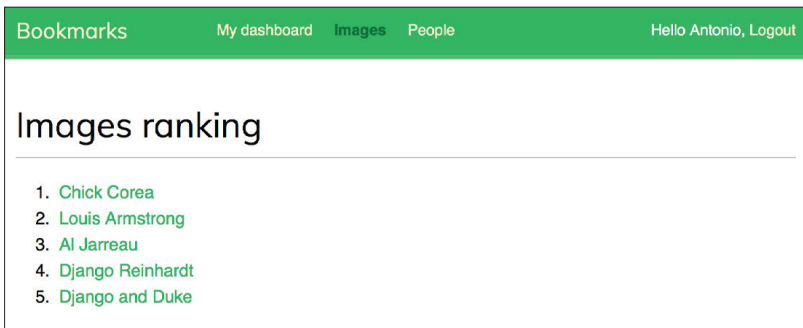


Рис. 6.8 ❖ Рейтинг самых просматриваемых картинок

Замечательно! Мы только что добавили рейтинг просмотров с помощью Redis.

Следующие шаги с Redis

Отметим, что Redis не является заменой баз данных. Это хранилище подходит для некоторых типов задач, когда необходим быстрый доступ к данным, сохраненным в памяти. Вот несколько примеров, в которых использование Redis является оправданным и может принести существенную пользу:

- подсчет событий. Как мы убедились в этой главе, реализация простого счетчика с Redis не составит труда. Используйте методы `incr()` и `incrby()` для увеличения значений;
- сохранение последних объектов. Вы можете добавлять элементы в конец или начало списков с помощью методов `lpush()` и `rpush()`. Также доступно и удаление первого или последнего элемента. Для этого используют методы `lpop()` и `rpop()`. Redis предоставляет метод `ltrim()`, который будет обрезать объекты, превышающие максимальное количество элементов для списка;
- очереди. Кроме стандартных для реализации очередей команд `push()` и `pop()`, Redis поддерживает блокировки очередей, что иногда бывает полезным;
- подписка на события. Redis предоставляет команды для рассылки и подписки на уведомления;
- кеширование. Методы `expire()` и `expireat()` позволяют использовать Redis в качестве слоя кеширования. Кроме того, существуют библиотеки для подключения этого хранилища к Django в качестве кеша;
- ранжирование. Сортированные наборы позволяют быстро формировать рейтинги и ранжировать объекты в соответствии с их позицией по некоторому критерию;
- отслеживание в реальном режиме. Быстрая система ввода-вывода Redis позволяет использовать его в режиме реального времени для отслеживания действий.

РЕЗЮМЕ

В этой главе мы реализовали систему подписок и ленту пользователей. Вы узнали, как работать с сигналами Django и подключить к проекту хранилище Redis.

В следующей главе мы создадим онлайн-магазин: определим модели и опишем логику для работы с каталогом товаров и корзиной покупок, попробуем работать с сессиями. Также вы научитесь настраивать и запускать Celery и напишете свою первую асинхронную задачу.

Глава 7

Создание онлайн-магазина

В предыдущей главе мы реализовали систему подписчиков и ленту активности пользователей. Вы познакомились с сигналами Django и подключили к проекту хранилище Redis для подсчета количества просмотров картинок. В этой главе мы выполним проект для онлайн-магазина. Вы создадите каталог товаров и добавите возможность формировать покупательскую корзину, используя сессии Django. Также вы реализуете собственный контекстный процессор и запустите первую асинхронную задачу в Celery.

В этой главе будут рассмотрены следующие темы:

- создание каталога товаров;
- реализация корзины товаров с помощью сессий Django;
- управление заказами покупателей;
- отправка асинхронных уведомлений с помощью Celery.

СОЗДАНИЕ ПРОЕКТА

Давайте начнем с создания нового Django-проекта. Наши пользователи смогут переходить по страницам магазина, изучать подробнее интересующий товар и класть его в корзину. Кроме этого, мы добавим возможность оформлять и оплачивать заказ. В этом разделе выполним следующие шаги по реализации магазина:

- создадим модели, добавим их на сайт администрирования и опишем обработчики для отображения данных на сайте;
- используя сессии Django, реализуем корзину, чтобы выбранные к покупке товары сохранялись при переходах между страницами сайта;
- создадим форму и опишем логику обработки заказов;
- подключим Celery для асинхронной рассылки e-mail-сообщений пользователям для подтверждения покупки.

Откройте консоль, создайте новое виртуальное окружение и активируйте его с помощью команд:

```
mkdir env
virtualenv env/myshop
source env/myshop/bin/activate
```

Установите в окружение Django:

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

Создайте новый проект myshop с приложением shop, выполнив команды:

```
django-admin startproject myshop
cd myshop/
django-admin startapp shop
```

Отредактируйте файл настроек settings.py и добавьте приложение shop в список ранее установленных, INSTALLED_APPS:

```
INSTALLED_APPS = [
    # ...
    'shop.apps.ShopConfig',
]
```

Приложение создано и подключено к проекту. Теперь нужно описать модель для товара в нашем магазине.

Добавление моделей каталога товаров

Каталог товаров в магазинах обычно содержит объекты, сгруппированные по различным категориям. Каждый товар имеет название (часто – описание и фотографии), цену и информацию о наличии. Добавьте соответствующую модель, отредактируйте файл models.py приложения shop, вставьте в него следующий код:

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, unique=True)

    class Meta:
        ordering = ('name',)
        verbose_name = 'category'
        verbose_name_plural = 'categories'

    def __str__(self):
        return self.name

class Product(models.Model):
    category = models.ForeignKey(Category,
                                related_name='products',
                                on_delete=models.CASCADE)
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True)
    image = models.ImageField(upload_to='products/%Y/%m/%d', blank=True)
    description = models.TextField(blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
```

```
class Meta:
    ordering = ('name',)
    index_together = (('id', 'slug'),)

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

Этот фрагмент описывает две модели: для категории, `Category`, и товара, `Product`. Модель `Category` состоит из двух полей: наименования, `name`, и уникального поля слага, `slug`. В модель `Product` добавлены такие поля:

- `category` – `ForeignKey` на модель `Category`. Это отношение «многие к одному» – каждый товар принадлежит к одной категории, но каждая категория может включать множество товаров;
- `name` – наименование;
- `slug` – уникальное поле слага, которое будем использовать для построения человекопонятных URL'ов;
- `image` – не являющееся обязательным поле для фотографии товара;
- `description` – не являющееся обязательным описание товара;
- `price` – цена товара. Это поле определено как тип `decimal.Decimal` и будет хранить значения с фиксированной точностью. Максимальное количество цифр, включая десятичные, задается параметром `max_digits`, а количество цифр после запятой – `decimal_places`;
- `available` – булево значение, которое говорит о наличии товара. Мы будем использовать его для скрытия из каталога закончившихся товаров;
- `created` – дата и время создания товара;
- `updated` – дата и время последнего изменения.

Мы определили поле `price` как `DecimalField` вместо `FloatField`, чтобы избежать проблем с округлением цен.

i Всегда используйте `DecimalField` для полей, которые хранят денежные значения. Поле `FloatField` представляется как число типа `float`, в то время как `DecimalField` преобразуется в тип `Decimal`. Применяя второй, вы избежите проблем с неточностью из-за внутренних округлений чисел типа `float`.

В опциях `Meta` модели `Product` мы определили индекс по двум полям, `id` и `slug`, с помощью атрибута `index_together`. Так как в дальнейшем мы будем запрашивать товары по их `id` и слаг, добавление такого индекса ускорит выборку объектов.

Поскольку наша модель содержит поле для изображений, необходимо установить пакет `Pillow` с помощью команды:

```
pip install Pillow==5.1.0
```

Теперь запустите команду создания миграций:

```
python manage.py makemigrations
```

Вы увидите, какие изменения Django отобразит в миграциях:

```
Migrations for 'shop':
shop/migrations/0001_initial.py
```

- Create model Category
- Create model Product
- Alter index_together for product (1 constraint(s))

Выполните синхронизацию миграций и базы данных:

```
python manage.py migrate
```

Вы увидите, что миграция успешно применилась:

```
Applying shop.0001_initial... OK
```

Теперь состояние базы данных полностью соответствует тому, что мы описали в моделях.

Регистрация моделей каталога на сайте администрирования

Давайте добавим модели на сайт администрирования, чтобы с легкостью управлять категориями и настраивать отображение товаров. Отредактируйте файл `admin.py` приложения `shop` и вставьте следующий код:

```
from django.contrib import admin
from .models import Category, Product

@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug']
    prepopulated_fields = {'slug': ('name',)}

@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price', 'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'available']
    prepopulated_fields = {'slug': ('name',)}
```

Помните, используя `prepopulated_fields`, мы настраиваем поле `slug` так, чтобы его значение формировалось автоматически из поля `name`. Атрибут `list_editable` в классе `ProductAdmin` добавляет возможность изменять перечисленные поля со страницы списка товаров, не переходя к форме редактирования товара. Отметим, что все поля, перечисленные в `list_editable`, должны быть добавлены в атрибут `list_display`, иначе они не будут видны на странице списка объектов.

Давайте создадим суперпользователя. Для этого выполните команду и задайте логин и пароль:

```
python manage.py createsuperuser
```

Запустите сервер командой `python manage.py runserver`. Перейдите на сайт администрирования по адресу: `http://127.0.0.1:8000/admin/shop/product/add/`, используя логин и пароль суперпользователя. Добавьте категорию и товар, после чего вы будете перенаправлены на страницу списка товаров, которая будет выглядеть аналогичным образом:

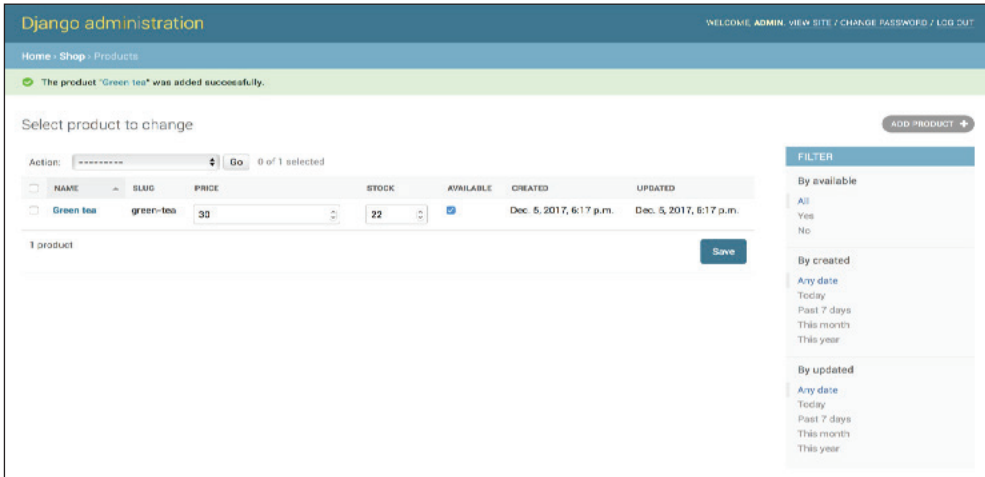


Рис. 7.1 ❖ Список товаров на сайте администрирования

Реализация обработчиков для каталога

Для того чтобы пользователи сайта могли видеть товары, нам нужно создать страницу списка товаров и осуществить их фильтрацию по категориям. Отредактируйте файл `views.py` приложения `shop` и добавьте следующий фрагмент:

```
from django.shortcuts import render, get_object_or_404
from .models import Category, Product

def product_list(request, category_slug=None):
    category = None
    categories = Category.objects.all()
    products = Product.objects.filter(available=True)
    if category_slug:
        category = get_object_or_404(Category, slug=category_slug)
        products = products.filter(category=category)
    return render(request, 'shop/product/list.html',
                  {'category': category,
                   'categories': categories,
                   'products': products})
```

Мы фильтруем `QuerySet` товаров по условию `available=True`, чтобы отображать только товары, имеющиеся в наличии. В обработчик также может быть передан не являющийся обязательным параметр `category_slug`. Если он указан, мы фильтруем товары по категориям.

Кроме отображения списка, в магазине нужно отображать страницу каждого товара с его подробным описанием. Для этого добавьте следующий обработчик:

```
def product_detail(request, id, slug):
    product = get_object_or_404(Product, id=id, slug=slug, available=True)
    return render(request, 'shop/product/detail.html', {'product': product})
```

Обработчик `product_detail` ожидает параметры `id` и `slug`, чтобы однозначно определить нужный товар. Мы могли бы получать объект, указав только ID, но, используя слаг, мы генерируем для товаров хорошие, с точки зрения SEO, ссылки, в результате чего поисковые системы смогут лучше проиндексировать наш сайт.

После описания обработчиков необходимо сформировать шаблоны их URL'ов. Создайте новый файл, `urls.py`, в папке приложения `shop`. Добавьте в него следующие строки:

```
from django.urls import path
from . import views
app_name = 'shop'

urlpatterns = [
    path('', views.product_list, name='product_list'),
    path('<slug:category_slug>/', views.product_list,
         name='product_list_by_category'),
    path('<int:id>/<slug:slug>/', views.product_detail, name='product_detail'),
]
```

Мы определили несколько шаблонов для обработчика `product_list`: шаблон `product_list` вызовет функцию без дополнительных параметров, а шаблон `product_list_by_category` вызовет функцию, передав в качестве аргумента слаг, `category_slug`. Также мы добавили шаблон для отображения страницы подробностей товара. Он вызовет функцию `product_detail` с двумя дополнительными параметрами: `id` и `slug`.

Отредактируйте файл `urls.py` проекта `myshop`, чтобы он выглядел таким образом:

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

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

Мы добавили конфигурацию URL'ов приложения `shop` в конфигурацию URL'ов проекта.

Давайте добавим в модели метод для получения канонического URL'а конкретного объекта. Отредактируйте файл `models.py` приложения `shop`. С этой целью импортируйте функцию `reverse()` и примените метод `get_absolute_url()` для моделей `Category` и `Product`, как показано ниже:

```
from django.urls import reverse
# ...

class Category(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('shop:product_list_by_category', args=[self.slug])
```

```
class Product(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('shop:product_detail', args=[self.id, self.slug])
```

Как вы уже знаете, метод `get_absolute_url()` используется в Django для формирования URL'а для конкретного объекта. В этих методах мы обращаемся по имени к шаблонам URL'ов приложения `shop`, которые только что определили.

Добавление шаблонов для отображения страниц каталога

Самое время описать шаблоны для отображения товаров. Создайте аналогичную структуру каталогов и файлов в папке приложения `shop`:

```
templates/
  shop/
    base.html
    product/
      list.html
      detail.html
```

Мы определим базовый шаблон, который будет задавать общую структуру для любой страницы сайта. Отредактируйте файл `shop/base.html` и вставьте в него такой код:

```
{% load static %}
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>{% block title %}My shop{% endblock %}</title>
    <link href="{% static 'css/base.css' %}" rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <a href="/" class="logo">My shop</a>
    </div>
    <div id="subheader">
      <div class="cart">Your cart is empty.</div>
    </div>
    <div id="content">
      {% block content %}{% endblock %}
    </div>
  </body>
</html>
```

От этого шаблона будут наследоваться страницы списка и подробностей товаров. Не забудьте скопировать статические файлы из кода – примера к данной главе в папку `static/` приложения `shop`.

Вставьте в файл `shop/product/list.html` следующий фрагмент:

```
{% extends "shop/base.html" %}
{% load static %}
```

```

{% block title %}
  {% if category %}{{ category.name }}{% else %}Products{% endif %}
{% endblock %}

{% block content %}
  <div id="sidebar">
    <h3>Categories</h3>
    <ul>
      <li {% if not category %}class="selected"{% endif %}>
        <a href="{% url "shop:product_list" %}">All</a>
      </li>
      {% for c in categories %}
        <li {% if category.slug == c.slug %}class="selected" {% endif %}>
          <a href="{{ c.get_absolute_url }}">{{ c.name }}</a>
        </li>
      {% endfor %}
    </ul>
  </div>
  <div id="main" class="product-list">
    <h1>{% if category %}{{ category.name }}{% else %}Products{% endif %}</h1>
    {% for product in products %}
      <div class="item">
        <a href="{{ product.get_absolute_url }}">
          
        </a>
        <a href="{{ product.get_absolute_url }}">{{ product.name }}</a>
        <br>
        ${{ product.price }}
      </div>
    {% endfor %}
  </div>
{% endblock %}

```

Это шаблон списка товаров. Он наследуется от базового шаблона `base.html` и использует переменные контекста `categories`, чтобы отобразить все доступные категории в боковой панели и `products` – для списка товаров. Этот код используется в обоих случаях: для показа всех товаров или только товаров определенной категории. Так как поле `image` модели `Product` может быть пустым, необходимо указать, какую картинку использовать по умолчанию, если фото товара не задано. Это изображение находится в статических файлах по пути `img/no_image.png`.

Так как для сохранения изображений мы используем поле `ImageField`, необходимо дополнительно настроить сервер для разработки, чтобы он находил загруженные пользователями картинки.

Отредактируйте файл `settings.py` проекта `myshop` и добавьте следующие строки:

```

MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')

```


`MEDIA_URL` – это базовый URL для доступа к файлам, загруженным пользователями. `MEDIA_ROOT` – путь в файловой системе, по которому хранятся эти файлы, по умолчанию полный путь формируется из настроек `BASE_DIR` и `MEDIA_ROOT`.

Чтобы сервер Django мог обращаться к файлам пользователей, отредактируйте файл `urls.py` проекта `myshop` и добавьте строки, как показано ниже:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ...
]
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Обратите внимание, что такой способ работы с файлами пользователей подходит только для их разработки и отладки. Никогда не используйте его в боевом режиме.

Добавьте несколько товаров через сайт администрирования и откройте главную страницу на `http://127.0.0.1:8000/`. Вы увидите список товаров, например таких:

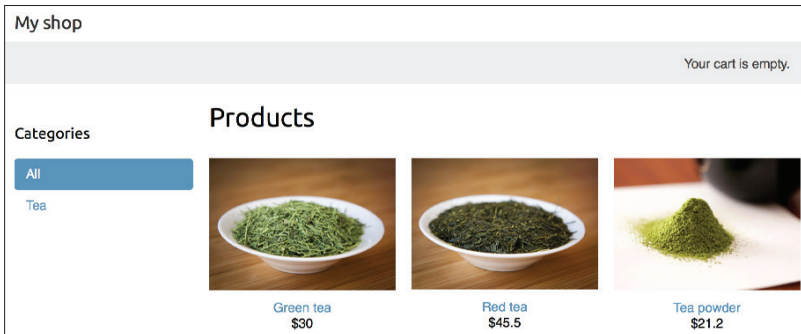


Рис. 7.2 ❖ Список товаров на сайте

Если вы создали какой-нибудь товар без фотографии, то вместо пустого места отобразится картинка по умолчанию, `no_image.png`:

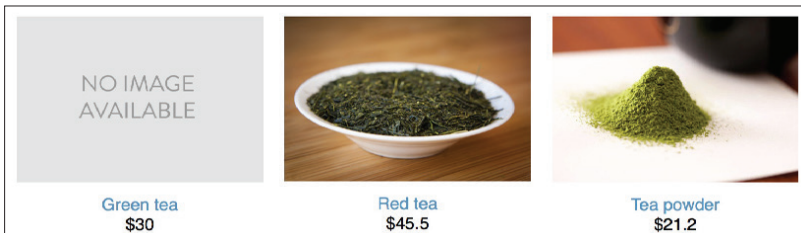


Рис. 7.3 ❖ Товар без фотографии

Давайте добавим страницу товара. Отредактируйте файл `shop/product/detail.html` и вставьте в него такой фрагмент:

```
{% extends "shop/base.html" %}
{% load static %}

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

{% block content %}
  <div class="product-detail">
    
    <h1>{{ product.name }}</h1>
    <h2>
      <a href="{{ product.category.get_absolute_url }}">{{product.category }}</a>
    </h2>
    <p class="price">${{ product.price }}</p>
    {{ product.description|linebreaks }}
  </div>
{% endblock %}
```

Мы обращаемся к методу `get_absolute_url()` категории товара, чтобы пользователь мог перейти со страницы подробностей к товарам этой категории. Откройте главную страницу `http://127.0.0.1:8000/` и кликните на любой товар. Вы будете перенаправлены на страницу с его подробным описанием, например такую:

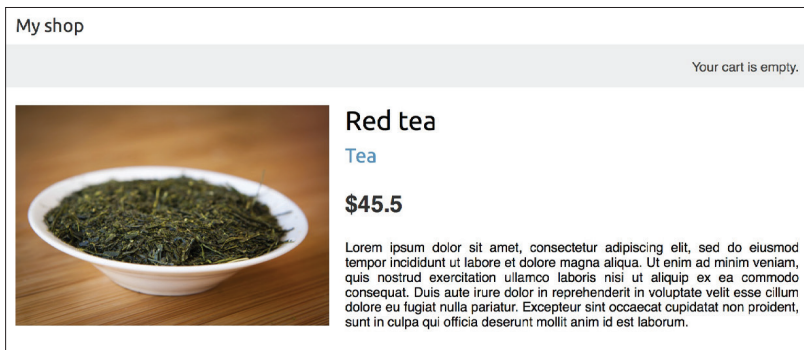


Рис. 7.4 ❖ Страница товара

Вы только что создали простой онлайн-магазин. Давайте добавим к базовой функциональности другие полезные возможности.

ДОБАВЛЕНИЕ КОРЗИНЫ ПОКУПОК

После создания каталога нужно добавить покупателям возможность выбрать и положить в корзину товары, которые они хотели бы приобрести. Во время поиска товаров корзина помогает пользователю отложить понравившийся и не потерять его до момента оформления заказа. Кроме этого, желательно сохранять корзину, чтобы, когда пользователь вернется в следующий раз и продолжит покупки, ему не пришлось заполнять ее заново.

Мы будем использовать *подсистему сессий* Django для сохранения данных корзины. Она будет запоминаться в сессии до тех пор, пока покупатель не оформит заказ. От нас потребуется описать дополнительную модель для корзины.

Использование сессий Django

Django предоставляет подсистему сессий, которая может работать как с авторизованными, так и с анонимными пользователями. С ее помощью вы можете привязать любую информацию к конкретному посетителю. Информация сессии сохраняется в базе данных на стороне сервера и в куках на стороне браузера, если вы используете сессии на их основе. Промежуточный слой сессий управляет обработкой и установкой куков для каждого запроса. По умолчанию подсистема сессий сохраняет их в базу данных, но и это поведение можно переопределить, если выбрать другой механизм хранения сессий.

Чтобы начать работу с сессиями, необходимо добавить в настройку MIDDLEWARE проекта строку `'django.contrib.sessions.middleware.SessionMiddleware'`. Это промежуточный слой для управления сессиями. Когда проект создается командой `startproject`, этот слой уже добавлен в список MIDDLEWARE.

Благодаря промежуточному слою текущая сессия становится доступна в объекте запроса, `request`, и вы можете обратиться к ней через запись `request.session`. Данные хранятся в виде словаря Python и могут быть представлены любым типом объектов. Единственное ограничение – ключи и значения должны быть сериализуемыми в JSON. Например, вы можете задать значение таким образом:

```
request.session['foo'] = 'bar'
```

Получить значение по ключу `foo`:

```
request.session.get('foo')
```

Удалить значение по ключу `foo`:

```
del request.session['foo']
```

То есть работа с сессиями в Django полностью аналогична работе с Python-словарями.



Когда пользователь авторизуется на сайте, его анонимная сессия теряется, и создается новая, ассоциированная с конкретным пользователем. Если вы храните в анонимной сессии данные, которые не должны быть утеряны после авторизации, необходимо копировать их в новую сессию при входе пользователя.

Настройки сессий

Django поддерживает несколько настроек сессий для вашего проекта. Самая значимая из них – `SESSION_ENGINE`. Она позволяет указать, каким образом хранить данные сессии. По умолчанию они сохраняются в базу данных как объекты модели `Session` приложения `django.contrib.sessions`.

Существуют следующие способы хранения данных сессии:

- на основе базы данных – информация сессии сохраняется в базе (этот способ используется по умолчанию);
- на основе файлов – данные сохраняются в файловой системе;
- на основе кеша – данные хранятся в бэкенде кеширования. Вы можете настроить его с помощью конфигурации `CACHES` файла `settings.py`. Отметим, что сессии на основе кеша – самый быстрый способ;
- на основе кеша и базы данных – информация сессий записывается в базу данных, но для доступа к ней обращение идет сначала в кеш, и только в том случае, если там этой информации уже нет, выполняется запрос в базу данных;
- на основе куков – данные сессий сохраняются в куках, отправляемых в браузер пользователя.

i Для лучшей производительности используйте сессии на основе кеша. Django поддерживает работу с `Memcached` и другими бэкендами кеширования, такими как `Redis`.

Кроме задания способа сохранения данных сессий, можно управлять ими и через такие настройки:

- `SESSION_COOKIE_AGE` – время жизни сессии на основе куков в секундах. Значение по умолчанию – `1209600` (2 недели);
- `SESSION_COOKIE_DOMAIN` – домен для сессий на основе куков. Установите эту настройку равной домену вашего сайта или `None`, чтобы избежать угрозы подмены куков;
- `SESSION_COOKIE_DOMAIN` – булево значение, говорящее о том, может ли сессия на основе куков быть задана через `HTTP-` и `HTTPS-` соединения или только через `HTTPS-`;
- `SESSION_EXPIRE_AT_BROWSER_CLOSE` – время жизни сессии на основе куков после закрытия браузера;
- `SESSION_SAVE_EVERY_REQUEST` – булево значение. Если оно равно `True`, сессия будет сохраняться в базу данных при каждом запросе. При этом время окончания ее действия будет автоматически обновляться.

Все доступные настройки подсистемы сессий вы можете найти на странице: <https://docs.djangoproject.com/en/2.0/ref/settings/#sessions>.

Время жизни сессии

Вы можете задать время жизни сессии в браузере с помощью настройки `SESSION_EXPIRE_AT_BROWSER_CLOSE`. По умолчанию она равна `False`. Сессия будет храниться в течение времени, указанного в `SESSION_COOKIE_AGE`. Если вы установите

`SESSION_EXPIRE_AT_BROWSER_CLOSE=True`, сессия будет заканчиваться при закрытии пользователем браузера. При этом Django не будет обращать внимания на время жизни, заданное в `SESSION_COOKIE_AGE`.

Кроме этого, можно изменить время жизни с помощью метода `set_expiry()` объекта сессии `request.session`.

Хранение данных корзины в сессии

Для сохранения корзины в сессии нам понадобится простая структура данных, которая может быть сериализована в JSON. Эта структура будет содержать список товаров. Каждый ее элемент может быть описан такими полями:

- ID объекта модели `Product`;
- выбранное количество товаров;
- цена за единицу.

Так как цены на товары могут меняться, мы будем сохранять текущую стоимость при добавлении товара в корзину. Так мы сохраним для пользователя ту цену, которую он видел при выборе товара, несмотря на то что она может поменяться к моменту оформления заказа.

Давайте реализуем функции для создания корзины и привязки ее к конкретному пользователю. Эти функции будут работать по такому принципу:

- когда корзина понадобится, мы проверяем, сохранена ли она в сессии. Если нет – создаем объект корзины и записываем его в сессию;
- когда корзина уже есть в сессии, получаем ее товары и связанные объекты `Product`.

Отредактируйте файл `settings.py` проекта и добавьте такую настройку:

```
CART_SESSION_ID = 'cart'
```

Это ключ, по которому мы будем хранить данные корзины в сессии. Так как сессии Django ассоциируются с конкретным посетителем сайта, мы можем использовать один и тот же ключ для разных пользователей. Это не приведет к конфликту данных.

Давайте создадим приложения для работы с корзиной покупок. Откройте консоль и выполните следующую команду из папки проекта:

```
python manage.py startapp cart
```

Затем добавьте его в список активных приложений проекта, отредактировав файл `settings.py` таким образом:

```
INSTALLED_APPS = [
    # ...
    'shop.apps.ShopConfig',
    'cart.apps.CartConfig',
]
```

Создайте в каталоге приложения `cart` файл `cart.py` с таким содержимым:

```
from decimal import Decimal
from django.conf import settings
```

```

from shop.models import Product

class Cart(object):

def __init__(self, request):
    """Инициализация объекта корзины."""
    self.session = request.session
    cart = self.session.get(settings.CART_SESSION_ID)
    if not cart:
        # Сохраняем в сессии пустую корзину.
        cart = self.session[settings.CART_SESSION_ID] = {}
    self.cart = cart

```

Это класс `Cart`, который будет отвечать за работу с корзинами покупок. При инициализации объекта этого класса необходимо передать в конструктор объект запроса `request`. Мы запоминаем текущую сессию в атрибуте `self.session` (она равна сессии из запроса `request.session`), чтобы иметь к ней доступ в других методах класса. Затем пытаемся получить данные корзины, обращаясь к `self.session.get(settings.CART_SESSION_ID)`. Если не получаем объект корзины, создаем ее как пустой словарь в сессии. В этом словаре ключами будут являться ID товаров, а значениями – количество и цена. Так мы будем уверены, что товар не добавлен в корзину больше одного раза. Хранение корзины в виде словаря упростит доступ к ее элементам.

Давайте реализуем подход для добавления товаров в корзину и обновления их количества. Создайте два метода – `add()` и `save()` – в классе `Cart`:

```

class Cart(object):
    # ...
def add(self, product, quantity=1, update_quantity=False):
    """Добавление товара в корзину или обновление его количества."""
    product_id = str(product.id)
    if product_id not in self.cart:
        self.cart[product_id] = {'quantity': 0, 'price': str(product.price)}
    if update_quantity:
        self.cart[product_id]['quantity'] = quantity
    else:
        self.cart[product_id]['quantity'] += quantity
    self.save()

def save(self):
    # Помечаем сессию как измененную
    self.session.modified = True

```

Метод `add()` принимает три входных параметра:

- `product` – объект модели `Product`, который нужно добавить или обновить;
- `quantity` – не являющееся обязательным количество объектов, по умолчанию 1;
- `update_quantity` – булево значение, которое говорит о том, нужно ли заменить значение количества товаров на новое (`True`) или следует добавить его к существующему (`False`).

Мы используем ID товара как ключ в словаре корзины. При этом он преобразуется в строку, потому что Django использует формат JSON для сериализации данных сессии, а в JSON-ключами могут быть только строки. Данные о цене также преобразуются в строку, чтобы их можно было сериализовать. В конце мы вызываем метод `save()`, чтобы сохранить сведения в сессию.

Метод `save()` помечает сессию как измененную с помощью атрибута `modified` – `session.modified = True`. Так мы говорим Django о том, что редактировали данные сессии, а теперь их необходимо сохранить.

Кроме этих методов, нужно реализовать удаление товаров из корзины. Добавьте метод `remove()` в класс `Cart`:

```
class Cart(object):
    # ...
    def remove(self, product):
        """Удаление товара из корзины."""
        product_id = str(product.id)
        if product_id in self.cart:
            del self.cart[product_id]
            self.save()
```

Метод `remove()` удаляет товар из корзины и сохраняет новые данные сессии, обращаясь к методу `save()`.

Для отображения списка товаров, отложенных в корзину, нужно иметь возможность проходить в цикле по объектам `Product`. С этой целью добавьте метод `__iter__()` в класс `Cart`, как показано ниже:

```
class Cart(object):
    # ...
    def __iter__(self):
        """Проходим по товарам корзины и получаем соответствующие объекты Product."""
        product_ids = self.cart.keys()
        # Получаем объекты модели Product и передаем их в корзину.
        products = Product.objects.filter(id__in=product_ids)

        cart = self.cart.copy()
        for product in products:
            cart[str(product.id)]['product'] = product
        for item in cart.values():
            item['price'] = Decimal(item['price'])
            item['total_price'] = item['price'] * item['quantity']
        yield item
```

В этом методе мы создаем копию объекта корзины, получаем товары, сохраненные в ней. Для каждого товара преобразуем цену из строки в число с фиксированной точностью, вычисляем общую стоимость, `total_price`, с учетом цены и количества. Теперь у нас есть все сведения о каждом товаре, сохраненные в сессии.

Для адекватного отображения корзины на сайте необходимо выводить количество товаров в корзине. Для этих целей можно использовать встроенную

Мы будем использовать эту форму для добавления товаров в корзину. Класс `CartAddProductForm` содержит такие поля:

- `quantity` – количество единиц товара (доступны значения от 1 до 20). Мы используем класс `TypedChoiceField` с параметром `coerce=int`, чтобы автоматически преобразовывать выбранное значение в целое число;
- `update` – обновить (значение `True`) или заменить (значение `False`) количество единиц для товара. Мы используем тип поля `HiddenInput`, чтобы пользователь не видел его в своей форме.

Следующий шаг – создание обработчика для добавления товаров в корзину. Откройте файл `views.py` приложения `cart` и вставьте такое содержимое:

```
from django.shortcuts import render, redirect, get_object_or_404
from django.views.decorators.http import require_POST
from shop.models import Product
from .cart import Cart
from .forms import CartAddProductForm

@require_POST
def cart_add(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    form = CartAddProductForm(request.POST)
    if form.is_valid():
        cd = form.cleaned_data
        cart.add(product=product,
                quantity=cd['quantity'],
                update_quantity=cd['update'])
    return redirect('cart:cart_detail')
```

Мы обернули функцию `cart_add()` декоратором `require_POST`, чтобы обратиться к ней можно было только методом `POST`. Обработчик принимает ID товара в качестве аргумента, по которому мы получаем объект `Product` из базы данных. Для работы с корзиной создаем форму `CartAddProductForm` и, если она валидна, добавляем или обновляем сведения по товару. В конце перенаправляем пользователя на URL с названием `cart_detail`. По этому адресу покупатель сможет увидеть содержимое своей корзины. Соответствующий обработчик мы добавим чуть позже.

Для удаления товара из корзины создадим обработчик `cart_remove()`. Допишите его код в файле `views.py` приложения `cart`:

```
def cart_remove(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    cart.remove(product)
    return redirect('cart:cart_detail')
```

Эта функция получает ID товара в качестве аргумента, вызывает его из базы данных и удаляет из корзины, после чего пользователь перенаправляется на страницу корзины.

Наконец, давайте создадим обработчик для страницы списка товаров, добавленных в корзину. Добавьте следующий фрагмент в файл `views.py` приложения `cart`:

```
def cart_detail(request):
    cart = Cart(request)
    return render(request, 'cart/detail.html', {'cart': cart})
```

Функция `cart_detail()` будет отображать корзину, основываясь на данных, сохраненных в сессии `request.session`.

Мы создали все обработчики, необходимые для работы с корзиной товаров. Самое время – описать шаблоны URL'ов для них. Создайте новый файл, `urls.py`, в папке приложения `cart` и добавьте в него такой код:

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

app_name = 'cart'

urlpatterns = [
    path('', views.cart_detail, name='cart_detail'),
    path('add/<int:product_id>/', views.cart_add, name='cart_add'),
    path('remove/<int:product_id>/', views.cart_remove, name='cart_remove'),
]
```

Подключите созданный файл в основном `urls.py` проекта `myshop`, как показано ниже:

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

Убедитесь, что шаблоны приложения `cart` добавлены перед шаблонами приложения `shop`, т. к. они более подробные, чем в приложении `shop`, и могут быть перекрыты, если изменить порядок подключения.

Шаблон для отображения содержимого корзины

Обработчики `cart_add` и `cart_remove`, в отличие от функции `cart_detail`, не формируют никаких страниц. Самое время создать шаблон для отображения содержимого корзины.

Создайте внутри каталога приложения `cart` такую структуру папок:

```
templates/
  cart/
    detail.html
```

Затем вставьте в шаблон `detail.html` следующий фрагмент кода:

```
{% extends "shop/base.html" %}
{% load static %}
```

```

{% block title %}Your shopping cart{% endblock %}

{% block content %}
<h1>Your shopping cart</h1>
<table class="cart">
  <thead>
    <tr>
      <th>Image</th>
      <th>Product</th>
      <th>Quantity</th>
      <th>Remove</th>
      <th>Unit price</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    {% for item in cart %}
      {% with product=item.product %}
        <tr>
          <td>
            <a href="{{ product.get_absolute_url }}">
              
            </a>
          </td>
          <td>{{ product.name }}</td>
          <td>{{ item.quantity }}</td>
          <td><a href="{% url "cart:cart_remove" product.id%">Remove</a></td>
          <td class="num">${{ item.price }}</td>
          <td class="num">${{ item.total_price }}</td>
        </tr>
      {% endwith %}
    {% endfor %}
    <tr class="total">
      <td>Total</td>
      <td colspan="4"></td>
      <td class="num">${{ cart.get_total_price }}</td>
    </tr>
  </tbody>
</table>
<p class="text-right">
  <a href="{% url "shop:product_list" %}" class="button light">Continue shopping</a>
  <a href="#" class="button">Checkout</a>
</p>
{% endblock %}

```

Этот шаблон сформирует страницу содержимого корзины. Товары будут размещены в виде таблицы. Пользователь сможет изменить количество каждого товара, отправляя форму на обработчик `cart_add`, или убрать его из корзины с помощью ссылки **Remove**.

Добавление товаров в шаблонах

Теперь нам нужно добавить кнопку **Add to cart** на страницу товара. Отредактируйте файл `views.py` приложения `shop` и добавьте форму `CartAddProductForm` в контекст обработчика `product_detail` таким образом:

```
from cart.forms import CartAddProductForm
def product_detail(request, id, slug):
    product = get_object_or_404(Product, id=id, slug=slug, available=True)
    cart_product_form = CartAddProductForm()
    return render(request, 'shop/product/detail.html',
                  {'product': product,
                   'cart_product_form': cart_product_form})
```

Теперь вставьте в шаблон `shop/product/detail.html` форму, которую мы только что добавили в контекст:

```
<p class="price">${{ product.price }}</p>
<form action="{% url 'cart:cart_add' product.id %}" method="post">
  {{ cart_product_form }}
  {% csrf_token %}
  <input type="submit" value="Add to cart">
</form>
{{ product.description|linebreaks }}
```

Убедитесь, что сервер разработки запущен. Если этого не произошло, выполните команду `python manage.py runserver`. Откройте сайт `http://127.0.0.1:8000/` и перейдите на страницу подробностей любого товара. Вы должны будете увидеть, что под ценой добавилась форма с кнопкой **Add to cart** и выбором количества единиц:

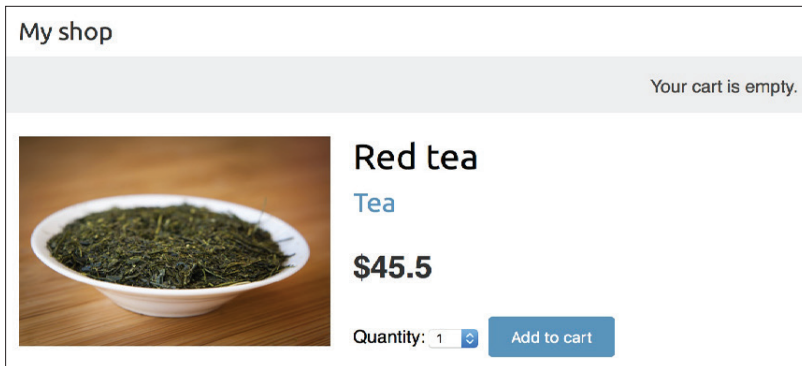


Рис. 7.5 ❖ Форма добавления товара в корзину

Задайте какое-нибудь число и нажмите кнопку **Add to cart**. Форма будет отправлена в обработчик `cart_add` через POST-запрос. Выбранный вами товар добавится в объект корзины, сохраненный в сессии. При этом сохранятся вы-

бранное количество и текущая цена. Затем вы будете перенаправлены на страницу корзины:

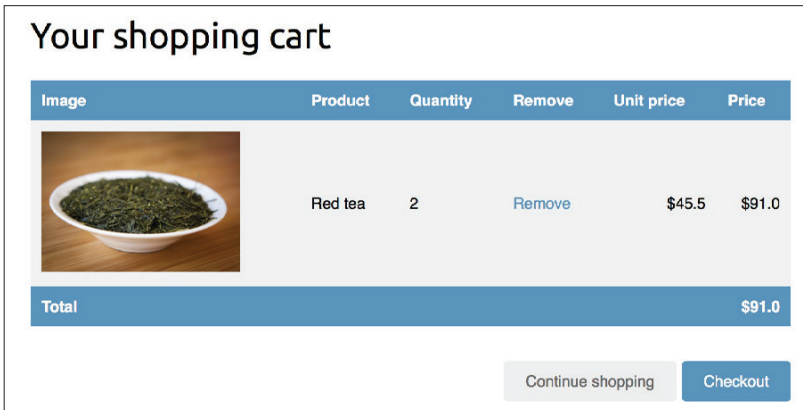


Рис. 7.6 ❖ Товары в корзине

Изменение количества товаров в корзине

Когда пользователи зайдут на страницу корзины, они могут захотеть изменить количество товаров, перед тем как оформить заказ. Давайте добавим обработку этих действий.

Отредактируйте файл `views.py` приложения `cart` и добавьте следующие строки в обработчик `cart_detail`:

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
            initial={'quantity': item['quantity'],
                    'update': True})
    return render(request, 'cart/detail.html', {'cart': cart})
```

Теперь мы будем создавать объект формы `CartAddProductForm` для каждого товара в корзине, чтобы пользователь мог сохранить новое количество единиц. Мы инициализируем форму, передавая текущее количество и значение `update`, равное `True`. Таким образом, когда пользователь отправит форму на сервер в обработчик `cart_add`, старое значение количества товаров заменится на новое.

Давайте добавим отображение формы на странице корзины. Для этого откройте файл `cart/detail.html` приложения `cart` и найдите такую строку:

```
<td>{{ item.quantity }}</td>
```

Замените ее на представленный ниже фрагмент:

```
<td>
<form action="{% url 'cart:cart_add' product.id %}" method="post">
  {{ item.update_quantity_form.quantity }}
```

```

{{ item.update_quantity_form.update }}
<input type="submit" value="Update">
{% csrf_token %}
</form>
</td>

```

Теперь откройте в браузере <http://127.0.0.1:8000/cart/>. Вы увидите, что для каждого товара в корзине появилась возможность обновить количество единиц, которое пользователь хочет купить:

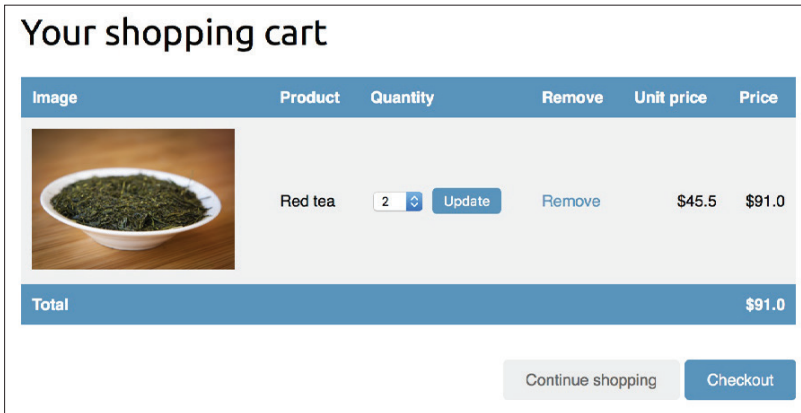


Рис. 7.7 ❖ Форма изменения количества единиц товара на странице корзины

Попробуйте изменить количество и нажать кнопку **Update**, чтобы протестировать новую функциональность. Также можете попробовать удалить товар из корзины с помощью ссылки **Remove**.

Создание контекстного процессора для корзины

Вы могли заметить, что в заголовке сайта отображается сообщение **Your cart is empty** даже в том случае, если корзина на самом деле не пуста. Для удобства пользователя нам нужно показывать актуальное количество товаров в корзине и общую стоимость. Так как эта информация будет отображаться на всех страницах сайта, мы создадим собственный контекстный процессор, который будет добавлять в контекст объект корзины для каждого запроса.

Контекстный процессор

Контекстный процессор – это функция Python, принимающая объект запроса `request` и возвращающая словарь, который будет добавлен в контекст запроса. Обычно к их использованию прибегают, когда нужно получить доступ к каким-либо объектам или переменным глобально, во всех шаблонах.

По умолчанию, когда вы создаете новый проект командой `startproject`, в него уже включено несколько контекстных процессоров. Они добавляются в раздел `context_processors` настройки `TEMPLATES`:

- `django.template.context_processors.debug` – добавляет булево значение `debug` и переменную `sql_queries`, содержащую выполненные для запроса SQL-инструкции в контекст шаблона;
- `django.template.context_processors.request` – добавляет объект запроса `request` в контекст;
- `django.contrib.auth.context_processors.auth` – добавляет объект текущего пользователя в переменную `user`;
- `django.contrib.messages.context_processors.messages` – добавляет переменную `messages`, содержащую уведомления, которые были сформированы для пользователя подсистемой сообщений Django.

Кроме этого, Django подключает процессор `django.template.context_processors.csrf`, чтобы обезопасить проект от CSRF-атак. Он не объявлен в настройках, но всегда активирован, и разработчик не может его отключить. Это сделано из соображений безопасности.

Полный список предопределенных контекстных процессоров вы можете найти на странице: <https://docs.djangoproject.com/en/2.0/ref/templates/api/#builtin-template-context-processors>.

Добавление корзины в контекст шаблонов

Давайте создадим контекстный процессор, который будет добавлять в контекст объект корзины. Так мы сможем получить к ней доступ из HTML-шаблонов.

Создайте новый файл, `context_processors.py`, в папке приложения `cart`. Процессоры могут быть описаны в любом месте вашего приложения, но создание их в отдельном файле с таким названием позволит структурировать ваш код и сделать его более понятным. Добавьте такой фрагмент в новый файл:

```
from .cart import Cart

def cart(request):
    return {'cart': Cart(request)}
```

Как мы уже говорили, контекстный процессор – это обычная функция, которой в качестве аргумента передается объект запроса, `request`, и которая должна возвращать словарь. Этот словарь будет добавляться в контекст любого шаблона, работающего с контекстом типа `RequestContext`. В нашей функции мы инициализируем корзину, передавая в конструктор объект текущего запроса, и добавляем в контекст в виде переменной `cart`.

Откройте файл `settings.py` проекта и допишите `cart.context_processors.cart` в раздел `context_processors` настройки `TEMPLATES` таким образом:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
```

```

    'OPTIONS': {
        'context_processors': [
            # ...
            'cart.context_processors.cart',
        ],
    },
},
]

```

Теперь процессор `cart` будет выполняться и добавлять переменную `cart` в контекст всякий раз, когда мы генерируем шаблон с помощью `RequestContext`.

i Контекстные процессоры выполняются при обработке всех запросов, использующих `RequestContext`. Вы можете создать собственный шаблонный тег и использовать его, если дополнительные переменные контекста не нужны во всех шаблонах. Стоит рассмотреть этот вариант, когда функция-процессор обращается к базе данных.

Откройте шаблон `shop/base.html` приложения `shop` и найдите такую строку:

```

<div class="cart">
    Your cart is empty.
</div>

```

Замените ее на фрагмент с использованием новой переменной контекста:

```

<div class="cart">
    {% with total_items=cart|length %}
        {% if cart|length > 0 %}
            Your cart:
            <a href="{% url 'cart:cart_detail' %}">
                {{ total_items }} item{{ total_items|pluralize }},
                ${{ cart.get_total_price }}
            </a>
        {% else %}
            Your cart is empty.
        {% endif %}
    {% endwith %}
</div>

```

Перезапустите сервер разработки с помощью команды `python manage.py runserver`. Откройте страницу: `http://127.0.0.1:8000/` и добавьте несколько товаров в корзину. В верхней части страницы вы увидите количество товаров и общую стоимость покупок корзины:



Рис. 7.8 ❖ Отображение количества и стоимости товаров в корзине в верхней части любой страницы

РЕГИСТРАЦИЯ ЗАКАЗОВ

Когда пользователь решает купить товары, добавленные в корзину, нам необходимо создать и сохранить заказ в базе данных. Объект заказа будет содержать сведения о покупателе и товарах из корзины.

Создайте новое приложение для управления заказами с помощью команды:

```
python manage.py startapp orders
```

Добавьте его в список установленных приложений `INSTALLED_APPS` в файле `settings.py`:

```
INSTALLED_APPS = [
    ...
    'orders.apps.OrdersConfig',
]
```

Приложение `orders` активировано.

Создание моделей заказа

Нам понадобится две модели: первая – для сохранения сведений о товаре; вторая – для связи заказа с покупаемыми товарами и указания их стоимости и количества. Отредактируйте файл `models.py` приложения `orders` и добавьте в него такой фрагмент кода:

```
from django.db import models
from shop.models import Product

class Order(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField()
    address = models.CharField(max_length=250)
    postal_code = models.CharField(max_length=20)
    city = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    paid = models.BooleanField(default=False)

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

    def __str__(self):
        return 'Order {}'.format(self.id)

    def get_total_cost(self):
        return sum(item.get_cost() for item in self.items.all())

class OrderItem(models.Model):
    order = models.ForeignKey(Order,
                              related_name='items',
                              on_delete=models.CASCADE)
    product = models.ForeignKey(Product,
```

```

        related_name='order_items',
        on_delete=models.CASCADE)
price = models.DecimalField(max_digits=10, decimal_places=2)
quantity = models.PositiveIntegerField(default=1)

def __str__(self):
    return '{}'.format(self.id)

def get_cost(self):
    return self.price * self.quantity

```

Модель `Order` определяет поля для сохранения информации о покупателе и флаг `paid`, который по умолчанию равен `False`. Позже мы будем использовать это поле, чтобы определять, оплачен заказ или нет. Также мы описали метод `get_total_cost()`, чтобы получить общую стоимость товаров в заказе.

В модели `OrderItem` мы будем сохранять товар, количество и стоимость для каждого элемента корзины. В этой модели есть метод `get_cost()` для получения общей стоимости позиции в корзине.

Выполните команду для создания начальной миграции в приложении `orders`:

```
python manage.py makemigrations
```

Вы увидите такой вывод:

```

Migrations for 'orders':
orders/migrations/0001_initial.py
- Create model Order
- Create model OrderItem

```

Запустите синхронизацию моделей и базы данных командой:

```
python manage.py migrate
```

Теперь состояние базы данных отражает созданные нами модели.

Добавление моделей на сайт администрирования

Давайте добавим эти модели на сайт администрирования. Отредактируйте файл `admin.py` приложения `orders`, добавив в него такой код:

```

from django.contrib import admin
from .models import Order, OrderItem

class OrderItemInline(admin.TabularInline):
    model = OrderItem
    raw_id_fields = ['product']

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                  'address', 'postal_code', 'city', 'paid',
                  'created', 'updated']
    list_filter = ['paid', 'created', 'updated']
    inlines = [OrderItemInline]

```

Мы используем класс `ModelInline` для модели `OrderItem`, чтобы добавить ее в виде списка связанных объектов на страницу заказа, зарегистрированную через `OrderAdmin`. Так администратор сможет редактировать данные по каждому товару напрямую со страницы заказа.

Запустите сервер для разработки командой `python manage.py runserver` и откройте страницу: `http://127.0.0.1:8000/admin/orders/order/add/`. Вы увидите такую форму:

The screenshot shows a web form titled "Add order". It contains several input fields: "First name:", "Last name:", "Email:", "Address:", "Postal code:", and "City:". Below these is a checkbox labeled "Paid". A blue header bar reads "ORDER ITEMS". Underneath is a table with four columns: "PRODUCT", "PRICE", "QUANTITY", and "DELETE?". Each row in the table has a search icon, a price input field, and a quantity spinner. At the bottom of the form, there are three buttons: "Save and add another", "Save and continue editing", and "SAVE".

Рис. 7.9 ❖ Страница оформления заказа на сайте администрирования

Обработка заказов покупателей

Мы будем использовать созданные модели, чтобы сохранить информацию о заказе, когда пользователь оформит его. Создание заказа состоит из следующих шагов:

- 1) отображение пользователю формы заказа;
- 2) создание нового объекта модели `Order` по данным, введенным покупателем, и создание объектов `OrderItem` для каждого товара из корзины;
- 3) очистка корзины и перенаправление пользователя на страницу об успешном оформлении заказа.

Для начала создайте форму для подробностей заказа. Добавьте новый файл, `forms.py`, в папке приложения `orders` и включите в него такой код:

```

from django import forms
from .models import Order

class OrderCreateForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address',
                  'postal_code', 'city']

```

Эта форма будет создавать новые объекты `Order`. Давайте добавим обработчик, который будет получать данные из запроса, инициировать, валидировать форму и создавать заказ. Отредактируйте файл `views.py` приложения `orders` и вставьте в него такой код:

```

from django.shortcuts import render
from .models import OrderItem
from .forms import OrderCreateForm
from cart.cart import Cart

def order_create(request):
    cart = Cart(request)
    if request.method == 'POST':
        form = OrderCreateForm(request.POST)
        if form.is_valid():
            order = form.save()
            for item in cart:
                OrderItem.objects.create(order=order,
                                         product=item['product'],
                                         price=item['price'],
                                         quantity=item['quantity'])

            # Очищаем корзину.
            cart.clear()
            return render(request,
                          'orders/order/created.html',
                          {'order': order})
    else:
        form = OrderCreateForm()
        return render(request,
                      'orders/order/create.html',
                      {'cart': cart, 'form': form})

```

В этом обработчике мы получаем объект корзины с помощью выражения `cart = Cart(request)`. В зависимости от метода выполняем следующие действия:

- при получении GET-запроса иницируем форму `OrderCreateForm` и передаем ее в шаблон `orders/order/create.html`;
- при получении POST-запроса валидируем данные формы. Если они корректны, записываем новый заказ в базу данных с помощью выражения `order = form.save()`. После этого проходим по всем товарам корзины и создаем для каждого объект `OrderItem`. Наконец, очищаем форму, хранящуюся в сессии, и формируем страницу ответа из шаблона `orders/order/created.html`.

Создайте новый файл в директории приложения `orders`, назовите его `urls.py` и вставьте в него такой фрагмент:

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

app_name = 'orders'

urlpatterns = [
    path('create/', views.order_create, name='order_create'),
]
```

Это шаблон URL'а для обработчика `order_create`. Теперь нужно добавить его в общий файл `urls.py` проекта `shop`. Не забудьте, что он должен быть расположен перед шаблонами приложения `shop`:

```
path('orders/', include('orders.urls', namespace='orders')),
```

Отредактируйте шаблон `cart/detail.html` приложения `cart` и замените эту строку:

```
<a href="#" class="button">Checkout</a>
```

Добавьте ссылку на обработчик `order_create`, как показано ниже:

```
<a href="{% url 'orders:order_create' %}" class="button">
Checkout
</a>
```

Теперь пользователи могут переходить из корзины на страницу оформления заказа. Но нам нужно добавить эти самые страницы. Создайте такую структуру папок в каталоге приложения `orders`:

```
templates/
  orders/
    order/
      create.html
      created.html
```

Откройте файл `orders/order/create.html` и разместите в нем следующий код:

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

{% block title %}
Checkout
{% endblock %}
```

```
{% block content %}
<h1>Checkout</h1>

<div class="order-info">
  <h3>Your order</h3>
  <ul>
    {% for item in cart %}
      <li>
        {{ item.quantity }}x {{ item.product.name }}
        <span>${{ item.total_price }}</span>
      </li>
    {% endfor %}
  </ul>
  <p>Total: ${{ cart.get_total_price }}</p>
</div>

<form action="." method="post" class="order-form">
  {{ form.as_p }}
  <p><input type="submit" value="Place order"></p>
  {% csrf_token %}
</form>
{% endblock %}
```

Этот шаблон отображает товары корзины и форму для оформления заказа. Отредактируйте файл `orders/order/created.html` и добавьте в него фрагмент:

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

{% block title %}
  Thank you
{% endblock %}

{% block content %}
  <h1>Thank you</h1>
  <p>Your order has been successfully completed. Your order number is
  <strong>{{ order.id }}</strong>.</p>
{% endblock %}
```

Этот шаблон будет формировать страницу об успешном оформлении заказа.

Перезапустите сервер для разработки, чтобы Django обнаружил новые файлы. Откройте страницу `http://127.0.0.1:8000/`, добавьте несколько товаров в корзину и перейдите к оформлению заказа. Вы увидите такую страницу:

My shop

Your cart: 3 items, \$112.2

Checkout

First name:

Last name:

Email:

Address:

Postal code:

City:

Your order

- 1x Tea powder \$21.2
- 2x Red tea \$91.0

Total: \$112.2

Рис. 7.10 ❖ Страница оформления заказа

Заполните форму и нажмите на кнопку **Place order**. Ваш заказ будет сохранен в базу данных, а вы увидите страницу об успешном создании:

Thank you

Your order has been successfully completed. Your order number is 1.

Рис. 7.11 ❖ Страница с сообщением об успешном создании заказа

ВЫПОЛНЕНИЕ АСИНХРОННЫХ ЗАДАЧ С CELERY

Для любых действий, которые выполняются в обработчике, требуется время. В некоторых ситуациях вам хочется вернуть пользователю ответ настолько быстро, насколько это возможно, и продолжить обработку запроса в асинхронном режиме. Это особенно актуально для трудоемких и долгих процессов или функций, при выполнении которых может произойти ошибка и, возможно, понадобятся повторные действия со стороны пользователя. Например, на платформе для обмена видео пользователи загружают свои файлы, но их обработка

на сервере может происходить продолжительное время. Такой сайт может вернуть пользователю сообщение о том, что его видео загружается, и выполнять обработку в асинхронном режиме. При этом действия пользователя не будут заблокированы, и он сможет дальше работать с платформой. Другой пример – отправка электронных сообщений пользователям. Если вы будете отправлять их напрямую из обработчика, то пользователь будет долго ждать ответа на свое действие, особенно в том случае, если SMTP-соединение будет медленным или разорвется. Во всех подобных ситуациях хорошей практикой считается применение асинхронных задач, которые предотвращают блокировку основного кода.

Celery – это очередь событий, которая может решать множество различных задач. Этот инструмент выполняет задачи из очереди в режиме реального времени (по мере поступления), но также позволяет задать расписание. Используя *Celery*, вы не только сможете выполнять трудоемкие процессы в асинхронном режиме, но и получите инструмент для отложенного выполнения задач по расписанию.

Полную документацию по *Celery* можно найти на странице: <http://docs.celeryproject.org/en/latest/index.html>.

Установка Celery

Давайте установим *Celery* и подключим к нашему проекту. Выполните такую команду:

```
pip install celery==4.1.0
```

Для работы с *Celery* из приложения необходим посредник. Часто его называют брокер. Этот посредник получает сообщения от внешних приложений и отправляет их рабочим потокам *Celery*, которые уже непосредственно выполняют нужные действия. Давайте установим брокер для *Celery*.

Установка RabbitMQ

Celery может работать с различными брокерами, например с хранилищами вида «ключ – значение», такими как *Redis*, или системами обмена сообщениями между приложениями, такими как *RabbitMQ*. Мы будем работать с последним, т. к. *RabbitMQ* является рекомендуемым брокером для *Celery*.

Если вы работаете с операционной системой *Linux*, откройте терминал и выполните команду:

```
apt-get install rabbitmq
```

Если вы работаете с *MacOS X* или *Windows*, перейдите по ссылке: <https://www.rabbitmq.com/download.html> и скачайте дистрибутив.

После установки запустите *RabbitMQ*, выполнив команду:

```
rabbitmq-server
```

Вы увидите вывод, который содержит такую строку:

```
Starting broker... completed with 10 plugins.
```

RabbitMQ запущен и готов принимать сообщения.

Подключение Celery к Django-проекту

Нам нужно описать конфигурацию для экземпляра Celery. Создайте новый файл, `celery.py`, и расположите его рядом с файлом `settings.py`. Здесь мы настроим наш проект на взаимодействие с Celery. Добавьте в новый файл такой фрагмент:

```
import os
from celery import Celery

# Задаем переменную окружения, содержащую название файла настроек нашего проекта.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myshop.settings')

app = Celery('myshop')

app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()
```

В этом коде мы выполняем следующие действия:

- 1) задаем переменную окружения `DJANGO_SETTINGS_MODULE` для консольных команд Celery;
- 2) создаем экземпляр приложения с помощью записи `app = Celery('myshop')`;
- 3) загружаем конфигурацию из настроек нашего проекта, вызывая метод `config_from_object()`. Параметр `namespace` определяет префикс, который мы будем добавлять для всех настроек, связанных с Celery. Таким образом, в файле `settings.py` можно будет задавать конфигурацию Celery через настройки вида `CELERY_`, например `CELERY_BROKER_URL`;
- 4) наконец, вызываем процесс поиска и загрузки асинхронных задач по нашему проекту. Celery пройдет по всем приложениям, указанным в настройке `INSTALLED_APPS`, и попытается найти файл `tasks.py`, чтобы загрузить код задач.

Вам необходимо импортировать модуль `celery.py` в файле `__init__.py` проекта, чтобы он выполнялся при старте проекта. Отредактируйте файл `myshop/__init__.py` и добавьте в него такой фрагмент:

```
# Подключение Celery.
from .celery import app as celery_app
```

Теперь мы можем приступить к созданию асинхронных задач для нашего проекта.

i Настройка `CELERY_ALWAYS_EAGER` позволит вам выполнять асинхронные задачи локально в синхронном режиме вместо отправки их в очередь. Это бывает полезно для запуска юнит-тестов или запуска приложения локально без установки Celery.

Добавление асинхронных задач

Мы добавим асинхронную задачу, которая будет отправлять уведомления на электронную почту пользователей после создания заказа. Существует договоренность, что асинхронные задачи должны быть расположены в файле `tasks.py` в папке приложения.

Создайте новый файл `tasks.py` в каталоге приложения `orders`. Здесь Celery будет искать асинхронные задачи при запуске проекта. Добавьте в файл следующий фрагмент:

```
from celery import task
from django.core.mail import send_mail
from .models import Order

@task
def order_created(order_id):
    """Задача отправки email-уведомлений при успешном оформлении заказа."""
    order = Order.objects.get(id=order_id)
    subject = 'Order nr. {}'.format(order.id)
    message = 'Dear {},\n\nYou have successfully placed an order.\n\nYour order id is {}'.format(order.first_name,
                                                                    order.id)

    mail_sent = send_mail(subject,
                          message,
                          'admin@myshop.com',
                          [order.email])

    return mail_sent
```

Как можно заметить, задача – это обычная Python-функция с декоратором `task`. Мы определили задачу `order_created`, которая получает один параметр, `order_id`. Рекомендуем передавать в качестве аргументов функции только идентификаторы и получать сами объекты из базы данных лишь во время выполнения задачи. Мы обращаемся к функции Django `send_mail()`, чтобы отправить покупателю сообщение на электронную почту.

Вы уже узнали, как настроить Django на работы с SMTP-сервером, в главе 2. Если вы хотите отладить работу асинхронной задачи без использования SMTP-сервера, добавьте настройку `EMAIL_BACKEND` в файл `settings.py` проекта, как показано ниже:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```



Используйте асинхронные задачи не только для трудоемких процессов, но и в случаях, когда выполняются процессы, которые могут прерваться и потребовать повторных действий со стороны пользователя.

Теперь нужно добавить вызов асинхронной задачи в обработчик `order_create`. Откройте файл `views.py` приложения `orders`, импортируйте функцию `order_created` и вызовите ее после очистки корзины при создании заказа:

```
from .tasks import order_created

def order_create(request):
    # ...
    if request.method == 'POST':
        # ...
        if form.is_valid():
            # ...
            cart.clear()
            # Запуск асинхронной задачи.
            order_created.delay(order.id)
        # ...
```

Мы используем метод `delay()`, чтобы запустить задачу асинхронно. Она будет добавлена в очередь Celery, и ее выполнит первый освободившийся поток.

Откройте другую консоль и запустите процесс Celery из папки проекта с помощью такой команды:

```
celery -A myshop worker -l info
```

Рабочий процесс запущен и готов выполнять задачи. Убедитесь, что сервер Django тоже запущен. Откройте в браузере `http://127.0.0.1:8000/` и добавьте несколько товаров в корзину, а затем оформите заказ. В консоли, где вы запустили рабочий процесс Celery, должен будет появиться такой вывод:

```
[2017-12-17 17:43:11,462: INFO/MainProcess] Received task:
orders.tasks.order_created[e990ddae-2e30-4e36-b0e4-78bbd4f2738e]
[2017-12-17 17:43:11,685: INFO/ForkPoolWorker-4] Task
orders.tasks.order_created[e990ddae-2e30-4e36-b0e4-78bbd4f2738e] succeeded
in 0.22019841300789267s: 1
```

Процесс выполнил задачу, и вы должны были получить электронное сообщение о совершении покупки.

Мониторинг Celery


Иногда разработчику необходимо отслеживать выполнение задач. Для этих целей отлично подходит инструмент Flower. Установите его командой:

```
pip install flower==0.9.2
```

После установки для запуска Flower выполните из папки проекта команду:

```
celery -A myshop flower
```

Откройте в браузере `http://localhost:5555/dashboard`. Вы увидите активные рабочие процессы Celery и статистику по ним:



| Flower | | Dashboard | Tasks | Broker | Monitor | Logout | Docs | Code |
|-------------------------------------|--------------|-----------|--------------|------------|-----------|---------|----------------|------|
| Active: 0 | Processed: 0 | Failed: 0 | Succeeded: 0 | Retried: 0 | | | | |
| Search: <input type="text"/> | | | | | | | | |
| Worker Name | Status | Active | Processed | Failed | Succeeded | Retried | Load Average | |
| celery@MacBook-Air-de-Antonio.local | Online | 0 | 0 | 0 | 0 | 0 | 2.2, 2.4, 2.36 | |
| Showing 1 to 1 of 1 entries | | | | | | | | |

Рис. 7.12 ❖ Мониторинг Celery с помощью Flower

Полную документацию по инструменту мониторинга Flower можно найти на странице <https://flower.readthedocs.io/>.

РЕЗЮМЕ

В этой главе вы создали основу для интернет-магазина: реализовали каталог товаров, корзину и возможность оформления заказов. Вы добавили в проект собственный контекстный процессор, узнали, как работать с асинхронными задачами в Celery.

В следующей главе вы узнаете, как интегрировать в проект платежную систему, и расширите возможности сайта администрирования, добавив экспорт заказов в CSV и формирование PDF-файлов.

Глава 8

Управление заказами и платежами

В предыдущей главе мы создали проект интернет-магазина с каталогом и корзиной, подключили к проекту Celery для выполнения асинхронных задач. В этой главе начнем с добавления на сайт платежной системы, чтобы пользователи могли оплатить заказы банковской картой. Также мы расширим возможности сайта администрирования (добавим экспорт заказов в CSV и формирование счетов в формате PDF).

В этой главе мы изучим следующие темы:

- подключение платежной системы к проекту;
- экспорт заказов в CSV;
- создание собственных обработчиков для сайта администрирования;
- динамическое формирование счетов в формате PDF.

Подключение платежного шлюза

Платежный шлюз позволит пользователям вашего сайта совершать оплату покупок онлайн. С его помощью владельцы сайта могут управлять заказами покупателей. При этом обработка денежных операций происходит в защищенной системе. Вам как разработчику не придется заботиться о безопасности работы с банковскими картами.

Существует несколько платежных систем. Мы сделаем выбор в пользу Braintree, которая используется популярными сервисами, такими как Uber, Airbnb. Braintree предоставляет API, с помощью которого наше приложение сможет обрабатывать оплату от различных источников: банковских карт, PayPal, Android Pay и Apple pay. Более подробно об этой системе можно узнать на странице <https://www.braintreepayments.com/>.

Braintree может быть подключена к проекту несколькими способами. Самый простой из них – использование стандартной формы оплаты. Поскольку нам нужно переопределить поведение оплаты, мы воспользуемся размещаемыми полями (Hosted Fields). Более подробную информацию об этом способе можно найти на странице <https://developers.braintreepayments.com/guides/hosted-fields/overview/javascript/v3>.

Часть полей формы оплаты, например номер банковской карты, CVV, срок окончания ее действия, должны обрабатываться защищенной системой. Технология размещаемых полей от Braintree позволяет показать пользователю фрейм с формой оплаты, который формируется и безопасно обрабатывается на стороне платежной системы. Вам как разработчику остается только настроить отображение полей формы, а весь процесс обработки денежных операций будет происходить на стороне защищенной платежной системы.

Создание аккаунта Braintree

Вам нужно зарегистрироваться на сайте Braintree, чтобы подключить онлайн-оплату на сайте магазина. Давайте создадим аккаунт, чтобы поэкспериментировать с API Braintree. Откройте страницу <https://www.braintreepayments.com/sandbox>. Вы увидите форму регистрации:

The image shows a registration form for Braintree's sandbox. On the left, there is a heading "Test Everything Braintree" and a sub-heading "Enter our sandbox allows you to get a feel for the Braintree experience before applying for a merchant account or going to production." Below this, there is a link "Already in the sandbox? [Sign in](#)." On the right, there is a form titled "Sign up for the sandbox" with the following fields: "Full name" (with sub-fields for "First name" and "Last name"), "Company name", "Where is your business located?" (with a dropdown menu showing "Spain"), and "Email address" (with the example "me@example.com"). At the bottom of the form is a blue button labeled "Try the sandbox".

Рис. 8.1 ❖ Страница регистрации в Braintree

Заполните форму и создайте новый аккаунт. Вам на почту придет сообщение со ссылкой на страницу настройки профиля. Перейдите по ней и завершите процедуру регистрации. После этого авторизуйтесь в системе по адресу <https://sandbox.braintreegateway.com/login>. Вы увидите свой уникальный ID, а также публичный и приватный ключи, как показано ниже:

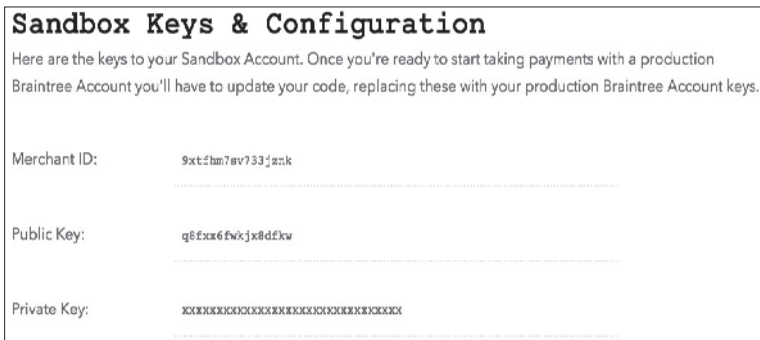


Рис.8.2 ❖ ID профиля Braintree, приватный и публичный ключи

Эти сведения понадобятся вам для того, чтобы система Braintree могла идентифицировать запросы с нашего сайта.

Установка Python-приложения Braintree

Braintree предоставляет Python-пакет, который облегчает взаимодействие с API. Исходный код этого проекта расположен в репозитории https://github.com/braintree/braintree_python. Мы подключим это приложение к нашему проекту, чтобы обращаться к API Braintree.

Выполните следующую команду для установки Python-пакета `braintree`:

```
pip install braintree==3.45.0
```

Добавьте строки конфигурации в файл настроек проекта, `settings.py`:

```
# Настройки Braintree.
BRAINTREE_MERCHANT_ID = 'XXX' # ID продавца.
BRAINTREE_PUBLIC_KEY = 'XXX' # Публичный ключ.
BRAINTREE_PRIVATE_KEY = 'XXX' # Секретный ключ.

from braintree import Configuration, Environment

Configuration.configure(
    Environment.Sandbox,
    BRAINTREE_MERCHANT_ID,
    BRAINTREE_PUBLIC_KEY,
    BRAINTREE_PRIVATE_KEY
)
```

Замените значения `BRAINTREE_MERCHANT_ID`, `BRAINTREE_PUBLIC_KEY` и `BRAINTREE_PRIVATE_KEY` на те, которые вы получили при регистрации аккаунта в системе Braintree.

i Обратите внимание, что во время разработки мы используем `Environment.Sandbox`. Когда вы захотите опубликовать ваше приложение, нужно будет зарегистрировать Braintree-аккаунт для боевого режима и заменить эту строку на `Environment.Production`. Также нужно будет указать новые ID, секретный и публичный ключи, соответствующие аккаунту.

Добавим форму оплаты в оформление заказа.

Интеграция платежного шлюза в проект

Теперь процесс оформления заказа будет состоять из трех шагов:

- добавление товаров в корзину;
- оформление заказа;
- ввод данных банковской карты и оплата.

Для этого нам нужно создать новое приложение, которое будет обрабатывать платежи. Создайте его, выполнив команду из папки проекта:

```
python manage.py startapp payment
```

Не забудьте добавить приложение `payment` в список `INSTALLED_APPS` настроек проекта `settings.py`:

```
INSTALLED_APPS = [
    # ...
    'payment.apps.PaymentConfig',
]
```

Теперь оно готово к работе.

После того как покупатель сделает заказ, нам необходимо перенаправить его на страницу оплаты. Отредактируйте файл `views.py` приложения `orders` и добавьте несколько импортов:

```
from django.urls import reverse
from django.shortcuts import render, redirect
```

В этом же файле найдите строки, формирующие страницу с ответом обработчика `order_create`:

```
# Запуск асинхронной задачи.
order_created.delay(order.id)
return render(request,
              'orders/order/created.html',
              locals())
```

Замените их, как показано в этом листинге:

```
# Запуск асинхронной задачи.
order_created.delay(order.id)
# Сохранение заказа в сессии.
request.session['order_id'] = order.id
# Перенаправление на страницу оплаты.
return redirect(reverse('payment:process'))
```

Этот код выполнит переход к оплате в случае успешного оформления заказа, ID которого мы сохраним в сессию по ключу `order_id`. Адрес, на который пользователь будет перенаправлен, определяется по имени шаблона URL'a, `payment:process`, который мы добавим чуть позже.

Не забудьте, что Celery должен быть запущен, чтобы выполнить задачу `order_created`.

Каждый раз при обработке платежа в Braintree формируется уникальный идентификатор транзакции. В нашем приложении необходимо сохранять этот идентификатор для объектов заказа, поэтому мы добавим новое поле в модель Order приложения orders. Это позволит нам однозначно связать каждый заказ с его платежной транзакцией.

Отредактируйте файл models.py приложения orders и добавьте новое поле в модель Order:

```
class Order(models.Model):
    # ...
    braintree_id = models.CharField(max_length=150, blank=True)
```

Теперь создайте миграции, чтобы поле появилось в базе данных:

```
python manage.py makemigrations
```

Вы увидите, что в миграции к модели Order добавится поле braintree_id:

```
Migrations for 'orders':
orders/migrations/0002_order_braintree_id.py
- Add field braintree_id to order
```

Выполните следующую команду:

```
python manage.py migrate
```

Миграция применится к базе данных:

```
Applying orders.0002_order_braintree_id... OK
```

Теперь приложение синхронизировано с базой данных, и мы можем сохранять идентификатор платежной транзакции для каждого заказа.

Переопределение вида платежной формы

Как уже было замечено, Braintree предоставляет возможность изменить стили и вид формы с помощью опции Hosted Fields. Мы воспользуемся JavaScript SDK от Braintree, и на странице оплаты форма ввода данных карты будет динамически формироваться во фрейме. Когда пользователь заполнит поля формы, Hosted Fields соберет данные карты и попытается сформировать защищенный токен. Если это удастся, мы сможем использовать сгенерированный токен, чтобы отправить в Braintree запрос на создание платежной транзакции.

Теперь следует добавить обработчик, который будет выполнять перечисленные ниже действия:

- 1) сформирует одноразовый токен с помощью Python-пакета braintree. Он будет использоваться, чтобы JavaScript SDK сгенерировал защищенный токен транзакции;
- 2) сформирует страницу оплаты. На ней будет загружаться JavaScript SDK от Braintree, который, зная токен из обработчика, сгенерирует защищенную форму;
- 3) при введении пользователем данных банковской карты и отправке формы JavaScript SDK сформирует защищенный токен, который будет отправлен в наш обработчик методом POST;

4) обработчик получает токен из запроса и формирует идентификатор транзакции с помощью Python-пакета braintree.

Создадим такой обработчик. Для этого отредактируйте файл `views.py` приложения `payment` и добавьте в него такой код:

```
import braintree
from django.shortcuts import render, redirect, get_object_or_404
from orders.models import Order

def payment_process(request):
    order_id = request.session.get('order_id')
    order = get_object_or_404(Order, id=order_id)

    if request.method == 'POST':
        # Получение токена для создания транзакции.
        nonce = request.POST.get('payment_method_nonce', None)
        # Создание и сохранение транзакции.
        result = braintree.Transaction.sale({
            'amount': '{:.2f}'.format(order.get_total_cost()),
            'payment_method_nonce': nonce,
            'options': {
                'submit_for_settlement': True
            }
        })
    if result.is_success:
        # Отметка заказа как оплаченного.
        order.paid = True
        # Сохранение ID транзакции в заказе.
        order.braintree_id = result.transaction.id
        order.save()
        return redirect('payment:done')
    else:
        return redirect('payment:canceled')

else:
    # Формирование одноразового токена для JavaScript SDK.
    client_token = braintree.ClientToken.generate()
    return render(request,
                  'payment/process.html',
                  {'order': order,
                  'client_token': client_token})
```

Мы определили функцию `payment_process`, которая обрабатывает процесс оплаты. Это происходит следующим образом:

- 1) получаем текущий заказ по добавленному в `order_create` после оформления покупателем заказа ключу `order_id` из сессии;
- 2) получаем соответствующий объект `Order` по переданному ID или возвращаем ответ `404 Not Found`, если не удалось найти заказ по такому ID;
- 3) если запрос происходил методом `POST`, получаем атрибут `payment_method_nonce`, с помощью которого формируем идентификатор платежной транзакции, обращаясь к методу `braintree.Transaction.sale()`. В качестве аргументов этот метод принимает:
 - `amount` (общую сумму заказа);

- `payment_method_nonce` (токен, сгенерированный Braintree для платежной транзакции. Он формируется на странице силами JavaScript SDK);
 - `options` (дополнительные параметры. Мы передали значение `submit_for_settlement`, равное `True`, благодаря чему транзакция будет обрабатываться автоматически);
- 4) если транзакция обработана успешно, отмечаем, что заказ оплачен. Для этого задаем в поле `paid` значение `True` и сохраняем уникальный ID транзакции в поле `braintree_id`. Перенаправляем пользователя на страницу шаблона URL'a `payment:done` или `payment:canceled`;
 - 5) если запрос выполнялся методом `GET`, формируем одноразовый токен для передачи в шаблон и его последующего использования в JavaScript SDK от Braintree.

Добавим обработчики, которые будут выполнять перенаправление пользователя после успешной или неудачной попытки оплаты. Добавьте эти строки в файл `views.py` приложения `payment`:

```
def payment_done(request):
    return render(request, 'payment/done.html')

def payment_canceled(request):
    return render(request, 'payment/canceled.html')
```

Теперь нужно сделать обработчики доступными извне. Для этого создайте в папке приложения `payment` новый файл, `urls.py`, и добавьте в него шаблоны URL'ов:

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

app_name = 'payment'

urlpatterns = [
    path('process/', views.payment_process, name='process'),
    path('done/', views.payment_done, name='done'),
    path('canceled/', views.payment_canceled, name='canceled'),
]
```

Это шаблоны, необходимые для работы с платежами:

- `process` (используется для формирования и обработки формы банковской карты);
- `done` (перенаправит пользователя на страницу об успешной оплате, если обработчику удалось отправить данный по платежу в Braintree);
- `canceled` (перенаправит покупателя на страницу с сообщением об ошибке, если что-то пошло не так).

Откройте файл `urls.py` проекта `myshop` и добавьте шаблоны, которые мы только что создали, как показано ниже:

```
urlpatterns = [
    # ...
    path('payment/', include('payment.urls', namespace='payment')),
    path('', include('shop.urls', namespace='shop')),
]
```

Не забудьте, что подключение шаблонов приложения payment должно происходить раньше, чем shop.urls, чтобы мы избежали нежелательных совпадений при поиске URL'ов.

Создайте отображенную ниже структуру в папке приложения payment:

```
templates/
  payment/
    process.html
    done.html
    canceled.html
```

Добавьте в файл payment/process.html такой код:

```
{% extends "shop/base.html" %}
{% block title %}Pay by credit card{% endblock %}
{% block content %}
<h1>Pay by credit card</h1>
<form action="." id="payment" method="post">
  <label for="card-number">Card Number</label>
  <div id="card-number" class="field"></div>
  <label for="cvv">CVV</label>
  <div id="cvv" class="field"></div>
  <label for="expiration-date">Expiration Date</label>
  <div id="expiration-date" class="field"></div>
  <input type="hidden" id="nonce" name="payment_method_nonce" value="">
  {% csrf_token %}
  <input type="submit" value="Pay">
</form>
<!--Подключаем клиента Braintree. -->
<script src="https://js.braintreegateway.com/web/3.29.0/js/client.min.js"></script>
<!--Подключаем компонент Hosted Fields. -->
<script src="https://js.braintreegateway.com/web/3.29.0/js/hosted-fields.min.js"></script>
<script>
  var form = document.querySelector('#payment');
  var submit = document.querySelector('input[type="submit"]');

  braintree.client.create({
    authorization: '{{ client_token }}'
  }, function (clientErr, clientInstance) {
    if (clientErr) {
      console.error(clientErr);
      return;
    }
  })

  braintree.hostedFields.create({
    client: clientInstance,
    styles: {
      'input': {'font-size': '13px'},
      'input.invalid': {'color': 'red'},
```

```

        'input.valid': {'color': 'green'}
    },
    fields: {
        number: {selector: '#card-number'},
        cvv: {selector: '#cvv'},
        expirationDate: {selector: '#expiration-date'}
    }
}, function (hostedFieldsErr, hostedFieldsInstance) {

    if (hostedFieldsErr) {
        console.error(hostedFieldsErr);
        return;
    }

    submit.removeAttribute('disabled');

    form.addEventListener('submit', function (event) {
        event.preventDefault();

        hostedFieldsInstance.tokenize(function (tokenizeErr, payload) {
            if (tokenizeErr) {
                console.error(tokenizeErr);
                return;
            }
            // Задаем значение поля для отправки токена на сервер.
            document.getElementById('nonce').value = payload.nonce;
            // Отправляем форму на сервер.
            document.getElementById('payment').submit();
        });
    }, false);
});
</script>
{% endblock %}

```

Это шаблон, который отображает платежную форму и формирует токен для дальнейшего создания транзакции. Мы определили `<div>`-элемент и несколько внутренних `<input>`-элементов для полей данных банковской карты, номера, CVV и даты окончания действия. Кроме этого, мы создали `<input>`-элемент `payment_method_nonce`, в который будет сохраняться одноразовый токен, сформированный JavaScript-клиентом Braintree.

В шаблоне загружаем два файла (Braintree JavaScript SDK, `client.min.js`) и компонент `Hosted Fields`, `hosted-fields.min.js`. Затем определяем код, который выполняет следующие шаги:

- 1) создает Braintree-клиента с помощью метода `braintree.client.create()`, передавая в него токен, сгенерированный в обработчике `payment_process`;
- 2) создает компонент `Hosted Fields`, вызывая метод `braintree.hostedFields.create()`;

- 3) определяет CSS-стили для `<input>`-элементов;
- 4) задает идентификаторы `id` для таких элементов, как `card-number`, `cvv`, `expiration-date`;
- 5) создает обработчик на событие отправки формы. Когда пользователь соберется отправить информацию на сервер, по введенным данным банковской карты будет сформирован токен для поля `payment_method_nonce`, после чего форма будет отправлена.

Отредактируйте файл `payment/done.html` и добавьте в него такой код:

```
{% extends "shop/base.html" %}
{% block content %}
  <h1>Your payment was successful</h1>
  <p>Your payment has been processed successfully.</p>
{% endblock %}
```

Эту страницу увидит покупатель в случае успешной обработки платежа. Добавьте в файл `payment/canceled.html` следующий фрагмент:

```
{% extends "shop/base.html" %}
{% block content %}
  <h1>Your payment has not been processed</h1>
  <p>There was a problem processing your payment.</p>
{% endblock %}
```

В том случае, если не удалось создать платежную транзакцию, пользователь будет перенаправлен на эту страницу. Давайте протестируем, что у нас получилось.

Тестирование платежей

Откройте консоль и запустите RabbitMQ, выполнив команду:

```
rabbitmq-server
```

Откройте еще одну консоль и запустите Celery из папки проекта:

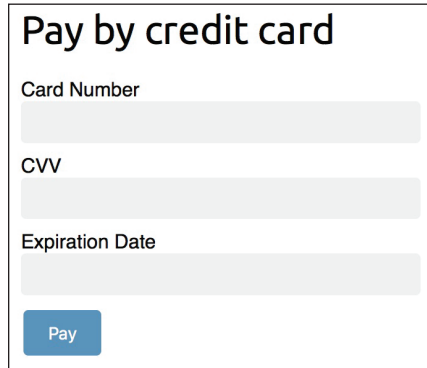
```
celery -A myshop worker -l info
```

Откройте третью консоль и запустите сервер для разработки командой:

```
python manage.py runserver
```

Перейдите в браузере на страницу: `http://127.0.0.1:8000/`, добавьте в корзину несколько товаров и заполните форму оформления заказа. Затем нажмите на кнопку **PLACE ORDER**. В этот момент выполнится обработчик `order_create`, благодаря чему заказ будет сохранен в базе данных, его ID добавлен в сессию, а вас перенаправят на страницу ввода данных банковской карты.

Страница оплаты получает ID заказа из сессии и готовит форму с помощью Hosted Fields во фрейме, как показано ниже:



Pay by credit card

Card Number

CVV

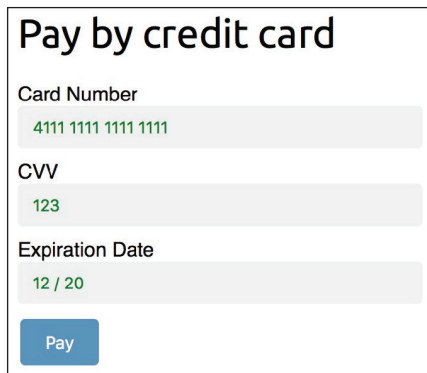
Expiration Date

Pay

Рис. 8.3 ❖ Форма оплаты заказа

Через инструменты отладки браузера вы можете посмотреть на HTML, который формируется для полей оплаты.

Braintree предоставляет тестовые данные банковских карт, чтобы разработчики могли попробовать различные сценарии оплаты (как успешные, так и ошибочные). Список доступных карт вы можете найти на странице <https://developers.braintreepayments.com/guides/credit-cards/testing-go-live/python>. Мы воспользуемся банковской картой VISA 4111 1111 1111 1111, которая используется для успешной оплаты. Ее CVV равен 123, а дата окончания действия – 12/24. Введите эти данные в форму, как показано ниже:



Pay by credit card

Card Number

CVV

Expiration Date

Pay

Рис. 8.4 ❖ Тестовые данные банковской карты

Нажмите кнопку **Pay**, и вы увидите такую страницу:



Рис. 8.5 ❖ Оплата прошла успешно

Платеж был обработан. Теперь вы можете зайти в свой аккаунт на сайте <https://sandbox.braintreegateway.com/login> и увидеть список созданных транзакций в разделе Transactions:


| ID | Transaction Date | Type | Status | Customer | Payment Information | Amount |
|----------|-------------------------------|------|-----------------------------|----------|--|-------------|
| 2bwkx5b6 | 02/05/2018 07:45:23 PM CST | Sale | Submitted For Settlement | |  411111*****1111 | 21,20 € EUR |

Рис. 8.6 ❖ Список платежных транзакций

Откройте страницу <http://127.0.0.1:8000/admin/orders/order/>. Вы увидите, что для созданного заказа сохранены ID транзакции и поле `paid`, равное `True`.

Paid

Braintree id:

Рис. 8.7 ❖ Сведения об оплате для заказа на сайте администрирования

Поздравляем! Вы подключили к проекту оплату через платежный шлюз.

Запуск в боевом режиме

После тестирования работы приложения вы можете создать полноценный аккаунт Braintree на странице <https://www.braintreepayments.com>. Как только вы будете готовы запускать проект в боевом режиме, не забудьте заменить в настройках `settings.py` ключи и ID приложения для доступа к Braintree, а также

подключить окружение `braintree.Environment.Production`. Пошаговую инструкцию, как настроить Braintree на работу с приложением в боевом режиме, можно найти на странице <https://developers.braintreepayments.com/start/go-live/python>.

Экспорт заказов в CSV-файл

Иногда в проекте вам может понадобиться экспорт информации, хранящейся в моделях, чтобы, например, загрузить их в другую систему и обрабатывать там. Одним из наиболее часто используемых для импорта и экспорта форматов является формат, хранящий данные в виде значений, разделенных запятыми (*Comma-separated Values, CSV*). CSV – это текстовый файл, состоящий из записей. Обычно каждая запись находится на отдельной строке, а поля записи разделены между собой символом, чаще всего запятой. Мы добавим на сайт администрирования возможность экспортировать заказы в CSV-файлы.

Добавление собственных действий на сайте администрирования

Django предоставляет широкие возможности по настройке сайта администрирования. Мы воспользуемся этим, чтобы изменить страницу отображения списка заказов и добавить действие по экспорту заказов.

На сайте администрирования пользователь может совершать различные действия над списком объектов. Для этого ему достаточно отметить элементы и выбрать, какое действие выполнить. В интерфейсе это выглядит следующим образом:

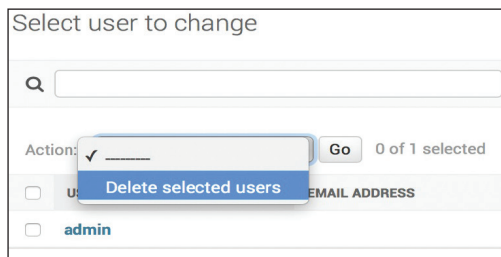


Рис. 8.8 ❖ Действия для списка объектов на сайте администрирования

- ☑ Создавайте собственные действия, чтобы администраторы сайта могли выполнять их для нескольких элементов сразу, а не отдельно для каждого.

Создать собственное действие – это значит создать функцию, которая принимает указанные ниже аргументы:

- объект `ModelAdmin` модели, которая отображается;
- объект запроса `request` типа `HttpRequest`;
- `QuerySet` объектов, которые выбрал пользователь.

Эта функция будет выполняться, когда пользователь выберет действие на сайте администрирования.

Мы создадим функцию, позволяющую скачать CSV-файл заказов. Отредактируйте файл `admin.py` приложения `orders` и добавьте такой код после определения класса `OrderAdmin`:

```
import csv
import datetime
from django.http import HttpResponse

def export_to_csv(modeladmin, request, queryset):
    opts = modeladmin.model._meta
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment;\n'
        'filename={}.csv'.format(opts.verbose_name)
    writer = csv.writer(response)

    fields = [field for field in opts.get_fields() if not field.many_to_many\
        and not field.one_to_many]
    # Записываем первую строку с заголовками полей.
    writer.writerow([field.verbose_name for field in fields])
    # Записываем данные.
    for obj in queryset:
        data_row = []
        for field in fields:
            value = getattr(obj, field.name)
            if isinstance(value, datetime.datetime):
                value = value.strftime('%d/%m/%Y')
            data_row.append(value)
        writer.writerow(data_row)
    return response
export_to_csv.short_description = 'Export to CSV'
```

В этой функции мы выполняем такие шаги:

- 1) создаем объект ответа класса `HttpResponse` с типом содержимого `text/csv`, чтобы браузер работал с файлом так же, как с CSV. Добавляем заголовок `Content-Disposition`, т. к. к ответу будет прикреплен файл;
- 2) создаем объект `writer`, который будет записывать данные файла в объект `response`;
- 3) динамически получаем поля модели с помощью метода `get_fields()` опций `meta` модели, исключая отношения «многие ко многим» и «один ко многим»;
- 4) заполняем строку-заголовок названиями полей;
- 5) проходим по каждому выбранному пользователем элементу и записываем его данные в строку. При этом выполняем форматирование объектов даты `datetime`, т. к. выходные данные для CSV должны быть представлены в виде строки;
- 6) определяем, как действие будет показано на сайте администрирования, с помощью атрибута `short_description` для функции.

Мы создали действие, которое не связано с конкретной моделью, поэтому может быть использовано многократно для любого объекта `ModelAdmin`.

Наконец, добавьте созданную функцию `export_to_csv` в список действий для `OrderAdmin`, как показано ниже:

```
class OrderAdmin(admin.ModelAdmin):
    # ...
    actions = [export_to_csv]
```

Откройте `http://127.0.0.1:8000/admin/orders/order/`, теперь список заказов должен выглядеть таким образом:

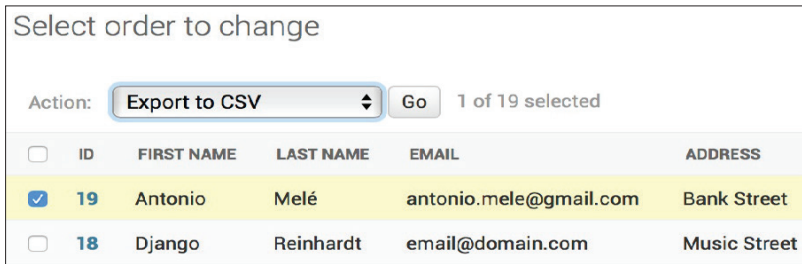


Рис. 8.9 ❖ Добавленное действие отображается на сайте администрирования

Выберите несколько заказов и попробуйте экспортировать их в CSV. Браузер начнет скачивать сформированный файл `order.csv`. Откройте его в текстовом редакторе, и вы увидите содержимое с заголовками:

```
ID,first name,last name,email,address,postal
code,city,created,updated,paid,braintree id
3,Antonio,Melé,antonio.mele@gmail.com,Bank Street,WS
J11,London,25/02/2018,25/02/2018,True,2bwkx5b6
...
```

Как вы можете заметить, добавление собственных действий на сайт администрирования – предельно простой процесс. Более подробно про формирование CSV-файлов с Django можно узнать на странице <https://docs.djangoproject.com/en/2.0/howto/outputting-csv/>.

РАСШИРЕНИЕ САЙТА АДМИНИСТРИРОВАНИЯ

Иногда вам нужно изменить сайт администрирования так, как это невозможно реализовать настройками `ModelAdmin`, созданием собственных действий или перепределением HTML-шаблонов. В этом случае придется добавить собственный обработчик для сайта администрирования. С его помощью вы реализуете необходимые возможности. Особенность такого подхода заключается в том, что к вашему обработчику будут иметь доступ только пользователи, которым

доступен сайт администрирования. В интерфейсе страницы выглядят так же, как и стандартные страницы Django.

Давайте создадим собственный обработчик, чтобы отображать информацию о заказе. Отредактируйте файл `views.py` приложения `orders`. Добавьте в него такой фрагмент:

```
from django.contrib.admin.views.decorators import staff_member_required
from django.shortcuts import get_object_or_404
from .models import Order

@staff_member_required
def admin_order_detail(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    return render(request,
                  'admin/orders/order/detail.html',
                  {'order': order})
```

Декоратор `staff_member_required` проверяет, что у пользователя в полях `is_active` и `is_staff` сохранено значение `True`. В этой функции мы получаем объект `Order` по переданному ID заказа и формируем HTML-шаблон.

Теперь отредактируйте файл `urls.py` приложения `orders` и добавьте такой шаблон для URL'a:

```
path('admin/order/', views.admin_order_detail, name='admin_order_detail'),
```

В папке `templates/` приложения `orders` создайте следующую структуру папок и файлов:

```
admin/
  orders/
    order/
      detail.html
```

Откройте файл `detail.html` и вставьте в него фрагмент:

```
{% extends "admin/base_site.html" %}
{% load static %}

{% block extrastyle %}
  <link rel="stylesheet" type="text/css" href="{% static "css/admin.css" %}" />
{% endblock %}

{% block title %}
  Order {{ order.id }} {{ block.super }}
{% endblock %}

{% block breadcrumbs %}
  <div class="breadcrumbs">
    <a href="{% url "admin:index" %}">Home</a> &rsaquo;
    <a href="{% url "admin:orders_order_changelist" %}">Orders</a>
    &rsaquo;
    <a href="{% url "admin:orders_order_change" order.id %}">Order {{ order.id }}</a>
    &rsaquo; Detail
  </div>
```

```

{% endblock %}
{% block content %}
<h1>Order {{ order.id }}</h1>
<ul class="object-tools">
  <li>
    <a href="#" onclick="window.print();">Print order</a>
  </li>
</ul>
<table>
  <tr>
    <th>Created</th>
    <td>{{ order.created }}</td>
  </tr>
  <tr>
    <th>Customer</th>
    <td>{{ order.first_name }} {{ order.last_name }}</td>
  </tr>
  <tr>
    <th>E-mail</th>
    <td><a href="mailto:{{ order.email }}">{{ order.email }}</a></td>
  </tr>
  <tr>
    <th>Address</th>
    <td>{{ order.address }}, {{ order.postal_code }} {{ order.city }}</td>
  </tr>
  <tr>
    <th>Total amount</th>
    <td>${{ order.get_total_cost }}</td>
  </tr>
  <tr>
    <th>Status</th>
    <td>{% if order.paid %}Paid{% else %}Pending payment{% endif %}</td>
  </tr>
</table>
<div class="module">
  <div class="tabular inline-related last-related">
    <table>
      <h2>Items bought</h2>
      <thead>
        <tr>
          <th>Product</th>
          <th>Price</th>
          <th>Quantity</th>
          <th>Total</th>
        </tr>
      </thead>
      <tbody>
        {% for item in order.items.all %}
          <tr class="row{% cycle "1" "2" %}">
            <td>{{ item.product.name }}</td>
            <td class="num">${{ item.price }}</td>
          </tr>
        {% endfor %}
      </tbody>
    </table>
  </div>
</div>

```

```

        <td class="num">{{ item.quantity }}</td>
        <td class="num">${{ item.get_cost }}</td>
    </tr>
    {% endfor %}
    <tr class="total">
        <td colspan="3">Total</td>
        <td class="num">${{ order.get_total_cost }}</td>
    </tr>
</tbody>
</table>
</div>
</div>
{% endblock %}

```

Это шаблон для отображения подробностей заказа. Он наследуется от базового шаблона Django для сайта администрирования, `admin/base_site.html`, поэтому стили и структура HTML будут схожи со стандартной страницей. Мы также подключаем файл собственных стилей, `css/admin.css`.

Чтобы использовать статические файлы, скопируйте их из кода – примера к этой главе. Они находятся в папке `static/` приложения `orders`. Добавьте их в такую же папку в вашем приложении.

Мы используем блоки, определенные в базовом шаблоне, чтобы отображать содержимое (информацию о заказе и купленных товарах).

Когда вы расширяете стандартные страницы сайта администрирования Django, вам нужно знать, какие блоки используются в базовом шаблоне. Для этого нужно перейти по адресу: <https://github.com/django/django/tree/2.0/django/contrib/admin/templates/admin>.

Вы можете также переопределить шаблоны. Чтобы сделать это, достаточно воспроизвести иерархию папок сайта администрирования Django в каталоге `templates/` вашего приложения. Тогда Django будет находить ваши файлы первыми и использовать их.

Давайте для каждого заказа в списке добавим ссылку на страницу подробностей. Отредактируйте файл `admin.py` приложения `orders` и вставьте в него такой код:

```

from django.urls import reverse
from django.utils.safestring import mark_safe

def order_detail(obj):
    return mark_safe('<a href="{}">View</a>'.format(
        reverse('orders:admin_order_detail', args=[obj.id])))

```

Эта функция получает в качестве аргумента объект заказа и возвращает ссылку обработчика `admin_order_detail`. Мы используем функцию `mark_safe()`, чтобы экранировать HTML-тег ссылки.



Используйте `mark_safe()`, чтобы избежать пропуска HTML-тегов. При этом важно не применять эту функцию к данным, введенным пользователями сайта, чтобы недобросовестные пользователи не могли вставить выполняемый код.

Добавьте функцию в класс `OrderAdmin`, чтобы ссылка появилась на странице списка:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id',
                  'first_name',
                  # ...
                  'updated',
                  'order_detail']
```

Откройте в браузере страницу `http://127.0.0.1:8000/admin/orders/order/`. Теперь каждый заказ содержит ссылку **View**:

| PAID | CREATED | UPDATED | ORDER DETAIL |
|------|-------------------------|-------------------------|----------------------|
| ✓ | Feb. 6, 2018, 1:35 a.m. | Feb. 6, 2018, 1:45 a.m. | View |

Рис. 8.10 ❖ Ссылка на страницу подробностей заказа

Кликните на нее, и откроется страница с HTML, который сформирован нашим собственным обработчиком:

| Django administration | | | |
|-----------------------------------|--|----------|-------------|
| Home · Orders · Order 19 · Detail | | | |
| Order 19 | | | PRINT ORDER |
| Created | Feb. 6, 2018, 1:35 a.m. | | |
| Customer | Antonio Melé | | |
| E-mail | antonio.mele@gmail.com | | |
| Address | Jazz Street, 28027 Madrid | | |
| Total amount | \$21.2 | | |
| Status | Paid | | |
| Items bought | | | |
| PRODUCT | PRICE | QUANTITY | TOTAL |
| Tea powder | \$21.2 | 1 | \$21.2 |
| Total | | | \$21.2 |

Рис. 8.11 ❖ Переопределенная страница подробностей заказа

ГЕНЕРАЦИЯ PDF-СЧЕТОВ

На текущий момент мы реализовали для магазина оформление и оплату заказов. Теперь нужно добавить возможность получить счет о покупке, который будем оформлять в PDF-документ. Существует несколько Python-библиотек

для работы с этим форматом. Одна из часто используемых – Reportlab. Более подробную информацию о формировании PDF в Django вы можете найти на странице <https://docs.djangoproject.com/en/2.0/howto/outputting-pdf/>.

В большинстве случаев требуется добавить стили оформления в PDF-документ. Для этих целей очень хорошо подходит предварительная генерация HTML, который затем сохраняется в виде PDF-файла. Мы воспользуемся этим способом и подключим Python-пакет WeasyPrint, который даст возможность работать с PDF из Django-проекта.

Установка WeasyPrint

Для начала установите зависимости, необходимые для работы WeasyPrint, следуя инструкциям для вашей операционной системы на странице <https://weasyprint.readthedocs.io/en/latest/install.html>. Затем установите пакет с помощью pip:

```
pip install WeasyPrint==0.42.3
```

Создание PDF-шаблона

Нам понадобится HTML-шаблон, который обработает шаблонизатор Django, а затем мы преобразуем его в PDF-документ с помощью WeasyPrint.

Создайте новый файл, pdf.html, в папке templates/orders/order/ приложения orders и добавьте в него такой фрагмент:

```
<html>
  <body>
    <h1>My Shop</h1>
    <p>
      Invoice no. {{ order.id }}<br>
      <span class="secondary">
        {{ order.created|date:"M d, Y" }}
      </span>
    </p>

    <h3>Bill to</h3>
    <p>
      {{ order.first_name }} {{ order.last_name }}<br>
      {{ order.email }}<br>
      {{ order.address }}<br>
      {{ order.postal_code }}, {{ order.city }}
    </p>

    <h3>Items bought</h3>
    <table>
      <thead>
        <tr>
          <th>Product</th>
          <th>Price</th>
          <th>Quantity</th>
          <th>Cost</th>
        </tr>
```



```

</thead>
<tbody>
  {% for item in order.items.all %}
    <tr class="row{% cycle "1" "2" %}">
      <td>{{ item.product.name }}</td>
      <td class="num">${{ item.price }}</td>
      <td class="num">{{ item.quantity }}</td>
      <td class="num">${{ item.get_cost }}</td>
    </tr>
  {% endfor %}
  <tr class="total">
    <td colspan="3">Total</td>
    <td class="num">${{ order.get_total_cost }}</td>
  </tr>
</tbody>
</table>
<span class="{% if order.paid %}paid{% else %}pending{% endif %}">
  {% if order.paid %}Paid{% else %}Pending payment{% endif %}
</span>
</body>
</html>

```

Это шаблон для формирования PDF-счета об оплаченном заказе. Мы отображаем подробные сведения о заказе внутри `<table>`-элемента и формируем сообщение о состоянии оплаты.

Формирование PDF-файлов

Следует добавить обработчик, который будет создавать документ через интерфейс сайта администрирования. Откройте файл `views.py` приложения `orders` и добавьте в него следующий фрагмент:

```

from django.conf import settings
from django.http import HttpResponse
from django.template.loader import render_to_string
import weasyprint

@staff_member_required
def admin_order_pdf(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    html = render_to_string('orders/order/pdf.html',
                          {'order': order})
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'filename=\
    "order_{}.pdf".format(order.id)
    weasyprint.HTML(string=html).write_pdf(response,
    stylesheets=[weasyprint.CSS(
    settings.STATIC_ROOT + 'css/pdf.css')])
    return response

```

Мы применили декоратор `staff_member_required`, чтобы доступ к функции имели только администраторы сайта. Чтобы сформировать документ, мы получаем заказ по ID и передаем его в функцию `render_to_string()`, которая воз-

вращает сгенерированный из шаблона `orders/order/pdf.html` HTML в виде строки. Мы заносим эту строку в переменную `html`, затем создаем объект ответа, `HttpResponse`, с типом содержимого `application/pdf` и заголовком `Content-Disposition`. Потом вызываем `WeasyPrint`, чтобы получить PDF-документ из строки `html`, и прикрепляем его к ответу. Для стилизации документа подключаем стили `css/pdf.css`. Файл физически создается в файловой системе по пути, определенному в настройке `STATIC_ROOT` файла `settings.py`. По окончании формирования документа отправляем объект ответа.

Если у вас не применились стили, убедитесь, что они скопированы из кода – примера к этой главе и находятся в папке `static/` приложения `shop`.

Так как мы используем настройку `STATIC_ROOT`, пора ее добавить. Это путь, благодаря которому Django будет искать статические файлы (стили, скрипты JavaScript, шрифты и картинки сайта). Отредактируйте файл `settings.py` проекта `myshop` и добавьте такую строку:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

Затем выполните команду:

```
python manage.py collectstatic
```

Вы увидите вывод, который заканчивается такой фразой:

```
120 static files copied to 'code/myshop/static'.
```

Команда `collectstatic` копирует все статические файлы приложения в общую для проекта папку, заданную этой настройкой. Так разработчики могут размещать файлы для каждого приложения в его отдельной папке `static/`. Вы можете добавить дополнительные источники статических файлов с помощью настройки `STATICFILES_DIRS`. Содержимое всех папок, заданных через эту переменную конфигурации, будет также скопировано в общий каталог. При последующих вызовах команды `collectstatic` Django будет предлагать заменить существующие файлы. Так вы сможете поддерживать их актуальность.

Отредактируйте файл `urls.py` приложения `orders` и добавьте следующий код:

```
urlpatterns = [
    # ...
    path('admin/order/<int:order_id>/pdf/',
         views.admin_order_pdf,
         name='admin_order_pdf'),
]
```

Теперь мы можем добавить ссылку на PDF-файл для каждого заказа из списка. Откройте файл `admin.py` приложения `orders` и добавьте такие строки в класс `OrderAdmin`:

```
def order_pdf(obj):
    return mark_safe('<a href="{}">PDF</a>'.format(
        reverse('orders:admin_order_pdf', args=[obj.id])))
order_pdf.short_description = 'Invoice'
```

Если вы укажете атрибут `short_description` для метода класса `ModelAdmin`, Django будет использовать его для заголовка соответствующей колонки в таблице объектов списка.

Добавьте метод `order_pdf` в список `list_display` класса `OrderModel`:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id',
                  # ...
                  order_detail,
                  order_pdf]
```

Откройте в браузере `http://127.0.0.1:8000/admin/orders/order/`. Теперь каждая строка таблицы заказов содержит ссылку на PDF-документ:

| UPDATED | ORDER DETAIL | INVOICE |
|--------------------------|----------------------|---------------------|
| Feb. 11, 2018, 3:17 p.m. | View | PDF |

Рис. 8.12 ❖ Ссылка на PDF-документ для заказа на сайте администрирования

Кликните на ссылку PDF для любого заказа. Вы увидите сформированный PDF-файл с подробностями заказа и отметкой о статусе оплаты:

My Shop

Invoice no. 16
Feb 01, 2018

Bill to

Antonio Mele
antonio.mele@gmail.com
Jazz Street
28033, Madrid

Items bought

| Product | Price | Quantity | Cost |
|--------------|-------|----------|-------------|
| Green tea | | \$30 | 1 \$30 |
| Total | | | \$30 |




Рис. 8.13 ❖ Информация о заказе в PDF

Если оплата заказа успешно обработана, то документ заказа будет выглядеть так:

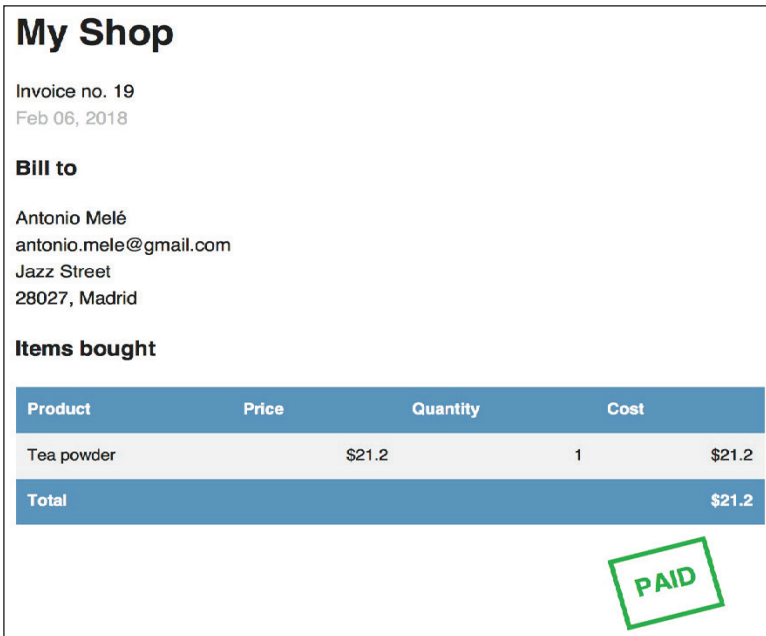


Рис. 8.14 ❖ Информация об оплаченном заказе в PDF

Отправка PDF-файла на электронную почту

Если оплата прошла успешно, покупателю на электронную почту автоматически придет PDF-счет с подробностями заказа. Чтобы получить такую функциональность, откройте файл `views.py` приложения `payment` и добавьте в него следующий код:

```
from django.template.loader import render_to_string
from django.core.mail import EmailMessage
from django.conf import settings
import weasyprint
from io import BytesIO
```

Затем измените обработчик `payment_process`, добавив после строки `order.save()` фрагмент кода для формирования PDF-документа заказа:

```
def payment_process(request):
    # ...
    if request.method == 'POST':
        # ...
        if result.is_success:
            # ...
```

```

order.save()
# Создание электронного сообщения.
subject = 'My Shop - Invoice no. {}'.format(order.id)
message = 'Please, find attached the invoice for your recent purchase.'
email = EmailMessage(subject,
                    message,
                    'admin@myshop.com',
                    [order.email])

# Формирование PDF.
html = render_to_string('orders/order/pdf.html', {'order': order})
out = BytesIO()
stylesheets=[weasyprint.CSS(settings.STATIC_ROOT + 'css/pdf.css')]
weasyprint.HTML(string=html).write_pdf(out, stylesheets=stylesheets)
# Прикрепляем PDF к электронному сообщению.
email.attach('order_{}.pdf'.format(order.id),
            out.getvalue(),
            'application/pdf')

# Отправка сообщения.
email.send()

return redirect('payment:done')
else:
    return redirect('payment:canceled')

else:
    # ...

```

Мы используем класс `EmailMessage`, чтобы создать объект `email`. Затем генерируем строку с HTML и помещаем ее в переменную `html`, формируем из этого шаблона PDF-документ и записываем его в объект `BytesIO`, который будет временно сохранен в оперативной памяти. В заключение прикрепляем созданный PDF-документ к электронному сообщению, вызывая метод `EmailMessage.attach()` с названием файла, его содержимым из `BytesIO` и типом данных в качестве аргументов, и отправляем сообщение покупателю.

Не забудьте, что проект должен быть настроен на взаимодействие с SMTP-сервером. Вспомнить, какие настройки необходимо использовать для этого, можно, прочитав главу 2.

РЕЗЮМЕ

В этой главе вы подключили к проекту платежную систему, создали собственное действие для административного сайта Django и научились динамически формировать CSV- и PDF-файлы.

В следующей главе узнаем, как добавить переводы и локализацию, чтобы пользователи из разных стран могли пользоваться сайтом на родном языке, а также реализуем систему купонов и подбор рекомендаций для покупателей.

Глава 9

Расширение онлайн-магазина

В предыдущей главе вы узнали, как подключить к магазину платежную систему и формировать CSV-файлы и PDF-документы. В этой главе мы узнаем, как добавить в магазин систему купонов и скидок, воспользоваться возможностями Django по локализации проекта и добавить рекомендации товаров.

Мы разберем следующие темы:

- создание системы купонов и скидок;
- добавление переводов на сайте;
- использование Rosetta для управления переводами;
- локализация моделей с помощью `django-parler`;
- добавление рекомендаций товаров для покупателей.

РЕАЛИЗАЦИЯ СИСТЕМЫ КУПОНОВ

Многие интернет-магазины поддерживают систему купонов для предоставления скидок покупателям. Обычно купон – это некоторый уникальный код, который действует в течение ограниченного периода времени и может быть использован один или несколько раз.

Мы добавим в магазин такую систему. Купоны будут предоставлять скидку на любое количество товаров. Их можно использовать несколько раз, но ограниченное время. Чтобы реализовать такую функциональность, нам понадобится модель для сохранения данных купона: кода, временных ограничений и размера скидки.

Создайте новое приложение, выполнив следующую команду внутри проекта `myshop`:

```
python manage.py startapp coupons
```

Отредактируйте файл `settings.py` проекта `myshop` и добавьте новое приложение в список установленных:

```
INSTALLED_APPS = [
    # ...
    'coupons.apps.CouponsConfig',
]
```

Приложение купонов активировано.

Создание моделей

Давайте создадим модель `Coupon`. Отредактируйте файл `models.py` приложения `coupons` и добавьте в него такой код:

```
from django.db import models
from django.core.validators import MinValueValidator, \
    MaxValueValidator

class Coupon(models.Model):
    code = models.CharField(max_length=50, unique=True)
    valid_from = models.DateTimeField()
    valid_to = models.DateTimeField()
    discount = models.IntegerField(
        validators=[MinValueValidator(0), MaxValueValidator(100)])
    active = models.BooleanField()

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

С помощью этой модели мы будем хранить информацию о купонах. Давайте разберем, какие поля нам для этого понадобятся:

- `code` – код, который будут использовать покупатели;
- `valid_from` – дата и время начала действия купона;
- `valid_to` – дата и время окончания действия купона;
- `discount` – размер скидки в процентах, принимающий значения от 0 до 100. Чтобы проверять введенные значения, мы добавили валидаторы;
- `active` – булево поле для отображения активности купона.

Запустите команду создания миграции для приложения `coupons`:

```
python manage.py makemigrations
```

Вы увидите, что новая миграция создаст в базе данных модель `Coupon`:

```
Migrations for 'coupons':
  coupons/migrations/0001_initial.py:
    - Create model Coupon
```

Затем выполните команду для применения миграции:

```
python manage.py migrate
```

Вы увидите вывод, содержащий такую строку:

```
Applying coupons.0001_initial... OK
```

Это значит, что миграции успешно применились. Давайте добавим новую модель на сайт администрирования. Отредактируйте файл `admin.py` приложения `coupons` и вставьте в него такой код:

```

from django.contrib import admin
from .models import Coupon

class CouponAdmin(admin.ModelAdmin):
    list_display = ['code', 'valid_from', 'valid_to', 'discount', 'active']
    list_filter = ['active', 'valid_from', 'valid_to']
    search_fields = ['code']
admin.site.register(Coupon, CouponAdmin)

```

Теперь модель `Coupon` зарегистрирована на сайте администрирования. Убедитесь, что сервер разработки запущен командой `python manage.py runserver`, и откройте страницу `http://127.0.0.1:8000/admin/coupons/coupon/add/`. Вы увидите такую форму:

The screenshot shows the Django administration interface for adding a coupon. The page title is "Django administration" and the breadcrumb is "Home > Coupons > Coupons > Add coupon". The form includes the following fields:

- Code:** A text input field.
- Valid from:** A date and time selection section. It includes a "Date:" field with a "Today" button and a calendar icon, and a "Time:" field with a "Now" button and a clock icon. A note below reads: "Note: You are 1 hour ahead of server time."
- Valid to:** A date and time selection section, identical in structure to the "Valid from" section, with a note: "Note: You are 1 hour ahead of server time."
- Discount:** A percentage input field with a percentage sign icon.
- Active:** A checkbox.

At the bottom of the form are three buttons: "Save and add another", "Save and continue editing", and "SAVE".

Рис. 9.1 ❖ Страница редактирования купона на сайте администрирования

Заполните форму корректными данными и поставьте чекбокс **Active**, после чего нажмите кнопку **SAVE**.

Добавление оплаты купонами

Мы реализовали создание купонов и просмотр информации о них. Теперь нам нужно добавить возможность использовать купон при оплате покупки. Этот процесс состоит из следующих шагов:

- 1) пользователь добавляет товары в корзину;
- 2) пользователю доступен ввод кода купона на странице корзины;

- 3) когда покупатель оформляет заказ, заполнив код купона, мы пытаемся найти действующий купон по введенной информации (он должен быть активен, а текущая дата должна находиться между `valid_from` и `valid_to`);
- 4) если купон найден, сохраняем его в сессию и отображаем корзину с учетом скидки;
- 5) когда пользователь оформляет заказ, привязываем к нему купон.

Создайте новый файл, `forms.py`, в папке приложения `coupons` и добавьте в него такой код:

```
from django import forms

class CouponApplyForm(forms.Form):
    code = forms.CharField()
```

Это форма, которую мы будем использовать для предоставления пользователям возможности ввести код купона. Отредактируйте файл `views.py` приложения `coupons` и добавьте следующий фрагмент:

```
from django.shortcuts import render, redirect
from django.utils import timezone
from django.views.decorators.http import require_POST
from .models import Coupon
from .forms import CouponApplyForm

@require_POST
def coupon_apply(request):
    now = timezone.now()
    form = CouponApplyForm(request.POST)
    if form.is_valid():
        code = form.cleaned_data['code']
        try:
            coupon = Coupon.objects.get(
                code__iexact=code,
                valid_from__lte=now,
                valid_to__gte=now,
                active=True)
            request.session['coupon_id'] = coupon.id
        except Coupon.DoesNotExist:
            request.session['coupon_id'] = None
    return redirect('cart:cart_detail')
```

Обработчик `coupon_apply` проверяет данные купона и сохраняет его в сессии пользователя. Мы используем декоратор `require_POST`, чтобы функция `coupon_apply` была доступна только при запросах методом `POST`. Давайте посмотрим, что происходит в этом обработчике:

- 1) мы создаем форму `CouponApplyForm` на основе переданных данных;
- 2) если форма валидна, получаем из словаря `cleaned_data` код `code`, введенный пользователем, и пытаемся получить соответствующий активный купон. Мы используем запрос вида `iexact`, чтобы проверить код без учета регистра. Купон должен быть активным (`active=True`) и подходящим по периоду действия. Мы используем функцию Django `timezone.now()`, чтобы

получить объект дат и времени с учетом временной зоны, а затем сравниваем его с датами начала и окончания действия купона (`valid_from` через запрос `lte`, меньше или равно, и `valid_to` через запрос `gte`, больше или равно, соответственно);

- 3) сохраняем ID купона в сессии;
- 4) перенаправляем пользователя на страницу по шаблону URL'а с именем `cart_detail`.

Нужно добавить шаблон URL'а для обработчика `coupon_apply`. Для этого создайте в папке приложения `coupons` файл `urls.py` и вставьте в него такой код:

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

app_name = 'coupons'

urlpatterns = [
    path('apply/', views.coupon_apply, name='apply'),
]
```

Затем отредактируйте файл `urls.py` проекта `myshop` и подключите файл `urls.py` приложения `coupons`, как показано ниже:

```
urlpatterns = [
    # ...
    path('coupons/', include('coupons.urls', namespace='coupons')),
    path('', include('shop.urls', namespace='shop')),
]
```

Не забудьте вставить его выше, чем шаблоны URL'ов приложения `shop`.

Теперь отредактируйте файл `cart.py` приложения `cart` и импортируйте модель купона:

```
from coupons.models import Coupon
```

Добавьте представленный ниже код в конец метода `__init__()` класса `Cart`, чтобы инициализировать объект купона из сессии.

```
class Cart(object):
    def __init__(self, request):
        # ...
        # Сохраняем купон.
        self.coupon_id = self.session.get('coupon_id')
```

В этом фрагменте мы пытаемся получить ID купона из сессии по ключу `coupon_id` и добавить его в качестве одноименного атрибута объекту типа `Cart`. Добавьте к классу `Cart` методы, которые будут работать с купонами:

```
class Cart(object):
    # ...
    @property
    def coupon(self):
        if self.coupon_id:
            return Coupon.objects.get(id=self.coupon_id)
```

```

    return None

def get_discount(self):
    if self.coupon:
        return (self.coupon.discount / Decimal('100')) \
            * self.get_total_price()
    return Decimal('0')

def get_total_price_after_discount(self):
    return self.get_total_price() - self.get_discount()

```

Эти методы выполняют следующие действия:

- `coupon()` – определен как свойство класса через декоратор `property`. Если у корзины задан атрибут `coupon_id`, это свойство будет возвращать соответствующий объект купона;
- `get_discount()` – возвращает размер скидки, если у корзины есть значение в атрибуте `coupon_id`;
- `get_total_price_after_discount()` – возвращает общую стоимость товаров в корзине с учетом скидки по купону.

Класс `Cart` теперь готов обрабатывать данные купонов, сохраненные в сессии.

Добавим форму купона в интерфейс магазина. Отредактируйте файл `views.py` приложения `cart` и выполните импорт класса формы:

```
from coupons.forms import CouponApplyForm
```

Затем добавьте несколько строк в обработчик `cart_detail`, как показано ниже:

```

def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
            initial={'quantity': item['quantity'],
                    'update': True})
    coupon_apply_form = CouponApplyForm()

    return render(request,
                  'cart/detail.html',
                  {'cart': cart,
                  'coupon_apply_form': coupon_apply_form})

```

Наконец, откройте файл `cart/detail.html` приложения `cart` и найдите строки:

```

<tr class="total">
  <td>Total</td>
  <td colspan="4"></td>
  <td class="num">${{ cart.get_total_price }}</td>
</tr>

```

Замените их на следующий фрагмент:

```
{% if cart.coupon %}
<tr class="subtotal">
  <td>Subtotal</td>
  <td colspan="4"></td>
  <td class="num">${{ cart.get_total_price|floatformat:"2" }}</td>
</tr>
<tr>
  <td>
    "{{{ cart.coupon.code }}" coupon
    ({{{ cart.coupon.discount }}}% off)
  </td>
  <td colspan="4"></td>
  <td class="num neg">
    - ${{ cart.get_discount|floatformat:"2" }}
  </td>
</tr>
{% endif %}
<tr class="total">
  <td>Total</td>
  <td colspan="4"></td>
  <td class="num">
    ${{ cart.get_total_price_after_discount|floatformat:"2" }}
  </td>
</tr>
</tr>
```

Этот фрагмент отвечает за отображение блока купона и размера скидки, которую он дает. Если у объекта корзины задан атрибут `coupon`, отображаем первую строку с информацией об общей стоимости товаров и вторую – с размером скидки. В третьей строке выводим итоговую стоимость, вызывая метод `get_total_price_after_discount()` объекта `cart`.

В этом же файле добавьте фрагмент после закрывающего тега `</table>`:

```
<p>Apply a coupon:</p>
<form action="{% url 'coupons:apply' %}" method="post">
  {{{ coupon_apply_form }}}
  <input type="submit" value="Apply">
  {% csrf_token %}
</form>
```

Он отобразит для пользователя форму ввода кода купона.

Откройте в браузере страницу <http://127.0.0.1:8000/>, добавьте товар в корзину и введите код купона. Страница корзины теперь выглядит аналогичным образом:

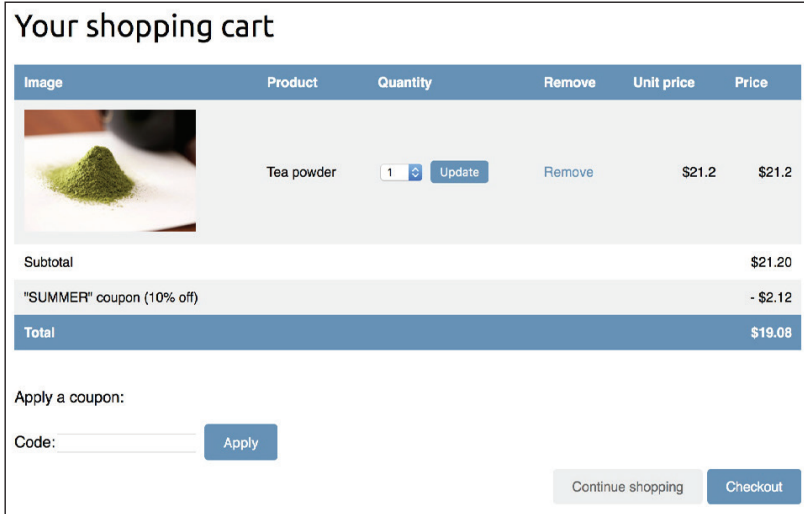


Рис. 9.2 ❖ Страница корзины с полем ввода кода купона

Давайте добавим обработку купона на следующем шаге, т. е. при оформлении заказа. Откройте файл `orders/order/create.html` и замените строки

```
<ul>
  {% for item in cart %}
    <li>
      {{ item.quantity }}x {{ item.product.name }}
      <span>${{ item.total_price }}</span>
    </li>
  {% endfor %}
</ul>
```

на фрагмент, который при необходимости будет отображать информацию о скидке:

```
<ul>
  {% for item in cart %}
    <li>
      {{ item.quantity }}x {{ item.product.name }}
      <span>${{ item.total_price|floatformat:"2" }}</span>
    </li>
  {% endfor %}
  {% if cart.coupon %}
    <li>
      "{{ cart.coupon.code }}" ({{ cart.coupon.discount }}% off)
      <span>- ${{ cart.get_discount|floatformat:"2" }}</span>
    </li>
  {% endif %}
</ul>
```


Добавив эти поля, мы сможем сохранять купон и размер скидки для каждого заказа. Модель `Coupon` содержит информацию о скидке. Однако нами было создано дополнительное поле `discount`, чтобы в случае удаления или изменения купона заказ мог быть обработан корректно. Также мы указали для поля `coupon` параметр `on_delete`, равный `models.SET_NULL`. При удалении купона поле автоматически будет изменено на `None`.

Теперь необходимо создать миграцию. Чтобы эти изменения для модели `Order` применились в базе данных, выполните команду:

```
python manage.py makemigrations
```

Вы увидите такой вывод:

```
Migrations for 'orders':
orders/migrations/0003_auto_20180307_2202.py:
- Add field coupon to order
- Add field discount to order
```

Выполните команду синхронизации моделей и базы данных:

```
python manage.py migrate orders
```

Вы должны будете увидеть сообщение о том, что миграция применилась. Теперь база данных соответствует состоянию моделей.

Вернитесь к файлу `models.py` и замените тело метода `get_total_cost()` модели `Order` таким образом:

```
class Order(models.Model):
    # ...
    def get_total_cost(self):
        total_cost = sum(item.get_cost() for item in self.items.all())
        return total_cost - total_cost * (self.discount / Decimal('100'))
```

Теперь метод `get_total_cost()` будет учитывать возможную скидку при расчете общей стоимости заказа.

Откройте файл `views.py` приложения `orders` и добавьте в обработчик `order_create` сохранение связанного купона. Для этого найдите строку:

```
order = form.save()
```

Замените ее таким образом:

```
order = form.save(commit=False)
if cart.coupon:
    order.coupon = cart.coupon
    order.discount = cart.coupon.discount
order.save()
```

В добавленном фрагменте мы создаем объект модели `Order` с помощью сохранения формы `OrderCreateForm`. При этом следует передать аргумент `commit=False`, чтобы объект пока не сохранялся в базу данных. Если для корзины задан купон, сохраняем ссылку на него и размер скидки, после чего заносим объект `order` в базу данных.

Убедитесь, что сервер для разработки запущен командой `python manage.py gunicornserver`.

Откройте в браузере страницу `http://127.0.0.1:8000/` и совершите покупку с купоном, который вы создали ранее на сайте администрирования. После этого перейдите по адресу `http://127.0.0.1:8000/admin/orders/order/` и проверьте, сохранился ли купон в объекте заказа:

The screenshot shows the Django admin interface for an order. At the top, there are three fields: 'Braintree id' with the value 'd31natz6', 'Coupon' with a dropdown menu showing 'SUMMER' and edit/delete icons, and 'Discount' with a dropdown menu showing '10'. Below these fields is a section titled 'ORDER ITEMS' which contains a table with the following data:

| PRODUCT | PRICE | QUANTITY | DELETE? |
|--|---------------------------|------------------------|--------------------------|
| 3 <input type="text"/> <input type="text"/> Tea powder | 21,2 <input type="text"/> | 1 <input type="text"/> | <input type="checkbox"/> |

Рис. 9.4 ❖ Купон и размер скидки сохранены в заказе

В качестве тренировки вы можете добавить отображение купона и скидки в шаблон PDF-документа для заказа по аналогии с тем, как мы делали это в прошлой главе.

Теперь мы добавим на сайт переводы.

ДОБАВЛЕНИЕ ИНТЕРНАЦИОНАЛИЗАЦИИ И ЛОКАЛИЗАЦИИ САЙТА

Django поддерживает локализацию и интернационализацию для проекта. Вы можете добавить переводы и специфичное форматирование дат, времени, чисел и временных зон для множества языков. Стоит пояснить различие между интернационализацией и локализацией. *Интернационализация* (часто используется аббревиатура i18n) – это процесс адаптации программы, для того чтобы ее потенциально можно было использовать на разных языках, т. е. написание программы таким образом, чтобы она не была привязана к конкретному языку. *Локализация* (аббревиатура l10n) – это процесс приведения программы к конкретному языку. Подсистема интернационализации Django поддерживает переводы для более чем 50 языков.

Интернационализация Django

Подсистема интернационализации Django позволяет разработчику пометить строки, которые необходимо перевести, и в Python-коде, и в HTML-шаблонах. Эта система использует возможности утилиты `gettext`, чтобы управлять файлами переводов. Файл перевода – это простой текстовый файл. Он содержит

часть строк, которые нужно перевести, и их перевод на конкретный язык. Такие файлы сохраняются с расширением `.po`.

После того как процесс перевода будет окончен, получившиеся документы компилируются в файлы с расширением `.mo` для более быстрого поиска переводов.

Настройки интернационализации и локализации

Django предоставляет настройки интернационализации. Давайте познакомимся с наиболее часто используемыми, такими как:

- `USE_I18N` – булево значение, указывает, включена ли подсистема интернационализации;
- `USE_L10N` – булево значение, указывает, включена ли локализация дат, времени и чисел. Когда оно равно `True`, Django будет использовать локализованные значения для дат, времени и чисел. По умолчанию же оно равно `False`;
- `USE_TZ` – булево значение, указывает, использовать ли даты с учетом временной зоны. Когда вы создаете проект командой `startproject`, настройка равна `True`;
- `LANGUAGE_CODE` – код проекта языка по умолчанию. Задается стандартным форматом языка, например `'en-us'` для американского английского, `'en-gb'` для британского и т. д. Чтобы эта настройка учитывалась, необходимо выставить `USE_I18N` в `True`. Список всех кодов вы можете найти на странице <http://www.i18nguy.com/unicode/language-identifiers.html>;
- `LANGUAGES` – кортеж, содержащий языки, которые могут быть использованы в проекте. Каждый язык задается кортежем из двух элементов: кода и названия. Вы можете посмотреть полный список поддерживаемых Django языков в `django.conf.global_settings`. Чтобы настроить проект на поддержку каких-то языков, нужно задать в настройке `LANGUAGES` кортежи из этого списка;
- `LOCALE_PATHS` – список путей в файловой системе, по которым Django будет находить переводы для проекта;
- `TIME_ZONE` – строка, задающая временную зону проекта. При создании проекта командой `startproject` выставляется значение `'UTC'`. Вы можете задать любую временную зону, например `'Europe/Madrid'`.

Это только часть настроек интернационализации и локализации. Полный список, поддерживаемый Django, вы можете найти на странице <https://docs.djangoproject.com/en/2.0/ref/settings/#globalization-i18n-l10n>.

Команды управления интернационализацией

Чтобы управлять файлами переводов в Django, существует две команды:

- `makemessages` – выполняется для того, чтобы найти все строки, помеченные к переводу, и создать или актуализировать файлы переводов `.po` в папке `locale`. Для каждого языка создается один файл `.po`;
- `compilemessages` – преобразует сформированные файлы `.po` в файлы `.mo`, используемые в запущенном проекте для применения переводов.

Для работы с этими командами необходимо установить утилиту `gettext`, которая умеет создавать, изменять и преобразовывать файлы переводов. В большинстве систем Linux `gettext` установлен по умолчанию. Если вы используете macOS X, самый простой способ установить его – использовать `Homeweb` на странице <https://brew.sh/> и загрузить пакет, выполнив команду `brew install gettext`. Возможно, вам понадобится использовать флаг `force`, таким образом: `brew link gettext --force`. Если вы работаете с Windows, установите пакет, следуя инструкциям на странице <https://docs.djangoproject.com/en/2.0/topics/i18n/translation/#gettext-on-windows>.

Добавление переводов в Django-проект

Давайте посмотрим, из каких этапов состоит процесс перевода проекта на другие языки:

- 1) отметить строки для перевода в Python-коде и HTML-шаблонах;
- 2) выполнить команду `makemessages`, чтобы создать или изменить файлы переводов, в которые будут добавлены все строки, отмеченные нами на первом этапе;
- 3) перевести строки в каждом `.po`-файле и преобразовать их в файлы с расширением `.mo` с помощью команды `compilemessages`.

Механизм Django для определения текущего языка

Django предоставляет промежуточный слой `LocaleMiddleware`, который определяет язык пользователя по данным из запроса. Он определен в модуле `django.middleware.locale` и работает следующим образом:

- 1) если вы используете `i18n_patterns`, т. е. все URL'ы локализованы, то промежуточный слой определяет язык из префикса в URL'е;
- 2) если префикс не найден, то класс обращается к атрибуту `LANGUAGE_SESSION_KEY` в сессии текущего пользователя;
- 3) если код не задан, то промежуточный слой пытается найти язык в куках пользователя. Имя, по которому будет происходить поиск, задается настройкой `LANGUAGE_COOKIE_NAME` и по умолчанию равно `django_language`;
- 4) если в куках нет нужного языка, то класс обращается к HTTP-заголовку `Accept-Language` запроса;
- 5) если в заголовке код не задан, Django по умолчанию использует язык, сохраненный в настройке `LANGUAGE_CODE`.

Если вы не подключаете промежуточный слой `LocaleMiddleware`, Django настраивает проект на язык, указанный в `LANGUAGE_CODE`. Описанный выше процесс определения языка при этом не выполняется.

Подготовка проекта к интернационализации

Подготовьте проект к работе с несколькими языками. Создайте английскую и испанскую версии магазина. Отредактируйте файл `settings.py` проекта и добавьте настройку `LANGUAGES` после `LANGUAGE_CODE`, как показано ниже:

```
LANGUAGES = (
    ('en', 'English'),
    ('es', 'Spanish'),
)
```

Настройка `LANGUAGES` содержит два кортежа, каждый из которых состоит из двух элементов: кода и названия языка. Коды могут быть заданы с вариантами языка (например, `en-us` или `en-gb`) или в общем виде (`en`). С помощью этой настройки мы указали, что магазин будет доступен только в испанской и английской версиях. Если бы мы не задали `LANGUAGES`, сайт был бы представлен для всех языков, которые поддерживает Django.

Измените `LANGUAGE_CODE` таким образом:

```
LANGUAGE_CODE = 'en'
```

Добавьте строку `'django.middleware.locale.LocaleMiddleware'` в список `MIDDLEWARE`. Убедитесь, что `LocaleMiddleware` находится ниже, чем `SessionMiddleware`, т. к. для переводов необходимо иметь доступ к сессии. Промежуточный слой `CommonMiddleware` должен находиться в списке после `LocaleMiddleware`, чтобы Django корректно определял префикс URL'а запроса. Теперь настройка `MIDDLEWARE` должна выглядеть таким образом:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
    # ...
]
```



Порядок, в котором классы промежуточных слоев указаны в `MIDDLEWARE`, важен, потому что слои могут быть зависимы от действий друг друга. При обработке запроса классы выполняются в том порядке, в каком это указано в настройке `MIDDLEWARE`.

Создайте на одном уровне с файлом `manage.py` такую структуру папок:

```
locale/
  en/
  es/
```

Все файлы переводов будут сохраняться в каталоге `locale`. Отредактируйте файл `settings.py` проекта и добавьте следующие строки:

```
LOCALE_PATHS = (
    os.path.join(BASE_DIR, 'locale/'),
)
```

Настройка `LOCALE_PATHS` определяет каталоги, в которых Django ищет файлы переводов. Как только найдется первый подходящий, дальнейший поиск прерывается.

Когда вы выполняете команду `makemessages` из каталога проекта, появляются файлы переводов в каталоге `locale/`, который мы только что добавили. Однако

в приложениях, для которых есть собственная папка `locale/`, переводы будут размещаться внутри этой папки.

Добавление переводов в Python-код

Чтобы переводить строки в Python-коде, нужно пометить их как переводимые с помощью функции `gettext()` из модуля `django.utils.translation`. Эта функция находит и возвращает перевод для фразы. Существует соглашение между разработчиками, чтобы использовать для этой функции сокращение `_` (нижнее подчеркивание).

Вы можете ознакомиться с полной документацией по переводам на странице <https://docs.djangoproject.com/en/2.0/topics/i18n/translation/>.

Переводы по умолчанию

Этот пример кода демонстрирует, как выполнить перевод для строки:

```
from django.utils.translation import gettext as _
output = _('Text to be translated.')
```

«Ленивый» перевод

Django предоставляет «ленивые» версии функций для переводов. Названия таких функций заканчиваются на `_lazy()`. Когда вы их используете, перевод строк выполняется в тот момент, когда он действительно понадобится, а не при вызове функции. «Ленивые» переводы бывают полезны, когда строки содержатся в файлах, которые выполняются в момент импортирования их модулей.

i Если вы используете `gettext_lazy()` вместо `gettext()`, фактический перевод строк будет выполнен не в момент вызова функции, а когда этот перевод понадобится (например, для отображения страницы). Такие «ленивые» версии есть для всех функций, работающих с переводами в Django.

Переводы с добавлением переменных

Строки, помеченные к переводу, могут быть не явно заданными, а сформированными динамически из значений каких-нибудь переменных, таких, например, как:

```
from django.utils.translation import gettext as _
month = _('April')
day = '14'
output = _('Today is %(month)s %(day)s') % {'month': month,
                                           'day': day}
```

Как вы можете заметить, мы задали наименования для переменных внутри строки: `month` и `day`. Таким образом, при переводе на разные языки мы сможем менять порядок переменных. Например, для английской версии результат будет таким: `'Today is April 14'`, а для испанской – `'Hoy es 14 de Abril'`. Всегда используйте именованные параметры для строк перевода вместо позиционных. Так вы сможете более гибко переводить динамические строки.

Множественное число в переводах

Для переводов слов во множественном числе используйте `ngettext()` и `ngettext_lazy()`. Эти функции выполняют перевод слов в единственном и множественном числе в зависимости от количества объектов. Этот пример демонстрирует возможности `ngettext()`:

```
output = ngettext('there is %(count)d product',
                 'there are %(count)d products',
                 count) % {'count': count}
```

Теперь вы знакомы с базовыми возможностями подсистемы переводов Django. Далее следует добавить переводы в проект магазина.

Перевод кода проекта

Отредактируйте файл `settings.py` проекта. Для этого импортируйте функцию `gettext_lazy()` и измените настройку `LANGUAGES`, как показано ниже:

```
from django.utils.translation import gettext_lazy as _
LANGUAGES = (
    ('en', _('English')),
    ('es', _('Spanish')),
)
```

Здесь мы используем `gettext_lazy()` вместо `gettext()`, чтобы исключить циклические импорты при переводе названий поддерживаемых языков.

Откройте консоль и выполните команду из папки проекта:

```
django-admin makemessages -all
```

Вы увидите вывод, содержащий такие строки:

```
processing locale es
processing locale en
```

Обратите внимание на каталог `locale/` проекта. Вы должны будете увидеть такую структуру файлов и папок:

```
en/
  LC_MESSAGES/
    django.po
es/
  LC_MESSAGES/
    django.po
```

Файлы переводов с расширением `.po` были созданы для каждого языка. Откройте `es/LC_MESSAGES/django.po` в текстовом редакторе. В конце файла вы увидите такие строки:

```
#: myshop/settings.py:117
msgid "English"
msgstr ""
```

```
#: myshop/settings.py:118
msgid "Spanish"
msgstr ""
```

Перевод каждой фразы состоит из комментария с дополнительной информацией о ней и двух строк, таких как:

- `msgid` – переводимая фраза из кода проекта;
- `msgstr` – перевод на целевой язык (по умолчанию – пустая строка).

Задайте переводы для фраз, как показано ниже:

```
#: myshop/settings.py:117
msgid "English"
msgstr "Inglés"
```

```
#: myshop/settings.py:118
msgid "Spanish"
msgstr "Español"
```

Сохраните файл, откройте консоль и выполните команду:

```
django-admin compilemessages
```

Если все прошло успешно, вы увидите вывод, содержащий такие строки:

```
processing file django.po in myshop/locale/en/LC_MESSAGES
processing file django.po in myshop/locale/es/LC_MESSAGES
```

Файлы переводов успешно преобразованы в файлы с расширением `.mo`. Взгляните на папку `locale/` проекта `myshop` еще раз. Вы увидите, что для каждого языка появился еще один файл:

```
en/
  LC_MESSAGES/
    django.mo
    django.po
es/
  LC_MESSAGES/
    django.mo
    django.po
```

Мы перевели названия используемых в проекте языков. Теперь добавим перевод для полей моделей, которые отображаются на сайте. Откройте файл `models.py` приложения `orders` и примените функцию `gettext_lazy`, как показано ниже:

```
from django.utils.translation import gettext_lazy as _

class Order(models.Model):
    first_name = models.CharField(_('first name'), max_length=50)
    last_name = models.CharField(_('last name'), max_length=50)
    email = models.EmailField(_('e-mail'))
    address = models.CharField(_('address'), max_length=250)
    postal_code = models.CharField(_('postal code'), max_length=20)
    city = models.CharField(_('city'), max_length=100)
    # ...
```

Мы поместили как переводимые названия полей, которые используются в форме оформления заказа: `first_name`, `last_name`, `address`, `postal_code` и `city`.

Создайте такую структуру папок в каталоге приложения `orders`:

```
locale/  
  en/  
  es/
```

Теперь переводы для строк из приложения `orders` будут сохраняться здесь, а не в общем файле проекта. Так мы можем формировать файлы переводов для каждого приложения по отдельности.

Откройте терминал и выполните команду из папки проекта:

```
django-admin makemessages -all
```

Вы увидите такой вывод:

```
processing locale es  
processing locale en
```

Откройте файл `locale/es/LC_MESSAGES/django.po` приложения `order` в текстовом редакторе. Вы увидите шаблоны переводов для полей модели `Order`. Заполните их, как показано ниже:

```
#: orders/models.py:10  
msgid "first name"  
msgstr "nombre"  
  
#: orders/models.py:11  
msgid "last name"  
msgstr "apellidos"  
  
#: orders/models.py:12  
msgid "e-mail"  
msgstr "e-mail"  
  
#: orders/models.py:13  
msgid "address"  
msgstr "dirección"  
  
#: orders/models.py:14  
msgid "postal code"  
msgstr "código postal"  
  
#: orders/models.py:15  
msgid "city"  
msgstr "ciudad"
```

Сохраните файл.

Кроме текстового редактора, вы можете использовать приложение Poedit, которое создано специально для обработки файлов переводов. Оно работает с операционными системами Linux, Windows и macOS X. Вы можете скачать Poedit на странице <https://poedit.net/>.

Давайте переведем формы. Класс `OrderCreateForm` приложения `order` не переводится, т. к. это класс модельной формы, которая использует `verbose_name` соответствующих полей модели `Order` для отображения заголовков на странице. Давайте посмотрим, что можно перевести в приложениях `cart` и `coupons`.

Отредактируйте файл `forms.py` приложения `cart` и добавьте параметр `label` в определение поля `quantity` класса `CartAddProductForm` таким образом:

```
from django import forms
from django.utils.translation import gettext_lazy as _

PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]

class CartAddProductForm(forms.Form):
    quantity = forms.TypedChoiceField(
        choices=PRODUCT_QUANTITY_CHOICES,
        coerce=int,
        label=_('Quantity'))
    update = forms.BooleanField(required=False,
                               initial=False,
                               widget=forms.HiddenInput)
```

Отредактируйте файл `forms.py` приложения `coupons` и измените форму `CouponApplyForm`, как показано ниже:

```
from django import forms
from django.utils.translation import gettext_lazy as _

class CouponApplyForm(forms.Form):
    code = forms.CharField(label=_('Coupon'))
```

Мы добавили заголовок и его перевод для поля `code`.

Перевод в HTML-шаблонах

Django предоставляет шаблонные теги `{% trans %}` и `{% blocktrans %}`, для того чтобы пометить переводимые строки в HTML-шаблонах. Чтобы применять их, нужно загружать модуль с тегами в начале каждого шаблона, используя запись `{% load i18n %}`.

Шаблонный тег `{% trans %}`

Тег `{% trans %}` позволяет пометить строку, константу или значение переменной как переводимое. Внутри Django применяет функцию `gettext()` к помеченной строке. Вот как это выглядит на практике:

```
{% trans "Text to be translated" %}
```

Вы можете добавить имя переменной, в которую будет сохранен перевод, чтобы он был доступен в шаблоне через эту переменную, например:

```
{% trans "Hello!" as greeting %}
<h1>{{ greeting }}</h1>
```


Тег `{% trans %}` удобен при переводе статичных сообщений, но с его помощью нельзя перевести фразы, содержащие переменные.

Шаблонный тег `{% blocktrans %}`

Тег `{% blocktrans %}` позволяет переводить фразы, которые состоят из статичных строк и переменных значений. В этом примере мы применим тег для перевода строки, в которой используется контекстная переменная `name`:

```
{% blocktrans %}Hello {{ name }}!{% endblocktrans %}
```

Внутри блока тега `{% blocktrans %}` нельзя использовать выражение или обращаться к методам и атрибутам какого-либо объекта. Вы можете применить ключевое слово `with`, чтобы создать в блоке перевода переменную и перевести ее. Давайте разберем это на конкретном примере. Для этого создадим переменную контекста, `name`, с помощью ключевого слова `with` и фильтра `capfirst` в имени пользователя:

```
{% blocktrans with name=user.name|capfirst %}
  Hello {{ name }}!
{% endblocktrans %}
```

i Когда вам нужно перевести фразу, содержащую переменную контекста, используйте тег `{% blocktrans %}` вместо `{% trans %}`.

Добавление переводов в шаблоны магазина

Отредактируйте файл `shop/base.html` приложения `shop`. Не забудьте загрузить модуль шаблонных тегов `i18n`:

```
{% load i18n %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>
    {% block title %}{% trans "My shop" %}{% endblock %}
  </title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
  <div id="header">
    <a href="/" class="logo">{% trans "My shop" %}</a>
  </div>
  <div id="subheader">
    <div class="cart">
      {% with total_items=cart|length %}
        {% if cart|length > 0 %}
          {% trans "Your cart" %}:
          <a href="{% url "cart:cart_detail" %}">
            {% blocktrans with total_items_plural=total_items|pluralize
```

```

        total_price=cart.get_total_price %}
        {{ total_items }} item{{ total_items_plural }},
        ${{ total_price }}
    {% endblocktrans %}
</a>
{% else %}
    {% trans "Your cart is empty." %}
{% endif %}
{% endwith %}
</div>
</div>
<div id="content">
    {% block content %}
    {% endblock %}
</div>
</body>
</html>

```

Обратите внимание, мы используем тег `{% blocktrans %}`, чтобы отобразить общую сумму покупки. Раньше мы делали это таким образом:

```
{{ total_items }} item{{ total_items|pluralize }}, ${{ cart.get_total_price }}
```

В шаблоне мы создаем новые переменные контекста с помощью записи `{% blocktrans with ... %}`, т. к. нам необходимо применить фильтр `total_items|pluralize` и обратиться к атрибуту объекта `cart.get_total_price`:

```
{% blocktrans with total_items_plural=total_items|pluralize
total_price=cart.get_total_price %}
    {{ total_items }} item{{ total_items_plural }},
    ${{ total_price }}
{% endblocktrans %}
```

Теперь отредактируйте файл `shop/product/detail.html` приложения `shop` и загрузите модуль `i18n`. Стоит отметить, что загрузка любых модулей шаблонных тегов и фильтров должна быть после вызова тега `{% extends %}`. Добавьте после него такой код:

```
{% load i18n %}
```

Найдите строку:

```
<input type="submit" value="Add to cart">
```

Замените ее на:

```
<input type="submit" value="{% trans "Add to cart" %}">
```

Теперь переведем шаблоны приложения `orders`. Откройте файл `orders/order/create.html` и пометьте строки к переводу:

```
{% extends "shop/base.html" %}
{% load i18n %}
```

```

{% block title %}
  {% trans "Checkout" %}
{% endblock %}

{% block content %}
  <h1>{% trans "Checkout" %}</h1>

  <div class="order-info">
    <h3>{% trans "Your order" %}</h3>
    <ul>
      {% for item in cart %}
        <li>
          {{ item.quantity }}x {{ item.product.name }}
          <span>${{ item.total_price }}</span>
        </li>
      {% endfor %}
      {% if cart.coupon %}
        <li>
          {% blocktrans with code=cart.coupon.code
            discount=cart.coupon.discount %}
            "{{ code }}" ({{ discount }}% off)
          {% endblocktrans %}
          <span>- ${{ cart.get_discount|floatformat:"2" }}</span>
        </li>
      {% endif %}
    </ul>
    <p>{% trans "Total" %}: ${{
      cart.get_total_price_after_discount|floatformat:"2" }}</p>
  </div>

  <form action="." method="post" class="order-form">
    {{ form.as_p }}
    <p><input type="submit" value="{% trans "Place order" %}"></p>
    {% csrf_token %}
  </form>
{% endblock %}

```

Посмотрите на следующие файлы из кода – примера к этой главе и пометьте строки в вашем проекте для перевода:

- приложение – shop, файл – shop/product/list.html;
- приложение – orders, файл – orders/order/created.html;
- приложение – cart, файл – cart/detail.html.

Давайте обновим файлы переводов. Откройте консоль и выполните команду:

```
django-admin makemessages -all
```

Файлы .po появились в папке locale проекта. Файлы переводов приложения orders теперь содержат все фразы, которые мы отметили к переводу.

Отредактируйте файл испанского перевода с расширением .po приложения orders, как это сделано в коде – примере к главе.

Выполните команду преобразования файлов переводов:

```
django-admin compilemessages
```

Вы увидите такой вывод в консоли:

```
processing file django.po in myshop/locale/en/LC_MESSAGES
processing file django.po in myshop/locale/es/LC_MESSAGES
processing file django.po in myshop/orders/locale/en/LC_MESSAGES
processing file django.po in myshop/orders/locale/es/LC_MESSAGES
```

Django сформировал для каждого .po-файла соответствующий файл с расширением .mo, который будет использоваться для поиска переводов.

Подключение Rosetta для перевода через сайт администрирования

Rosetta – это стороннее приложение, которое добавляет возможность редактировать файлы переводов через сайт администрирования. С его помощью редактирование .po-файлов и обновление .mo-файлов становятся невероятно простыми. Давайте подключим Rosetta к проекту.

Установите приложение с помощью pip:

```
pip install django-rosetta==0.8.1
```

Затем добавьте 'rosetta' в список установленных приложений, INSTALLED_APPS, файла settings.py проекта:

```
INSTALLED_APPS = [
    # ...
    'rosetta',
]
```

Вам также необходимо добавить шаблоны URL'ов приложения Rosetta в файле urls.py:

```
urlpatterns = [
    # ...
    path('rosetta/', include('rosetta.urls')),
    path('', include('shop.urls', namespace='shop')),
]
```

Убедитесь, что шаблоны приложения shop находятся ниже шаблонов URL'ов приложения Rosetta.

Откройте в браузере страницу <http://127.0.0.1:8000/admin/> и войдите как администратор. Затем перейдите по адресу <http://127.0.0.1:8000/rosetta/>. В меню **Filter** кликните **THIRD PARTY**, чтобы увидеть список всех файлов переводов приложения orders:

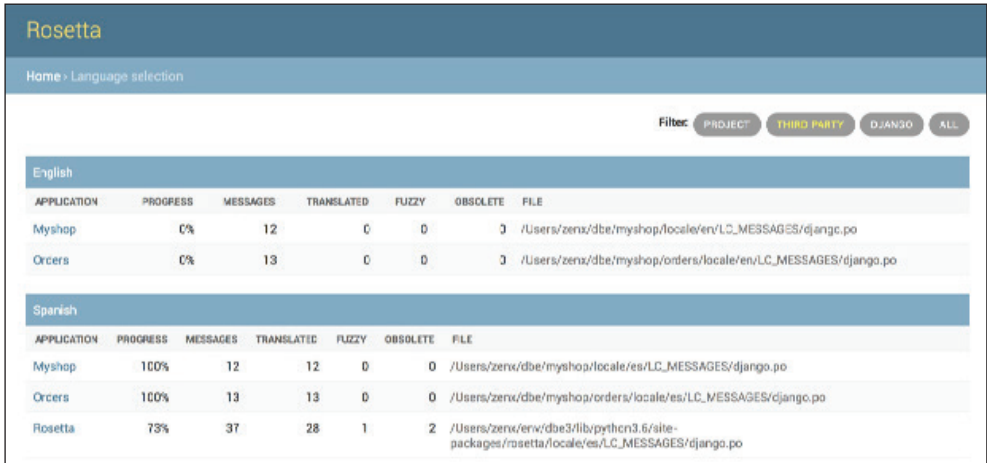


Рис. 9.5 ❖ Интерфейс управления переводами Rosetta

Кликните на ссылке **Myshop** в разделе **Spanish**, чтобы редактировать испанские переводы. Вы увидите их в такой форме:

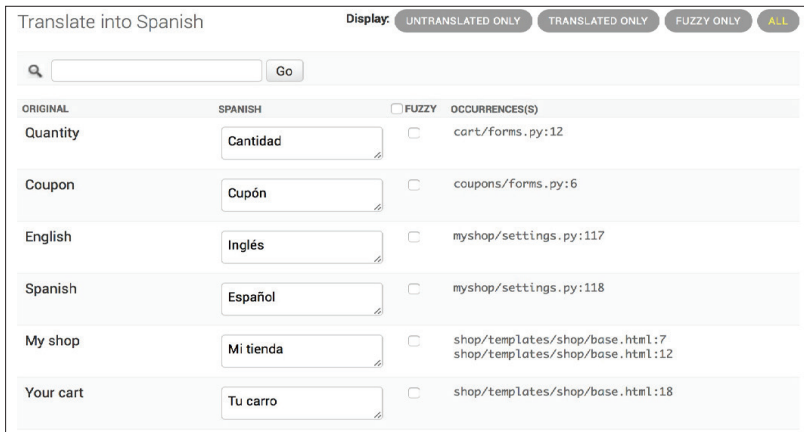


Рис. 9.6 ❖ Фразы, помеченные к переводу

Вы можете ввести перевод в колонке **Spanish**. Колонка **OCCURRENCES(S)** показывает файлы и строки, в которых встречается переводимая фраза. Переводы, которые содержат переменные, выделяются цветом:

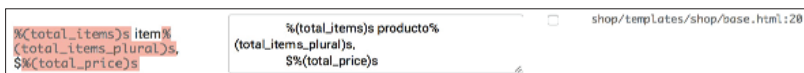


Рис. 9.7 ❖ Отображение фразы, содержащей переменные

Rosetta использует разные цвета для выделения переменных. Когда вы выполняете перевод, будьте внимательны и пропускайте имена переменных. Например, строка

```
%(total_items)s item%(total_items_plural)s, $%(total_price)s
```

может быть переведена на испанский следующим образом:


```
%(total_items)s producto%(total_items_plural)s, $%(total_price)s
```

В коде – примере к этой главе заполнены все файлы переводов. Вы можете использовать их.

После того как вы закончите редактирование, кликните на кнопку **Save and translate next block**. Соответствующий файл .po будет сохранен. При этом Rosetta выполнит преобразование изменений в .mo-файл, поэтому вам вовсе не обязательно запускать команду `compilemessages`. Важно убедиться, что Rosetta имеет доступ на запись в папке `locale`.

Если вы хотите, чтобы другие пользователи тоже могли редактировать переводы, перейдите по ссылке <http://127.0.0.1:8000/admin/auth/group/add/> и создайте новую группу `translators`. Затем откройте страницу: <http://127.0.0.1:8000/admin/auth/user/> и добавьте пользователям, которым вы хотите дать доступ к переводам, эту группу. Для этого в форме редактирования пользователя в разделе **Permissions** выберите группу `translators`. Возможности приложения Rosetta доступны только пользователям с правами администратора или с группой `translators`.

С полной документацией по Rosetta можно ознакомиться на странице <https://django-rosetta.readthedocs.io/en/>.

 В процессе добавления новых файлов переводов на боевом сервере вам необходимо перезагружать его каждый раз после сохранения переводов через интерфейс Rosetta или после выполнения команды `compilemessages`.

Грязный перевод

Вы могли заметить, что в интерфейсе Rosetta есть колонка **FUZZY** для переводов. Это поле заполняется утилитой `gettext`. Если чекбокс активирован, фраза не будет включена в .mo-файл. Этот флаг помечает фразы, которые были изменены и нуждаются в проверке переводчиком. Когда обновляются файлы .po, `gettext` может автоматически проставить флаг `fuzzy` для фраз, которые были немного изменены. Утилита пытается сопоставить новый вариант с какой-нибудь из существующих строк и, если находит соответствие, помечает ее. Чтобы эта фраза была добавлена в .mo-файл, переводчику необходимо снять флаг и убедиться, что перевод корректен.

Шаблоны URL'ов для интернационализации

Django дает возможность использовать локализованные URL'ы, что реализуется двумя способами:

- префикс языка в URL'е. Для каждого адреса добавляется префикс, чтобы любая языковая версия сайта находилась относительно своего базового URL'а;
- перевод шаблонов URL'ов. Все шаблоны переводятся, и для каждого языка используется соответствующая версия.

Добавление префикса языка в шаблоны

Django поддерживает языковые префиксы для URL'ов. Например, в английской версии сайта каждый адрес будет начинаться с /en/, в испанской – с /es/ и т. д.

Такой способ переводов реализуется средствами класса `LocaleMiddleware`. Фреймворк будет использовать этот промежуточный слой, чтобы определить язык сайта для текущего запроса. Мы описали его ранее в настройке `MIDDLEWARE`, поэтому дополнительных действий с файлом `settings.py` не требуется.

Давайте добавим префикс ко всем шаблонам URL'ов проекта. Откройте файл `urls.py` приложения `myshop` и задействуйте функцию `i18n_patterns`, как показано ниже:

```
from django.conf.urls.i18n import i18n_patterns

urlpatterns = i18n_patterns(
    path('admin/', admin.site.urls),
    path('cart/', include('cart.urls', namespace='cart')),
    path('orders/', include('orders.urls', namespace='orders')),
    path('payment/', include('payment.urls', namespace='payment')),
    path('coupons/', include('coupons.urls', namespace='coupons')),
    path('rosetta/', include('rosetta.urls')),
    path('', include('shop.urls', namespace='shop')),
)
```

Вы можете комбинировать переводимые и непередаваемые URL'ы. Однако рекомендуем все же при подключении переводов использовать только шаблоны с префиксами, чтобы избежать потенциальных ошибок из-за совпадений шаблонов.

Запустите сервер разработки и откройте страницу `http://127.0.0.1:8000/`. Django выполнит шаги, описанные в разделе «Механизм Django для определения текущего языка», и перенаправит вас на запрашиваемую страницу, добавив префикс языка. Обратите внимание на адресную строку вашего браузера. Она будет содержать URL вида `http://127.0.0.1:8000/en/`. Префикс был установлен на основе значений заголовка `Accept-Language`, если там передан язык вашего браузера, или настройки `LANGUAGE_CODE`.

Перевод шаблонов

Другой способ локализации URL'ов – их перевод. Вы можете создать несколько вариантов переводов шаблона (для нескольких языков). Для этого достаточно пометить строки шаблонов к переводу, как мы делали это раньше с помощью функции `gettext_lazy()`.

Отредактируйте файл `urls.py` приложения `myshop` и добавьте переводы для URL'ов приложений `cart`, `orders`, `payment` и `coupons`, как показано ниже:

```
from django.utils.translation import gettext_lazy as _
urlpatterns = i18n_patterns(
    path(_('admin/'), admin.site.urls),
    path(_('cart/'), include('cart.urls', namespace='cart')),
    path(_('orders/'), include('orders.urls', namespace='orders')),
    path(_('payment/'), include('payment.urls', namespace='payment')),
    path(_('coupons/'), include('coupons.urls', namespace='coupons')),
    path('rosetta/', include('rosetta.urls')),
    path('', include('shop.urls', namespace='shop')),
)
```

Отредактируйте файл `urls.py` приложения `orders` таким образом:

```
from django.utils.translation import gettext_lazy as _
urlpatterns = [
    path(_('create/'), views.order_create, name='order_create'),
    # ...
]
```

Аналогично добавьте перевод для шаблонов URL'ов в файле `urls.py` приложения `payment`:

```
from django.utils.translation import gettext_lazy as _
urlpatterns = [
    path(_('process/'), views.payment_process, name='process'),
    path(_('done/'), views.payment_done, name='done'),
    path(_('canceled/'), views.payment_canceled, name='canceled'),
]
```

Шаблоны URL'ов для приложения `shop` не нуждаются в переводах, т. к. они строятся из переменных и не содержат строк.

Откройте консоль и выполните команду обновления файлов переводов:

```
django-admin makemessages -all
```

Убедитесь, что сервер разработки запущен, откройте страницу `http://127.0.0.1:8000/en/rosetta/` и кликните на ссылку **Myshop** в разделе **Spanish**. Теперь вы увидите переводы для URL'ов, которые мы только что пометили. Чтобы отображать только непереуведенные фразы, можете воспользоваться фильтром **Untranslated only**.

Добавление возможности сменить язык сайта

На текущий момент сайт настроен так, чтобы отображать содержимое на английском и испанском языках. Но пользователи не могут выбрать языковую версию через интерфейс. Давайте добавим такую возможность, для чего разместим на сайте список с доступными языками.

Откройте шаблон `shop/base.html` приложения `shop` и найдите такой блок:

```
<div id="header">
  <a href="/" class="logo">{% trans "My shop" %}</a>
</div>
```


Замените его на код блока доступных языков:

```
<div id="header">
  <a href="/" class="logo">{% trans "My shop" %}</a>

  {% get_current_language as LANGUAGE_CODE %}
  {% get_available_languages as LANGUAGES %}
  {% get_language_info_list for LANGUAGES as languages %}
  <div class="languages">
    <p>{% trans "Language" %}</p>
    <ul class="languages">
      {% for language in languages %}
        <li>
          <a href="{% language.code %}"
            {% if language.code == LANGUAGE_CODE %} class="selected"{% endif %}>
            {% language.name_local %}
          </a>
        </li>
      {% endfor %}
    </ul>
  </div>
</div>
```

Выбор языка реализуется таким образом:

- 1) мы подключаем в шаблоне теги интернационализации с помощью записи `{% load i18n %}`;
- 2) используем тег `{% get_current_language %}`, чтобы получить текущий язык;
- 3) получаем языки, указанные в настройке проекта `LANGUAGES`, с помощью тега `{% get_available_languages %}`;
- 4) обращаемся к тегу `{% get_language_info_list %}`, чтобы получить объект списка языков в удобном для отображения виде;
- 5) формируем HTML, который содержит список доступных языков с выделением текущего посредством CSS-класса `selected`.

В том фрагменте мы используем теги из модуля `i18n`. Теперь откройте в браузере `http://127.0.0.1:8000/`. Вы увидите блок выбора языка в правом верхнем углу:

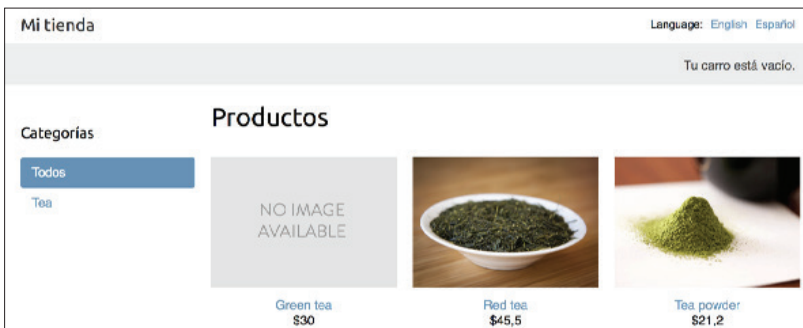


Рис. 9.8 ❖ Доступные языки интерфейса в шапке сайта

Теперь пользователи могут легко изменить языковую версию сайта.

Перевод данных в моделях с `django-parler`

Django по умолчанию не предоставляет функциональности для перевода данных, сохраненных в моделях. Для реализации этого вам необходимо придумать свое решение или применить одно из сторонних приложений. Существует несколько Python-пакетов, позволяющих переводить содержимое моделей. Каждый из них использует разные способы хранения и получения переводов. Один из таких пакетов – `django-parler`. Он весьма эффективно организует работу с переводами для Django-проектов и легко интегрируется на сайт администрирования.

Приложение `django-parler` создает в базе данных несколько таблиц с переводами для каждой модели. Таблица содержит все переведенные поля, внешний ключ на оригинальный объект и поле для того, чтобы задать язык перевода, т. к. каждая запись в этой таблице хранит переводы для одного языка.

Установка `django-parler`

Установите пакет `django-parler` с помощью `pip` таким образом:

```
pip install django-parler==1.9.2
```

Добавьте его в список установленных приложений `INSTALLED_APPS` файла `settings.py` проекта:

```
INSTALLED_APPS = [
    # ...
    'parler',
]
```

Добавьте в файл `settings.py` конфигурацию перевода:

```
PARLER_LANGUAGES = {
    None: (
        {'code': 'en'},
        {'code': 'es'},
    ),
    'default': {
        'fallback': 'en',
        'hide_untranslated': False,
    }
}
```

Эта конфигурация указывает, что в проекте есть две версии: английская и испанская. Мы задали язык по умолчанию `en` и настроили `django-parler` так, чтобы непереуведенное содержимое не скрывалось при показе, а отображалось из оригинального объекта.

Перевод полей моделей

Давайте переведем сведения о товарах нашего магазина. Пакет `django-parler` предоставляет классы модели `TranslatedModel` и поля модели `TranslatedFields`. Импортируйте их в файле `models.py` приложения `shop`:

```
from parler.models import TranslatableModel, TranslatedFields
```

Затем добавьте в модели `Category` атрибут `translations`, чтобы сделать поля `name` и `slug` переводимыми:

```
class Category(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=200, db_index=True),
        slug = models.SlugField(max_length=200,
                                db_index=True,
                                unique=True)
    )
```

Теперь модель `Category` наследуется от класса `TranslatedModel` вместо `models.Model`, а поля `name` и `slug` обернуты в `TranslatedFields`.

Отредактируйте аналогично модель `Product`, добавив переводы для полей `name`, `slug` и `description`:

```
class Product(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=200, db_index=True),
        slug = models.SlugField(max_length=200, db_index=True),
        description = models.TextField(blank=True)
    )
    category = models.ForeignKey(Category, related_name='products')
    image = models.ImageField(upload_to='products/%Y/%m/%d', blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
```

Приложение `django-parler` реализует несколько вариантов содержимого путем создания дополнительной таблицы с данными, переведенными на определенный язык. На схеме показано, как будут представлены в базе данных таблицы для модели `Product`:

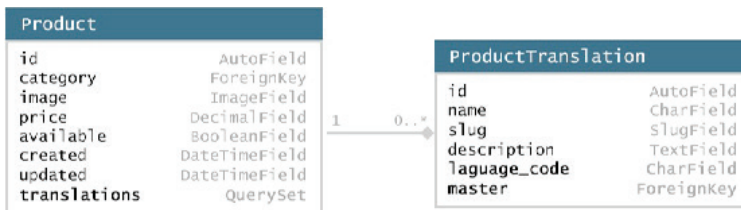


Рис. 9.9 ❖ Представление перевода модели `Product`

Модель `ProductTranslation` была сгенерирована приложением `django-parler`. Она содержит поля `name`, `slug`, `description`, `language_code` и внешний ключ на оригинальный объект. Между моделями `Product` и `ProductTranslation` существует от-

ношение «одни ко многим» (с одним объектом класса `Product` может быть создано несколько объектов `ProductTranslation`).

Так как Django использует разные таблицы для переводов, некоторые возможности фреймворка нельзя применить при подключении переводов моделей. Например, мы не можем задать сортировку по переводимым полям через опции `Meta` модели. При этом фильтрация и сортировка через методы `filter()` и `order_by()` `QuerySet`'ов остаются доступными.

Отредактируйте файл `models.py` приложения `shop` и прокомментируйте строку, определяющую сортировку по умолчанию:

```
class Category(TranslatableModel):
    # ...
    class Meta:
        # ordering = ('name',)
        verbose_name = 'category'
        verbose_name_plural = 'categories'
```

Также необходимо скрыть сортировку и индекс в классе `Meta` модели `Product`. Текущая версия `django-parler` не поддерживает работу с индексами в базе данных, определенными через атрибут `index_together`. Приведите модель `Product` к такому виду:

```
class Product(TranslatableModel):
    # ...

    # class Meta:
    #     ordering = ('-name',)
    #     index_together = (('id', 'slug'),)
```

Более подробно о `django-parler` и ограничениях, которые он накладывает на Django-проекты, можно прочесть на странице <https://django-parler.readthedocs.io/en/latest/compatibility.html>.

Добавление переводов на сайт администрирования

Как мы уже отмечали, возможности `django-parler` можно легко добавить на сайт администрирования. Для этого пакет предоставляет класс `TranslatableAdmin`, который расширяет возможности `ModelAdmin`, добавляя переводы.

Отредактируйте файл `admin.py` приложения `shop` и импортируйте этот класс:

```
from parler.admin import TranslatableAdmin
```

Измените родительский класс для `CategoryAdmin` и `ProductAdmin` на `TranslatableAdmin`. Приложение `django-parler` не умеет работать с атрибутом `prepopulated_fields`, но позволяет настроить ту же самую функциональность с помощью метода `get_prepopulated_fields()`. Отредактируйте файл `admin.py`, чтобы он выглядел таким образом:

```
from django.contrib import admin
from .models import Category, Product
from parler.admin import TranslatableAdmin
```

```

@admin.register(Category)
class CategoryAdmin(TranslatableAdmin):
    list_display = ['name', 'slug']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}

@admin.register(Product)
class ProductAdmin(TranslatableAdmin):
    list_display = ['name', 'slug', 'price',
                  'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'available']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}

```

Мы только что адаптировали сайт администрирования для работы с переводимым содержимым моделей. Теперь следует синхронизировать приложение и базу данных.

Создание миграций для переведенных моделей

Откройте консоль и выполните команду, чтобы создать миграции:

```
python manage.py makemigrations shop --name "translations"
```

Вы увидите такой вывод:

```

Migrations for 'shop':
shop/migrations/0002_translations.py
- Create model CategoryTranslation
- Create model ProductTranslation
- Change Meta options on category
- Change Meta options on product
- Remove field name from category
- Remove field slug from category
- Alter index_together for product (0 constraint(s))
- Add field master to producttranslation
- Add field master to categorytranslation
- Remove field description from product
- Remove field name from product
- Remove field slug from product
- Alter unique_together for producttranslation (1 constraint(s))
- Alter unique_together for categorytranslation (1 constraint(s))

```

Эта миграция автоматически создаст таблицы для моделей `CategoryTranslation` и `ProductTranslation`, т. к. мы добавили к ним переводы. Стоит отметить, что эта миграция удалит поля, которые раньше существовали в этих моделях. Это приведет к потере данных, так что после ее применения необходимо заново задать переводимые поля для категорий и товаров.

Выполните миграцию с помощью команды:

```
python manage.py migrate shop
```

Вы увидите такой вывод:

```
Applying shop.0002_translations... OK
```

Теперь модели и база данных синхронизированы.

Запустите сервер разработки командой `python manage.py runserver` и откройте страницу `http://127.0.0.1:8000/en/admin/shop/category/`. Вы увидите, что у категорий нет названий и слагов, т. к. соответствующие столбцы в базе данных были удалены во время выполнения миграции. Кликните на категорию и отредактируйте ее. Можно заметить, что теперь страница редактирования содержит две вкладки (для английской и испанской версий):

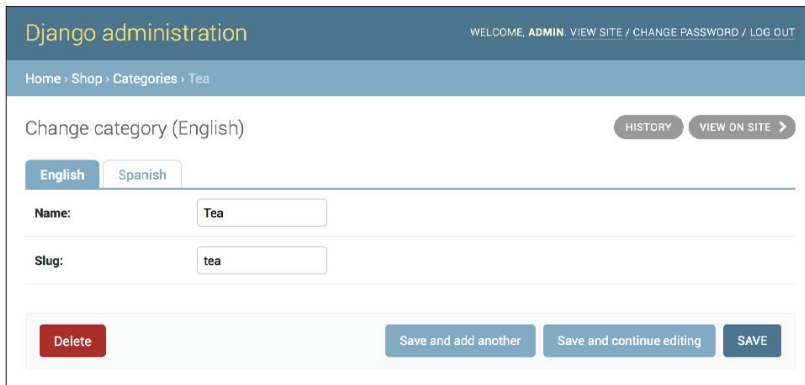


Рис. 9.10 ❖ Страница редактирования категории на сайте администрирования после подключения переводов для моделей

Задайте название и слаг для категории. Также добавьте испанский перевод для этих полей и нажмите кнопку **SAVE**. Сохранять изменения нужно для каждой вкладки отдельно.

После того как закончите заполнять утерянные данные, перейдите на страницу `http://127.0.0.1:8000/en/admin/shop/product/` и добавьте переводы на оба языка для товаров.

Адаптация обработчиков для работы с переводимыми моделями

Теперь нам нужно изменить обработчики, чтобы они могли работать с QuerySet'ами переводимых моделей. Выполните в консоли команду:

```
python manage.py shell
```

Давайте попробуем получить выборку для конкретного языка. Чтобы запросить из базы данных объекты, переведенные на определенный язык, можно вызвать функцию `activate()`, как показано ниже:

```
>>> from shop.models import Product
>>> from django.utils.translation import activate
```

```
>>> activate('es')
>>> product=Product.objects.first()
>>> product.name
'Té verde'
```

Еще один способ – использовать менеджер моделей `language()`, который предоставляет пакет `django-parler`:

```
>>> product=Product.objects.language('en').first()
>>> product.name
'Green tea'
```

Когда вы обращаетесь к переводимым полям, они стремятся к тому языку, который вы указали для `QuerySet`'а. Кроме этого, вы можете задать перевод для самого объекта:

```
>>> product.set_current_language('es')
>>> product.name
'Té verde'
>>> product.get_current_language()
'es'
```

Когда такие `QuerySet`'ы фильтруются методом `filter()`, можно задать поиск среди переводимых полей с помощью нотации `translations__`:

```
>>> Product.objects.filter(translations__name='Green tea')
<TranslatableQuerySet [<Product: Té verde>]>
```

Давайте адаптируем обработчик каталога товаров. Отредактируйте файл `views.py` приложения `shop`. Найдите в функции `product_list` такую строку:

```
category = get_object_or_404(Category, slug=category_slug)
```

Замените ее на следующий фрагмент:

```
language = request.LANGUAGE_CODE
category = get_object_or_404(Category,
                             translations__language_code=language,
                             translations__slug=category_slug)
```

Теперь найдите в обработчике `product_detail` такой код:

```
product = get_object_or_404(Product,
                             id=id,
                             slug=slug,
                             available=True)
```

Замените его на фрагмент с указанием языка:

```
language = request.LANGUAGE_CODE
product = get_object_or_404(Product,
                             id=id,
                             translations__language_code=language,
                             translations__slug=slug,
                             available=True)
```

Обработчики `product_list` и `product_detail` адаптированы к работе с языком, который пользователь выберет в меню. Запустите сервер разработки и откройте страницу <http://127.0.0.1:8000/es/>. Вы увидите каталог, переведенный на испанский язык:

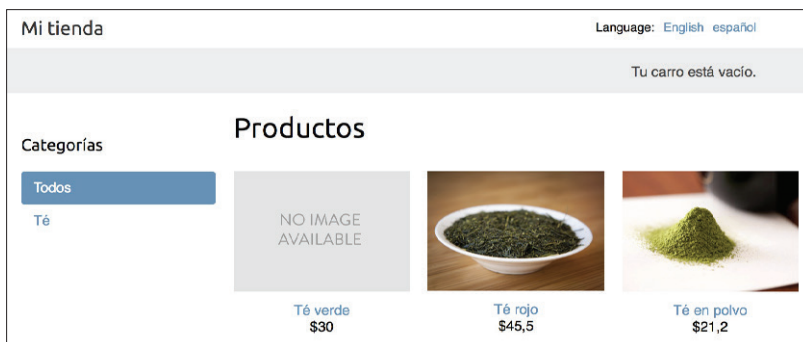


Рис. 9.11 ❖ Переведенная на испанский язык страница каталога

Теперь URL для каждого товара формируется с учетом слога, переведенного на текущий язык. Например, адресу в испанской версии сайта <http://127.0.0.1:8000/es/2/te-rojo/> будет соответствовать адрес для английского языка: <http://127.0.0.1:8000/en/2/red-tea/>. Если вы перейдете на страницу товара, то увидите, что ее содержимое также переведено на язык, выбранный в меню:



Рис. 9.12 ❖ Переведенная страница товара

Если вы хотите подробнее изучить возможности пакета `django-parler`, обратитесь к документации на странице <https://django-parler.readthedocs.io/en/latest/>.

Мы узнали, как добавить поддержку нескольких языков в обработчики, модели, шаблоны и URL'ы. Чтобы полностью перевести сайт, осталось добавить локализацию для дат, времени и чисел.

Настройка формата локализации

В зависимости от настроек локализации пользователя мы можем отображать даты, время и числа по-разному для каждого из них. Форматирование этих объектов можно включить, установив настройку `USE_L10N` в `True` в файле `settings.py`.

Когда локализация активна, Django при отображении будет пытаться использовать формат, специфичный для пользователя. Вы можете настроить показ чисел с плавающей запятой, так чтобы в английской версии сайта в качестве десятичного разделителя использовалась точка, в то время как в испанской версии – запятая. Это произойдет в соответствии с настройками Django для испанского языка. Полный список конфигурации локализации можно найти на странице <https://github.com/django/django/blob/stable/2.0.x/django/conf/locale/es/formats.py>.

По умолчанию, когда вы выставляете настройку `USE_L10N`, равную `True`, Django применяет формат для всех версий языков. Но иногда вам нужно избежать преобразования и использовать нелокализованные значения, например в том случае, когда вы работаете с JSON.

Django предоставляет шаблонный тег `{% localize %}`, который позволит вам включить или отключить локализацию для части шаблона. Это дает возможность более гибко настраивать отображение дат, времени и чисел. Чтобы использовать данный тег, необходимо подключить модуль `l10n`. В этом примере демонстрируются возможности шаблонизатора Django по настройке локализации:

```
{% load l10n %}
{% localize on %}
  {{ value }}
{% endlocalize %}
{% localize off %}
  {{ value }}
{% endlocalize %}
```

Кроме этого, в Django определены шаблонные фильтры `localize` и `unlocalize`, чтобы задать формат для переменной:

```
{{ value|localize }}
{{ value|unlocalize }}
```

Вы можете создать собственный формат локализации. Более подробно об этом можно прочитать на странице <https://docs.djangoproject.com/en/2.0/topics/i18n/formatting/>.

Валидация форм с django-localflavor

Приложение `django-localflavor` – это библиотека, которая содержит полезные функции и классы для работы с полями форм или моделей, отображаемых по-разному. Оно бывает очень полезно при валидации данных форм, имеющих специфичный формат в разных языковых версиях (например, телефонных номеров, данных банковских карт и т. п.). Пакет состоит из нескольких модулей, каждый из которых именуется кодом определенной страны в соответствии со стандартом ISO 3166.

Установите приложение с помощью `pip`:

```
pip install django-localflavor==2.0
```

Добавьте его в список `INSTALLED_APPS` файла `settings.py`:

```
INSTALLED_APPS = [
    # ...
    'localflavor',
]
```

Добавим в форму поле для почтового индекса США, которое будет обязательно к заполнению при оформлении заказа.

Отредактируйте файл `forms.py` приложения `orders` таким образом:

```
from django import forms
from .models import Order
from localflavor.us.forms import USZipCodeField

class OrderCreateForm(forms.ModelForm):
    postal_code = USZipCodeField()
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address',
                 'postal_code', 'city']
```

Мы импортируем класс `USZipCodeField` из модуля `us` приложения `localflavor` и добавляем поле `postal_code` этого типа в форму `OrderCreateForm`.

Запустите сервер разработки и перейдите на страницу `http://127.0.0.1:8000/en/orders/create/`. Заполните все поля, а в поле индекса введите код из трех букв, после чего отправьте форму на сервер. В результате вы увидите ошибку валидации такого содержания:

```
Enter a zip code in the format XXXXX or XXXXX-XXXX.
```

Это простой пример того, как легко можно добавить поле из пакета `localflavor`, которое будет обрабатывать данные с учетом особенностей конкретной страны. Классы и функции этого приложения очень полезны, когда вы адаптируете сайт для нескольких языковых версий. Вы можете найти полную документацию по пакету `django-localflavor` на странице <https://django-localflavor.readthedocs.io/en/latest/>.

А теперь пришло время добавить систему рекомендаций товаров в наш магазин.

РЕАЛИЗАЦИЯ СИСТЕМЫ РЕКОМЕНДАЦИЙ ТОВАРОВ

Система рекомендаций – это приложение, пытающееся предугадать предпочтения пользователя и предложить товары, которые кажутся подходящими для него. Такие системы подбирают рекомендации в зависимости от предыдущего поведения пользователя. Сегодня системы рекомендаций используются во многих онлайн-сервисах. Они показывают товары или услуги, которые могут заинтересовать покупателя, скрывая неподходящие. Хорошие рекомендации улучшают опыт пользователя по взаимодействию с системой. Коммерческие сайты часто используют такой подход, чтобы увеличить уровень продаж.

Мы собираемся добавить на наш сайт простую, но в то же время достаточно мощную систему, которая будет рекомендовать товары, купленные совместно с просматриваемыми. При формировании списка рекомендаций мы будем опираться на исторические сведения о покупках. После реализации пользователи смогут видеть схожие товары на двух страницах:

- подробности товара. В список попадут товары, которые покупают наиболее часто с просматриваемым. Список будет показан как «С этим товаром часто покупают X, Y, Z». Нам понадобится структура данных для хранения информации о товарах, купленных в одном заказе;
- страница корзины. В список попадут товары, которые часто покупают вместе с добавленными в корзину. В этом случае нам понадобится дополнительно суммировать рейтинг рекомендаций по каждому товару из списка.

Для реализации нам понадобится Redis, чтобы сохранять сведения о составе покупок. Мы уже подключали его в главе 6. Если вы еще не установили Redis, обратитесь к этой главе за инструкцией.

Добавление рекомендаций товаров на основе совершенных заказов

Мы добавим систему, которая будет выдавать покупателям рекомендуемые товары в зависимости от того, что они положили в корзину. Для этого нам понадобится хранить ключ каждого купленного товара в Redis. По этим ключам будет формироваться рейтинг. Каждый раз, когда два товара будут куплены вместе, их рейтинг будет увеличиваться на 1.

При обработке покупки мы будем сохранять в хранилище купленный товар и отсортированный список других товаров заказа.

Не забудьте установить `redis-ру` с помощью команды:

```
pip install redis==2.10.6
```

Отредактируйте файл `settings.py` проекта и добавьте такие настройки:

```

REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 1

```

Эти настройки понадобятся, чтобы Django-приложение могло подключиться к серверу Redis. Создайте в папке приложения `shop` новый файл, `recommender.py`, и добавьте в него следующий фрагмент:

```

import redis
from django.conf import settings
from .models import Product

# connect to redis
r = redis.StrictRedis(host=settings.REDIS_HOST,
                      port=settings.REDIS_PORT,
                      db=settings.REDIS_DB)

class Recommender(object):

    def get_product_key(self, id):
        return 'product: {}:purchased_with'.format(id)

    def products_bought(self, products):
        product_ids = [p.id for p in products]
        for product_id in product_ids:
            for with_id in product_ids:
                # Получаем товары, купленные вместе с текущим.
                if product_id != with_id:
                    # Увеличиваем их рейтинг.
                    r.zincrby(self.get_product_key(product_id),
                             with_id,
                             amount=1)

```

Это класс `Recommender`, который реализует сохранение информации о купленных вместе товарах и подбор рекомендаций. Метод `get_product_key()` получает ID товара и формирует ключ для хранилища Redis вида `product:[id]:purchased_with`.

Метод `products_bought()` получает список объектов `Product`, которые были куплены вместе, после чего выполняет следующие шаги:

- 1) для каждого товара из списка получаем ID, формируя список;
- 2) проходим по каждому идентификатору из списка и получаем товары, которые были куплены вместе с текущим;
- 3) для этого формируем ключ Redis, вызывая метод `get_product_id()`. Для ID, равного 33, этот метод вернет строку `product:33:purchased_with`. Она является ключом хранилища, по которому можно получить список товаров, купленных вместе с текущим;
- 4) увеличиваем рейтинг каждого товара из списка, сохраняя информацию о том, что эти товары часто покупают вместе.

Мы реализовали методы для сохранения и ранжирования объектов. Теперь нужно добавить возможность получать рекомендуемые объекты. Добавьте метод `suggest_products_for()` в класс `Recommender`:

```

def suggest_products_for(self, products, max_results=6):
    product_ids = [p.id for p in products]
    if len(products) == 1:
        # Передан только один товар.
        suggestions = r.zrange(
            self.get_product_key(product_ids[0]),
            0, -1, desc=True)[:max_results]
    else:
        # Формируем временный ключ хранилища.
        flat_ids = ''.join([str(id) for id in product_ids])
        tmp_key = 'tmp_{}'.format(flat_ids)
        # Передано несколько товаров, суммируем рейтинги их рекомендаций.
        # Сохраняем суммы во временном ключе.
        keys = [self.get_product_key(id) for id in product_ids]
        r.zunionstore(tmp_key, keys)
        # Удаляем ID товаров, которые были переданы в списке.
        r.zrem(tmp_key, *product_ids)
        # Получаем товары, отсортированные по рейтингу.
        suggestions = r.zrange(tmp_key, 0, -1,
                               desc=True)[:max_results]
        # Удаляем временный ключ.
        r.delete(tmp_key)
    suggested_products_ids = [int(id) for id in suggestions]

    # Получаем рекомендуемые товары и сортируем их.
    suggested_products = list(Product.objects.filter(id__in=suggested_products_ids))
    suggested_products.sort(key=lambda x: suggested_products_ids.index(x.id))
    return suggested_products

```

Метод `suggest_products_for()` получает два аргумента:

- `products` – список объектов типа `Product`, для которых нужно подобрать рекомендации. Здесь можно задать один или несколько товаров;
- `max_results` – целое число, обозначающее максимальное количество рекомендованных товаров, которые вернет метод.

В этой функции мы выполняем следующие действия:

- 1) получаем идентификаторы переданных объектов;
- 2) если переданный в качестве аргумента список товаров содержит единственное значение, сразу получаем список товаров, купленных с ним, при этом учитывая их рейтинг. Для этого используем команду `Redis`, `ZRANGE`. Ограничиваем результат максимальным количеством, `max_results`, который по умолчанию равен 6;
- 3) если в списке – аргументе функции задано более одного товара, формируем временный ключ для `Redis`, используя идентификаторы товаров;
- 4) суммируем рейтинги для каждого товара, который был куплен вместе с каким-либо из переданных в аргументе. Для этого обращаемся к команде `ZUNIONSTORE`. Она выполняет объединение множеств по указанным ключам и сохраняет в `Redis` агрегированное значение по новому ключу. Более подробно об этой команде можно прочесть на странице <https://>

redis.io/commands/ZUNIONSTORE. Сохраняем результат суммирования во временном ключе, который генерировали на предыдущем шаге;

- 5) чтобы товары, которые были переданы в функцию в списке `products`, не попали в рекомендации, удаляем их с помощью команды `ZREM`;
- 6) затем получаем идентификаторы всех товаров по временному ключу, сортируя их командой `ZRANGE`. Ограничиваем результат в соответствии с переменной `max_results`. Удаляем из хранилища временный ключ;
- 7) наконец, получаем объекты `Product` по вычисленным на предыдущих шагах идентификаторам.

Давайте добавим метод для очистки рекомендаций. Вставьте в класс `Recommender` такой код:

```
def clear_purchases(self):
    for id in Product.objects.values_list('id', flat=True):
        r.delete(self.get_product_key(id))
```

Самое время протестировать нашу систему рекомендаций. Убедитесь, что в каталоге есть несколько товаров, и запустите сервер Redis командой из папки Redis:

`src/redis-server`

Откройте другую консоль и выполните команду:

```
python manage.py shell
```

Нам понадобится хотя бы четыре товара в каталоге. Если у вас их меньше, создайте несколько, а затем получите их по имени, как показано ниже:

```
>>> from shop.models import Product
>>> black_tea = Product.objects.get(translations__name='Black tea')
>>> red_tea = Product.objects.get(translations__name='Red tea')
>>> green_tea = Product.objects.get(translations__name='Green tea')
>>> tea_powder = Product.objects.get(translations__name='Tea powder')
```

Потом обратитесь к системе рекомендаций, передав несколько покупок:

```
>>> from shop.recommender import Recommender
>>> r = Recommender()
>>> r.products_bought([black_tea, red_tea])
>>> r.products_bought([black_tea, green_tea])
>>> r.products_bought([red_tea, black_tea, tea_powder])
>>> r.products_bought([green_tea, tea_powder])
>>> r.products_bought([black_tea, tea_powder])
>>> r.products_bought([red_tea, green_tea])
```

Так в Redis появятся несколько записей о том, какие товары были куплены одновременно:

```
black_tea: red_tea (2), tea_powder (2), green_tea (1)
red_tea: black_tea (2), tea_powder (1), green_tea (1)
green_tea: black_tea (1), tea_powder (1), red_tea(1)
tea_powder: black_tea (2), red_tea (1), green_tea (1)
```

Давайте активировать другую языковую версию и посмотрим, как получить рекомендации:

```
>>> from django.utils.translation import activate
>>> activate('en')
>>> r.suggest_products_for([black_tea])
[<Product: Tea powder>, <Product: Red tea>, <Product: Green tea>]
>>> r.suggest_products_for([red_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Green tea>]
>>> r.suggest_products_for([green_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Red tea>]
>>> r.suggest_products_for([tea_powder])
[<Product: Black tea>, <Product: Red tea>, <Product: Green tea>]
```

Вы можете заметить, что порядок товаров в списке рекомендаций зависит от их рейтинга. Теперь попробуйте получить рекомендации для нескольких товаров таким образом:

```
>>> r.suggest_products_for([black_tea, red_tea])
[<Product: Tea powder>, <Product: Green tea>]
>>> r.suggest_products_for([green_tea, red_tea])
[<Product: Black tea>, <Product: Tea powder>]
>>> r.suggest_products_for([tea_powder, black_tea])
[<Product: Red tea>, <Product: Green tea>]
```

Как и ожидалось, в этом случае порядок соответствует суммарному рейтингу товаров. Например, рекомендованными товарами для `black_tea` и `red_tea` являются `tea_powder(2+1)` и `green_tea(1+1)`.

Мы проверили, что механизм рекомендаций работает корректно. Теперь можно добавить отображение на страницах.

Отредактируйте файл `views.py` приложения `shop`, добавьте ограничение на максимальное количество товаров в рекомендации (не более 4). Обработчик `product_detail` теперь будет выглядеть таким образом:

```
from .recommender import Recommender

def product_detail(request, id, slug):
    language = request.LANGUAGE_CODE
    product = get_object_or_404(Product,
                                id=id,
                                translations__language_code=language,
                                translations__slug=slug,
                                available=True)
    cart_product_form = CartAddProductForm()

    r = Recommender()
    recommended_products = r.suggest_products_for([product], 4)

    return render(request,
                  'shop/product/detail.html',
                  {'product': product,
                  'cart_product_form': cart_product_form,
                  'recommended_products': recommended_products})
```

Откройте файл `shop/product/detail.html` приложения `shop` и добавьте приведенный ниже фрагмент после строки `{{ product.description|linebreaks }}`:

```
{% if recommended_products %}
<div class="recommendations">
  <h3>{% trans "People who bought this also bought" %}</h3>
  {% for p in recommended_products %}
    <div class="item">
      <a href="{{ p.get_absolute_url }}">
        
        </a>
      <p><a href="{{ p.get_absolute_url }}">{{ p.name }}</a></p>
    </div>
  {% endfor %}
</div>
{% endif %}
```

Запустите сервер разработки и перейдите на страницу `http://127.0.0.1:8000/en/`. Щелкните по любому товару. На странице подробностей вы увидите список рекомендованных товаров:

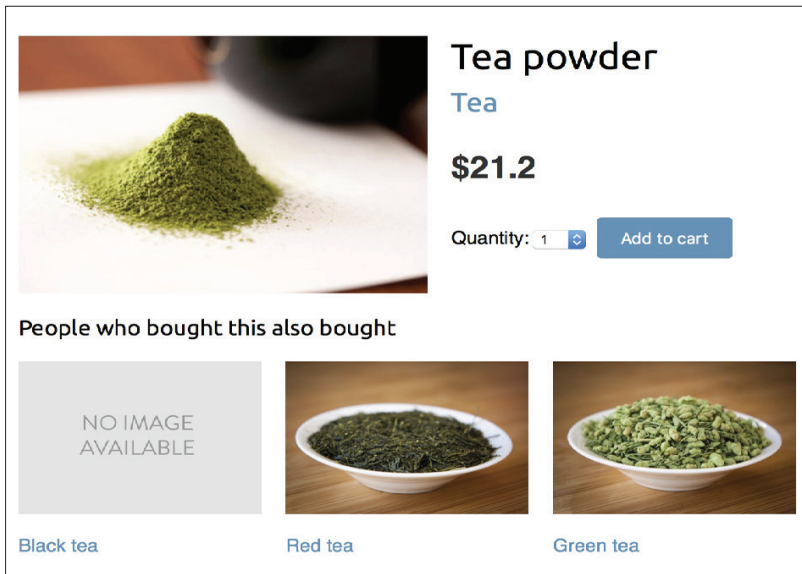


Рис. 9.13 ❖ Страница товара с рекомендациями

Также мы добавим рекомендации на страницу оформления заказа. Отредактируйте файл `views.py` приложения `cart`. Для этого импортируйте класс `Recommender` и измените обработчик `cart_view`, чтобы он выглядел таким образом:


```

from shop.recommender import Recommender

def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
            initial={'quantity': item['quantity'],
                    'update': True})

    coupon_apply_form = CouponApplyForm()

    r = Recommender()
    cart_products = [item['product'] for item in cart]
    recommended_products = r.suggest_products_for(cart_products, max_results=4)

    return render(request,
                  'cart/detail.html',
                  {'cart': cart,
                  'coupon_apply_form': coupon_apply_form,
                  'recommended_products': recommended_products})

```

Отредактируйте файл `cart/detail.html` приложения `cart` и вставьте после тега `</table>` такой фрагмент:



```

{% if recommended_products %}
<div class="recommendations cart">
  <h3>{% trans "People who bought this also bought" %}</h3>
  {% for p in recommended_products %}
    <div class="item">
      <a href="{{ p.get_absolute_url }}">
        
        </a>
      <p><a href="{{ p.get_absolute_url }}">{{ p.name }}</a></p>
    </div>
  {% endfor %}
</div>
{% endif %}

```

Откройте в браузере страницу `http://127.0.0.1:8000/en/` и добавьте несколько товаров в корзину. Когда вы перейдете к оформлению заказа на `http://127.0.0.1:8000/en/cart/`, то увидите список рекомендаций к товарам, которые добавлены в корзину:


Your shopping cart

| Image | Product | Quantity | Remove | Unit price | Price |
|---|------------|---|---------------------------------------|------------|----------------|
|  | Tea powder | 1 <input type="button" value="Update"/> | <input type="button" value="Remove"/> | \$21.2 | \$21.2 |
|  | Green tea | 1 <input type="button" value="Update"/> | <input type="button" value="Remove"/> | \$30 | \$30 |
| Total | | | | | \$51.20 |

People who bought this also bought


Apply a coupon:

Coupon:



NO IMAGE AVAILABLE

Black tea



Red tea

Рис. 9.14 ❖ Рекомендации на странице корзины

Поздравляем! Вы реализовали полноценную систему рекомендаций с помощью Django и Redis.

РЕЗЮМЕ

В этой главе вы создали систему купонов с помощью сессий Django, узнали об интернационализации и локализации, а также реализовали систему рекомендаций товаров.

В следующей главе мы начнем новый проект. Вы создадите платформу для онлайн-обучения, попробуете применить обработчики на основе классов и реализуете собственную систему управления содержимым.

Глава 10

Создание платформы для онлайн-обучения

Из предыдущей главы вы узнали, как добавить на сайт несколько языковых версий, реализовать системы скидочных купонов и рекомендаций товаров. В этой главе мы начнем с создания нового проекта. Это будет платформа для онлайн-обучения с собственной *системой управления содержимым* (Content Management System, CMS).

Мы рассмотрим следующие темы:

- создание фикстур для моделей;
- наследование моделей;
- реализация собственного поля модели;
- использование обработчиков-классов и примесей;
- работа с наборами форм;
- управление доступом к содержимому сайта с помощью групп и разрешений;
- создание CMS.

СОЗДАНИЕ ПРОЕКТА

В качестве заключительного проекта мы создадим платформу для онлайн-обучения. В этой главе мы узнаем, как реализовать собственную CMS, которая позволит создавать курсы и уроки и настраивать их содержимое.

Для начала создайте виртуальное окружение и активируйте его с помощью этих команд:

```
mkdir env
virtualenv env/educa
source env/educa/bin/activate
```

Установите Django с помощью pip:

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

Мы будем работать с изображениями, поэтому нужно установить пакет Pillow:

```
pip install Pillow==5.1.0
```

Теперь создайте новый проект, выполнив команду:

```
django-admin startproject educa
```

Перейдите в папку educa и создайте приложения с помощью команд:

```
cd educa
```

```
django-admin startapp courses
```

Отредактируйте файл settings.py проекта educa и добавьте приложение courses в список INSTALLED_APPS:

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

Приложение courses активно, пришло время описать модели, которые мы будем использовать.

ОПРЕДЕЛЕНИЕ МОДЕЛЕЙ ДЛЯ КУРСОВ ОБУЧЕНИЯ

Платформа для онлайн-обучения работает с такими объектами, как курсы и предметы. Каждый курс может быть разбит на несколько модулей. Каждый модуль может содержать несколько разделов. Количество модулей и разделов будет настраиваемым. Каждый раздел содержит различные материалы (Content): тексты, файлы, картинки или видео. Этот пример демонстрирует, как может быть организован один из курсов:

```
Subject 1
  Course 1
    Module 1
      Content 1 (image)
      Content 2 (text)
    Module 2
      Content 3 (text)
      Content 4 (file)
      Content 5 (video)
    ...
```

Давайте создадим модели. Отредактируйте файл models.py приложения courses и добавьте в него такой код:

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

class Subject(models.Model):
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)

    class Meta:
        ordering = ['title']

    def __str__(self):
        return self.title

class Course(models.Model):
    owner = models.ForeignKey(User,
                              related_name='courses_created',
                              on_delete=models.CASCADE)
    subject = models.ForeignKey(Subject,
                                 related_name='courses',
                                 on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)
    overview = models.TextField()
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ['-created']

    def __str__(self):
        return self.title

class Module(models.Model):
    course = models.ForeignKey(Course,
                              related_name='modules',
                              on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)

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

Мы описали модели `Subject`, `Course` и `Module`. Класс `Course` содержит следующие поля:

- `owner` – преподаватель, который создал курс;
- `subjects` – предмет, к которому привязан курс. Это внешний ключ, `ForeignKey`, на модель `Subject`;
- `title` – название курса;
- `slug` – слаг курса, мы будем использовать его для формирования человекопонятных URL'ов;
- `overview` – текстовое поле типа `TextField` для создания краткого описания курса;
- `created` – дата и время создания курса, которые будут устанавливаться автоматически, т. к. мы указали `auto_now_add=True`.

В каждый курс может входить несколько модулей, поэтому для модели `Module` мы определили внешний ключ, `ForeignKey`, на модель `Course`.

Откройте консоль и выполните команду создания миграций:

```
python manage.py makemigrations
```

Вы увидите такой вывод:

```
Migrations for 'courses':
  0001_initial.py:
    - Create model Course
    - Create model Module
    - Create model Subject
    - Add field subject to course
```

Затем выполните команду:

```
python manage.py migrate
```

Вы увидите, что применились наша миграция и миграции Django. Вывод будет содержать такую строку:

```
Applying courses.0001_initial... OK
```

Теперь модели приложения `courses` синхронизированы с базой данных.

Регистрация моделей на сайте администрирования

Давайте добавим модели курсов на сайт администрирования. Отредактируйте файл `admin.py` приложения `courses`, добавьте в него такой фрагмент:

```
from django.contrib import admin
from .models import Subject, Course, Module

@admin.register(Subject)
class SubjectAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug']
    prepopulated_fields = {'slug': ('title',)}

class ModuleInline(admin.StackedInline):
    model = Module

@admin.register(Course)
class CourseAdmin(admin.ModelAdmin):
    list_display = ['title', 'subject', 'created']
    list_filter = ['created', 'subject']
    search_fields = ['title', 'overview']
    prepopulated_fields = {'slug': ('title',)}
    inlines = [ModuleInline]
```

Работа завершена. Мы использовали декоратор `@admin.register()`, чтобы зарегистрировать модель на сайте администрирования.

Задание начальных данных с помощью фикстур

Иногда необходимо заполнить базу данных какой-то начальной информацией, чтобы объекты были доступны сразу после разворачивания проекта и вам не пришлось добавлять их в систему вручную. Django предоставляет такую возможность. Для этого вам понадобятся *фикстуры* – специальные файлы, которые содержат начальные данные для проекта.

Django поддерживает форматы JSON, XML и YAML. Мы создадим фикстуру, для того чтобы предварительно заполнить несколько объектов Subject.

Для начала нам понадобится суперпользователь. Создайте его такой командой:

```
python manage.py createsuperuser
```

Теперь запустите сервер для разработки, выполнив в консоли команду:

```
python manage.py runserver
```

Откройте в браузере страницу <http://127.0.0.1:8000/admin/courses/subject/>, создайте несколько предметов через сайт администрирования. Список предметов должен будет выглядеть подобным образом:

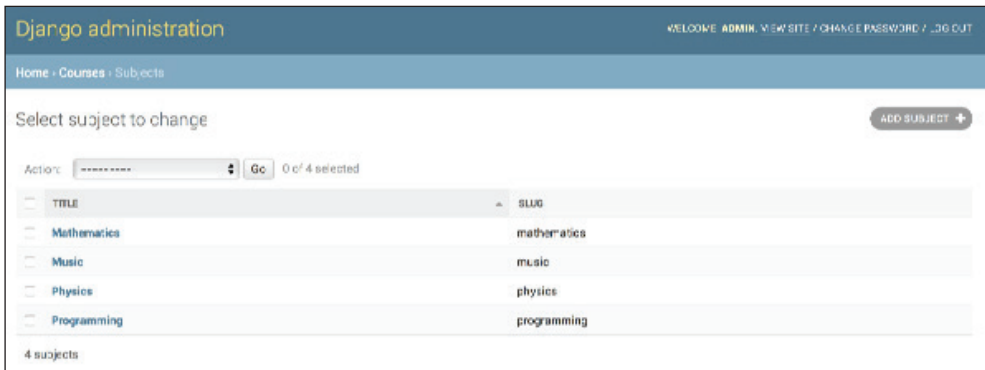


Рис. 10.1 ❖ Список предметов на сайте администрирования

Выполните такую команду:

```
python manage.py dumpdata courses --indent=2
```

Вы увидите вывод, похожий на этот:

```
{
  "model": "courses.subject",
  "pk": 1,
  "fields": {
    "title": "Mathematics",
    "slug": "mathematics"
  }
}
```

```

},
{
  "model": "courses.subject",
  "pk": 2,
  "fields": {
    "title": "Music",
    "slug": "music"
  }
},
{
  "model": "courses.subject",
  "pk": 3,
  "fields": {
    "title": "Physics",
    "slug": "physics"
  }
},
{
  "model": "courses.subject",
  "pk": 4,
  "fields": {
    "title": "Programming",
    "slug": "programming"
  }
}
]

```

Команда `dumpdata` по умолчанию выводит информацию по существующим объектам в консоль в формате JSON. Структура данных, которую мы видим, описывает модели, их поля и значения, так чтобы Django смог загрузить их и сохранить в базу.

Вы можете ограничить список моделей, которые следует выводить. Для этого нужно указать имя приложения и модели в виде `app.Model`. Также можно применить флаг `format`, чтобы задать формат вывода. По умолчанию `dumpdata` выводит результат в консоль. Но это поведение можно изменить с помощью флага `output`. Флаг `indent` можно использовать, если вы хотите задать другой размер отступа. Для получения более подробной информации о параметрах `dumpdata` выполните в консоли `python manage.py dumpdata -- help`.

Сохраните вывод в файл в папку `fixtures/` приложения `orders`, выполнив команды:

```

mkdir courses/fixtures
python manage.py dumpdata courses --indent=2 --
output=courses/fixtures/subjects.json

```

Запустите сервер для разработки и через сайт администрирования удалите все предметы, которые создали ранее. Затем загрузите фикстуру в базу данных с помощью команды:

```
python manage.py loaddata subjects.json
```


Все объекты Subject, которые были добавлены в фикстуру, должны снова появиться в базе данных.

По умолчанию Django ищет файлы фикстуры в папке fixtures/ каждого приложения, но можно указать полный путь до них. Также есть возможность задать каталоги, в которых нужно осуществлять поиск, с помощью настройки FIXTURE_DIRS.

i Фикстуры – это хороший способ заполнить базу данных начальными объектами, а также можно применять их при тестировании.

О том, как использовать фикстуры для тестирования, можно прочесть на странице <https://docs.djangoproject.com/en/2.0/topics/testing/tools/#fixture-loading>.

Если вы хотите загружать фикстуры не вручную, а через миграции, обратитесь к документации на странице <https://docs.djangoproject.com/en/2.0/topics/migrations/#data-migrations>.

СОЗДАНИЕ МОДЕЛЕЙ ДЛЯ СОДЕРЖИМОГО КУРСОВ

Наша цель – создать платформу, в которой модули курсов будут содержать различные типы содержимого. Это может быть текст, картинка, видео или какие-то файлы. Поэтому нам нужны модели, которые смогут хранить такую информацию. В главе 6 вы познакомились с таким понятием, как обобщенные типы. Они применяются в случаях, когда объекты одной модели могут ссылаться на несколько объектов других моделей. Мы снова применим тот механизм и создадим класс Content, объекты которого будут ссылаться на другие модели, представляющие конкретный тип содержимого модуля.

Отредактируйте файл models.py приложения courses и импортируйте следующее:

```
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey
```

Затем опишите новый класс таким образом:

```
class Content(models.Model):
    module = models.ForeignKey(Module,
                              related_name='contents',
                              on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType,
                                     on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    item = GenericForeignKey('content_type', 'object_id')
```

Это модель Content. Модуль курса может содержать множество объектов этого типа, поэтому мы используем ForeignKey на модель Module. Также мы выполнили обобщенную связь, чтобы соединить объекты типа Content с любой другой моделью, представляющей тип содержимого. Помните, чтобы обобщенные связи работали, нам необходимо создать три поля в модели:

- `content_type` – внешний ключ, `ForeignKey`, на модель `ContentType`;
- `object_id` – идентификатор связанного объекта типа `PositiveIntegerField`;
- `item` – поле типа `GenericForeignKey`, которое обобщает данные из предыдущих двух.

Только поля `content_type` и `object_id` будут представлены в базе данных соответствующими столбцами. Поле `item` используется только в Python-коде и позволяет вам получить или задать связанный объект.

Для каждого типа содержимого мы создадим отдельные модели. Они будут иметь общие данные и отличаться только полем, которое представляет содержимое модуля.

Виды наследования моделей Django

Для реализации таких моделей нам пригодится наследование. В Django существует *наследование моделей*, которое работает аналогично наследованию класса и может быть реализовано одним из трех способов:

- абстрактные модели полезны, когда вы хотите описать некоторую общую информацию. Для них не создается таблица в базе данных;
- наследование с помощью нескольких таблиц применимо, когда каждая модель в иерархии рассматривается как обособленная и может быть использована отдельно от других. В этом случае для каждой модели создается ее собственная таблица;
- прокси-модели полезны, когда вы хотите хранить те же данные для всех моделей, но реализовать для каждой из них какую-то отдельную функциональность (например, создать методы, переопределить или добавить менеджеры, использовать другие опции класса `Meta`). Для прокси-моделей таблицы в базе данных не создаются.

Давайте рассмотрим каждый из этих способов подробнее.

Абстрактные модели

Абстрактная модель – это базовый класс. В нем необходимо определить поля, которые будут общими для всех дочерних классов. Django не добавляет в базу данных таблицу для абстрактной модели, но для каждого дочернего класса включает столбцы, соответствующие полям родительского класса.

Чтобы указать, что модель является абстрактной, достаточно задать атрибут `abstract=True` в опциях класса `Meta`. Django распознает такую настройку и не создает таблицу в базе данных. Чтобы создать дочернюю модель, просто унаследуйте ее от абстрактной.

Этот пример демонстрирует абстрактную модель, `Content`, и дочернюю, `Text`:

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
```

```
        abstract = True

class Text(BaseContent):
    body = models.TextField()
```

В этом случае в базе данных будет создана только таблица для модели `Text` со столбцами `title`, `created` и `body`.

Наследование с несколькими таблицами

В случае *наследования с несколькими таблицами* для каждой модели создается соответствующая таблица. Django делает ссылку типа `OneToOneField` на родительскую модель из дочерней.

Чтобы применить этот способ, достаточно унаследовать класс дочерней модели от родительской, как это показано в примере:

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class Text(BaseContent):
    body = models.TextField()
```

Django автоматически добавит поле `OneToOneField` в модель `Text` и создаст две таблицы в базе данных.

Прокси-модели

Они используются, когда модели хранят одинаковые данные, но поведение классов отличается (например, у них разные методы или другие опции). Все они работают с одной таблицей. Чтобы создать прокси-модель, добавьте в класс `Meta` атрибут `proxy=True`.

Давайте посмотрим, как это выглядит, на конкретном примере:

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

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class OrderedContent(BaseContent):
    class Meta:
        proxy = True
        ordering = ['created']

    def created_delta(self):
        return timezone.now() - self.created
```

Здесь мы создали прокси-модель `OrderedContent`. Она добавляет сортировку по умолчанию для `QuerySet`'ов и метод `created_delta()`. Обе модели работают с одной и той же таблицей в базе данных, и обратиться к объектам можно через ORM для каждой модели.

Создание моделей содержимого курса

Модель Content приложения courses определяет обобщенную связь с различными типами содержимого. Мы создадим несколько моделей для каждого типа. Все они будут иметь общие и некоторые специфичные поля. Мы будем использовать абстрактную модель.

Отредактируйте файл `models.py` приложения courses и добавьте в него такой код:

```
class ItemBase(models.Model):
    owner = models.ForeignKey(User,
                              related_name='%s_related',
                              on_delete=models.CASCADE)
    title = models.CharField(max_length=250)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True

    def __str__(self):
        return self.title

class Text(ItemBase):
    content = models.TextField()

class File(ItemBase):
    file = models.FileField(upload_to='files')

class Image(ItemBase):
    file = models.FileField(upload_to='images')

class Video(ItemBase):
    url = models.URLField()
```

В этом фрагменте вы создали абстрактную модель `ItemBase`, задав в опциях класса `Meta` атрибут `abstract=True`. Она содержит четыре поля: `owner`, `title`, `created` и `updated`, – которые будут общими для всех дочерних моделей. Поле `owner` содержит данные пользователя, который создал объект. Так как в дочерних классах будет присутствовать это поле, необходимо задать `related_name` для каждого из них. Django задает связанное наименование `related_name` в виде `%(class)s`, но мы определили его как `'%(class)s_related'`. Таким образом, объекты каждой дочерней модели будут доступны по именам `text_related`, `file_related`, `image_related` и `video_related`.

Мы определили несколько типов содержимого, унаследованных от класса `ItemBase`:

- `Text` – для текста;
- `File` – для файлов, например PDF;
- `Image` – для картинок;
- `Video` – для видео. Мы применили поле `URLField`, чтобы сохранять URL видео для его скачивания.

Каждая дочерняя модель определяет дополнительные поля. Таблицы в базе данных будут созданы для моделей Text, File, Image и Video. Для ItemBase Django не добавит таблицу, т. к. эта модель является абстрактной.

Отредактируйте модель Content, которую создали раньше, измените ее поле, content_type, как показано ниже:

```
content_type = models.ForeignKey(ContentType,
                                on_delete=models.CASCADE,
                                limit_choices_to={'model__in':(
                                    'text',
                                    'video',
                                    'image',
                                    'file'}})
```

Мы добавили атрибут limit_choices_to, чтобы ограничить типы содержимого ContentType, которые могут участвовать в связи. Чтобы фильтровать объекты ContentType при запросах, указали условие model__in и значения 'text', 'video', 'image' и 'file'.

Давайте создадим миграции. Выполните команду:

```
python manage.py makemigrations
```

Вы увидите такой вывод:

```
Migrations for 'courses':
  courses/migrations/0002_content_file_image_text_video.py
  - Create model Content
  - Create model File
  - Create model Image
  - Create model Text
  - Create model Video
```

Запустите синхронизацию моделей с базой данных:

```
python manage.py migrate
```

Вывод в консоли будет содержать такую строку:

```
Applying courses.0002_content_file_image_text_video... OK
```

Мы создали модели, которые позволят сохранять различное содержимое для модулей курсов, но упустили небольшую деталь. Модули в курсах и содержимое в модулях должны следовать в определенном порядке. Поэтому нам понадобится поле, в котором он будет соблюдаться.

Создание собственных типов полей для модели

В Django предопределено множество полей, с помощью которых можно создавать модели. Но вы можете создать собственное поле, например для того, чтобы хранить особый тип данных или изменить поведение стандартных классов.

Нам нужно поле, которое позволит определить порядок для содержимого курсов. Самый простой способ – использовать поле PositiveIntegerField. Целочисленные значения будут задавать порядок объектов. Но мы не ограничимся

этим, а создадим на основе класса `PositiveIntegerField` свое поле, которое будет выполнять некоторые дополнительные действия.

В нашем поле будут реализованы две дополнительные функции:

- автоматическое назначение порядкового номера, если он не был задан явно. Когда создается новый объект и пользователь не указывает порядок, поле будет заполняться автоматически, основываясь на том, сколько объектов уже создано для модуля. Например, если уже есть два объекта с порядковыми номерами 1 и 2, то новому будет присвоен 3;
- сортировка объектов по порядку номеров. Модули курсов и содержимое модулей всегда будут возвращаться отсортированными внутри своего родительского объекта.

Создайте файл `fields.py` в папке приложения `courses` и добавьте в него такой код:

```
from django.db import models
from django.core.exceptions import ObjectDoesNotExist

class OrderField(models.PositiveIntegerField):
    def __init__(self, for_fields=None, *args, **kwargs):
        self.for_fields = for_fields
        super(OrderField, self).__init__(*args, **kwargs)

    def pre_save(self, model_instance, add):
        if getattr(model_instance, self.attname) is None:
            # Значение пусто.
            try:
                qs = self.model.objects.all()
                if self.for_fields:
                    # Фильтруем объекты с такими же значениями полей,
                    # перечисленных в "for_fields".
                    query = {field: getattr(model_instance, field)\
                             for field in self.for_fields}
                    qs = qs.filter(**query)
                # Получаем заказ последнего объекта.
                last_item = qs.latest(self.attname)
                value = last_item.order + 1
            except ObjectDoesNotExist:
                value = 0
            setattr(model_instance, self.attname, value)
            return value
        else:
            return super(OrderField, self).pre_save(model_instance, add)
```

Это класс поля `OrderField`. Он наследуется от `PositiveIntegerField`, который определен в Django. Конструктор принимает необязательный параметр `for_fields`, чтобы определить поле родительского объекта, относительно которого будет вычислен порядок.

Мы переопределили метод `pre_save()` класса `PositiveIntegerField`. Он выполняется перед тем, как Django сохранит поле в базу данных. В этом методе мы выполняем следующие действия:

- 1) проверяем, существует ли такое значение для объектов модели. Для того чтобы получить имя поля, по которому оно было определено в модели, обращаемся к атрибуту `self.attname`. Если значение поля равно `None`, считываем, чему оно должно быть равно:
 - формируем `QuerySet`, чтобы получить все объекты модели. Класс модели, для которой определено текущее поле, получаем через запись `self.model`;
 - если был задан параметр `for_fields`, получаем для текущего объекта значения этих полей и фильтруем `QuerySet` по ним. Так мы получим только те объекты, которые принадлежат одному родительскому, например все модули курса;
 - получаем объект с максимальным значением порядкового номера из результата фильтрации с помощью записи `last_item = qs.latest(self.attname)`. Если не найдено ни одного объекта, присваиваем текущему порядковый номер 0;
 - если объект найден, то присваиваем текущему номер, больший на единицу;
- 2) если поле заполнено пользователем, ничего не делаем.

i Когда вы создаете собственные поля для моделей, старайтесь делать их универсальными. Избегайте явного задания данных или полей моделей, от которых они могут зависеть. Ваши поля должны быть применимы не для одной модели, а для всех.

Больше информации о создании своих классов для полей моделей можно найти на странице <https://docs.djangoproject.com/en/2.0/howto/custom-model-fields/>.

Добавление собственного поля сортировки в модели

Давайте применим в модели поле, которое мы только что создали. Отредактируйте файл `models.py` приложения `courses`, импортируйте класс `OrderField` и добавьте поле в модель `Module`:

```
from .fields import OrderField

class Module(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['course'])
```

Новое поле называется `order`. Оно будет рассчитываться автоматически для каждого модуля в рамках одного курса, т. к. мы указали `for_fields=['course']`. Таким образом, при создании нового модуля его порядок будет больше на единицу, чем у предыдущего модуля курса. Теперь отредактируйте метод `__str__()` модели `Module`, как показано ниже:

```
class Module(models.Model):
    # ...
    def __str__(self):
        return '{}. {}'.format(self.order, self.title)
```

Содержимое модулей также должно сортироваться по порядку. Добавьте новое поле `OrderField` в модель `Content`:

```
class Content(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['module'])
```

Здесь мы использовали запись `for_fields=['module']`, следовательно, порядковые номера объектов будут задаваться в рамках одного модуля. Теперь определите сортировку по умолчанию в классе `Meta` для моделей `Module` и `Content`:

```
class Module(models.Model):
    # ...
    class Meta:
        ordering = ['order']

class Content(models.Model):
    # ...
    class Meta:
        ordering = ['order']
```

На текущий момент модели `Module` и `Content` должны выглядеть таким образом:

```
class Module(models.Model):
    course = models.ForeignKey(Course,
                              related_name='modules',
                              on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    order = OrderField(blank=True, for_fields=['course'])

    class Meta:
        ordering = ['order']

    def __str__(self):
        return '{}. {}'.format(self.order, self.title)

class Content(models.Model):
    module = models.ForeignKey(Module,
                              related_name='contents',
                              on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType,
                                     on_delete=models.CASCADE,
                                     limit_choices_to={'model__in':(
                                         'text',
                                         'video',
                                         'image',
                                     )})
    object_id = models.PositiveIntegerField()
    item = GenericForeignKey('content_type', 'object_id')
    order = OrderField(blank=True, for_fields=['module'])

    class Meta:
        ordering = ['order']
```


Давайте создадим миграцию, которая перенесет эти изменения в базу данных. Откройте консоль и выполните команду:

```
python manage.py makemigrations courses
```

Вы увидите вывод, который содержит такие строки:

```
You are trying to add a non-nullable field 'order' to content without a default; we can't do that (the database needs something to populate existing rows).
```

```
Please select a fix:
```

- 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
- 2) Quit, and let me add a default in models.py Select an option:

Django сообщает о том, что мы не указали значение по умолчанию для полей `order`. Если поле определено с параметром `null=True`, то для всех существующих записей будет проставлено пустое значение, а Django добавит миграцию автоматически, без запроса действия. Мы можем задать значение по умолчанию или отменить создание миграции и добавить параметр `default` в поля `order` моделей, после чего снова выполнить команду.

Введите `1` и нажмите **Enter**, чтобы проставить значение по умолчанию для существующих объектов. Вы увидите такой вывод:

```
Please enter the default value now, as valid Python
```

```
The datetime and django.utils.timezone modules are available, so you can do e.g. timezone.now
```

```
Type 'exit' to exit this prompt
```

```
>>>
```

Введите `0` и снова нажмите **Enter**. Django попросит вас ввести значение по умолчанию для модели `Module`. Выберите первую опцию и задайте `0`. Наконец, миграция будет создана, вы увидите такой вывод:

```
Migrations for 'courses':
```

- ```
courses/migrations/0003_auto_20180326_0704.py
- Change Meta options on content
- Change Meta options on module
- Add field order to content
- Add field order to module
```

Примените изменения в базе данных, выполнив команду:

```
python manage.py migrate
```

Вы увидите сообщение о том, что миграции успешно применились, оканчивающееся такой строкой:

```
Applying courses.0003_auto_20180326_0704... OK
```

Сейчас нужно протестировать наше поле! Откройте консоль и выполните команду:

```
python manage.py shell
```

Создайте новый курс:

```
>>> from django.contrib.auth.models import User
>>> from courses.models import Subject, Course, Module
>>> user = User.objects.last()
>>> subject = Subject.objects.last()
>>> c1 = Course.objects.create(subject=subject, owner=user, title='Course
1', slug='course1')
```

Курс создан и сохранен в базу данных. Добавим в него модулей и посмотрим, какие порядковые номера будут им присвоены. Это можно выполнить следующим образом:

```
>>> m1 = Module.objects.create(course=c1, title='Module 1')
>>> m1.order
0
```

Класс `OrderField` задал значение по умолчанию, 0, т. к. это первый объект модели `Module` для соответствующего курса. Теперь создайте второй модуль:

```
>>> m2 = Module.objects.create(course=c1, title='Module 2')
>>> m2.order
1
```

Порядковый номер был вычислен автоматически путем добавления единицы к номеру последнего модуля этого курса. Давайте добавим третий модуль, явно указав его номер:

```
>>> m3 = Module.objects.create(course=c1, title='Module 3', order=5)
>>> m3.order
5
```

Вы можете убедиться, что если задать номер явно, то класс `OrderField` не будет выполнять никаких действий и просто сохранит переданное число.

Добавьте четвертый модуль:

```
>>> m4 = Module.objects.create(course=c1, title='Module 4')
>>> m4.order
6
```

Мы не указали порядок, поэтому он был задан автоматически. То, как мы реализовали поле `OrderField`, не гарантирует, что порядковые номера будут последовательными. Но при автоматическом формировании поле никогда не задаст предыдущее значение, а использует следующий порядковый номер.

Давайте посмотрим, что произойдет, если добавить новый курс и один модуль для него:

```
>>> c2 = Course.objects.create(subject=subject, title='Course 2',
slug='course2', owner=user)
>>> m5 = Module.objects.create(course=c2, title='Module 1')
>>> m5.order
0
```

Чтобы вычислить номер для нового модуля, использовались только модули второго курса. Так как это первый объект, порядковый номер задан как 0. Это происходит потому, что при определении поля `order` модели `Module` мы указали атрибут `for_fields=['course']`.

Поздравляем! Вы только что создали собственное поле для модели.

## СОЗДАНИЕ СИСТЕМЫ УПРАВЛЕНИЯ СОДЕРЖИМЫМ (CMS)

В предыдущих разделах мы описали модели для платформы онлайн-обучения. Теперь необходимо создать систему управления содержимым сайтов (CMS). CMS позволит преподавателям создавать курсы и управлять их модулями. Нам нужно реализовать такие возможности, как:

- авторизация в системе;
- отображение списка курсов преподавателя;
- создание, редактирование и удаление курсов;
- добавление модулей к курсам;
- добавление содержимого различных типов в модули.

### Добавление системы аутентификации

Мы будем использовать стандартную подсистему аутентификации Django. Преподаватели и студенты курсов будут представлены моделью `User` фреймворка. Таким образом, за доступ пользователя к аккаунту будет отвечать пакет `django.contrib.auth`.

Отредактируйте файл `urls.py` проекта `educa` и добавьте обработчики входа и выхода подсистемы аутентификации Django:

```
from django.contrib import admin
from django.urls import path
from django.contrib.auth import views as auth_views

urlpatterns = [
 path('accounts/login/', auth_views.LoginView.as_view(), name='login'),
 path('accounts/logout/', auth_views.LogoutView.as_view(), name='logout'),
 path('admin/', admin.site.urls),
]
```

### Создание шаблонов аутентификации

Добавьте папку для шаблонов в каталог приложения `courses` и создайте в ней такую структуру файлов:

```
templates/
 base.html
 registration/
 login.html
 logged_out.html
```

Перед тем как определить шаблоны страниц сайта, нам нужно определить базовый шаблон. Отредактируйте файл `base.html` и добавьте в него такой код:

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8" />
 <title>{% block title %}Educa{% endblock %}</title>
 <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
 <div id="header">
 Educa
 <ul class="menu">
 {% if request.user.is_authenticated %}
 Sign out
 {% else %}
 Sign in
 {% endif %}

 </div>
 <div id="content">
 {% block content %}
 {% endblock %}
 </div>

 <script src="https://ajax.googleapis.com/ajax/libs/jquery/
 3.3.1/jquery.min.js"></script>
 <script>
 $(document).ready(function() {
 {% block domready %}
 {% endblock %}
 });
 </script>
</body>
</html>
```

Это базовый шаблон для всех страниц. Он содержит три блока:

- `title` – блок для подстановки заголовка;
- `content` – блок для содержимого страницы;
- `domready` – блок для JavaScript-кода, размещенный внутри функции `$(document).ready()`. Он позволит задать операции, которые будут выполнены после загрузки DOM-страницы.

CSS-стили находятся в папке `static/` каталога приложения `courses` в коде – примере к этой главе. Скопируйте эту папку в свой проект по тому же пути, чтобы они применились.

Отредактируйте файл `registration/login.html` и добавьте в него такой фрагмент:

```
{% extends "base.html" %}
{% block title %}Log-in{% endblock %}
{% block content %}
<h1>Log-in</h1>
<div class="module">
 {% if form.errors %}
 <p>Your username and password didn't match. Please try again.</p>
 {% else %}
 <p>Please, use the following form to log-in:</p>
 {% endif %}
 <div class="login-form">
 <form action="{% url 'login' %}" method="post">
 {{ form.as_p }}
 {% csrf_token %}
 <input type="hidden" name="next" value="{{ next }}" />
 <p><input type="submit" value="Log-in"></p>
 </form>
 </div>
</div>
{% endblock %}
```

Это стандартный шаблон для страницы входа Django.

Отредактируйте файл `registration/logged_out.html`, добавив такой код:

```
{% extends "base.html" %}
{% block title %}Logged out{% endblock %}
{% block content %}
<h1>Logged out</h1>
<div class="module">
 <p>You have been successfully logged out.
 You can log-in again.</p>
</div>
{% endblock %}
```

Эта страница будет показана пользователю, после того как он выйдет из своего аккаунта. Запустите сервер для разработки командой:

```
python manage.py runserver
```

Откройте в браузере `http://127.0.0.1:8000/accounts/login/`. Вы увидите такую страницу входа:

Рис. 10.2 ❖ Страница входа в аккаунт

## Определение обработчиков-классов

Мы определим обработчики для создания, редактирования и удаления курсов, используя для этого не функции, а классы. Откройте файл `views.py` приложения `courses` и вставьте в него такой фрагмент:

```
from django.views.generic.list import ListView
from .models import Course

class ManageCourseListView(ListView):
 model = Course
 template_name = 'courses/manage/course/list.html'

 def get_queryset(self):
 qs = super(ManageCourseListView, self).get_queryset()
 return qs.filter(owner=self.request.user)
```

Это класс `ManageCourseListView`, который будет выступать в роли обработчика запросов. Он наследуется от класса `ListView` Django. Мы переопределили метод `get_queryset()`, чтобы получать курсы, созданные текущим пользователем. Чтобы не дать пользователям возможности редактировать курсы, которые они не создавали, мы переопределим этот метод аналогичным образом в соответствующих обработчиках. Когда появляется необходимость в реализации одинаковой логики в нескольких обработчиках, на помощь приходят примеси.

## Использование примесей для обработчиков

*Примесь*, или миксин, – это класс, который используется при множественном наследовании. При определении вашего класса можно задействовать несколько примесей, каждая из которых добавит часть функций в класс. Примеси удобны в двух случаях:

- вы хотите использовать несколько различных функций в рамках вашего класса;
- вы хотите реализовать одну и ту же функциональность в нескольких классах.

Django предоставляет набор примесей для обработчиков на основе классов. Более подробно о них можно прочесть на странице <https://docs.djangoproject.com/en/2.0/topics/class-based-views/mixins/>.

Мы создадим собственную примесь, чтобы не дублировать код для обработчиков действий с курсами. Откройте файл `views.py` приложения `courses` и измените его таким образом:

```
from django.urls import reverse_lazy
from django.views.generic.list import ListView
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from .models import Course

class OwnerMixin(object):
 def get_queryset(self):
 qs = super(OwnerMixin, self).get_queryset()
 return qs.filter(owner=self.request.user)

class OwnerEditMixin(object):
 def form_valid(self, form):
 form.instance.owner = self.request.user
 return super(OwnerEditMixin, self).form_valid(form)

class OwnerCourseMixin(OwnerMixin):
 model = Course

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin):
 fields = ['subject', 'title', 'slug', 'overview']
 success_url = reverse_lazy('manage_course_list')
 template_name = 'courses/manage/course/form.html'

class ManageCourseListView(OwnerCourseMixin, ListView):
 template_name = 'courses/manage/course/list.html'

class CourseCreateView(OwnerCourseEditMixin, CreateView):
 pass

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):
 pass

class CourseDeleteView(OwnerCourseMixin, DeleteView):
 template_name = 'courses/manage/course/delete.html'
 success_url = reverse_lazy('manage_course_list')
```

В этом фрагменте мы определили две примеси: `OwnerMixin` и `OwnerEditMixin`. Они будут добавлены к нашим обработчикам вместе с такими классами Django, как `ListView`, `CreateView`, `UpdateView` и `DeleteView`. Примесь `OwnerMixin` определяет метод `get_queryset()`. Он используется для получения базового `QuerySet`'а, с которым будет работать обработчик. Мы переопределили этот метод, так чтобы получать только объекты, владельцем которых является текущий пользователь (`request.user`).

Примесь `OwnerEditMixin` определяет метод `form_valid()`. Django вызывает его для обработчиков, которые наследуются от `ModelFormMixin` и работают с формами и модельными формами, например `CreateView` или `UpdateView`. Методы выполняются, когда форма успешно проходит валидацию. Поведение по умолчанию для примеси Django – сохранение объекта в базу данных (для модельных форм) и перенаправление пользователя на страницу по адресу `success_url` (для обычных форм). Мы переопределили этот метод, чтобы автоматически заполнять поле `owner` сохраняемого объекта.

Примесь `OwnerMixin` можно применять для любого обработчика, который работает с моделью, содержащей поле `owner`.

Мы также создали класс `OwnerCourseMixin`, который наследуется от `OwnerMixin`, и добавили для него атрибут `model` – модель, с которой работает обработчик.

Еще одна примесь, `OwnerCourseEditMixin`, содержит такие атрибуты:

- `fields` – поля модели, из которых будет формироваться объект обработчиками `CreateView` и `UpdateView`;
- `success_url` – адрес, на который пользователь будет перенаправлен после успешной обработки формы классами `CreateView` и `UpdateView` или их наследниками. Мы указали шаблон URL'a с именем `manage_course_list`, который добавим чуть позже.

Наконец, мы создали такие обработчики-наследники от класса `OwnerCourseMixin`, как:

- `ManageCourseListView` – список курсов, созданных пользователем. Наследуется от `OwnerCourseMixin` и `ListView`;
- `CourseCreateView` – использует модельную форму для создания нового курса. При создании объекта из данных запроса учитывает поля, определенные в родительском классе `OwnerCourseEditMixin`. Является наследником `CreateView`;
- `CourseUpdateView` – позволяет владельцу курса редактировать его. Наследник двух классов – `OwnerCourseEditMixin` и `UpdateView`;
- `CourseDeleteView` – наследуется от примеси `OwnerCourseMixin` и обработчика `DeleteView`. Задает атрибут `success_url` – адрес, на который пользователь будет перенаправлен после успешного удаления объекта.

## Работа с группами и правами

Мы создали классы для обработки действий с курсами. Сейчас любой пользователь может получить доступ к этим обработчикам, зная URL. Необходимо исключить такую возможность и разрешить редактирование курсов только их владельцам. В подсистему аутентификации Django включена система разрешений, которая позволяет администратору создать группы пользователей и ограничения на доступ к тем или иным действиям. Это то, что нам нужно. Давайте создадим группу «Владелец курса» (`Instructors`) и разрешим ей доступ к обработчикам создания, изменения и удаления курсов.

Запустите сервер разработки и перейдите по адресу `http://127.0.0.1:8000/admin/auth/group/add/`, чтобы создать новую группу пользователей. Добавьте на-



звание Instructors и выберите все разрешения приложения courses, кроме разрешений на модель Subject, как показано ниже:

The screenshot shows the 'Add group' form in Django Admin. The 'Name' field is filled with 'instructors'. Under 'Permissions', there are two panels: 'Available permissions' and 'Chosen permissions'. The 'Available permissions' panel lists various permissions such as 'admin | log entry | Can add log entry', 'admin | log entry | Can change log entry', 'admin | log entry | Can delete log entry', 'auth | group | Can add group', 'auth | group | Can change group', 'auth | group | Can delete group', 'auth | permission | Can add permission', 'auth | permission | Can change permission', 'auth | permission | Can delete permission', 'auth | user | Can add user', and 'auth | user | Can change user'. The 'Chosen permissions' panel lists: 'courses | content | Can add content', 'courses | content | Can change content', 'courses | content | Can delete content', 'courses | course | Can add course', 'courses | course | Can change course', 'courses | course | Can delete course', 'courses | file | Can add file', 'courses | file | Can change file', 'courses | file | Can delete file', 'courses | image | Can add image', and 'courses | image | Can change image'. A 'Remove all' button is at the bottom of the 'Chosen permissions' panel. At the bottom of the form, there are three buttons: 'Save and add another', 'Save and continue editing', and 'SAVE'.

Рис. 10.3 ❖ Создание группы пользователей «Владелец курса»

Как вы можете заметить, для каждой модели существует три разрешения: can add, can change и can delete. После выбора сохраните группу, нажав кнопку **SAVE**.

Django создает разрешения для моделей проекта автоматически, но можно реализовать собственные разрешения. Более подробно об этом можно прочесть на странице <https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#custom-permissions>.

Откройте страницу <http://127.0.0.1:8000/admin/auth/user/add/> и создайте нового пользователя. Добавьте его в группу «Владелец курса», как показано ниже:

The screenshot shows the 'Groups' form in Django Admin. The 'Available groups' panel is empty. The 'Chosen groups' panel contains the 'Instructors' group. A green plus sign is visible at the top right of the 'Chosen groups' panel.

Рис. 10.4 ❖ Добавление пользователя в группу «Владелец курса»

Разрешения группы распространяются на всех пользователей, состоящих в ней. Но Django позволяет выдавать их и индивидуально, т. е. для каждого пользователя. Это можно сделать и через сайт администрирования. У пользователя есть атрибут `is_superuser`, который определяет его принадлежность к суперпользователям сайта. Если флаг равен `True`, то система разрешений Django даст доступ такому пользователю ко всем возможностям системы, несмотря на его связи с группами.

### **Ограничение доступа к обработчикам-классам**

Мы хотим ограничить доступ к обработчикам, так чтобы только пользователи, имеющие права на добавление, редактирование или удаление объектов `Course`, могли к ним обратиться. Для этого применим две примеси, которые определены в пакете `django.contrib.auth`:

- `LoginRequiredMixin` – примесь разрешает доступ к обработчику только авторизованным пользователям, аналогична декоратору `login_required` для обработчиков-функций;
- `PermissionRequiredMixin` – разрешает доступ только пользователям, у которых есть заданное разрешение. Исключением являются суперпользователи. Для них обработчик будет доступен всегда.

Отредактируйте файл `views.py` приложения `courses` и импортируйте следующее:

```
from django.contrib.auth.mixins import LoginRequiredMixin, PermissionRequiredMixin
```

Добавьте еще один родительский класс `LoginRequiredMixin` в примесь `OwnerCourseMixin`:

```
class OwnerCourseMixin(LoginRequiredMixin,
 model = Course
 fields = ['subject', 'title', 'slug', 'overview']
 success_url = reverse_lazy('manage_course_list'))
```

Теперь добавьте атрибут `permission_required`, чтобы задать разрешения, которые будут проверяться при доступе к обработчикам:

```
class CourseCreateView(PermissionRequiredMixin,
 OwnerCourseEditMixin,
 CreateView):
 permission_required = 'courses.add_course'

class CourseUpdateView(PermissionRequiredMixin,
 OwnerCourseEditMixin,
 UpdateView):
 permission_required = 'courses.change_course'

class CourseDeleteView(PermissionRequiredMixin,
 OwnerCourseMixin,
 DeleteView):
 template_name = 'courses/manage/course/delete.html'
 success_url = reverse_lazy('manage_course_list')
 permission_required = 'courses.delete_course'
```

Примесь `PermissionRequiredMixin` добавляет проверку наличия у пользователя разрешения, указанного в атрибуте `permission_required`. Теперь наши обработчики доступны только пользователям, которые имеют соответствующие разрешения.

Давайте добавим шаблоны URL'ов. Создайте новый файл, `urls.py`, в папке приложения `courses` и вставьте в него такой код:

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

urlpatterns = [
 path('mine/',
 views.ManageCourseListView.as_view(),
 name='manage_course_list'),
 path('create/',
 views.CourseCreateView.as_view(),
 name='course_create'),
 path('<pk>/edit/',
 views.CourseUpdateView.as_view(),
 name='course_edit'),
 path('<pk>/delete/',
 views.CourseDeleteView.as_view(),
 name='course_delete'),
]
```

Это шаблоны URL'ов, по которым будет формироваться адрес до обработчиков создания, редактирования и удаления курсов. Отредактируйте файл `urls.py` проекта `edusa` и подключите шаблоны, как показано ниже:

```
from django.urls import path, include

urlpatterns = [
 path('accounts/login/', auth_views.LoginView.as_view(), name='login'),
 path('accounts/logout/', auth_views.LogoutView.as_view(), name='logout'),
 path('admin/', admin.site.urls),
 path('course/', include('courses.urls')),
]
```

Самое время добавить HTML-шаблоны для обработчиков. Создайте в каталоге `templates/` приложения `courses` такую структуру папок и файлов:

```
courses/
 manage/
 course/
 list.html
 form.html
 delete.html
```

Отредактируйте файл `courses/manage/course/list.html`, добавьте в него следующий код:

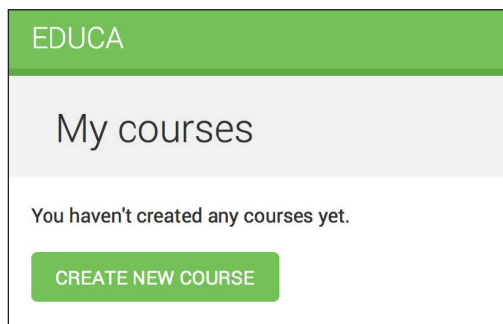
```

{% extends "base.html" %}
{% block title %}My courses{% endblock %}
{% block content %}
<h1>My courses</h1>
<div class="module">
 {% for course in object_list %}
 <div class="course-info">
 <h3>{{ course.title }}</h3>
 <p>
 Edit
 Delete
 </p>
 </div>
 {% empty %}
 <p>You haven't created any courses yet.</p>
 {% endfor %}
 <p>
 Create new course
 </p>
</div>
{% endblock %}

```

Этот шаблон будет использоваться обработчиком `ManageCourseListView`. Он отображает список курсов, созданных текущим пользователем. Мы добавили ссылку на страницу редактирования курса и на страницу создания нового.

Запустите сервер разработки командой `python manage.py runserver` и перейдите по адресу `http://127.0.0.1:8000/accounts/login/?next=/course/mine/`, зарегистрируйтесь как пользователь с группой «Владелец курса». После входа вы будете перенаправлены на `http://127.0.0.1:8000/course/mine/` и увидите такую страницу:



**Рис. 10.5** ❖ Список курсов, созданных пользователем

На этой странице будут появляться курсы, владельцем которых является пользователь.

Давайте добавим шаблон, который покажет форму для создания и редактирования курсов. Откройте файл `courses/manage/course/form.html` и вставьте в него такой код:

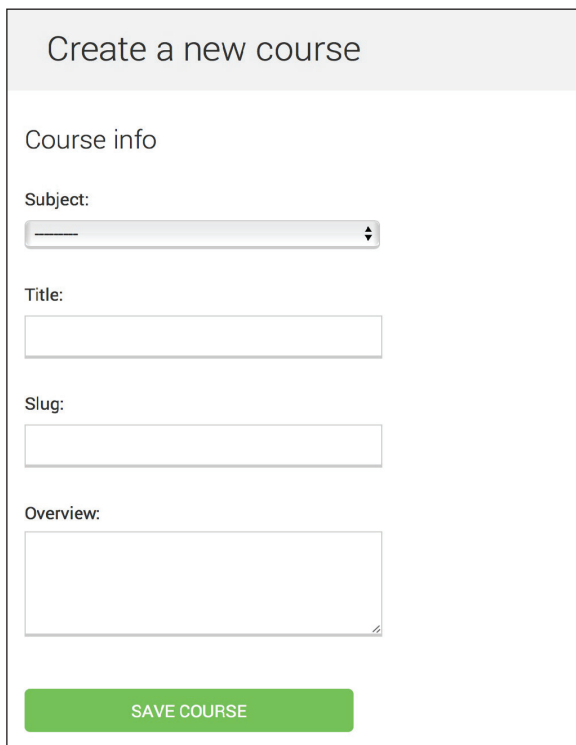
```
{% extends "base.html" %}

{% block title %}
 {% if object %}
 Edit course "{{ object.title }}"
 {% else %}
 Create a new course
 {% endif %}
{% endblock %}

{% block content %}
<h1>
 {% if object %}
 Edit course "{{ object.title }}"
 {% else %}
 Create a new course
 {% endif %}
</h1>
<div class="module">
 <h2>Course info</h2>
 <form action="." method="post">
 {{ form.as_p }}
 {% csrf_token %}
 <p><input type="submit" value="Save course"></p>
 </form>
</div>
{% endblock %}
```

Шаблон `form.html` используется двумя обработчиками: `CourseCreateView` и `CourseUpdateView`. В нем мы проверяем, есть ли переменная `object` в контексте. Если есть, показываем страницу редактирования курса, в противном случае – создания.

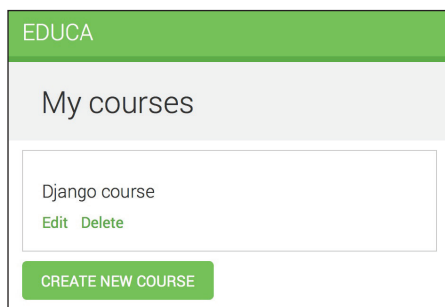
Откройте в браузере <http://127.0.0.1:8000/course/mine/> и нажмите на кнопку **CREATE NEW COURSE**. Вы увидите такую страницу:



The image shows a web form titled "Create a new course". It contains several input fields: a dropdown menu for "Subject", text boxes for "Title" and "Slug", and a larger text area for "Overview". At the bottom of the form is a green button labeled "SAVE COURSE".

Рис. 10.6 ❖ Форма создания курса

Заполните форму и нажмите на кнопку **SAVE COURSE**. Курс сохранится в базу данных, а вы будете перенаправлены на список курсов:



The image shows a web interface for a user named "EDUCA". The page title is "My courses". There is a list of courses, with one course titled "Django course" visible. Below the course name are two links: "Edit" and "Delete". At the bottom of the list is a green button labeled "CREATE NEW COURSE".

Рис. 10.7 ❖ Список курсов содержит добавленный объект

Кликните по ссылке **Edit** рядом с курсом, который вы только что создали. Вы увидите ту же форму, но поля будут заполнены данными курса.

Наконец, отредактируйте файл `courses/manage/course/delete.html` и вставьте в него такой фрагмент:

```
{% extends "base.html" %}
{% block title %}Delete course{% endblock %}
{% block content %}
<h1>Delete course "{{ object.title }}"</h1>
<div class="module">
 <form action="" method="post">
 {% csrf_token %}
 <p>Are you sure you want to delete "{{ object }}"?</p>
 <input type="submit" class="button" value="Confirm">
 </form>
</div>
{% endblock %}
```

Эта страница используется обработчиком `CourseDeleteView`. Он унаследован от стандартного класса Django, `DeleteView`, и реализует удаление объекта.

Откройте браузер и нажмите на ссылку **Delete** рядом с курсом. Вы увидите страницу подтверждения удаления:

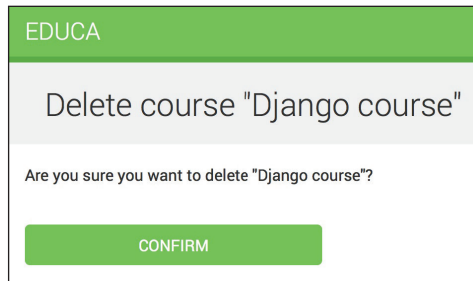


Рис. 10.8 ❖ Подтверждение действия обработчика `CourseDeleteView`

Нажмите кнопку **CONFIRM**. Курс будет удален, а вы перенаправитесь на страницу списка курсов.

На текущий момент владельцы курсов могут управлять ими: создавать, редактировать и удалять. Следующий шаг – добавление CMS, с помощью которой можно будет управлять модулями и их содержимым.

## УПРАВЛЕНИЕ МОДУЛЯМИ КУРСОВ И ИХ СОДЕРЖИМЫМ

Мы создадим систему, которая позволит управлять модулями и их содержимым. Для этого нам понадобятся формы для нескольких модулей курса и не-

скольких объектов содержимого для модуля. Внутри модуля и курса вложенные объекты должны быть расположены в определенном порядке.

## Использование наборов форм для модулей курсов

В Django реализован механизм работы с несколькими формами на одной странице. Такая группа, состоящая из нескольких форм, называется *набором форм*, или формсетом. Набор форм может обработать несколько объектов типа `Form` или `ModelForm`. Все они отправляются на сервер за один раз, а формсет отвечает за то, чтобы определить их количество, понять, сколько форм нужно отображать при редактировании объектов, установить ограничение максимального количества создаваемых объектов и валидацию.

У класса формсета есть метод `is_valid()`, позволяющий проверить корректность всех входящих в него форм за один раз. Кроме того, есть возможность предоставить начальные данные, которыми будут заполнены формы, и указать, сколько пустых форм для каждого типа объектов следует отображать на странице.

Более подробная информация о формсетах приведена в документации на страницах <https://docs.djangoproject.com/en/2.0/topics/forms/formsets/> и <https://docs.djangoproject.com/en/2.0/topics/forms/modelforms/#model-formsets> (для модельных форм).

Так как курс может состоять из нескольких модулей, имеет смысл применить набор форм. Создайте файл `forms.py` в папке приложения `courses` и добавьте в него такой код:

```
from django import forms
from django.forms.models import inlineformset_factory
from .models import Course, Module

ModuleFormSet = inlineformset_factory(Course,
 Module,
 fields=['title', 'description'],
 extra=2,
 can_delete=True)
```

Это набор форм `ModuleFormSet`. Мы формируем его с помощью фабричной функции Django `inlineformset_factory()`. Получаем набор форм, когда объекты одного типа, модули, будут связаны с объектом другого типа, курсами.

Мы также передали в фабричную функцию следующие аргументы:

- `fields` – поля, которые будут добавлены для каждой формы набора;
- `extra` – количество дополнительных пустых форм модулей;
- `can_delete`. Если установить его в `True`, Django добавит для каждой формы модулей чекбокс, с помощью которого можно отметить объект к удалению.

Отредактируйте файл `views.py` приложения `courses` и добавьте в него такой код:

```
from django.shortcuts import redirect, get_object_or_404
from django.views.generic.base import TemplateResponseMixin, View
```



```

from .forms import ModuleFormSet

class CourseModuleUpdateView(TemplateResponseMixin, View):
 template_name = 'courses/manage/module/formset.html'
 course = None

 def get_formset(self, data=None):
 return ModuleFormSet(instance=self.course,
 data=data)

 def dispatch(self, request, pk):
 self.course = get_object_or_404(Course,
 id=pk,
 owner=request.user)
 return super(CourseModuleUpdateView, self).dispatch(request, pk)

 def get(self, request, *args, **kwargs):
 formset = self.get_formset()
 return self.render_to_response({'course': self.course, 'formset': formset})

 def post(self, request, *args, **kwargs):
 formset = self.get_formset(data=request.POST)
 if formset.is_valid():
 formset.save()
 return redirect('manage_course_list')
 return self.render_to_response({'course': self.course,
 'formset': formset})

```

Класс `CourseModuleUpdateView` обрабатывает действия, связанные с набором форм по сохранению, редактированию и удалению модулей для конкретного курса. Этот обработчик наследуется от таких классов:

- `TemplateResponseMixin` – примесь, которая добавит формирование HTML-шаблона и вернет его в качестве ответа на запрос. Она использует шаблон, имя которого задано в атрибуте `template_name`, и добавляет в дочерние классы метод `render_to_response()`, который сформирует результирующую страницу;

- `View` – базовый класс для обработчиков Django.

В этом обработчике мы описали четыре метода:

- `get_formset()` – метод, позволяющий избежать дублирования кода, который отвечает за формирование набора форм;
- `dispatch()` – метод, определенный в базовом классе `View`. Он принимает объект запроса и его параметры и пытается вызвать метод, который соответствует HTTP-методу запроса. Если запрос отправлен с помощью GET, его обработка будет делегирована методу `get()` обработчика; если POST – то методу `post()`. Внутри функции мы пытаемся получить объект типа `Course` с помощью `get_object_or_404()`, т. к. курс необходим как для GET-, так и для POST-запросов, а затем сохраняем его в атрибут класса `course`;
- `get()` – метод, обрабатывающий GET-запрос. Мы создаем пустой набор форм `ModuleFormSet` и отображаем его в шаблоне с данными курса, используя для этого метод родительского класса `render_to_response()`;

- `post()` – метод, обрабатывающий POST-запросы. При этом выполняются следующие шаги:
  - создается набор форм `ModuleFormSet` по отправленным данным;
  - происходит проверка данных с помощью метода `is_valid()` набора форм;
  - если все формы заполнены корректно, сохраняем объекты, вызвав метод `save()`. На этом этапе в базу данных будут сохранены не только данные курса, но и все изменения модулей. Затем перенаправляем пользователя на страницу по URL'у `manage_course_list`. Если хотя бы одна форма набора заполнена некорректно, формируем страницу с отображением ошибок.

Отредактируйте файл `urls.py` приложения `courses` и добавьте в него такой шаблон:

```
path('<pk>/module/',
 views.CourseModuleUpdateView.as_view(),
 name='course_module_update'),
```

Создайте новую папку в каталоге `courses/manage/` и назовите ее `module`. Добавьте файл `courses/manage/module/formset.html` с таким содержимым:

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

{% block title %}
 Edit "{{ course.title }}"
{% endblock %}

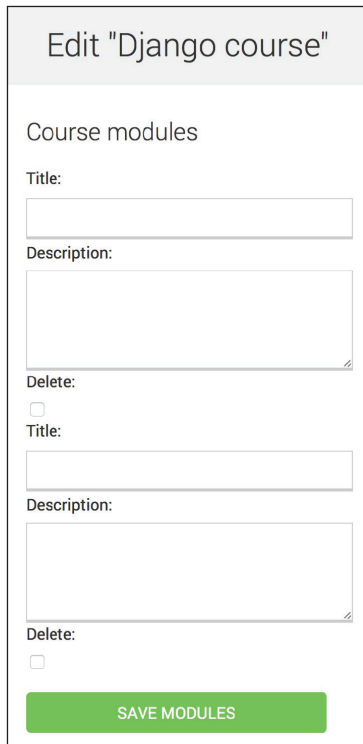
{% block content %}
 <h1>Edit "{{ course.title }}"</h1>
 <div class="module">
 <h2>Course modules</h2>
 <form action="" method="post">
 {{ formset }}
 {{ formset.management_form }}
 {% csrf_token %}
 <input type="submit" class="button" value="Save modules">
 </form>
 </div>
{% endblock %}
```

В этот фрагмент мы добавили тег `<form>`, в котором будет отображена переменная контекста `formset`. Мы также добавили форму управления формсетом с помощью записи `{{ formset.management_form }}`. Она невидима для пользователя и содержит служебную информацию о количестве форм в наборе. Работа завершена. Вы можете убедиться, как легко добавить отображение формсета на странице.

Отредактируйте файл `courses/manage/course/list.html`, добавьте в него ссылку на страницу редактирования модулей курса:

```
Edit
Delete
Edit modules
```

Теперь откройте в браузере страницу <http://127.0.0.1:8000/course/mine/>, создайте новый курс и нажмите ссылку **Edit modules**. Вы увидите такой набор форм:



The screenshot shows a web form titled "Edit 'Django course'". Under the heading "Course modules", there are two identical form entries. Each entry consists of a "Title:" label followed by a text input field, a "Description:" label followed by a larger text area, and a "Delete:" label with an unchecked checkbox. At the bottom of the form is a green button labeled "SAVE MODULES".

Рис. 10.9 ❖ Форма добавления модуля для курса из формсета

Набор содержит форму для каждого объекта `Module`, связанного с курсом. После заполненных форм будут отображены две пустые, т. к. мы указали параметр `extra=2` для `ModuleFormSet`. Когда вы сохраняете такой формсет, Django записывает в базу данных все модули, которые заполнены в формах.

## Добавление содержимого в модуль

Для каждого модуля владельцы курсов могут задавать сколько угодно объектов содержимого: текста, файлов, картинок или видео. Для этого необходимо добавить обработчики, которые позволят создать, изменить или удалить содержимое модуля. На этот раз мы найдем более универсальное решение, т. к. будем работать с объектами четырех моделей.

Вставьте в файл `views.py` приложения `courses` такой код:

```
from django.forms.models import modelform_factory
from django.apps import apps
from .models import Module, Content

class ContentCreateUpdateView(TemplateResponseMixin, View):
 module = None
 model = None
 obj = None
 template_name = 'courses/manage/content/form.html'

 def get_model(self, model_name):
 if model_name in ['text', 'video', 'image', 'file']:
 return apps.get_model(app_label='courses',
 model_name=model_name)

 return None

 def get_form(self, model, *args, **kwargs):
 Form = modelform_factory(model, exclude=['owner',
 'order',
 'created',
 'updated'])

 return Form(*args, **kwargs)

 def dispatch(self, request, module_id, model_name, id=None):
 self.module = get_object_or_404(Module,
 id=module_id,
 course_owner=request.user)

 self.model = self.get_model(model_name)
 if id:
 self.obj = get_object_or_404(self.model,
 id=id,
 owner=request.user)

 return super(ContentCreateUpdateView,
 self).dispatch(request, module_id, model_name, id)
```

Это первая часть обработчика `ContentCreateUpdateView`. Код, который мы описали выше, позволит создавать и редактировать содержимое различных типов. В обработчик добавлены такие методы:

- `get_model()` – возвращает класс модели по переданному имени. Допустимые значения – `Text`, `Video`, `Image` и `File`. Мы обращаемся к модулю `apps` Django, чтобы получить класс модели. Если его не удалось найти по переданному имени, возвращаем `None`;
- `get_form()` – создает форму в зависимости от типа содержимого с помощью функции `modelform_factory()`. Так как модели `Text`, `Video`, `Image` и `File` содержат общие поля, исключим их из формы, чтобы пользователь заполнял только поле непосредственного содержимого (файл, текст, картинку или видео);
- `dispatch()` – получает приведенные ниже данные из запроса и создает соответствующие объекты модуля, модели содержимого:

- `module_id` – идентификатор модуля, к которому привязано содержимое;
- `model_name` – имя модели содержимого;
- `id` – идентификатор изменяемого объекта.

Добавьте методы `get()` и `post()` в класс `ContentCreateUpdateView`:

```
def get(self, request, module_id, model_name, id=None):
 form = self.get_form(self.model, instance=self.obj)
 return self.render_to_response({'form': form, 'object': self.obj})

def post(self, request, module_id, model_name, id=None):
 form = self.get_form(self.model,
 instance=self.obj,
 data=request.POST,
 files=request.FILES)

 if form.is_valid():
 obj = form.save(commit=False)
 obj.owner = request.user
 obj.save()
 if not id:
 # Создаем новый объект.
 Content.objects.create(module=self.module, item=obj)
 return redirect('module_content_list', self.module.id)
 return self.render_to_response({'form': form, 'object': self.obj})
```

Они выполняют следующие действия:

- `get()` – извлекает из GET-параметров запроса данные. Формирует модельные формы для объектов `Text`, `Video`, `Image` или `File`, если объект редактируется, т. е. указан `self.obj`. В противном случае мы отображаем пустую форму для создания объекта;
- `post()` – обрабатывает данные POST-запроса, для чего создает модельную форму и валидирует ее. Если форма заполнена корректно, создает новый объект, указав текущего пользователя, `request.user`, владельцем. Если в запросе был передан ID, значит, объект изменяют, а не создают.

Отредактируйте файл `urls.py` приложения `courses` и добавьте такие шаблоны URL'ов:

```
path('module/<int:module_id>/content/<model_name>/create/',
 views.ContentCreateUpdateView.as_view(),
 name='module_content_create'),

path('module/<int:module_id>/content/<model_name>/<id>/',
 views.ContentCreateUpdateView.as_view(),
 name='module_content_update'),
```

Мы добавили два шаблона, которые будут использоваться для:

- `module_content_create` – дает доступ к обработчику создания нового содержимого и привязки его к модулю. Он содержит параметры `module_id` и `model_name`. ID используется для получения модуля, которому добавляю содержимое, а имя модели нужно, чтобы определить тип содержимого;

- `module_content_update` – дает доступ к обработчику редактирования содержимого. По аналогии с первым шаблоном принимает два параметра (`module_id` и `module_id`), которые использует таким же образом.

Создайте новую папку в каталоге `courses/manage/` и назовите ее `content`. Добавьте файл `courses/manage/content/form.html` с таким содержимым:

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

{% block title %}
 {% if object %}
 Edit content "{{ object.title }}"
 {% else %}
 Add a new content
 {% endif %}
{% endblock %}

{% block content %}
<h1>
 {% if object %}
 Edit content "{{ object.title }}"
 {% else %}
 Add a new content
 {% endif %}
</h1>
<div class="module">
 <h2>Course info</h2>
 <form action="" method="post" enctype="multipart/form-data">
 {{ form.as_p }}
 {% csrf_token %}
 <p><input type="submit" value="Save content"></p>
 </form>
</div>
{% endblock %}
```

Это шаблон для обработчика `ContentCreateUpdateView`. При формировании HTML-страницы мы проверяем, передан ли объект содержимого в переменной `object`. Если передан, отображаем форму редактирования, если нет – форму создания.

Вы могли заметить, что мы указали атрибут `enctype="multipart/form-data"` для тега `<form>`. Это необходимо, чтобы при загрузке картинок и файлов их содержимое было передано на сервер.

Запустите сервер для разработки и перейдите на страницу `http://127.0.0.1:8000/course/mine/`. Нажмите на ссылку **Edit modules** на странице курса и создайте модуль. Откройте консоль Python с помощью команды `python manage.py shell` и запросите ID самого последнего модуля:

```
>>> from courses.models import Module
>>> Module.objects.latest('id').id
```

Запустите сервер для разработки и перейдите по адресу вида `http://127.0.0.1:8000/course/module/6/content/image/create/`, заменив ID модуля на тот, который получили благодаря выполнению предыдущего шага. Вы увидите форму добавления картинки:

**Рис. 10.10** ❖ Форма создания содержимого модуля

Пока не отправляйте ее. Если вы попытаетесь это сделать, получите ошибку, потому что мы пока не определили шаблон URL'а `module_content_list`. Сделаем это чуть-чуть позже.

На текущий момент нам осталось добавить обработчик для удаления содержимого модулей. Откройте файл `views.py` приложения `courses` и добавьте такой класс:

```
class ContentDeleteView(View):
 def post(self, request, id):
 content = get_object_or_404(Content,
 id=id,
 module__course__owner=request.user)

 module = content.module
 content.item.delete()
 content.delete()
 return redirect('module_content_list', module.id)
```

Обработчик `ContentDeleteView` получает объект типа `Content` по переданному ID и удаляет соответствующий объект модели `Text`, `Video`, `Image` или `File`, после

чего ликвидирует объект Content. При успешном завершении действия перенаправляет пользователя на страницу по URL'у с именем module\_content\_list.

Добавьте шаблон URL'а для этого обработчика. В файл urls.py приложения courses вставьте такие строки:

```
path('content/<int:id>/delete/',
 views.ContentDeleteView.as_view(),
 name='module_content_delete'),
```

Теперь владельцы курсов смогут добавлять, изменять и удалять содержимое модулей.

## Управление модулями и их содержимым

Мы создали обработчики для модулей и содержимого. Давайте добавим возможность отображать список всех модулей курса и всего содержимого модуля.

Отредактируйте файл views.py приложения courses и добавьте в него такой код:

```
class ModuleContentListView(TemplateResponseMixin, View):
 template_name = 'courses/manage/module/content_list.html'

 def get(self, request, module_id):
 module = get_object_or_404(Module,
 id=module_id,
 course__owner=request.user)

 return self.render_to_response({'module': module})
```

Это обработчик ModuleContentListView. Он получает из базы данных модуль по переданному ID и генерирует для него страницу подробностей.

Добавьте в файл urls.py приложения courses такие строки:

```
path('module/<int:module_id>/',
 views.ModuleContentListView.as_view(),
 name='module_content_list'),
```

Создайте новый файл, content\_list.html, в папке templates/courses/manage/module/ и вставьте в него следующий код:

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

{% block title %}
Module {{ module.order|add:1 }}: {{ module.title }}
{% endblock %}

{% block content %}
{% with course=module.course %}
<h1>Course "{{ course.title }}"</h1>
<div class="contents">
<h3>Modules</h3>
<ul id="modules">
 {% for m in course.modules.all %}
 <li data-id="{{ m.id }}" {% if m == module %}
 class="selected"{% endif %}>
```



```


 Module {{ m.order|add:1 }}

 {{ m.title }}

 {% empty %}
 No modules yet.
 {% endfor %}

<p>
Edit modules</p>
</div>
<div class="module">
 <h2>Module {{ module.order|add:1 }}: {{ module.title }}</h2>
 <h3>Module contents:</h3>

 <div id="module-contents">
 {% for content in module.contents.all %}
 <div data-id="{{ content.id }}">
 {% with item=content.item %}
 <p>{{ item }}</p>
 Edit
 <form action="{% url "module_content_delete" content.id %}"
 method="post">
 <input type="submit" value="Delete">
 {% csrf_token %}
 </form>
 {% endwith %}
 </div>
 {% empty %}
 <p>This module has no contents yet.</p>
 {% endfor %}
 </div>
 <h3>Add new content:</h3>
 <ul class="content-types">

 Text

 Image

 Video

 File

</div>
{% endwith %}
{% endblock %}

```

Этот шаблон будет формировать все модули курса и их содержимое. Мы обращаемся к списку модулей, чтобы отобразить их в боковой панели. Затем про-

ходим по списку содержимого каждого модуля, обращаясь к `content.item`, чтобы получить соответствующий объект одного из типов `Text`, `Video`, `Image` или `File`. Также отображаем ссылку для добавления содержимого к модулю.

Мы хотим знать, какому типу принадлежит каждый объект содержимого `item`, т. к. нам необходимо формировать ссылку, основываясь на имени модели. Кроме этого, содержимое разных типов по-разному формируется в HTML, поэтому мы могли бы обратиться к опциям класса `Meta` модели через атрибут `_meta`. Но шаблонизатор Django не допускает такую запись, и вы получите ошибку, если используете этот атрибут в шаблоне. Для решения данной проблемы нам понадобится собственный шаблонный фильтр.

Создайте такую структуру папок и файлов в каталоге приложения `courses`:

```
templatetags/
 __init__.py
 course.py
```

Добавьте в файл `course.py` фрагмент кода:

```
from django import template
register = template.Library()

@register.filter
def model_name(obj):
 try:
 return obj._meta.model_name
 except AttributeError:
 return None
```

Только что мы определили шаблонный фильтр `model_name`, который можно применить в шаблонах, чтобы получить имя модели объекта. Для этого используйте запись вида `object|model_name`.

Отредактируйте файл `templates/courses/manage/module/content_list.html` и добавьте такую строку после тега `{% extends %}`:

```
{% load course %}
```

Мы подключили модуль `course`, и только что созданный нами шаблонный фильтр теперь доступен. Найдите следующие строки в файле:

```
<p>{{ item }}</p>
Edit
```

Замените их таким образом:

```
<p>{{ item }} ({{ item|model_name }})</p>
Edit
```

Теперь мы показываем ссылку на редактирование объекта нужного типа. Измените файл `courses/manage/course/list.html`, чтобы ссылка на URL `module_content_list` приобрела указанный ниже вид:

```

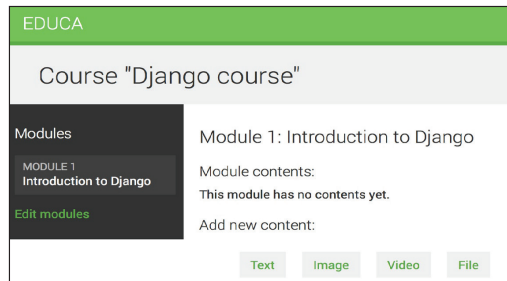
Edit modules
{% if course.modules.count > 0 %}

 Manage contents
{% endif %}

```

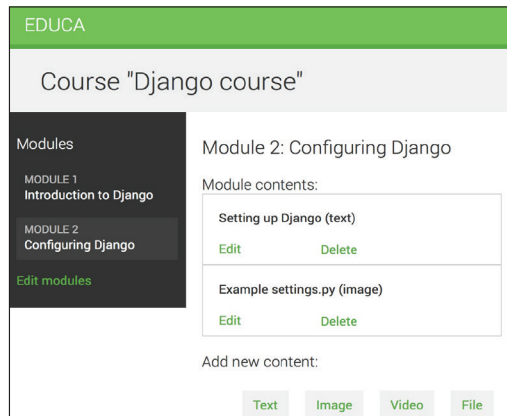
Эта ссылка позволит владельцу курса настроить содержимое первого модуля.

Откройте в браузере страницу <http://127.0.0.1:8000/course/mine/>. Кликните на ссылку Manage contents курса, в котором есть хотя бы один модуль. Вы увидите похожую страницу:



**Рис. 10.11** ❖ Страница модуля со ссылками на добавление нового содержимого

Когда вы выбираете модуль в левой боковой панели, его содержимое отображается в основном блоке страницы. При этом пользователю доступны ссылки на добавление в модуль нового содержимого. Добавьте несколько объектов и посмотрите на страницу модуля. Содержимое должно будет появиться в основном блоке страницы:



**Рис. 10.12** ❖ Страница модуля с содержимым

## Изменения порядка модулей и их содержимого

Мы предусмотрели возможность задавать порядок модулей и содержимого. Теперь нам необходимо добавить возможность сделать это через интерфейс. Для этого мы применим drag-n-drop виджет JavaScript, с помощью которого пользователь сможет перетаскивать объекты списка и благодаря этому задавать их порядок. Для сохранения действий в базе данных реализуем AJAX-запрос.

### Использование примесей из *django-braces*

Пакет *django-braces* – это сторонняя библиотека, которая содержит примеси для Django. Полный список всех примесей можно найти на странице <https://django-braces.readthedocs.io/>.

Мы добавим в проект следующие классы из пакета *django-braces*:

- `CsrfExemptMixin`, чтобы отключить проверку CSRF-токена для некоторых POST-запросов. Нам нужно иметь возможность посылать AJAX-запросы без генерации токена;
- `JsonRequestResponseMixin` – преобразует данные запроса в JSON и сериализует ответ в JSON, возвращая его с типом `application/json`.

Установите пакет *django-braces* с помощью `pip`:

```
pip install django-braces==1.13.0
```

Нам нужен обработчик, который будет получать новый порядок модулей курса в формате JSON. Отредактируйте файл `views.py` приложения `courses` и добавьте в него такой код:

```
from braces.views import CsrfExemptMixin, JsonRequestResponseMixin

class ModuleOrderView(CsrfExemptMixin, JsonRequestResponseMixin, View):
 def post(self, request):
 for id, order in self.request_json.items():
 Module.objects.filter(id=id,
 course__owner=request.user).update(order=order)
 return self.render_json_response({'saved': 'OK'})
```

Это обработчик `ModuleOrderView` для модулей.

Теперь добавим аналогичный обработчик для содержимого модулей. Вставьте в файл `views.py` фрагмент:

```
class ContentOrderView(CsrfExemptMixin, JsonRequestResponseMixin, View):
 def post(self, request):
 for id, order in self.request_json.items():
 Content.objects.filter(id=id,
 module__course__owner=request.user).update(order=order)
 return self.render_json_response({'saved': 'OK'})
```

Добавьте в файл `urls.py` приложения `courses` шаблоны URL'ов для этих обработчиков:

```
path('module/order/',
 views.ModuleOrderView.as_view(),
```

```

 name='module_order'),
path('content/order/',
 views.ContentOrderView.as_view(),
 name='content_order'),

```

Наконец, давайте подключим виджет в шаблоне. Мы будем использовать библиотеку jQuery UI. Она основывается на jQuery и предоставляет плагины для интерфейса (различных виджетов и визуальных эффектов). Для начала необходимо загрузить эту библиотеку. Откройте файл `templates/base.html` приложения `courses` и добавьте подключение скрипта после jQuery таким образом:

```

<script
src="//ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
<script
src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.12.1/jquery-ui.min.js"></script>

```

Теперь отредактируйте файл `courses/manage/module/content_list.html` и добавьте следующий фрагмент:

```

{% block domready %}
$('#modules').sortable({
 stop: function(event, ui) {
 modules_order = {};
 $('#modules').children().each(function(){
 // Обновляем поле порядкового номера.
 $(this).find('.order').text($(this).index() + 1);
 // Связываем порядковый номер с идентификатором объекта.
 modules_order[$(this).data('id')] = $(this).index();
 });
 $.ajax({
 type: 'POST',
 url: '{% url "module_order" %}',
 contentType: 'application/json; charset=utf-8',
 dataType: 'json',
 data: JSON.stringify(modules_order)
 });
 }
});

$('#module-contents').sortable({
 stop: function(event, ui) {
 contents_order = {};
 $('#module-contents').children().each(function(){
 // Связываем порядковый номер с идентификатором объекта.
 contents_order[$(this).data('id')] = $(this).index();
 });

 $.ajax({
 type: 'POST',
 url: '{% url "content_order" %}',
 contentType: 'application/json; charset=utf-8',
 dataType: 'json',

```

```

 data: JSON.stringify(contents_order),
 });
}
});
{% endblock %}

```

Этот код размещен в блоке `{% block domready %}`, поэтому будет выполняться после построения DOM-дерева страницы. Таким образом, мы гарантируем, что все HTML-элементы, необходимые для работы виджета, уже будут созданы на странице. Мы определили объект списка модулей из боковой панели как `sortable`. То же самое сделали для списка содержимого каждого модуля. В этом фрагменте кода мы выполняем следующие действия:

- 1) определяем списки как `sortable`-объекты, используя селекторы jQuery вида `#modules`;
- 2) определяем функцию для события `stop`, срабатывающего каждый раз, когда пользователь заканчивает сортировку объектов;
- 3) создаем пустой словарь `modules_order`. Ключами являются идентификаторы модулей, а значениями – их порядковые номера в списке;
- 4) проходим по списку модулей, пересчитываем порядок для каждого из них и получаем ID, обратившись к атрибуту `data-id`. Добавляем пару «идентификатор – номер» в словарь `modules_order`;
- 5) выполняем AJAX-запрос методом POST на обработчик по URL'у с именем `content_order`. В качестве параметра передаем переменную `modules_order` в виде JSON. Обработчик `ModuleOrderView` сохраняет изменения в базу данных.

Элемент `sortable` для содержимого модуля работает аналогично. Перейдите в браузер и перезагрузите страницу. Теперь вы можете перетаскивать модули в боковой панели, меняя их порядок в рамках курса:

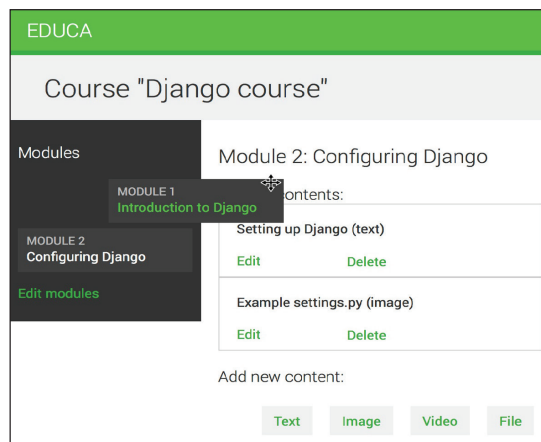


Рис. 10.13 ❖ Изменение порядка модулей курса

Поздравляем! Теперь владельцы курсов могут менять порядок модулей и их содержимого.

## РЕЗЮМЕ

Благодаря этой главе вы узнали, как создать простую, но в то же время достаточно мощную CMS. Вы применили наследование моделей и реализовали собственный класс для поля модели, а также научились работать с наборами форм для различных типов объектов.

В следующей главе вы узнаете, как создавать системы для регистрации студентов курсов, будете формировать различный вид страниц для разных типов объектов и узнаете, как работать с подсистемой кеширования Django.

# Глава 11

## Отображение и кеширование содержимого курсов

В предыдущей главе вы применили такой прием, как наследование моделей, и реализовали с помощью обобщенных отношений платформу для онлайн-обучения с настраиваемыми модулями и содержимым. Для этого вы использовали обработчики в виде классов, наборы форм и AJAX-запросы. В этой главе мы рассмотрим следующие темы:

- создание обработчиков для показа содержимого курсов студентам;
- добавление системы регистрации пользователей;
- управление участием студентов в курсах;
- формирование вида содержимого курсов в зависимости от типа;
- кеширование с помощью стороннего приложения.

Начнем знакомство с создания каталога курсов, чтобы студенты могли искать интересные им и записываться.

### ОТОБРАЖЕНИЕ КУРСОВ

Для каталога нам понадобятся два обработчика:

- для списка курсов с возможностью фильтрации;
- для страницы подробностей курса.

Отредактируйте файл `views.py` приложения `courses` и добавьте в него такой код:

```
from django.db.models import Count
from .models import Subject

class CourseListView(TemplateResponseMixin, View):
 model = Course
 template_name = 'courses/course/list.html'

 def get(self, request, subject=None):
```



```

 subjects = Subject.objects.annotate(
 total_courses=Count('courses'))
 courses = Course.objects.annotate(total_modules=Count('modules'))
 if subject:
 subject = get_object_or_404(Subject, slug=subject)
 courses = courses.filter(subject=subject)
 return self.render_to_response({'subjects': subjects,
 'subject': subject,
 'courses': courses})

```

Это обработчик `CourseListView`. Он является наследником примеси `TemplateResponseMixin` и класса `View`. При обработке запроса на получение курсов мы выполняем следующие действия:

- 1) получаем список всех предметов, добавляя количество курсов по каждому из них. Для этого применяем метод `annotate()` `QuerySet`'а и функцию агрегации `Count()`;
- 2) получаем все доступные курсы, включая количество модулей для каждого из них;
- 3) если в URL'e задан слаг предмета, получаем объект предмета и фильтруем список курсов по нему;
- 4) для формирования результата используем метод `render_to_response()` из примеси `TemplateResponseMixin`.

Давайте создадим обработчик, который будет показывать страницу курса. Добавьте такой фрагмент кода в файл `views.py`:

```

from django.views.generic.detail import DetailView
class CourseDetailView(DetailView):
 model = Course
 template_name = 'courses/course/detail.html'

```

Этот обработчик наследуется от стандартного класса Django, `DetailView`, для отображения объекта конкретной модели. Мы указали два атрибута: `model` и `template_name`. При обработке запроса Django ожидает, что в URL будет передан идентификатор (`pk`) объекта, по которому можно его получить. Затем формирует результат в виде HTML-страницы, сгенерированной из шаблона с именем `template_name`. В контекст шаблона добавляется переменная – объект модели.

Отредактируйте файл `urls.py` проекта `edusa` и добавьте в него подключение шаблонов URL'a, как показано ниже:

```

from courses.views import CourseListView
urlpatterns = [
 # ...
 path('', CourseListView.as_view(), name='course_list'),
]

```

Мы добавили шаблон с именем `course_list` в файл `urls.py` проекта, т. к. хотим обращаться к нему без префикса приложения, по адресу: `http://127.0.0.1:8000/`. Для всех остальных шаблонов, которые будем размещать в соответствующем файле приложения `courses`, в URL будет добавляться префикс `/course/`.

Откройте файл `urls.py` приложения `courses` и добавьте в него следующие строки:

```
path('subject/<slug:subject>/',
 views.CourseListView.as_view(),
 name='course_list_subject'),
path('<slug:slug>/',
 views.CourseDetailView.as_view(),
 name='course_detail'),
```

Мы определили два шаблона URL'ов:

- `course_list_subject` – для отображения всех курсов по выбранному предмету;
- `course_detail` – для страницы подробного описания курса.

Теперь давайте создадим HTML-шаблоны для обработчиков `CourseListView` и `CourseDetailView`. Создайте в каталоге `templates/courses/` приложения `courses` такую структуру папок и файлов:

```
course/
 list.html
 detail.html
```

Вставьте в файл `courses/course/list.html` следующий фрагмент:

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

{% block title %}
 {% if subject %}
 {{ subject.title }} courses
 {% else %}
 All courses
 {% endif %}
{% endblock %}

{% block content %}
<h1>
 {% if subject %}
 {{ subject.title }} courses
 {% else %}
 All courses
 {% endif %}
</h1>
<div class="contents">
 <h3>Subjects</h3>
 <ul id="modules">
 <li {% if not subject %}class="selected"{% endif %}>
 All

 {% for s in subjects %}
 <li {% if subject == s %}class="selected"{% endif %}>

 {{ s.title }}

</div>
```

```


{{ s.total_courses }} courses

{% endfor %}

</div>
<div class="module">
 {% for course in courses %}
 {% with subject=course.subject %}
 <h3>
 {{ course.title }}</h3>
 <p>

 {{ subject }}.
 {{ course.total_modules }} modules.
 Instructor: {{ course.owner.get_full_name }}
 </p>
 {% endwith %}
 {% endfor %}
</div>
{% endblock %}

```

Этот шаблон будет формировать список всех курсов. Мы создаем список предметов Subject. Нажатие на любой его элемент будет перенаправлять пользователя на страницу отфильтрованных курсов, `course_list_subject`. Для выделения предмета, если он выбран, используем CSS-класс `selected`. В заключение проходим по каждому объекту `Course` и показываем количество модулей и имя преподавателя.

Запустите сервер для разработки и перейдите на страницу `http://127.0.0.1:8000/`. Вы увидите похожую страницу:

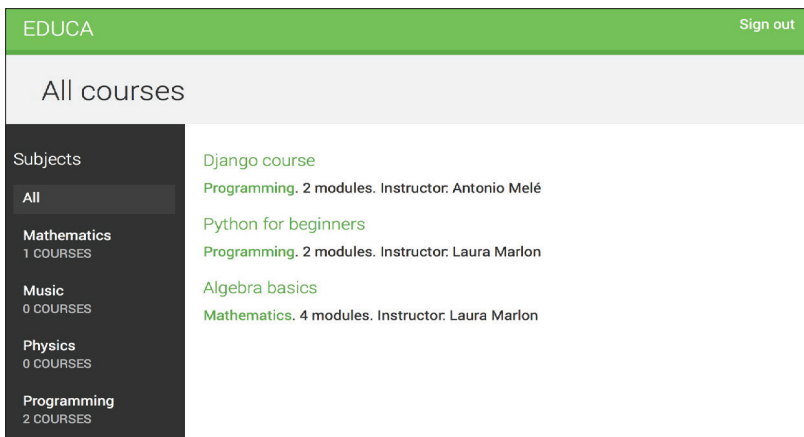


Рис. 11.1 ❖ Список курсов

В левой боковой панели перечислены все предметы, включая количество курсов. Вы можете кликнуть на любой из них и попадете на страницу с отфильтрованным списком курсов.

Отредактируйте файл `courses/course/detail.html` и добавьте такой код:

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

{% block title %}
 {{ object.title }}
{% endblock %}

{% block content %}
 {% with subject=course.subject %}
 <h1>
 {{ object.title }}
 </h1>
 <div class="module">
 <h2>Overview</h2>
 <p>

 {{ subject.title }}.
 {{ course.modules.count }} modules.
 Instructor: {{ course.owner.get_full_name }}
 </p>
 {{ object.overview|linebreaks }}
 </div>
 {% endwith %}
 {% endblock %}
```

В этом шаблоне мы показываем подробную информацию о курсе. Откройте в браузере `http://127.0.0.1:8000/` и кликните на какой-нибудь курс. Вы должны будете увидеть страницу с такой структурой:



Рис. 11.2 ❖ Страница описания курса

Мы создали публичные страницы для доступа всех пользователей к курсам. Далее необходимо добавить возможность регистрации на сайте и записи на курс.

## ДОБАВЛЕНИЕ РЕГИСТРАЦИИ ОБУЧАЮЩИХСЯ

Создайте новое приложение, выполнив в консоли следующую команду:

```
python manage.py startapp students
```

Отредактируйте файл `settings.py` проекта `edusa` и добавьте созданное приложение в список `INSTALLED_APPS`:

```
INSTALLED_APPS = [
 # ...
 'students.apps.StudentsConfig',
]
```

## Обработка регистрации обучающихся в системе

Откройте файл `views.py` приложения `students` и вставьте в него следующий фрагмент:

```
from django.urls import reverse_lazy
from django.views.generic.edit import CreateView
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import authenticate, login

class StudentRegistrationView(CreateView):
 template_name = 'students/student/registration.html'
 form_class = UserCreationForm
 success_url = reverse_lazy('student_course_list')

 def form_valid(self, form):
 result = super(StudentRegistrationView, self).form_valid(form)
 cd = form.cleaned_data
 user = authenticate(username=cd['username'],
 password=cd['password1'])
 login(self.request, user)
 return result
```

Это обработчик регистрации студентов на сайте. Мы используем специальный класс `CreateView`, который предоставляет методы создания объектов заданной модели. Также мы определяем несколько атрибутов модели:

- `template_name` – имя HTML-шаблона, который будет использоваться;
- `form_class` – класс формы для создания объекта. Указанный класс должен быть наследником `ModelForm`. Мы указали форму `UserCreationForm` для создания объектов модели `User`;
- `success_url` – адрес, на который пользователь будет перенаправлен после успешной обработки формы регистрации. Мы получаем URL по имени `student_course_list`, но создадим его чуть позже.

Метод `form_valid()` обработчика будет выполняться при успешной валидации формы. Он должен возвращать объект HTTP-ответа. Мы переопределили его в нашем обработчике, чтобы после регистрации автоматически авторизовать пользователя на сайте.

Добавьте новый файл, `urls.py`, в папку приложения `students` и вставьте в него такой код:

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

urlpatterns = [
 path('register/',
 views.StudentRegistrationView.as_view(),
 name='student_registration'),
]
```

Затем подключите созданный файл в основной конфигурации URL'ов проекта `educa`. Для этого добавьте в файл `urls.py` проекта строку, как показано ниже:

```
urlpatterns = [
 # ...
 path('students/', include('students.urls')),
]
```

Создайте в каталоге приложения `students` следующую структуру папок и файлов:

```
templates/
 students/
 student/
 registration.html
```

Теперь вставьте в шаблон `students/student/registration.html` такой фрагмент:

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

{% block title %}
 Sign up
{% endblock %}

{% block content %}
<h1>
 Sign up
</h1>
<div class="module">
<p>Enter your details to create an account:</p>
 <form action="" method="post">
 {{ form.as_p }}
 {% csrf_token %}
 <p><input type="submit" value="Create my account"></p>
 </form>
</div>
{% endblock %}
```

Запустите сервер разработки и откройте в браузере `http://127.0.0.1:8000/students/register/`. Вы увидите такую страницу:

Рис. 11.3 ❖ Форма регистрации студентов

Не забудьте, что мы пока не определили URL с именем `student_course_list`, который задали как атрибут `success_url` обработчика `StudentRegistrationView`. Если вы попытаетесь зарегистрироваться, Django не сможет найти этот адрес, вследствие чего будет сгенерировано исключение. Мы создадим шаблоны URL'а и соответствующий обработчик чуть позже.

## Реализация записи на курсы

После регистрации пользователи должны иметь возможность записываться на интересные им курсы. Чтобы хранить сведения о том, в каких курсах участвуют студенты, нам понадобится создать связь вида «многие ко многим» между моделями `Course` и `User`.

Отредактируйте файл `models.py` приложения `courses` и добавьте такую строку в модель `Course`:

```
students = models.ManyToManyField(User,
 related_name='courses_joined',
 blank=True)
```

В консоли выполните команду создания миграций:

```
python manage.py makemigrations
```

Вы увидите такой вывод:

```
Migrations for 'courses':
courses/migrations/0004_course_students.py
- Add field students to course
```

Затем выполните команду синхронизации с базой данных:

```
python manage.py migrate
```

Вы увидите вывод, заканчивающийся такой строкой:

```
Applying courses.0004_course_students... OK
```

Миграции успешно применены, теперь вы можете связывать студентов и курсы. Самое время реализовать возможность записываться на курсы.

Создайте новый файл, `forms.py`, в папке приложения `students` и добавьте в него следующий фрагмент:

```
from django import forms
from courses.models import Course

class CourseEnrollForm(forms.Form):
 course = forms.ModelChoiceField(queryset=Course.objects.all(),
 widget=forms.HiddenInput)
```

Мы будем использовать эту форму при записи студентов на курсы. В поле `course` будет содержаться идентификатор курса, на который происходит запись. Мы определили его тип как `ModelChoiceField` и указали виджет `HiddenInput`, т. к. не хотим, чтобы пользователь видел поле. Эта форма будет использоваться в обработчике `CourseDetailView`.

Отредактируйте файл `views.py` приложения `students` и вставьте такой фрагмент:

```
from django.views.generic.edit import FormView
from django.contrib.auth.mixins import LoginRequiredMixin
from .forms import CourseEnrollForm

class StudentEnrollCourseView(LoginRequiredMixin, FormView):
 course = None
 form_class = CourseEnrollForm

 def form_valid(self, form):
 self.course = form.cleaned_data['course']
 self.course.students.add(self.request.user)
 return super(StudentEnrollCourseView, self).form_valid(form)

 def get_success_url(self):
 return reverse_lazy('student_course_detail', args=[self.course.id])
```

Это обработчик `StudentEnrollCourseView`. Он занимается зачислением студентов на курсы. Мы указали родительский класс `LoginRequiredMixin`, поэтому только авторизованные пользователи смогут записываться. Также родительским классом является базовый обработчик Django, `FormView`, который реализует ра-



боту с формой. В атрибуте `form_class` мы указали форму `CourseEnrollForm`, которая при успешной валидации будет создавать связь между студентом и курсом.

Метод `get_success_url()` возвращает адрес, на который пользователь будет перенаправлен после успешной обработки формы. Этот метод аналогичен атрибуту `success_url`. Мы формируем адрес по шаблону URL'а с именем `student_course_detail`, который добавим в следующей части этой главы.

Отредактируйте файл `urls.py` приложения `students` и добавьте такой шаблон:

```
path('enroll-course/',
 views.StudentEnrollCourseView.as_view(),
 name='student_enroll_course'),
```

Давайте добавим на страницу курса кнопку, нажав на которую, можно будет записаться на него. Откройте файл `views.py` приложения `courses` и отредактируйте класс `CourseDetailView`, как показано ниже:

```
from students.forms import CourseEnrollForm

class CourseDetailView(DetailView):
 model = Course
 template_name = 'courses/course/detail.html'

 def get_context_data(self, **kwargs):
 context = super(CourseDetailView, self).get_context_data(**kwargs)
 context['enroll_form'] = CourseEnrollForm(
 initial={'course':self.object})
 return context
```

Мы переопределяем метод базового класса `get_context_data()`, чтобы добавить форму в контекст шаблона. Объект формы при этом содержит скрытое поле с ID курса, поэтому при нажатии кнопки на сервер будут отправлены данные курса и пользователя.

Откройте файл `courses/course/detail.html` и найдите следующую строку:

```
{{ object.overview|linebreaks }}
```

Допишите следом такой фрагмент:

```
{{ object.overview|linebreaks }}
{% if request.user.is_authenticated %}
<form action="{% url 'student_enroll_course' %}" method="post">
 {{ enroll_form }}
 {% csrf_token %}
 <input type="submit" class="button" value="Enroll now">
</form>
{% else %}

 Register to enroll

{% endif %}
```

Это кнопка, по которой студенты смогут записаться на курс, и форма для отправки данных на URL с именем `student_enroll_course`. Они видны на страни-

це только в том случае, если пользователь авторизован на сайте. В противном случае мы отображаем ссылку на страницу регистрации.

Убедитесь, что сервер разработки запущен, и откройте страницу `http://127.0.0.1:8000/`, затем кликните на любой курс. Если вы авторизованы, то увидите кнопку **ENROLL NOW** под описанием:

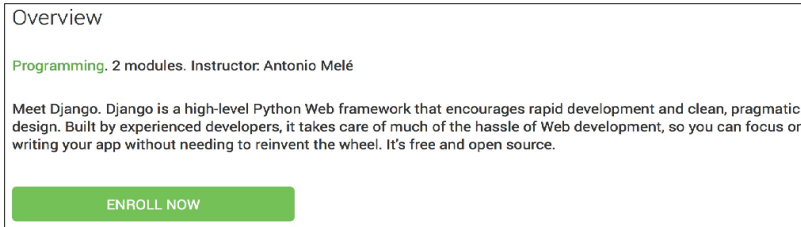


Рис. 11.4 ❖ Страница курса для авторизованных пользователей

Если же вы не авторизованы, то увидите кнопку **REGISTER TO ENROLL**.

## Доступ к СОДЕРЖИМОМУ КУРСОВ

Теперь необходимо добавить страницу, показывающую студентам курсы, в которых они уже принимают участие, и страницу содержимого курса. Для этого отредактируйте файл `views.py` приложения `students` и добавьте в него следующий код:

```
from django.views.generic.list import ListView
from courses.models import Course

class StudentCourseListView(LoginRequiredMixin, ListView):
 model = Course
 template_name = 'students/course/list.html'

 def get_queryset(self):
 qs = super(StudentCourseListView, self).get_queryset()
 return qs.filter(students__in=[self.request.user])
```

Этот обработчик будет формировать список курсов, слушателем которых является студент. Мы используем примесь `LoginRequiredMixin`, чтобы только авторизованные пользователи могли иметь доступ к этой странице. Наш обработчик также наследуется от класса `ListView`, чтобы отображать объекты модели `Course` в виде списка. Чтобы получить только курсы, связанные с текущим пользователем, мы переопределили метод `get_queryset()` и отфильтровали `QuerySet` курсов по связи `ManyToManyField` со студентом.

Теперь добавьте в файл `views.py` такой фрагмент:

```
from django.views.generic.detail import DetailView

class StudentCourseDetailView(DetailView):
 model = Course
```

```

template_name = 'students/course/detail.html'

def get_queryset(self):
 qs = super(StudentCourseDetailView, self).get_queryset()
 return qs.filter(students__in=[self.request.user])

def get_context_data(self, **kwargs):
 context = super(StudentCourseDetailView, self).get_context_data(**kwargs)
 # Получаем объект курса.
 course = self.get_object()
 if 'module_id' in self.kwargs:
 # Получаем текущий модуль по параметрам запроса.
 context['module'] = course.modules.get(id=self.kwargs['module_id'])
 else:
 # Получаем первый модуль.
 context['module'] = course.modules.all()[0]
 return context

```

Это обработчик `StudentCourseDetailView`. Мы переопределили метод `get_queryset()`, чтобы ограничить `QuerySet` курсов и работать только с теми, на которые записан текущий пользователь. Мы также переопределили метод `get_context_data()`, чтобы добавить в контекст шаблона данные о модуле, если его идентификатор был передан в параметре `module_id` URL'а. В противном случае мы показываем содержимое первого модуля. Так студенты смогут переходить от одного модуля курса к другому.

Отредактируйте файл `urls.py` приложения `students` и добавьте в него следующие шаблоны URL'ов:

```

path('courses/',
 views.StudentCourseListView.as_view(),
 name='student_course_list'),

path('course/<pk>/',
 views.StudentCourseDetailView.as_view(),
 name='student_course_detail'),

path('course/<pk>/<module_id>/',
 views.StudentCourseDetailView.as_view(),
 name='student_course_detail_module'),

```

Самое время добавить HTML-шаблоны. Для этого создайте в каталоге `templates/students/` приложения `students` следующую структуру файлов:

```

course/
 detail.html
 list.html

```

Добавьте в шаблон `students/course/list.html` приведенный ниже фрагмент разметки:

```

{% extends "base.html" %}

{% block title %}My courses{% endblock %}

{% block content %}
 <h1>My courses</h1>

```

```

<div class="module">
 {% for course in object_list %}
 <div class="course-info">
 <h3>{{ course.title }}</h3>
 <p>
 Access contents</p>
 </div>
 {% empty %}
 <p>
 You are not enrolled in any courses yet.
 Browse courses
 to enroll in a course.
 </p>
 {% endfor %}
</div>
{% endblock %}

```

Этот шаблон мы будем использовать для списка курсов студента. Помните, что после регистрации пользователь перенаправляется на URL с именем `student_course_list`. Давайте будем перенаправлять студентов на эту же страницу после авторизации на сайте.

Для реализации перенаправления добавьте в файл `settings.py` проекта `educa` следующие строки:

```

from django.urls import reverse_lazy
LOGIN_REDIRECT_URL = reverse_lazy('student_course_list')

```

Это настройка, которую использует пакет Django, `auth`, для определения адреса, куда перенаправлять пользователя после успешной авторизации на сайте, если параметр `next` не задан явно. Теперь при входе в аккаунт студент будет автоматически переходить на список своих курсов.

Отредактируйте файл `students/course/detail.html` и добавьте такой фрагмент:

```

{% extends "base.html" %}

{% block title %}
 {{ object.title }}
{% endblock %}

{% block content %}
 <h1>
 {{ module.title }}
 </h1>
 <div class="contents">
 <h3>Modules</h3>
 <ul id="modules">
 {% for m in object.modules.all %}
 <li data-id="{{ m.id }}" {% if m == module %}
 class="selected"
 {% endif %}>

 Module {{ m.order|add:1 }}


```

```


 {{ m.title }}

{% empty %}
 No modules yet.
{% endfor %}

</div>
<div class="module">
 {% for content in module.contents.all %}
 {% with item=content.item %}
 <h2>{{ item.title }}</h2>
 {{ item.render }}
 {% endwith %}
 {% endfor %}
</div>
{% endblock %}

```

Этот шаблон предназначен для отображения курса и его содержимого студентам, которые на него записались. Для начала мы генерируем список всех модулей курса, выделяя текущий. Затем проходим по содержимому текущего модуля и формируем соответствующий фрагмент разметки с помощью записи вида `{{ item.render }}`. Метод `render()` для модели содержимого курса мы добавим чуть позже.

## Отображение различного типа содержимого

На нашем сайте может быть несколько типов содержимого модуля: текст, картинки, видео или файлы. Нам необходимо по-разному отображать страницу с содержимым курса в зависимости от типа. Давайте реализуем это. Отредактируйте файл `models.py` приложения `courses` и добавьте метод `render()` в модель `ItemBase`, как показано ниже:

```

from django.template.loader import render_to_string
from django.utils.safestring import mark_safe

class ItemBase(models.Model):
 # ...

 def render(self):
 return render_to_string('courses/content/{}.html'.format(
 self._meta.model_name), {'item': self})

```

В этом методе мы вызываем функцию `render_to_string()`, чтобы сгенерировать шаблон с контекстом и получить результат в виде строки. Каждый тип содержимого будет использовать соответствующий ему шаблон, полученный по названию модели. Чтобы динамически формировать имя шаблона, обратимся к атрибуту `self._meta.model_name` модели. Метод `render()` предоставляет общий интерфейс для генерации шаблона под конкретный тип содержимого.

Создайте в каталоге `templates/courses/` приложения `courses` следующую структуру файлов:

```
content/
 text.html
 file.html
 image.html
 video.html
```

В файл `courses/content/text.html` вставьте такую строку:

```
{{ item.content|linebreaks|safe }}
```

Отредактируйте шаблон `courses/content/file.html` и добавьте следующий код:

```
<p>Download file</p>
```

Затем откройте файл `courses/content/image.html` и вставьте в него строку:

```
<p></p>
```

При использовании полей модели `ImageField` и `FileField` необходимо настроить проект в режиме разработки на работу с файлами, загружаемыми пользователями. Для этого откройте `settings.py` проекта `educa` и добавьте две строки:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

Помните, что настройка `MEDIA_URL` – это базовый URL, на основе которого будут формироваться ссылки к медиафайлам, расположенным в файловой системе в папке `MEDIA_ROOT`.

Отредактируйте файл `urls.py` проекта и добавьте в начало файла такой фрагмент:

```
from django.conf import settings
from django.conf.urls.static import static
```

Затем перейдите на последнюю строку файла и вставьте приведенный ниже код:

```
if settings.DEBUG:
 urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Теперь проект готов к загрузке и отображению файлов пользователей. Сервер разработки Django возьмет на себя обязанность по поиску медиафайлов в системе. Этот способ используется только при разработке (настройка `DEBUG` равна `True`). Никогда не применяйте такую конфигурацию в боевом режиме. Об этом мы узнаем в главе 13.

Осталось создать шаблон для отображения видео. Мы будем использовать пакет `django-embed-video`. Это стороннее приложение для Django, которое позволяет встраивать в шаблоны видео из таких источников, как YouTube, Vimeo. Пользователю будет достаточно указать адрес видео.

Установите пакет с помощью `pip`:

```
pip install django-embed-video==1.1.2
```

Подключите его в список установленных приложений в файле `settings.py` проекта:

```
INSTALLED_APPS = [
 # ...
 'embed_video',
]
```

Полную документацию пакета `django-embed-video` можно найти на странице <https://django-embed-video.readthedocs.io/en/latest/>.

Отредактируйте файл `courses/content/video.html` и вставьте такие строки:

```
{% load embed_video_tags %}
{% video item.url "small" %}
```

Теперь запустите сервер для разработки и перейдите в браузере на страницу `http://127.0.0.1:8000/course/mine/`.

Войдите на сайт под именем пользователя, который принадлежит группе «Владелец курса» (Instructors), и добавьте в модуль содержимое разных типов. Чтобы создать видео, просто скопируйте его URL с сайта YouTube (например, <https://www.youtube.com/watch?v=bgV39DlmZ2U>) и вставьте его в поле `url` формы.

После того как вы закончите редактировать курс, перейдите на главную страницу по адресу `http://127.0.0.1:8000/`, кликните на курс и нажмите кнопку **ENROLL NOW**. Вы будете зачислены на курс и перенаправлены по адресу с именем шаблона `student_course_detail`. Вы увидите содержимое курса такого вида:

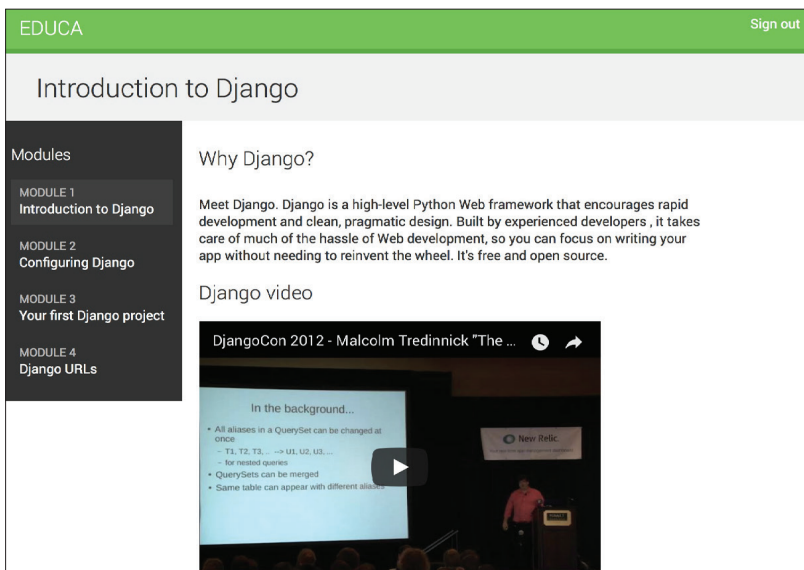


Рис. 11.5 ❖ Страница курса с несколькими типами содержимого (текст и видео)

Поздравляем! Вы только что создали единый интерфейс для формирования шаблонов содержимого разных типов.

## ИСПОЛЬЗОВАНИЕ ФРЕЙМВОРКА ДЛЯ КЕШИРОВАНИЯ

Зачастую обработка запроса в веб-приложении подразумевает три этапа: обращение к базе данных, какие-либо вычисления и преобразования результата выборки и формирование ответа. Это гораздо более требовательный к производительности системы процесс, чем статичный сайт.

При этом для обработки некоторых запросов могут возникать задержки при росте числа пользователей приложения. В таких случаях может помочь кеширование. Вы можете кешировать в базу данных результаты запросов, результаты каких-то вычислений, сгенерированный HTML-шаблон или любые другие дорогостоящие операции. Так вам удастся снизить время отклика на запрос.

Django предоставляет *подсистему кеширования*, которая позволит определить, какие фрагменты кода и результаты вычислений должны быть кешированы. Например, это могут быть отдельные запросы, результат работы конкретного обработчика, часть или весь HTML-шаблон или даже весь сайт. Элементы хранятся в подсистеме кеширования в течение определенного срока, который вы можете переопределить.

Вот каким образом обычно работает подсистема кеширования при обработке HTTP-запроса:

- 1) пытается найти запрашиваемые данные в кеше;
- 2) если это удалось, возвращает ответ;
- 3) если данные не нашлись, выполняет такие шаги:
  - а) делает запрос или вычисления в соответствии с логикой обработчика;
  - б) сохраняет результат в кеш;
  - в) возвращает данные.

Более подробно про работу подсистемы кеширования Django можно прочесть на странице документации <https://docs.djangoproject.com/en/2.0/topics/cache/>.

## Доступные бэкэнды кеширования

В Django реализовано несколько бэкэндов кеширования. Вот их краткое описание:

- `backends.memcached.MemcachedCache`, или `backends.memcached.PyLibMCCache`, – бэкэнды для Memcached. Memcached. Это быстрая и эффективная система кеширования, работающая с оперативной памятью. Какой именно из классов использовать, зависит от того, как вы настроите взаимодействие Memcached со своим Python-кодом;
- `backends.db.DatabaseCache` – использует в качестве хранилища кешей базу данных;



- `backends.filebased.FileBasedCache` – сохраняет результаты в файловую систему, сериализует и хранит каждое кешированное значение в отдельном файле;
- `backends.locmem.LocMemCache` – бэкэнд для кеширования в памяти. Используется по умолчанию;
- `backends.dummy.DummyCache` – фиктивный бэкэнд кеширования, применяемый только при разработке. Он реализует интерфейс бэкэнда кеширования, но фактически не сохраняет никакие результаты.

**i** Для хорошей производительности системы обычно используют системы кеширования, основанные на оперативной памяти, например Memcached.

## Установка Memcached

В проекте мы будем использовать Memcached. Это приложение при запуске получает от операционной системы заданный в настройках объем оперативной памяти для кеширования. Как только в хранилище становится недостаточно места для новых значений, Memcached перезаписывает старые.

Скачайте приложение с сайта: <https://memcached.org/downloads>. Если вы работаете с операционной системой Linux, для установки достаточно выполнить команду:

```
./configure && make && make test && sudo make install
```

Если вы работаете с macOS X, можете установить Memcached через менеджер пакетов Homebrew с помощью команды `brew install memcached`. Чтобы скачать Homebrew, перейдите на сайт <https://brew.sh/>.

После установки Memcached откройте консоль и выполните команду:

```
memcached -l 127.0.0.1:11211
```

По умолчанию приложение будет запущено на порту 11211. Но вы можете задать любой другой с помощью флага `-l`. Более подробную информацию о Memcached можно найти в официальной документации на странице <https://memcached.org/>.

После установки приложения необходимо установить также и Python-пакет для взаимодействия с Memcached из Python-кода. Выполните такую команду:

```
pip install python-memcached==1.59
```

## Настройки кеширования

Django поддерживает несколько настроек для конфигурации бэкэнда кеширования:

- `CACHES` – словарь используемых в проекте систем кеширования;
- `CACHE_MIDDLEWARE_ALIAS` – псевдонимы кешей;
- `CACHE_MIDDLEWARE_KEY_PREFIX` – префиксы для ключей кешей. Когда ваш проект работает с несколькими сайтами, префиксы позволяют избежать коллизий имен ключей;

- `CACHE_MIDDLEWARE_SECONDS` – продолжительность хранения кешированных страниц в секундах.

Подсистему кеширования проекта конфигурируют с помощью настройки `CACHES`. Это словарь, который задает параметры конфигурации для каждого бэкэнда, например такие:

- `BACKEND` – используемый класс для бэкэнда;
- `KEY_FUNCTION` – функция для получения ключа кеша. Она принимает в качестве аргументов префикс, версию и некоторый начальный ключ в виде строки. Затем преобразует их в ключ, используемый для кеширования;
- `KEY_PREFIX` – префикс для всех ключей бэкэнда;
- `LOCATION` – расположение результата кеширования. В зависимости от используемого класса бэкэнда эта настройка может принимать значения в виде пути в файловой системе, хоста и порта или имени для бэкэндов на основе оперативной памяти;
- `OPTIONS` – любые дополнительные параметры, которые может принимать конкретный класс бэкэнда;
- `TIMEOUT` – время хранения результатов кеширования в секундах. По умолчанию оно равно 300, т. е. 5 мин. Если установить эту настройку равной `None`, срок хранения данных не будет ограничен;
- `VERSION` – номер версии кешированных данных. Нужно для добавления версий при кешировании.

## Добавление Memcached в проект

Давайте настроим проект для взаимодействия с Memcached. Откройте файл `settings.py` проекта `edusa` и вставьте в него такой фрагмент:

```
CACHES = {
 'default': {
 'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
 'LOCATION': '127.0.0.1:11211',
 }
}
```

Мы указали в качестве бэкэнда класс `MemcachedCache`. Адрес, по которому Django сможет к нему обратиться, задали в виде `address:port`. При этом, если вы используете несколько рабочих процессов Memcached, в настройке `LOCATION` необходимо задать список их адресов.

## Мониторинг Memcached

Чтобы анализировать работу Memcached, воспользуемся сторонней библиотекой `django-memcache-status`. Это приложение собирает статистику по каждому рабочему процессу Memcached и отображает ее на сайте администрирования. Установите пакет с помощью команды:

```
pip install django-memcache-status==1.3
```

Отредактируйте файл `settings.py` проекта и добавьте приложение `'memcache_status'` в настройку `INSTALLED_APPS`:

```
INSTALLED_APPS = [
 # ...
 'memcache_status',
]
```

Убедитесь, что Memcached и сервер для разработки запущены, после чего откройте в браузере страницу `http://127.0.0.1:8000/admin/`. Войдите под логином администратора системы. Вы увидите такой блок:

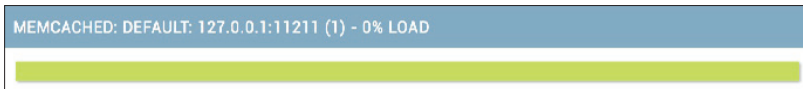


Рис. 11.6 ❖ Информация об использовании памяти Memcached

Этот график демонстрирует статистику использования Memcached. Зеленый цвет соответствует количеству свободного пространства, а красный – занятого кешированными данными. Если вы кликнете на заголовок блока, то увидите детализированную статистику рабочего процесса Memcached.

Мы добавили систему кеширования Memcached и инструмент для анализа ее работы. Самое время реализовать кеширование на сайте!

## Уровни кеширования

Django поддерживает кеширование данных на нескольких уровнях, представленных ниже:

- низкоуровневый API – предоставляет возможность кешировать наименьшую единицу вычислений (запросы или вычисления);
- уровень обработчиков – кешируются результаты обработки одного HTTP-запроса;
- уровень шаблонов – применяется для добавления в кеш результата генерации HTML-шаблона или его фрагмента;
- уровень сайта – кеширует весь сайт.

**i** Хорошо обдумайте стратегии кеширования, перед тем как применять их в проекте. Для начала необходимо оптимизировать наиболее дорогие с точки зрения вычислений запросы и операции.

## Использование низкоуровневого API кеширования

Этот способ позволяет сохранять в кеше данные любого размера. Все необходимые классы и функции реализованы в пакете `django.core.cache`. Чтобы начать использовать низкоуровневый кеш Django, достаточно импортировать его:

```
from django.core.cache import cache
```

Так вы получите доступ к кешу по умолчанию. Запись эквивалентна записи `caches['default']`. Чтобы обратиться к другому кешу, описанному в настройках приложения, используйте следующие строки:

```
from django.core.cache import caches
my_cache = caches['alias']
```

Давайте посмотрим, как работает низкоуровневый API кеширования. Откройте консоль и выполните команду `python manage.py shell`, после чего введите такие строки:

```
>>> from django.core.cache import cache
>>> cache.set('musician', 'Django Reinhardt', 20)
```

Мы обратились к кешу по умолчанию и установили значение `'Django Reinhardt'` по ключу `'musician'` на 20 с с помощью записи `set(key, value, timeout)`. Если не задать третий аргумент метода, Django использует значение по умолчанию из настройки `CACHES`.

Теперь выполните такой код:

```
>>> cache.get('musician')
'Django Reinhardt'
```

Мы получили значение, которое ранее добавили в кеш. Подождите 20 с и попытайтесь снова обратиться к кешу по этому ключу:

```
>>> cache.get('musician')
```

Вы не увидите результата, т. к. ключ `'musician'` был удален из кеша по истечении срока действия.



Избегайте установки неограниченного срока хранения данных в кеше, так вы обезопасите себя от получения неактуальной информации.

Давайте попробуем кешировать результат выполнения `QuerySet`'а:

```
>>> from courses.models import Subject
>>> subjects = Subject.objects.all()
>>> cache.set('all_subjects', subjects)
```

Мы обратились к `QuerySet`'у модели `Subject` и сохранили результат выборки в кеше по ключу `'all_subjects'`. Попробуйте получить его из кеша вызовом метода `get()`:

```
>>> cache.get('all_subjects')
<QuerySet [<Subject: Mathematics>, <Subject: Music>, <Subject: Physics>,
<Subject: Programming>]>
```

Мы можем применить кеширование `QuerySet`'ов в некоторых наших обработчиках. Отредактируйте файл `views.py` приложения `courses` и добавьте такой импорт:

```
from django.core.cache import cache
```

В методе `get()` обработчика `CourseListView` найдите строку

```
subjects = Subject.objects.annotate(total_courses=Count('courses'))
```

и замените ее следующим образом:

```
subjects = cache.get('all_subjects')
if not subjects:
 subjects = Subject.objects.annotate(
 total_courses=Count('courses'))
 cache.set('all_subjects', subjects)
```

В этом коде мы сначала пытаемся получить данные из кеша по ключу `all_students`. Если метод `cache.get()` возвращает `None`, следовательно, `QuerySet` не находится в кеше (например, еще не был туда добавлен или истек срок его хранения). В этом случае выполняем запрос в базу данных, чтобы получить объекты модели `Subject` и количество курсов для каждого из них, после чего сохраняем результат запроса в кеш.

Запустите сервер для разработки и откройте в браузере страницу `http://127.0.0.1:8000/`. Когда обработчик выполняется первый раз, кеш пуст, осуществляется запрос в базу данных. Откройте страницу `http://127.0.0.1:8000/admin/` и разверните блок статистики Memcached. Вы увидите что-то похожее:

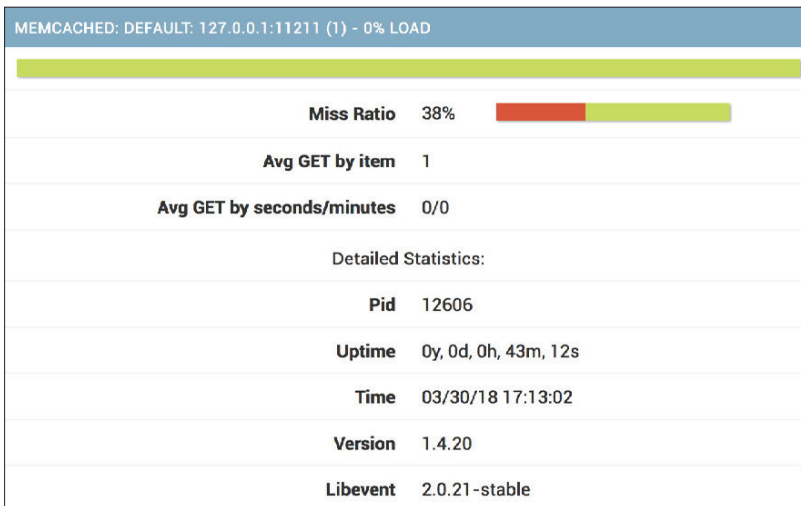


Рис. 11.7 ❖ Статистика использования Memcached

Обратите внимание на поле **Curr Items**, его значение должно быть равно **1**. Это говорит нам о том, что сейчас в кеше находится только один объект. Поле **Get Hits** показывает, сколько запросов в кеш было произведено успешно, а **Get Misses** – наоборот, количество неудачных обращений к кешу. Поле **Miss Ratio** выводит результирующую оценку по этим двум параметрам.

Теперь перейдите обратно на главную страницу <http://127.0.0.1:8000/> и перезагрузите ее несколько раз. Теперь в статистике использования кеша вы увидите обновленные данные с увеличенным количеством успешных запросов **Get Hits** и **Cmd Get**.

### *Кеширование на основе динамических данных*

Часто на практике появляется необходимость кешировать динамические данные, которые формируются в зависимости от каких-то входных параметров. В таких случаях необходимо формировать уникальные ключи кеширования. Отредактируйте файл `views.py` приложения `courses` и измените обработчик `CourseListView`, чтобы он выглядел следующим образом:

```
class CourseListView(TemplateResponseMixin, View):
 model = Course
 template_name = 'courses/course/list.html'

 def get(self, request, subject=None):
 subjects = cache.get('all_subjects')
 if not subjects:
 subjects = Subject.objects.annotate(total_courses=Count('courses'))
 cache.set('all_subjects', subjects)
 all_courses = Course.objects.annotate(total_modules=Count('modules'))
 if subject:
 subject = get_object_or_404(Subject, slug=subject)
 key = 'subject_{}_courses'.format(subject.id)
 courses = cache.get(key)
 if not courses:
 courses = all_courses.filter(subject=subject)
 cache.set(key, courses)
 else:
 courses = cache.get('all_courses')
 if not courses:
 courses = all_courses
 cache.set('all_courses', courses)
 return self.render_to_response({'subjects': subjects,
 'subject': subject,
 'courses': courses})
```

Мы добавили кеширование курсов, отфильтрованных по конкретному предмету. Для первого случая мы используем ключ `all_courses`. Если в запросе был передан ID предмета, создаем ключ кеширования из строки `'subject_{}_courses'.format(subject.id)`.

Важно понимать, что когда вы получаете `QuerySet` из кеша, то нельзя получить на его основе новый, т. е. такая запись не будет работать:

```
courses = cache.get('all_courses')
courses.filter(subject=subject)
```

Вместо этого вам нужно создать новый объект `QuerySet`'а `Course.objects.annotate(total_modules=Count('modules'))`, который не будет выполняться до непосредственного обращения к нему. Затем, если соответствующее значение

не было найдено в кеше, следует применить к нему фильтрацию, например `all_courses.filter(subject=subject)`.

## Кеширование фрагментов шаблонов

Кеширование шаблонов – это более высокоуровневый способ. Чтобы применить его, загрузите библиотеку тегов с помощью записи `{% load cache %}`. После этого вы сможете использовать тег `{% cache %}`, чтобы отметить фрагмент шаблона, подлежащий кешированию. Например:

```
{% cache 300 fragment_name %}
...
{% endcache %}
```

Тэг `{% cache %}` принимает два обязательных аргумента: время жизни значения в секундах и название кешируемого фрагмента. Если вы хотите кешировать фрагмент в зависимости от параметров, можете передавать в тег `{% cache %}` дополнительные параметры для формирования уникального ключа кеша.

Отредактируйте файл `/students/course/detail.html` приложения `students` и добавьте в него следующую строку после тега `{% extends %}`:

```
{% load cache %}
```

Затем найдите такой фрагмент:

```
{% for content in module.contents.all %}
 {% with item=content.item %}
 <h2>{{ item.title }}</h2>
 {{ item.render }}
 {% endwith %}
{% endfor %}
```

Оберните его в шаблонный тег кеширования:

```
{% cache 600 module_contents module %}
 {% for content in module.contents.all %}
 {% with item=content.item %}
 <h2>{{ item.title }}</h2>
 {{ item.render }}
 {% endwith %}
 {% endfor %}
{% endcache %}
```

Теперь фрагмент с именем `module_contents` будет кешироваться для каждого модуля. Мы гарантируем уникальность ключа, передавая текущий модуль из переменной контекста `module`. При кешировании шаблонов контроль уникальности ключей очень важен, чтобы пользователи видели корректные страницы.



Если установить настройку `USE_I18N` равной `True`, при кешировании будет учитываться текущий язык сайта пользователя. Чтобы кешировать фрагменты шаблонов в зависимости от языка, добавьте код в тег `{% cache %}`, например, таким образом: `{% cache 600 name request.LANGUAGE_CODE %}`.

## Кеширование результатов работы обработчиков

Для добавления в кеш результатов обработки HTTP-запросов в пакете `django.views.decorators.cache` реализован декоратор `cache_page`. При его использовании необходимо передать обязательный параметр `timeout` – срок жизни кешированного значения в секундах.

Давайте применим его в проекте. Отредактируйте файл `urls.py` приложения `students` и добавьте такой импорт:

```
from django.views.decorators.cache import cache_page
```

Затем оберните обработчики `student_course_detail` и `student_course_detail_module` в декоратор `cache_page`, как показано ниже:

```
path('course/<pk>/',
 cache_page(60 * 15)(views.StudentCourseDetailView.as_view()),
 name='student_course_detail'),
path('course/<pk>/<module_id>/',
 cache_page(60 * 15)(views.StudentCourseDetailView.as_view()),
 name='student_course_detail_module'),
```

Теперь результаты обработчика `StudentCourseDetailView` будут кешироваться на 15 мин.

**i** При кешировании обработчиков в качестве ключа используется имя соответствующих шаблонов URL'ов. Поэтому если несколько шаблонов ссылаются на один и тот же обработчик, они будут кешированы с разными именами.

## Использование кеширования для сайта

Этот способ самый высокоуровневый, он позволяет реализовать кеширование всего сайта.

Чтобы проект мог работать с кешированием сайтов, отредактируйте файл `settings.py` и добавьте классы `UpdateCacheMiddleware` и `UpdateCacheMiddleware` в настройку `MIDDLEWARE`:

```
MIDDLEWARE = [
 'django.middleware.security.SecurityMiddleware',
 'django.contrib.sessions.middleware.SessionMiddleware',
 'django.middleware.cache.UpdateCacheMiddleware',
 'django.middleware.common.CommonMiddleware',
 'django.middleware.cache.FetchFromCacheMiddleware',
 # ...
]
```

Помните, что промежуточные слои при обработке запроса выполняются в том порядке, в каком указаны в настройке, а при возврате ответа – в противоположном. Поэтому необходимо разместить класс `UpdateCacheMiddleware` перед `CommonMiddleware`, а класс `FetchFromCacheMiddleware` – после.

Теперь добавьте эти строки в файл `settings.py` проекта:



```
CACHE_MIDDLEWARE_ALIAS = 'default'
CACHE_MIDDLEWARE_SECONDS = 60 * 15 # 15 minutes
CACHE_MIDDLEWARE_KEY_PREFIX = 'educa'
```

С помощью таких настроек мы задаем псевдоним кеша и время жизни кешированных данных в 15 мин по умолчанию. Мы также указали префикс для всех ключей, чтобы избежать их пересечения при использовании одного рабочего процесса Memcached с несколькими проектами. Теперь наш сайт будет кеширован и станет возвращать одно и то же содержимое для всех GET-запросов.

Мы добавили кеширование на уровне сайта только для того, чтобы продемонстрировать возможности этого способа. На самом деле это не подходит для платформы онлайн-обучения, т. к. курсы могут меняться, а пользователям необходимо показывать актуальную информацию. В нашем случае наиболее подходящий вариант – кеширование на уровне обработчиков и шаблонов, которые используются при просмотре студентами курсов.

Мы рассмотрели способы кеширования, которые поддерживает Django. В реальных проектах стоит относиться к кешированию с умом и тщательно продумывать стратегии кеширования (начинать с наиболее затратных операций, а только после этого переходить к наименее критичным частям проекта).

## РЕЗЮМЕ

В этой главе мы создали страницы для доступа к курсам и реализовали регистрацию пользователей и запись на курсы, а также подключили к проекту приложение Memcached и рассмотрели несколько уровней кеширования.

В следующей главе мы реализуем RESTful API для проекта.

# Глава 12

## Реализация API

В предыдущей главе вы узнали, как добавить в проект возможность регистрации студентов и записи на курсы, реализовать обработчики для страниц курсов и подключить к Django-проекту систему кеширования. В этой главе мы рассмотрим следующие темы:

- создание RESTful API;
- аутентификация и ограничение доступа к API-обработчикам;
- создание наборов обработчиков и маршрутизаторов.

### Создание RESTful API

Иногда в проектах появляется необходимость взаимодействия с другими системами для отображения данных вашего веб-приложения. Для этих целей реализуют специальный интерфейс, API, который определяет точки взаимодействия двух систем.

Существует несколько способов организации API, но наиболее часто применяют REST. *REST* – это аббревиатура от термина *передача состояния управления* (Representational State Transfer). Такой способ взаимодействия двух систем основан на ресурсах. Модель приложений представляет собой ресурсы, а HTTP-заголовки – действия. Например, GET, POST, PUT и DELETE используются для получения, изменения, создания или удаления объектов. В этом подходе также активно применяют коды HTTP-ответов (например, статусы вида 2xx означают успешную операцию, 4xx – ошибки при обработке запроса и т. д.).

Общепринятые форматы взаимодействия RESTful – JSON и XML. Мы реализуем в проекте поддержку REST API посредством сериализации объектов в JSON. Наш интерфейс взаимодействия будет предоставлять такие точки доступа, как:

- список предметов;
- список всех курсов;
- содержимое курсов;
- запись на курс.

Мы можем реализовать API с помощью обработчиков Django. Однако существует несколько сторонних приложений, которые упрощают создание

REST API. Наиболее часто для этих целей применяют фреймворк Django REST Framework.

## УСТАНОВКА DJANGO REST FRAMEWORK

Фреймворк Django REST позволяет вам легко добавить REST API к Django-проекту. Полную документацию по этому приложению можно найти на странице <https://www.django-rest-framework.org/>.

Откройте консоль и установите его с помощью pip:

```
pip install djangorestframework==3.8.2
```

Отредактируйте файл settings.py проекта educa и добавьте rest\_framework в список INSTALLED\_APPS для активации приложения:

```
INSTALLED_APPS = [
 # ...
 'rest_framework',
]
```

Затем допишите такую настройку:

```
REST_FRAMEWORK = {
 'DEFAULT_PERMISSION_CLASSES': [
 'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
]
}
```

С помощью настройки REST\_FRAMEWORK можно дополнительно сконфигурировать фреймворк Django REST, который поддерживает множество различных параметров для переопределения поведения по умолчанию. Например, настройка DEFAULT\_PERMISSION\_CLASSES отвечает за разрешения по умолчанию для доступа к чтению, созданию, изменению и удалению объектов. Мы указали класс DjangoModelPermissionsOrAnonReadOnly, поэтому анонимные пользователи смогут только просматривать, но не изменять данные, а авторизованные будут иметь доступ ко всем четырем действиям. Более подробно разграничение доступа мы разберем чуть позже в секции «Обработка аутентификации пользователей».

Полный список доступных настроек фреймворка приведен на странице <https://www.django-rest-framework.org/api-guide/settings/>.

## Определение сериализаторов

После установки Django REST нам необходимо задать способ, с помощью которого данные моделей будут преобразованы в JSON и обратно. Фреймворк предоставляет несколько базовых классов сериализаторов, которые работают с объектами:

- Serializer – сериализует обычные Python-классы;
- ModelSerializer – преобразует объекты моделей Django;

- `HyperlinkedModelSerializer` – аналогичен классу `ModelSerializer`, но для связанных объектов формирует http-ссылки, а не их внешние ключи.

Давайте создадим наш первый сериализатор. Добавьте в каталог приложения `courses` такую структуру файлов:

```
api/
 __init__.py
 serializers.py
```

Мы будем описывать всю функциональность, касающуюся API, в папке `api`, чтобы код был структурирован. Отредактируйте файл `serializers.py` и добавьте в него следующий код:

```
from rest_framework import serializers
from ..models import Subject

class SubjectSerializer(serializers.ModelSerializer):
 class Meta:
 model = Subject
 fields = ['id', 'title', 'slug']
```

Это сериализатор для модели `Subject`. Как можно заметить, он определяется аналогично формам Django. Класс `Meta` задает модель и ее поля, которые необходимо сериализовать. Если не указать атрибут `fields`, Django REST будет преобразовывать все поля модели.

Давайте посмотрим, как это работает. Откройте консоль и выполните команду:

```
python manage.py shell
```

Затем введите такие инструкции:

```
>>> from courses.models import Subject
>>> from courses.api.serializers import SubjectSerializer
>>> subject = Subject.objects.latest('id')
>>> serializer = SubjectSerializer(subject)
>>> serializer.data
{'id': 4, 'title': 'Programming', 'slug': 'programming'}
```

В этом примере мы получаем объект предмета `Subject`, создаем его сериализатор `SubjectSerializer` и обращаемся к результату преобразования. Сериализованные данные представлены стандартными типами данных Python: числами, строками и др.

## Принцип работы парсеров и рендереров

Прежде чем вернуть данные в ответе на запрос API, необходимо преобразовать их в определенный формат. Для этого используют рендереры. Верно и обратное утверждение: прежде чем обрабатывать запрос, необходимо сформировать из входных данных подходящие объекты, а в дальнейшем работать с ними. Для этого применяют парсеры.

Давайте посмотрим, как преобразовать входные параметры в подходящие объекты. Выполните в Python-консоли следующие команды:

```
>>> from io import BytesIO
>>> from rest_framework.parsers import JSONParser
>>> data = b'{"id":4, "title": "Programming", "slug": "programming"}'
>>> JSONParser().parse(BytesIO(data))
{'id': 4, 'title': 'Programming', 'slug': 'programming'}
```

Когда у вас есть строка в формате JSON, для получения соответствующего Python-объекта используйте класс `JSONParser`.

Фреймворк Django REST также предоставляет класс `Renderer`, который позволяет формировать ответы на запросы к API. Для определения конкретного класса рендера фреймворк смотрит на тип объекта, который нужно преобразовать. Тип ответа зависит от значения HTTP-заголовка запроса `Accept`. Дополнительно можно настроить формат ответа по префиксам в URL'ах запросов. Например, для формирования JSON-ответа будет задействован класс `JSONRenderer`.

Вернитесь к консоли и выполните следующий код, чтобы сформировать ответ из объекта `serializer` предыдущего примера:

```
>>> from rest_framework.renderers import JSONRenderer
>>> JSONRenderer().render(serializer.data)
```

Вы увидите такой вывод:

```
b'{"id":4, "title": "Programming", "slug": "programming"}'
```

Мы применили класс `JSONRenderer`, чтобы сформировать JSON-ответ. По умолчанию Django REST использует два рендера: `JSONRenderer` и `BrowsableAPIRenderer`. Второй предоставляет веб-интерфейс для просмотра API. Вы можете указать необходимый рендер по умолчанию с помощью атрибута `DEFAULT_RENDERER_CLASSES` в настройке `REST_FRAMEWORK`.

Более подробную информацию о рендерах и парсерах можно найти на страницах <https://www.django-rest-framework.org/api-guide/renderers/> и <https://www.django-rest-framework.org/api-guide/parsers/> соответственно.

## Создание обработчиков списка и подробностей

Во фреймворке Django REST реализован базовый набор классов и примесей, которые помогают разработчику создавать обработчики запросов к API. Они могут использоваться для точек доступа к функциям получения, изменения, создания или удаления объектов. Список всех базовых классов и примесей можно найти в официальной документации на странице <http://www.django-rest-framework.org/api-guide/generic-views/>.

Давайте создадим обработчики списка и подробностей предметов – объектов модели `Subject`. Добавьте новый файл `views.py` в папке `courses/api/` и вставьте в него такой код:

```

from rest_framework import generics
from ..models import Subject
from .serializers import SubjectSerializer

class SubjectListView(generics.ListAPIView):
 queryset = Subject.objects.all()
 serializer_class = SubjectSerializer

class SubjectDetailView(generics.RetrieveAPIView):
 queryset = Subject.objects.all()
 serializer_class = SubjectSerializer

```

В этом коде мы используем базовые классы `ListAPIView` и `RetrieveAPIView`. Мы добавим значение идентификатора предмета, поле `pk`, в качестве параметра в URL, чтобы получить информацию по конкретному предмету. У обоих обработчиков мы определили два атрибута:

- `queryset` – начальный `QuerySet` для получения объектов;
- `serializer_class` – класс для сериализации объектов.

Давайте добавим шаблоны URL'ов для этих обработчиков. Создайте новый файл `urls.py` в папке `courses/api/` и добавьте в него следующий фрагмент:

```

from django.urls import path
from . import views

app_name = 'courses'

urlpatterns = [
 path('subjects/',
 views.SubjectListView.as_view(),
 name='subject_list'),

 path('subjects/<pk>/',
 views.SubjectDetailView.as_view(),
 name='subject_detail'),
]

```

Откройте файл `urls.py` проекта `educa` и подключите шаблоны, которые мы только что определили:

```

urlpatterns = [
 # ...
 path('api/', include('courses.api.urls', namespace='api')),
]

```

Мы добавили параметр `namespace`, чтобы определить пространство имен для URL'ов внешнего API. Убедитесь, что сервер запущен командой `python manage.py runserver`. Откройте консоль и обратитесь к приложению по адресу `http://127.0.0.1:8000/api/subjects/` с помощью утилиты `curl`:

```
curl http://127.0.0.1:8000/api/subjects/
```

Вы увидите ответ, похожий на этот:

```
[
 {"id":1,"title":"Mathematics","slug":"mathematics"},

```

```
{ "id": 2, "title": "Music", "slug": "music" },
{ "id": 3, "title": "Physics", "slug": "physics" },
{ "id": 4, "title": "Programming", "slug": "programming" }
```

```
]
```

HTTP-ответ содержит список всех предметов, которые созданы в вашем приложении, в формате JSON. Если ваша операционная система не поддерживает утилиту `curl` по умолчанию, скачайте ее с сайта <https://curl.haxx.se/dlwiz/>. Кроме этого, вместо `curl` можно использовать любой другой инструмент по отправке HTTP-запросов, например расширение для браузеров Postman. Для его скачивания обратитесь к сайту <https://www.getpostman.com/>.

Откройте в браузере страницу <http://127.0.0.1:8000/api/subjects/>. Вы увидите описание API, которое формирует фреймворк Django REST:

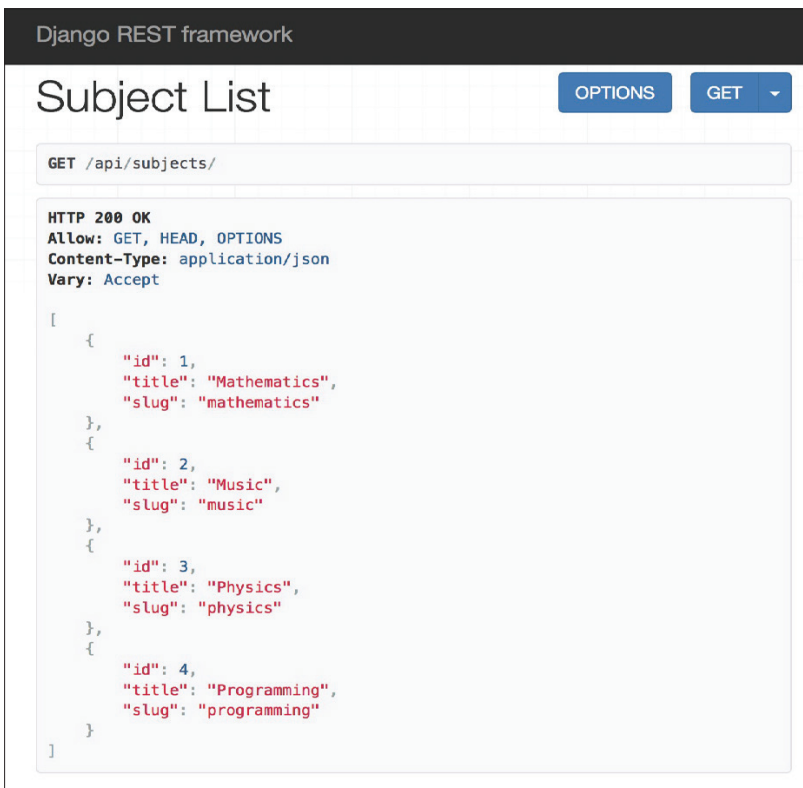


Рис. 12.1 ❖ Описание API в интерфейсе Django REST Framework

Такие страницы генерируются благодаря рендереру `BrowsableAPIRenderer`. Он показывает заголовки и содержимое HTTP-ответа на запрос. Аналогичным образом вы можете посмотреть результат выполнения запроса подроб-

ностей о предмете, если добавите его ID в URL. Например, откройте страницу `http://127.0.0.1:8000/api/subjects/1/`. Вы увидите объект модели `Subject`, преобразованный в формат JSON.

## Создание вложенных сериализаторов

Мы собираемся реализовать сериализатор для модели `Course`. Отредактируйте файл `api/serializers.py` приложения `courses` и добавьте следующий фрагмент:

```
from ..models import Course

class CourseSerializer(serializers.ModelSerializer):
 class Meta:
 model = Course
 fields = ['id', 'subject', 'title', 'slug', 'overview',
 'created', 'owner', 'modules']
```

Давайте посмотрим, как сериализуются объекты модели `Course`. Откройте консоль и выполните команду `python manage.py shell`, затем введите такие строки:

```
>>> from rest_framework.renderers import JSONRenderer
>>> from courses.models import Course
>>> from courses.api.serializers import CourseSerializer
>>> course = Course.objects.latest('id')
>>> serializer = CourseSerializer(course)
>>> JSONRenderer().render(serializer.data)
```

Мы получаем JSON-объект с набором полей, который определили в классе `CourseSerializer`. Обратите внимание на поле связанных объектов, `modules`, которое преобразовано в список идентификаторов модулей курса:

```
"modules": [6, 7, 9, 10]
```

Что делать, если мы хотим не просто перечислить идентификаторы каждого модуля, а добавить дополнительную информацию о них? В этом случае нам нужно создать сериализатор для модели `Module`. Измените файл `api/serializers.py` приложения `courses`, чтобы он выглядел таким образом:

```
from rest_framework import serializers
from ..models import Module

class ModuleSerializer(serializers.ModelSerializer):
 class Meta:
 model = Module
 fields = ['order', 'title', 'description']

class CourseSerializer(serializers.ModelSerializer):
 modules = ModuleSerializer(many=True, read_only=True)

 class Meta:
 model = Course
 fields = ['id', 'subject', 'title', 'slug', 'overview',
 'created', 'owner', 'modules']
```



Мы создали новый класс `ModuleSerializer` для сериализации модели `Module`. Затем добавили его в атрибут `modules` сериализатора курсов `CourseSerializer`, указав, что он является вложенным. В качестве параметра передали `many=True`, чтобы обозначить, что для одного курса может быть множество модулей. Параметр `read_only` сообщает фреймворку, что данные вложенного сериализатора не являются доступными для редактирования.

Откройте консоль и создайте новый объект класса `CourseSerializer`. Сгенерируйте данные курсов с помощью класса `JSONRenderer`, как мы делали это выше. На этот раз информация в теге `modules` будет в виде не списка идентификаторов, а вложенного объекта, описывающего модули курса:

```
"modules": [
 {
 "order": 0,
 "title": "Introduction to overview",
 "description": "A brief overview about the Web Framework."
 },
 {
 "order": 1,
 "title": "Configuring Django",
 "description": "How to install Django."
 },
 ...
]
```

Полное описание и особенности работы сериализаторов приведены на странице <https://www.django-rest-framework.org/api-guide/serializers/>.

## Реализация собственных обработчиков

Во фреймворке Django REST описан класс `APIView`, который реализует внешний API поверх базового класса обработчика Django, `View`. Отличие заключается в том, что `APIView` использует собственные классы для объектов запроса и ответа (`Request` и `Response`) и может обрабатывать исключения `APIException`, возвращая необходимые коды ошибок. Также он реализует методы авторизации и аутентификации, чтобы была возможность ограничить доступ к обработчику API-запросов.

Мы создадим обработчик для записи пользователей на курс. Отредактируйте файл `api/views.py` приложения `courses`, чтобы он выглядел таким образом:

```
from django.shortcuts import get_object_or_404
from rest_framework.views import APIView
from rest_framework.response import Response
from ..models import Course

class CourseEnrollView(APIView):
 def post(self, request, pk, format=None):
 course = get_object_or_404(Course, pk=pk)
 course.students.add(request.user)
 return Response({'enrolled': True})
```

Обработчик `CourseEnrollView` зачисляет студентов на курсы. В представленном выше фрагменте мы выполнили следующие действия:

- 1) создали обработчик, который наследуется от базового класса `APIView`;
- 2) определили метод `post()` для обработки POST-запросов. Другие методы не будут обрабатываться;
- 3) для получения курса, на который происходит зачисление, обратились к POST-параметру запроса `pk`. Если соответствующего курса не существует, будет сгенерировано исключение, вследствие чего пользователь получит ответ с кодом 404;
- 4) если удалось определить курс, создали связь между студентом и объектом модели `Course`, после чего возвратили ответ с успешным статусом.

Отредактируйте файл `api/urls.py` и добавьте шаблон URL'а для обработчика `CourseEnrollView`:

```
path('courses/<pk>/enroll/',
 views.CourseEnrollView.as_view(),
 name='course_enroll'),
```

Сейчас обработчик готов и будет пытаться зачислить всех пользователей, которые обратятся к нему. Но это не совсем корректно, т. к. даже неавторизованные пользователи имеют к нему доступ. Давайте посмотрим, как Django REST работает с разрешениями и ограничивает доступ.

## Обработка аутентификации пользователей

Фреймворк Django REST предоставляет классы для управления аутентификацией. Если пользователь аутентифицирован, в объект запроса добавляется атрибут `request.user` с объектом модели `User`. В противном случае добавляется объект `AnonymousUser`.

Django REST Framework предоставляет такие бэкэнды аутентификации:

- `BasicAuthentication` – бэкэнд для базовой аутентификации по HTTP. Пользователь и пароль отправляются в зашифрованном виде в HTTP-заголовке `Authorization`. Подробное описание этого механизма приведено на странице [https://en.wikipedia.org/wiki/Basic\\_access\\_authentication](https://en.wikipedia.org/wiki/Basic_access_authentication);
- `TokenAuthentication` – бэкэнд для авторизации по токену. В этом случае используется модель `Token` для сохранения токенов пользователей, которые добавляются к запросам в HTTP-заголовке `Authorization`;
- `SessionAuthentication` – бэкэнд на основе сессий Django. Такой способ может быть полезен, когда браузерный код отправляет много AJAX-запросов к API сайта;
- `RemoteUserAuthentication` – делегирует процесс авторизации пользователя веб-серверу, который устанавливает переменную окружения `REMOTE_USER`.

Вы можете реализовать собственный бэкэнд. Для этого достаточно создать класс-наследник от `BaseAuthentication` и описать для него метод `authenticate()`.

В Django REST предусмотрена возможность использования аутентификации на уровне обработчиков или глобально, на весь проект. Управлять этим поведением можно с помощью настройки `DEFAULT_AUTHENTICATION_CLASSES`.

**i** Аутентификация только определяет конкретного пользователя, выполняющего запрос, но не ограничивает доступ к обработчикам по ролям. Чтобы скрыть какие-либо обработчики от части пользователей, используйте разрешения.

Более подробную информацию об аутентификации во фреймворке Django REST можно найти на странице <https://www.django-restframework.org/api-guide/authentication/>.

Давайте добавим к нашему обработчику класс `BasicAuthentication`. Откройте файл `api/views.py` приложения `courses` и задайте атрибут `authentication_classes` обработчику `CourseEnrollView` таким образом:

```
from rest_framework.authentication import BasicAuthentication

class CourseEnrollView(APIView):
 authentication_classes = (BasicAuthentication,)
 # ...
```

Теперь мы будем определять пользователя по HTTP-заголовку `Authorization` запроса.

## Ограничение доступа к обработчикам с помощью разрешений

В Django REST реализована подсистема управления доступом пользователей к сайту по аналогии с тем, как это сделано в Django. Вот, например, несколько разрешений, определенных во фреймворке:

- `AllowAny` – доступ разрешен всем;
- `IsAuthenticated` – доступ имеют только авторизованные пользователи;
- `IsAuthenticatedOrReadOnly` – доступ имеют все, но анонимные пользователи не могут изменять данные;
- `IsAuthenticatedOrReadOnly` – разрешение на основе `django.contrib.auth`. У обработчика с таким уровнем доступа обязательно должен быть задан атрибут `queryset`. Он будет обрабатывать запросы пользователей, которые имеют разрешение на обращение к указанной модели;
- `DjangoObjectPermissions` – разрешения Django по отношению к конкретным объектам.

Если доступ запрещен, автоматически формируется ответ с одним из таких HTTP-статусов:

- HTTP 401 – пользователь не авторизован;
- HTTP 403 – доступ запрещен.

Подробная информация о работе с разрешениями в Django REST Framework с примерами приведена на странице <https://www.django-restframework.org/api-guide/permissions/>.

Отредактируйте файл `api/views.py` приложения `courses` и добавьте атрибут `permission_classes` в обработчик `CourseEnrollView`:

```

from rest_framework.authentication import BasicAuthentication
from rest_framework.permissions import IsAuthenticated

class CourseEnrollView(APIView):
 authentication_classes = (BasicAuthentication,)
 permission_classes = (IsAuthenticated,)
 # ...

```

Мы добавили разрешение `IsAuthenticated`. Теперь анонимные пользователи не смогут получить доступ к обработчику. Давайте проверим, действительно ли это так, отправив POST-запрос к API.

Убедитесь, что сервер для разработки запущен. Откройте другой сеанс в консоли и выполните команду:

```
curl -i -X POST http://127.0.0.1:8000/api/courses/1/enroll/
```

Вы увидите такой вывод:

```

HTTP/1.1 401 Unauthorized
...
{"detail": "Authentication credentials were not provided."}

```

Вы получили ошибку 401, потому что выполнили запрос как анонимный пользователь. Давайте укажем данные студента, зарегистрированного в системе, и проверим, как работает ограничение доступа. Выполните следующую команду, заменив `student:password` на актуальные логин и пароль вашего пользователя:

```
curl -i -X POST -u student:password http://127.0.0.1:8000/api/courses/1/enroll/
```

На этот раз вы увидите вывод с успешным статусом ответа:

```

HTTP/1.1 200 OK
...
{"enrolled": true}

```

Если перейти на сайт администрирования, то можно легко убедиться, что студент действительно зачислен на курс.

## Создание блоков обработчиков и их маршрутизаторов

Класс `ViewSet` позволяет вам реализовать динамический API с несколькими обработчиками и URL'ами, формируемыми с помощью объектов типа `Router`. Используя набор, вы избегаете повторения кода. Такой набор содержит обработчики типичных действий (создания, получения, изменения и удаления объектов) с помощью таких методов, как `list()`, `create()`, `retrieve()`, `update()`, `partial_update()` и `destroy()`.

Давайте создадим набор обработчиков для модели `Course`. Вставьте в файл `api/views.py` следующий фрагмент:

```

from rest_framework import viewsets
from .serializers import CourseSerializer

```

```
class CourseViewSet(viewsets.ReadOnlyModelViewSet):
 queryset = Course.objects.all()
 serializer_class = CourseSerializer
```

Мы унаследовали класс от `ReadOnlyModelViewSet`, который реализует доступ только для чтения через методы `retrieve()` и `list()`. Первый из них возвращает единственный объект, а второй – список. Отредактируйте файл `api/urls.py` и добавьте маршрутизатор к набору обработчиков, который мы только что создали:

```
from django.urls import path, include
from rest_framework import routers
from . import views

router = routers.DefaultRouter()
router.register('courses', views.CourseViewSet)

urlpatterns = [
 # ...
 path('', include(router.urls)),
]
```

Мы создали объект `DefaultRouter` и зарегистрировали набор обработчиков с префиксом `courses`. Маршрутизатор автоматически сформирует необходимые URL'ы и будет передавать запрос в подходящий обработчик.

Откройте в браузере страницу `http://127.0.0.1:8000/api/`. Вы увидите все обработчики набора, соответствующие маршрутизатору `DefaultRouter`:



Рис. 12.2 ❖ Маршрутизатор и соответствующие обработчики запросов

Вы можете сделать запрос по адресу `http://127.0.0.1:8000/api/courses/`, чтобы получить список всех курсов.

Полная документация по набору обработчиков приведена на сайте <https://www.django-rest-framework.org/apiguide/viewsets/>. Подробнее о работе с маршрутизаторами Django REST можно прочитать на странице <https://www.django-rest-framework.org/api-guide/routers/>.

## Добавление собственных обработчиков в набор

Вы можете задать новые обработчики в наборе. Давайте добавим класс `CourseEnrollView` и будем использовать его как собственный обработчик создания объекта. Отредактируйте файл `api/views.py`, чтобы класс `CourseViewSet` выглядел таким образом:

```
from rest_framework.decorators import detail_route

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
 queryset = Course.objects.all()
 serializer_class = CourseSerializer

 @detail_route(methods=['post'],
 authentication_classes=[BasicAuthentication],
 permission_classes=[IsAuthenticated])
 def enroll(self, request, *args, **kwargs):
 course = self.get_object()
 course.students.add(request.user)
 return Response({'enrolled': True})
```

Мы добавили новый метод `enroll()`, который будет обрабатывать нестандартное действие – зачисление студентов на курсы. В этом фрагменте кода мы выполняем следующие шаги:

- 1) используем декоратор `detail_route`, чтобы указать, что метод работает с одним объектом, а не списком;
- 2) в качестве аргумента передаем в декоратор список HTTP-методов, с которыми может работать функция `enroll()`, а также классы для авторизации пользователей и проверки прав доступа;
- 3) вызываем метод `self.get_object()`, чтобы получить объект `Course`;
- 4) добавляем связь `students` текущего пользователя и курса, на который он пытается записаться.

Отредактируйте файл `api/urls.py` и удалите этот шаблон URL'а, т. к. теперь он будет генерироваться автоматически посредством маршрутизатора:

```
path('courses/<pk>/enroll/',
 views.CourseEnrollView.as_view(),
 name='course_enroll'),
```

Удалите также из файла `api/views.py` класс `CourseEnrollView`.

Теперь передача запроса и обработка зачисления студента на курс полностью реализованы с помощью маршрутизатора и набора обработчиков. Адрес, по которому пользователь может записаться на курс, формируется динамически на основе имени метода, `enroll`.

## Создание собственных разрешений

Давайте добавим ограничение доступа к содержимому курсов для пользователей, не записанных на него. Только те, кто зачислен на курс, смогут просматривать модули и содержимое. Самый простой и надежный способ – создать

собственное разрешение. Django предоставляет базовый класс `BasePermission`. Нас интересуют два его метода:

- `has_permission()` – выполняет проверку доступа на уровне обработчика;
- `has_object_permission()` – проверяет доступ к объекту.

Чтобы разрешить доступ, метод должен возвращать `True`, а чтобы запретить – `False`. Создайте новый файл `permissions.py` в папке `courses/api/` и добавьте в него такой код:

```
from rest_framework.permissions import BasePermission

class IsEnrolled(BasePermission):
 def has_object_permission(self, request, view, obj):
 return obj.students.filter(id=request.user.id).exists()
```

Мы создали собственную реализацию разрешения. Класс `IsEnrolled` является наследником `BasePermission` и переопределяет метод `has_object_permission()`. Мы проверяем, является ли текущий пользователь слушателем курса, через атрибут `students` объекта модели `Course`. Чуть позже мы применим это разрешение.

## Сериализация содержимого курсов

Для передачи данных с помощью Django REST нам необходимо сериализовать содержимое курсов. Модель `Content` содержит обобщенную связь с несколькими моделями. В предыдущей главе мы уже добавили метод `gender()` для каждого типа содержимого. Давайте задействуем его для формирования ответа на запросы к API.

Отредактируйте файл `api/serializers.py` приложения `courses` и добавьте такой код:

```
from ..models import Content

class ItemRelatedField(serializers.RelatedField):
 def to_representation(self, value):
 return value.render()

class ContentSerializer(serializers.ModelSerializer):
 item = ItemRelatedField(read_only=True)

 class Meta:
 model = Content
 fields = ['order', 'item']
```

В этом фрагменте мы создали собственное поле, являющееся наследником класса `RelatedField` фреймворка Django REST, и переопределили метод `to_representation()`. Также мы создали сериализатор `ContentSerializer` для модели `Content`, в котором определили поле `item` типа `ItemRelatedField`.

Необходимо создать сериализатор для модели `Module`, который будет формировать данные входящего в модуль содержимого. Затем следует расширить сериализатор модели `Course`. Отредактируйте файл `api/serializers.py` и добавьте в него такой код:

```

class ModuleWithContentsSerializer(serializers.ModelSerializer):
 contents = ContentSerializer(many=True)

 class Meta:
 model = Module
 fields = ['order', 'title', 'description', 'contents']

class CourseWithContentsSerializer(serializers.ModelSerializer):
 modules = ModuleWithContentsSerializer(many=True)

 class Meta:
 model = Course
 fields = ['id', 'subject', 'title', 'slug',
 'overview', 'created', 'owner', 'modules']

```

Давайте реализуем обработчик, работающий аналогично методу `retrieve()`. Он будет возвращать данные курса, его модулей и содержимого. Откройте файл `api/views.py` и измените класс `CourseViewSet` таким образом:

```

from .permissions import IsEnrolled
from .serializers import CourseWithContentsSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
 # ...
 @detail_route(methods=['get'],
 serializer_class=CourseWithContentsSerializer,
 authentication_classes=[BasicAuthentication],
 permission_classes=[IsAuthenticated, IsEnrolled])
 def contents(self, request, *args, **kwargs):
 return self.retrieve(request, *args, **kwargs)

```

В этом методе мы выполняем следующие действия:

- оборачиваем метод в декоратор `detail_route`, т. к. он работает с одним объектом;
- указываем, что метод обрабатывает только GET-запросы;
- обращаемся к сериализатору `CourseWithContentsSerializer` для формирования содержимого курса для ответа;
- используем разрешения `IsAuthenticated` и `IsEnrolled`. Так мы ограничим доступ к курсам, и их содержимое смогут просматривать только записавшиеся студенты;
- вызываем существующий метод `retrieve()`, чтобы вернуть объект модели `Course`.

Откройте в браузере страницу `http://127.0.0.1:8000/api/courses/1/contents/`. Если вы авторизованы как пользователь, зачисленный на курс, с идентификатором, равным 1, то увидите JSON-представление модулей и содержимого курса:

```

{
 "order": 0,
 "title": "Introduction to Django",
 "description": "Brief introduction to the Django Web Framework.",
 "contents": [

```



```
{
 "order": 0,
 "item": "<p>Meet Django. Django is a high-level
Python Web framework
...</p>"
},
{
 "order": 1,
 "item": "\n<iframe width=\"480\" height=\"360\"
src=\"http://www.youtube.com/embed/bgV39DlmZ2U?
wmode=opaque\"
frameborder=\"0\" allowfullscreen></iframe>\n"
}
]
}
```

Вы реализовали простой интерфейс к платформе онлайн-обучения, который позволит другим системам получать данные о курсах. Фреймворк Django REST содержит методы для создания и редактирования объектов с помощью набора обработчиков `ModelViewSet`. Мы затронули основные и наиболее значимые на практике стороны фреймворка Django REST. Ознакомиться с подробным описанием его возможностей и полной документацией вы можете на сайте <https://www.django-rest-framework.org/>.

## РЕЗЮМЕ

В этой главе мы создали RESTful API для взаимодействия приложения с другими системами.

В следующей главе вы узнаете, как развернуть проект в боевом окружении с помощью `uWSGI` и `NGINX`, а также реализуете собственный промежуточный слой и создадите команду управления.

# Глава 13

## Запуск в боевом режиме

В предыдущей главе вы реализовали на сайте RESTful API. В этой вы узнаете, как подготовить проект к запуску в боевом кружении, мы рассмотрим такие темы:

- конфигурация сервера;
- создание собственных промежуточных слоев;
- реализация собственной команды управления.

### СОЗДАНИЕ ОКРУЖЕНИЯ ДЛЯ ЗАПУСКА

Настало время открыть свету наш проект и запустить его на реальном сервере. Для этого нам понадобится выполнить следующие шаги.

1. Настроить проект.
2. Запустить СУБД PostgreSQL.
3. Установить веб-сервер NGINX и настроить uWSGI.
4. Задать доступ к статическим файлам.
5. Защитить сайт с помощью SSL.

### Управление настройками для нескольких окружений

В неучебных проектах зачастую приходится иметь дело с несколькими конфигурациями проекта. У вас будет как минимум два окружения – для локального и боевого запуска, а иногда и дополнительные, например для запуска автотестов. Некоторые настройки будут общими для всех этих конфигураций, но часть из них может отличаться в зависимости от режима запуска. Давайте добавим возможность использовать разные конфигурации для проекта.

Создайте каталог `settings/` рядом с файлом `settings.py` в папке проекта `educa`. Переименуйте файл `settings.py` в `base.py` и переместите его в только что созданный каталог `settings/`. Создайте еще несколько файлов в этой папке, так чтобы получилась следующая структура:

```
settings/
 __init__.py
 base.py
 local.py
 pro.py
```

Мы будем использовать эти файлы для таких целей:

- `base.py` – файл базовых настроек (раньше назывался `settings.py`);
- `local.py` – дополнительная конфигурация для локального запуска разработчиком;
- `pro.py` – настройки, специфичные для боевого режима.

Найдите в файле `settings/base.py` эту строку:

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

И измените ее таким образом:

```
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(os.path.join(__file__,
os.pardir))))
```

Мы переместили файл настроек на один уровень внутрь папки, поэтому путь `BASE_DIR` должен указывать не на текущий каталог, а на родительский. Чтобы получить внешнюю по отношению к файлу папку, используем функцию `os.pardir`.

Отредактируйте файл `settings/local.py`, вставив в него такое содержимое:

```
from .base import *
DEBUG = True
DATABASES = {
 'default': {
 'ENGINE': 'django.db.backends.sqlite3',
 'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
 }
}
```

Мы изменили файл локальных настроек. Как вы можете заметить, здесь импортируется все содержимое базового файла `base.py`. Мы не дублируем его, а только добавляем дополнительные настройки. Мы задали атрибуты `DEBUG` и `DATABASES`, которые будут отличаться для локального и боевого окружения. Теперь вы можете удалить эти две настройки из файла `base.py`.

Отредактируйте файл `settings/pro.py`, чтобы он выглядел таким образом:

```
from .base import *
DEBUG = False
ADMINS = (
 ('Antonio M', 'email@mydomain.com'),
)
ALLOWED_HOSTS = ['*']
DATABASES = {
 'default': {}
}
```

Эти настройки будут использоваться в боевом окружении. Давайте рассмотрим каждую из них подробнее:

- DEBUG – должна быть всегда равна False для боевого окружения. Если это не так, то при возникновении ошибок на сайте пользователям будет показан отладочный экран с приватной информацией о проекте и его настройках;
- ADMINS – когда режим отладки выключен (DEBUG=False), при возникновении ошибок в работе сайта на перечисленные в этом списке электронные адреса будет отправлено соответствующее сообщение. Не забывайте указывать здесь актуальную информацию;
- ALLOWED\_HOSTS – список доменов, на которых может работать текущий сайт. Эта настройка обеспечивает дополнительную безопасность. В боевом режиме Django будет выбрасывать исключение, если запрос придет с другого домена. В текущей реализации мы разрешаем все домены, так как указали \* (символ звездочки), но позже исправим это;
- DATABASES – настройка баз данных, пока что пустая. Мы заполним ее чуть позже.



При работе с несколькими окружениями создайте базовый файл настроек и отдельные для окружений, каждый из которых импортирует основные настройки, возможно, перепределяет их, и добавляет свои.

Мы разместили настройки в нескольких файлах, поэтому сейчас вы не сможете выполнить какую-либо команду управления через `manage.py`. Чтобы исправить это, необходимо явно указать, какой файл настроек использовать, для этого можно применить флаг `--settings` или задать переменную окружения `DJANGO_SETTINGS_MODULE`.

Откройте консоль и выполните такую команду:

```
export DJANGO_SETTINGS_MODULE=educa.settings.pro
```

Так вы установите в переменной окружения `DJANGO_SETTINGS_MODULE` файл, который будет использоваться вместо `settings.py`. Эту команду придется выполнять в каждом новом сеансе консоли, или можно настроить автоматическое выставление переменной при запуске терминала. Для этого достаточно добавить приведенную выше строку в файлы `.bashrc` и `.bash_profile`. Если вы все же решили воспользоваться флагом `--settings`, команды будут выглядеть таким образом:

```
python manage.py migrate --settings=educa.settings.pro
```

Мы создали структуру настроек проекта, которая позволит запускать его в различных окружениях.

## Настройка PostgreSQL

На протяжении всей книги мы преимущественно использовали СУБД SQLite. Это хорошая система для быстрого старта и прототипирования приложений, но для запуска в боевом режиме необходима более мощная СУБД, например

PostgreSQL, MySQL или Oracle. Из главы 3 вы уже знаете, как настроить проект на работу с PostgreSQL, сейчас мы настроим проект аналогичным образом.

Давайте создадим пользователя PostgreSQL. Откройте консоль и выполните такие команды:

```
su postgres
createuser -dP educa
```

Вам необходимо будет ввести пароль и выбрать права, которые нужно назначить пользователю. Введите эти данные и затем создайте новую базу данных такой командой:

```
createdb -E utf8 -U educa educa
```

Отредактируйте файл `settings/pro.py`, изменив раздел `DATABASES`, чтобы он выглядел аналогично:

```
DATABASES = {
 'default': {
 'ENGINE': 'django.db.backends.postgresql',
 'NAME': 'educa',
 'USER': 'educa',
 'PASSWORD': '*****',
 }
}
```

Замените название базы данных и информацию о пользователе на актуальные. Сейчас созданная база пуста, давайте добавим в нее необходимые таблицы. Выполните команду синхронизации миграций:

```
python manage.py migrate
```

Теперь можно создать суперпользователя нашего сайта:

```
python manage.py createsuperuser
```

## Проверка проекта

Django предоставляет специальную команду управления `check` для добавления проверок проекта. Когда вы запускаете ее, Django проверяет все установленные приложения и, если находит проверки, запускает их. Если вы добавите флаг `-deploy`, дополнительно будут запущены те, которые предназначены для боевого режима. Откройте консоль и выполните команду:

```
python manage.py check --deploy
```

Вы увидите вывод без ошибок. Это значит, проверки прошли успешно, но Django может вывести предупреждения, всегда просматривайте их перед запуском проекта в боевом режиме, так как некоторые могут повлиять на безопасность приложения. Мы не будем углубляться в эту тему, но имейте в виду, что Django имеет встроенную возможность для проверки кода до его запуска.

## Запуск Django в режиме WSGI-приложения

Для запуска Django-приложений используют WSGI. WSGI (Web Server Gateway Interface) – это интерфейс взаимодействия Python-приложения и веб-сервера, и он является своего рода стандартом при запуске Django-приложений.

Когда вы создаете новый проект с помощью команды `startproject`, Django добавляет файл `wsgi.py`. Он содержит вызываемый объект, который используется WSGI в качестве точки входа в приложение. WSGI применим как для запуска проекта при разработке, так и в боевом режиме.

Более подробно о WSGI можно прочесть на странице <https://wsgi.readthedocs.io/en/latest/>.

## Установка uWSGI

На протяжении всей книги для запуска приложения мы использовали сервер для разработки Django. Но теперь нам понадобится реальный веб-сервер для запуска в боевом режиме.

Программа *uWSGI* – это невероятно быстрый веб-сервер для Python. Он взаимодействует с Python-приложениями посредством WSGI. Именно uWSGI занимается преобразованием запросов в формат, с которым может работать Django.

Установите uWSGI такой командой:

```
pip install uwsgi==2.0.17
```

Чтобы сгенерировать исполняемый файл uWSGI, вам понадобится компилятор языка C, например `gcc` или `clang`. Если вы работаете с Linux, для установки просто выполните команду `apt-get install build-essential`.

Если вы используете macOS X, можете установить uWSGI через пакетный менеджер Homebrew, выполнив `brew install uwsgi`. Если ваша операционная система – Windows, установите приложение Cygwin с сайта <https://www.cygwin.com/>. Стоит отметить, что предпочтительной операционной системой для uWSGI являются UNIX-системы.

Документация по веб-серверу uWSGI приведена на сайте <https://uwsgi-docs.readthedocs.io/en/latest/>.

## Конфигурация uWSGI

Сервер uWSGI можно запускать из консоли. Откройте ее и выполните такую команду из каталога проекта `educa`:

```
sudo uwsgi --module=educa.wsgi:application \
--env=DJANGO_SETTINGS_MODULE=educa.settings.pro \
--master --pidfile=/tmp/project-master.pid \
--http=127.0.0.1:8000 \
--uid=1000 \
--virtualenv=/home/env/educa/
```

Если у вас нет прав администратора операционной системы, необходимо добавить перед командой `sudo`, как в примере выше.

Давайте рассмотрим, какие настройки мы задали для uWSGI, выполнив эту команду:

- указали исполняемый файл проекта `educa.wsgi:application`;
- задали файл настроек для боевого режима;
- указали, какое виртуальное окружение использовать. Замените путь в опции `virtualenv` на путь до папки вашего виртуального окружения.

Если вы не используете его, можно не указывать эту настройку.

Если вы выполняете команду `uwsgi` не из папки проекта, обязательно задайте опцию `--chdir=/path/to/educa/` с указанием пути до проекта.

Откройте в браузере страницу `http://127.0.0.1:8000/`. Вы увидите HTML без CSS-стилей и изображений. И это нормально, ведь мы не настроили веб-сервер на работу со статическими и медиафайлами, загруженными пользователями, давайте сделаем это.

uWSGI позволяет прописать конфигурацию сервера в файле с расширением `.ini`. Это более удобный способ, чем перечисление всех настроек в команде.

Создайте в каталоге проекта `educa/` такую структуру:

```
config/
 uwsgi.ini
```

Отредактируйте файл `uwsgi.ini` и добавьте в него такие строки:

```
[uwsgi]
Переменные.
projectname = educa
base = /home/projects/educa
Настройки веб-сервера.
master = true
virtualenv = /home/env/%(projectname)
pythonpath = %(base)
chdir = %(base)
env = DJANGO_SETTINGS_MODULE=%(projectname).settings.pro
module = educa.wsgi:application
socket = /tmp/%(projectname).sock
```

Здесь мы определили две переменные:

- `projectname` – имя Django-проекта, `educa`;
- `base` – абсолютный путь до проекта `educa`. Замените эту настройку на актуальный для вас путь.

Это переменные, которые мы используем в конфигурации веб-сервера. Вы можете задать и другие, главное, чтобы имена переменных не пересекались с названиями настроек uWSGI.

Давайте рассмотрим, что мы задали в конфигурации:

- `master` – активировали главный процесс;
- `virtualenv` – путь к папке виртуального окружения. Не забудьте заменить на ваш путь;

- `pythonpath` – путь до исполняемого файла Python;
- `chdir` – путь к папке проекта, uWSGI будет работать с этим каталогом при запуске проекта;
- `env` – переменные окружения. Мы добавили `DJANGO_SETTINGS_MODULE`, указав файл настроек Django-проекта;
- `module` – модуль WSGI, который будет использоваться. Мы указали вызываемый объект `application`, описанный в файле `wsgi.py` проекта;
- `socket` – сокет UNIX/TCP для подключения к серверу.

Настройка `socket` задается для того, чтобы uWSGI мог взаимодействовать с внешним маршрутизатором, например NGINX, а настройка `http` предназначена, чтобы назначить обработчиком HTTP-запросов непосредственно uWSGI. Мы будем запускать uWSGI с сокетом, так как в следующих разделах узнаем, как работать с NGINX, и настроим его для нашего проекта.

Подробная документация и uWSGI приведена на странице <https://uwsgi-docs.readthedocs.io/>.

Теперь вы можете запускать uWSGI короткой командой, а настройки сервера будут получены из файла конфигурации:

```
uwsgi --ini config/uwsgi.ini
```

На текущем этапе вы не сможете проверить работу сайта в браузере, так как мы используем сокет. Давайте закончим настройку боевого окружения.

## Установка NGINX

Когда вы запускаете сайт, вам нужно обеспечить работу браузера не только с динамическим содержимым, страницами, но и со статическими файлами, которые, например, содержат CSS, JavaScript и картинки. Можно использовать средства uWSGI, но это накладывает дополнительную нагрузку на приложение, поэтому на практике чаще применяют внешний веб-сервер, который отвечает за работу со статическими файлами.

NGINX – это веб-сервер, который обеспечивает высокую производительность и параллелизм, при этом использует мало памяти. Также часто NGINX используют в качестве прокси-сервера, который принимает на себя все входящие запросы и распределяет их между множеством веб-приложений. В общем случае NGINX применяют для того, чтобы обрабатывать запросы на получение статических файлов, а остальные передавать на рабочие процессы uWSGI. Еще одним преимуществом NGINX можно считать то, что он очень гибок в настройке и мы можем задавать различные правила для обработки запросов.

Установите NGINX с помощью команды

```
sudo apt-get install nginx
```

Если вы используете macOS X, то можете установить сервер, выполнив команду `brew install nginx`. Дистрибутив для пользователей Windows находится на странице <https://nginx.org/en/download.html>.



## Боевое окружение

Представленная ниже диаграмма демонстрирует взаимодействие всех систем, которые мы рассмотрели ранее в этой главе:

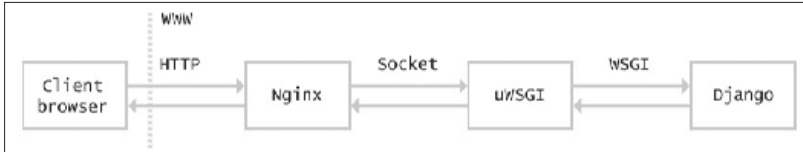


Рис. 13.1 ❖ Взаимодействие систем веб-приложения в боевом режиме

В такой цепочке запрос от клиента обрабатывается в несколько шагов.

1. NGINX принимает HTTP-запрос.
2. Если это запрос на получение статических файлов, его обрабатывает сам NGINX. Если это запрос на что-то другое, NGINX делегирует его обработку веб-серверу uWSGI через сокет.
3. uWSGI принимает входящий запрос и передает его в Django-приложение. Результирующий HTTP-ответ передается по цепочке в обратном порядке, и NGINX отправляет его клиенту.

## Конфигурация NGINX

Создайте в папке `config/` новый файл, `nginx.conf`, и добавьте в него такой код:

```
Сокет, через который будет взаимодействовать NGINX
upstream educa {
 server unix:///tmp/educa.sock;
}

server {
 listen 80;
 server_name www.educaproject.com educaproject.com;

 location / {
 include /etc/nginx/uwsgi_params;
 uwsgi_pass educa;
 }
}
```

Это базовая конфигурация NGINX. Мы указали канал `educa`, через который NGINX будет взаимодействовать с веб-сервером uWSGI посредством сокета. Также в файле задана настройка `server` с таким описанием:

- NGINX будет прослушивать порт 80;
- запросы будут обрабатываться при условии, что пришли с домена `www.educaproject.com` или `educaproject.com`;
- все запросы относительно пути `/` будут переданы в сокет канала `educa` (на uWSGI). Мы также задали стандартные настройки NGINX для работы с uWSGI.

Документация по NGINX приведена на странице <https://nginx.org/en/docs/>.

Основная конфигурация NGINX описана в файле `/etc/nginx/nginx.conf`. Кроме этого, подключаются настройки из файлов, расположенных в папке `/etc/nginx/sites-enabled/`. Чтобы создать свой конфигурационный файл и применить его, создайте ссылку таким образом:

```
sudo ln -s /home/projects/educa/config/nginx.conf /etc/nginx/sitesenabled/educa.conf
```

Замените в этой строке `/home/projects/educa/` на абсолютный путь до проекта в вашей файловой системе. Затем откройте консоль и запустите `uwsgi`, если не сделали этого раньше:

```
uwsgi --ini config/uwsgi.ini
```

Откройте вторую консоль и запустите NGINX:

```
service nginx start
```

Так как мы указали доменные имена в настройках NGINX, нужно добавить перенаправление с них на локальный хост. Отредактируйте файл `/etc/hosts`, добавив такие строки:

```
127.0.0.1 educaproject.com
127.0.0.1 www.educaproject.com
```

Таким образом запросы на оба адреса будут перенаправляться на IP `127.0.0.1`. Когда вы развернете проект на удаленном сервере, этот шаг выполнять не нужно, так как у вас будет задан IP-адрес, но нужно будет настроить имя хоста на DNS-сервере.

Откройте в браузере <http://educaproject.com/>. Вы увидите, что на сайте до сих пор не появились статические файлы. Осталось совсем чуть-чуть, и наш сайт будет полностью готов.

Нам нужно ограничить имена доменов, которые Django будет распознавать как корректные. Отредактируйте файл `settings/pro.py` проекта и добавьте в настройку `ALLOWED_HOSTS` такие значения:

```
ALLOWED_HOSTS = ['educaproject.com', 'www.educaproject.com']
```

Теперь Django не будет выбрасывать исключение при получении запроса. Более подробно о настройке `ALLOWED_HOSTS` можно прочесть на странице <https://docs.djangoproject.com/en/2.0/ref/settings/#allowed-hosts>.

## Настройка отдачи статических и медиафайлов

NGINX прекрасно подходит для обработки запросов на получение статических файлов. Для наилучшей производительности мы воспользуемся его возможностями. Задействуем NGINX как для отдачи статических файлов, так и для файлов, загруженных пользователями при создании содержимого курсов.

Отредактируйте файл `settings/base.py`, добавив такую строку:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

Теперь нам необходимо собрать все статические файлы проекта. Команда `collectstatic` копирует файлы из каждого приложения в папку, указанную в настройке `STATIC_ROOT`. Откройте консоль и выполните такую команду:

```
python manage.py collectstatic
```

Вы увидите сообщение:

```
160 static files copied to '/educa/static'.
```

Теперь откройте файл `config/nginx.conf` и добавьте в блок `server` такие строки:

```
location /static/ {
 alias /home/projects/educa/static;
}
location /media/ {
 alias /home/projects/educa/media;
}
```

Не забудьте заменить `/home/projects/educa/` на путь до вашего проекта. Эти настройки говорят о том, что запросы с адресами, начинающимися на `/static/` и `/media/`, должен обрабатывать NGINX. Давайте рассмотрим, для чего предназначена каждая из этих настроек:

- `/static/` – адрес, соответствующий настройке `STATIC_URL` Django-проекта, указывает путь в файловой системе, где искать CSS, JavaScript и картинки. Путь должен совпадать с путем из настройки `STATIC_ROOT`;
- `/media/` – адрес, соответствующий настройке `MEDIA_URL`, указывает путь в файловой системе, где искать медиафайлы. Путь должен совпадать с настройкой `MEDIA_ROOT` Django-проекта.

После каждого изменения конфигурации NGINX необходимо перезапускать его, чтобы правки применились. Выполните такую команду:

```
service nginx reload
```

Откройте в браузере страницу <http://educaproject.com/>, на сайте должны появиться стили и картинки. Теперь именно NGINX занимается обработкой запросов на получение статических файлов, вместо того чтобы нагружать этим Python-приложение.

Поздравляем! Вы успешно настроили NGINX на работу со статическими файлами.

## Защита подключений с помощью SSL

*Протокол SSL* (уровень защищенных сокетов, Secure Sockets Layer) – это еще один способ защитить ваш сайт с помощью SSL-сертификатов. Этот механизм становится общепринятым в веб-разработке. Рекомендуется настраивать сайты так, чтобы они могли работать по HTTPS. Мы добавим такую возможность в наш проект.

## Создание SSL-сертификата

Создайте новую папку `ssl` в каталоге проекта `educa`. Затем сгенерируйте SSL-сертификат, выполнив в консоли команду

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ssl/educa.key -out ssl/educa.crt
```

Мы формируем закрытый ключ и 2048-битный SSL-сертификат, который будет действовать в течение года. Вам необходимо будет заполнить данные:

```
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []: educaproject.com
Email Address []: email@domain.com
```

Задайте ваши данные, наиболее важное поле – **Common Name**. Необходимо указать имя домена, для которого формируется сертификат, вставьте `educaproject.com`.

По окончании ввода вы увидите, что в папке `ssl/` созданы два файла – закрытый ключ, `educa.key`, и файл сертификата, `educa.crt`.

## Настройка NGINX на использование SSL

Откройте файл `nginx.conf` и измените блок `server`, добавьте SSL, как показано ниже:

```
server {
 listen 80;
 listen 443 ssl;
 ssl_certificate /home/projects/educa/ssl/educa.crt;
 ssl_certificate_key /home/projects/educa/ssl/educa.key;
 server_name www.educaproject.com educaproject.com;
 # ...
}
```

Теперь мы настроили NGINX на прослушивание HTTP-запросов на 80 порту, а запросов по протоколу HTTPS – на 443. Мы указали файлы SSL-сертификата и закрытого ключа с помощью параметров `ssl_certificate` и `ssl_certificate_key` соответственно.

Перезапустите веб-сервер NGINX:

```
sudo service nginx restart
```

NGINX применил изменения в конфигурации, и вы можете открыть в браузере страницу <https://educaproject.com/>. Она должна будет выглядеть примерно так:

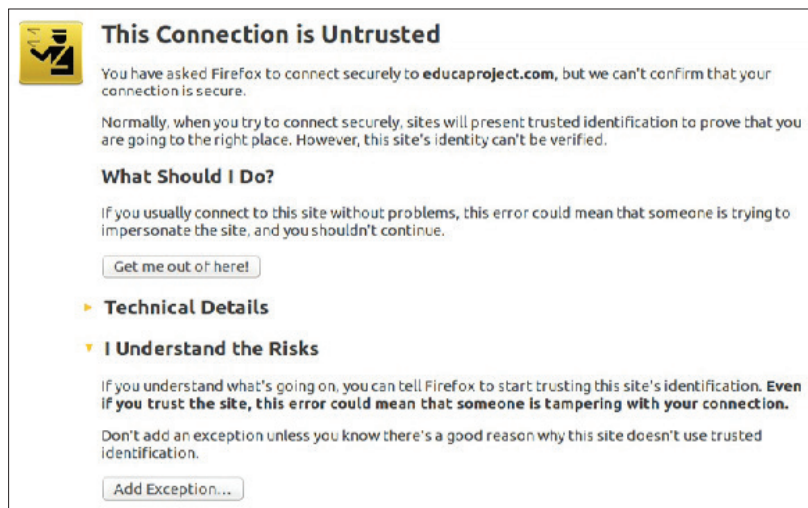


Рис. 13.2 ❖ Предупреждение браузера об использовании SSL

Сообщение и его вид могут отличаться в зависимости от браузера. Оно говорит о том, что сайт использует непроверенный сертификат – браузер не может убедиться в его достоверности. Вы видите это сообщение, потому что мы сгенерировали SSL-сертификат самостоятельно, а не запросили его в центре сертификации. Когда вы будете запускать реальный сайт в боевом окружении, сделайте это, тогда пользователи не увидят подобных сообщений.

Если вы хотите добавить сертификат, можете перейти к проекту *Let's Encrypt* от компании Linux Foundation. Он значительно упрощает создание и обновление SSL-сертификатов. Более подробную информацию об этом можно найти на странице <https://letsencrypt.org/>.

Нажмите на кнопку **Add Exception** (Добавить исключение), чтобы браузер больше не показывал это сообщение для нашего сайта. Вы увидите, что в адресной строке рядом с URL'ом появился значок замочка:

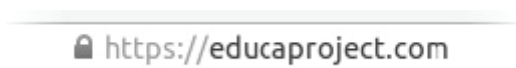


Рис. 13.3 ❖ Отметка браузера о том, что сайт использует непроверенный сертификат

При клике на эту иконку вы увидите подробности об SSL на текущем сайте.

### **Настройка проекта на использование SSL**

Чтобы Django смог работать с SSL, необходимо задать дополнительные настройки. Откройте файл `settings/pro.py` и вставьте такие строки:

```
SECURE_SSL_REDIRECT = True
CSRF_COOKIE_SECURE = True
```

Они отвечают за следующую функциональность:

- SECURE\_SSL\_REDIRECT – любой HTTP-запрос будет перенаправлен на HTTPS;
- CSRF\_COOKIE\_SECURE – при работе с куками и CSRF-токенами будет учитываться SSL.

Поздравляем! Вы настроили проект на работу в боевом режиме.

## СОЗДАНИЕ СОБСТВЕННОГО ПРОМЕЖУТОЧНОГО СЛОЯ

Мы уже говорили о настройке MIDDLEWARE, которая содержит промежуточные слои проекта. Вы можете рассматривать это как низкоуровневую систему плагинов, которая дает возможность добавить какие-либо действия в цикле обработки запроса и формирования ответа. Каждый слой отвечает за единственное атомарное действие, которое выполняется для всех HTTP-запросов и ответов.

**i** Избегайте сложных вычислений в коде промежуточных слоев, так как он выполняется для каждого запроса.

Когда Django получает HTTP-запрос, он выполняет промежуточные слои в порядке, как они указаны в списке MIDDLEWARE. Когда Django формирует HTTP-ответ, слои вызываются в обратном порядке.

Промежуточный слой может быть реализован в виде функции, например так:

```
def my_middleware(get_response):
 def middleware(request):
 # Код, который будет выполняться при каждом запросе
 # до его передачи в обработчик (или следующий слой).

 response = get_response(request)

 # Код выполняется после вызова обработчика.

 return response

 return middleware
```

Функция `my_middleware` принимает в качестве аргумента другую функцию `get_response` и возвращает промежуточный слой. Промежуточный слой – это вызываемый объект, который принимает запрос и возвращает ответ, аналогично тому, как это делают обработчики. В качестве аргумента `get_response` может выступать следующий промежуточный слой или непосредственный обработчик запроса.

Если в каком-то из слоев не вызывается `get_response`, процесс обработки запроса прерывается и следующие в цепочке слои или обработчик не будут задействованы, а ответ будет сформирован на этом же уровне.

Порядок, в котором слои описаны в MIDDLEWARE, важен, так как некоторые из них добавляют в объект запроса переменные, необходимые для работы зависящих слоев.

**i** При добавлении нового элемента в настройку MIDDLEWARE убедитесь, что разместили его на нужной позиции. Промежуточные слои выполняются в том порядке, как они описаны в списке MIDDLEWARE, для запроса и в обратном – для ответа.

Более подробную информацию о промежуточных слоях можно найти на странице <https://docs.djangoproject.com/en/2.0/topics/http/middleware/>.

## Создание промежуточного слоя для доступа через поддомен

Мы создадим собственный промежуточный слой, который позволит разместить страницы курсов в поддоменах. URL'ы, которые сейчас выглядят как <https://educaproject.com/course/django/>, будут заменены на формат с поддоменом, сформированным из слага курса, – <https://django.educaproject.com/>. Пользователи с помощью такой записи смогут быстро переходить к содержимому курса. Каждый запрос через поддомен будет передаваться в обработчик, соответствующий начальной версии URL'а.

Код промежуточных слоев можно размещать в любом месте проекта. Но общепринятый подход – создать для них отдельный файл `middleware.py` в папке конкретного приложения.

Создайте в каталоге приложения `courses` новый файл `middleware.py`, и добавьте в него такой фрагмент:

```
from django.urls import reverse
from django.shortcuts import get_object_or_404, redirect
from .models import Course

def subdomain_course_middleware(get_response):
 """
 Обрабатывает запросы к курсам через поддомены.
 """
 def middleware(request):
 host_parts = request.get_host().split('.')
 if len(host_parts) > 2 and host_parts[0] != 'www':
 # Получение объекта курса по данным из URL.
 course = get_object_or_404(Course, slug=host_parts[0])
 course_url = reverse('course_detail', args=[course.slug])
 # Переенаправление на страницу курса.
 url = '{://}{}'.format(request.scheme,
 '.'.join(host_parts[1:]),
 course_url)

 return redirect(url)

 response = get_response(request)
 return response

 return middleware
```

При обработке запроса мы выполняем такие действия.

1. Получаем имя домена по частям. Например, если URL содержит домен вида `mycourse.educaproject.com`, мы получим список `['mycourse', 'educaproject', 'com']`.
2. Проверяем, что поддомен существует, то есть в списке содержится более двух элементов, и первый из них не равен `www`. Извлекаем слаг курса и пытаемся получить соответствующий объект `Course`.
3. Если курс не найден, выбрасываем исключение `404`, иначе – перенаправляем пользователя на страницу подробностей.

Отредактируйте файл `settings/base.py` и подключите слой, который мы только что создали, добавив строку `'courses.middleware.subdomain_course_middleware'` в конец списка `MIDDLEWARE`:

```
MIDDLEWARE = [
 # ...
 'courses.middleware.subdomain_course_middleware',
]
```

Теперь код промежуточного слоя будет выполняться для каждого запроса.

Помните, что имена доменов, с которым Django будет работать, задаются в настройке `ALLOWED_HOSTS`? Давайте изменим ее так, чтобы поддомены рассматривались как корректный URL.

Откройте файл `settings/pro.py` и замените настройку `ALLOWED_HOSTS` следующим образом:

```
ALLOWED_HOSTS = ['.educaproject.com']
```

Запись, начинающаяся с точки, `'.educaproject.com'`, подразумевает наличие поддомена, поэтому теперь Django может обрабатывать запросы вида `educaproject.com`, `course.educaproject.com`, `django.educaproject.com` и тому подобные.

## Настройка NGINX на работу с несколькими поддоменами

Нам необходимо изменить конфигурацию NGINX, чтобы он тоже понимал поддомены и делегировал обработку таких запросов. Откройте файл `config/nginx.conf`, найдите в нем строку:

```
server_name www.educaproject.com educaproject.com;
```

Замените ее на такую:

```
server_name *.educaproject.com educaproject.com;
```

Мы добавили звездочку, поэтому теперь NGINX обрабатывает все запросы с поддоменами для домена `educaproject.com`. Чтобы протестировать эту возможность локально, необходимо добавить маршрут для адреса `django.educaproject.com`. Для проверки работы курса со слагом `django` добавьте в файл `/etc/hosts` строку

```
127.0.0.1 django.educaproject.com
```



Теперь перейдите в браузере на страницу <https://django.educaproject.com/>. Промежуточный слой найдет нужный курс по поддомену и перенаправит вас на страницу <https://educaproject.com/course/django/>.

## ДОБАВЛЕНИЕ СОБСТВЕННЫХ КОМАНД УПРАВЛЕНИЯ

В Django реализована возможность создать свои команды управления для утилиты `manage.py`. Например, в главе 9 мы использовали `makemessages` и `compilemessages`, чтобы создавать файлы переводов.

*Команда управления* – это Python-модуль, который содержит класс `Command`, являющийся наследником базового класса `django.core.management.base.BaseCommand`. Вы можете создать простую команду или с передачей параметров.

Django ищет файлы с классами команд в папке `management/commands/` каждого приложения, подключенного в списке `INSTALLED_APPS`. Каждый найденный модуль регистрируется как команда с именем модуля.

Более подробно о реализации собственных команд управления можно прочесть на странице <https://docs.djangoproject.com/en/2.0/howto/custom-management-commands/>.

Мы создадим команду, которая будет рассылать студентам напоминания о том, чтобы они записались хотя бы на один курс. Письмо будет отправляться пользователям, которые в течение некоторого времени не принимали участия ни в одном курсе.

Создайте такую структуру папок и файлов в каталоге приложения `students`:

```
management/
 __init__.py
 commands/
 __init__.py
 enroll_reminder.py
```

Отредактируйте файл `enroll_reminder.py`, добавив в него такой код:

```
import datetime
from django.conf import settings
from django.core.management.base import BaseCommand
from django.core.mail import send_mass_mail
from django.contrib.auth.models import User
from django.db.models import Count

class Command(BaseCommand):
 help = 'Sends an e-mail reminder to users registered more \
 than N days that are not enrolled into any courses yet'

 def add_arguments(self, parser):
 parser.add_argument('--days', dest='days', type=int)

 def handle(self, *args, **options):
 emails = []
 subject = 'Enroll in a course'
 date_joined = datetime.date.today() - \

```

```

datetime.timedelta(days=options['days'])
users = User.objects.annotate(course_count=Count('courses_joined'))\
 .filter(course_count=0, date_joined__lte=date_joined)
for user in users:
 message = 'Dear {},\n\n We noticed that you didn't\
 enroll in any courses yet. What are you waiting\
 for?'.format(user.first_name)
 emails.append((subject,
 message,
 settings.DEFAULT_FROM_EMAIL,
 [user.email]))
send_mass_mail(emails)
self.stdout.write('Sent {} reminders'.format(len(emails)))

```

Это и есть наша команда, которая будет называться `enroll_reminder`. Давайте посмотрим, что происходит в этом фрагменте:

- мы создали класс `Command`, унаследованный от `BaseCommand`;
- добавили в него атрибут `help`. Он используется для отображения подсказки, когда вы запрашиваете ее в консоли с помощью записи вида `python manage.py help enroll_reminder`;
- используем метод `add_arguments()`, чтобы в команду можно было передать аргумент `-days`. Этот параметр задает количество дней, по прошествии которых пользователям необходимо отправить напоминание;
- определили метод `handle()`, который занимается отправкой сообщения. Мы получаем количество дней `days` из аргументов консоли. Фильтруем пользователей и получаем тех, кто не записан более чем `days` дней хотя бы на один курс. Количество курсов добавляем к объектам `QuerySet`'а с помощью метода `annotate()`. Формируем сообщение для каждого пользователя. Наконец, вызываем функцию `send_mass_mail()`, чтобы за одно SMTP-подключение к почтовому серверу отправить сообщения всем пользователям.

Вы только что создали свою первую команду управления. Давайте проверим, как она работает. Откройте консоль и выполните ее:

```
python manage.py enroll_reminder --days=20
```

Если у вас не настроен SMTP-сервер, обратитесь к главе 2, где описан процесс подключения проекта к почтовому серверу. Или вы можете временно указать в настройках `settings/base.py` бэкэнд для вывода в консоль:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Давайте сделаем запуск команды периодичным, чтобы рассылка происходила каждый день в 8 часов. Если вы используете Linux или macOS X, откройте консоль и выполните команду `crontab -e`, чтобы отредактировать `crontab`. Добавьте в файл такую строку:

```
0 8 * * * python /path/to/educa/manage.py enroll_reminder --days=20 -- settings=educa.settings.pro
```

Если вы не знаете, как работать с приложением `cron`, обратитесь к его документации на странице <http://www.unixgeeks.org/security/newbie/unix/cron-1.html>.

Если вы работаете с операционной системой Windows, можете задать периодичность с помощью планировщика задач. Более подробную информацию об этом можно найти на странице [https://msdn.microsoft.com/en-us/library/windows/desktop/aa383614\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa383614(v=vs.85).aspx).

Другой возможный вариант создания периодических задач – использование Celery. Помните, в главе 7 мы использовали это приложение для запуска асинхронных задач? Вместо создания собственной команды управления и ее планирования с помощью `cron` можно создать асинхронную задачу и выполнять ее, используя Celery и Celery beat. Более подробно об этом можно прочесть на странице <https://celery.readthedocs.io/en/latest/userguide/periodic-tasks.html>.



Создавайте команды управления для выполнения обособленных блоков кода, которые необходимо запускать с некоторой периодичностью. Для этого используйте `cron` или планировщик задач Windows.

В Django реализована возможность вызывать команды управления из Python-кода. Вы можете сделать это таким образом:

```
from django.core import management
management.call_command('enroll_reminder', days=20)
```

Поздравляем! Теперь вы знаете, как создать команду управления и запланировать ее периодичное выполнение.

## РЕЗЮМЕ

В этой главе мы изменили проект так, чтобы его можно было запускать в боевом окружении, настроили работу с веб-серверами uWSGI и NGINX, вы создали собственную команду управления.

Вы подошли к концу. Поздравляем! Вы получили знания, необходимые для реализации веб-приложений с помощью Django, и закрепили их на практике и реальных примерах. Эта книга провела вас через весь процесс создания полноценных проектов и интеграции их со сторонними сервисами. Теперь вы готовы к реализации собственного проекта, будь то небольшой прототип или сложная масштабируемая система.

Удачи в ваших следующих приключениях с Django!

# Предметный указатель

- AJAX, 154
- Celery, 227
- Comma-separated Values, 244
- DOM, 156
- Markdown, 76
- NGINX, 395
- ORM, 35
- QuerySet, 36
- Redis, 188
- REST, 373
- uWSGI, 393
- WSGI, 393
- Абстрактная модель, 309
- Букмарклет, 143
- Бэкэнд аутентификации, 120
- Валидация форм, 49
- Вычисление QuerySet'a, 37
- Декоратор, 160
- Денормализация, 184
- Иерархия документа, 156
- Интернационализация, 267
- Интерфейс, 26
- Команда управления, 404
- Контекстный процессор, 217
- Конфигурация проекта, 24
- Локализация, 267
- Модель, 26
- Набор, 192
  - форм, 331
- Наследование
  - моделей, 309
  - с несколькими таблицами, 310
- Обобщенные связи, 177
- Обработчик Django, 38
- Отношение
  - «многие ко многим», 137
  - один к одному, 113
  - один ко многим, 27
- Передача состояния управления, 373
- Подделка межсайтовых запросов, 54
- Подсистема
  - аутентификации, 92
  - карты сайта, 78
  - кеширования, 363
  - миграций, 28
  - сессий, 206
  - сообщений, 118
  - типов содержимого, 176
  - форм, 48
- Приложение, 25
- Примесь, 321
- Проект, 25
- Промежуточная модель, 166
- Промежуточный слой, 93
- Протокол SSL, 398
- Процессор контекста, 39
- Ранжирование результатов поиска, 87
- Семантический URL, 27
- Сигнал модели, 185
- Система объектно-реляционного отображения, 35
- Система управления содержимым, 302
- Слаг, 27
- Стемминг, 87
- Триграмма, 89
- Фид, 81
- фикстура, 306
- Функция агрегации, 68
- Шаблон
  - HTML, 41
  - URL, 40
  - адресов, 22
- Шаблонный фильтр, 42

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу:

115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: [www.a-planet.ru](http://www.a-planet.ru).

Оптовые закупки: тел. (499) 782-38-89.

Электронный адрес: [books@aliants-kniga.ru](mailto:books@aliants-kniga.ru).

Антонио Меле

## **Django 2 в примерах**

Главный редактор *Мовчан Д. А.*  
[dmkpress@gmail.com](mailto:dmkpress@gmail.com)

Перевод *Плотникова Д. В.*

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура «PT Serif». Печать офсетная.

Усл. печ. л. 33,15. Тираж 200 экз.

Веб-сайт издательства: [www.dmkpress.com](http://www.dmkpress.com)