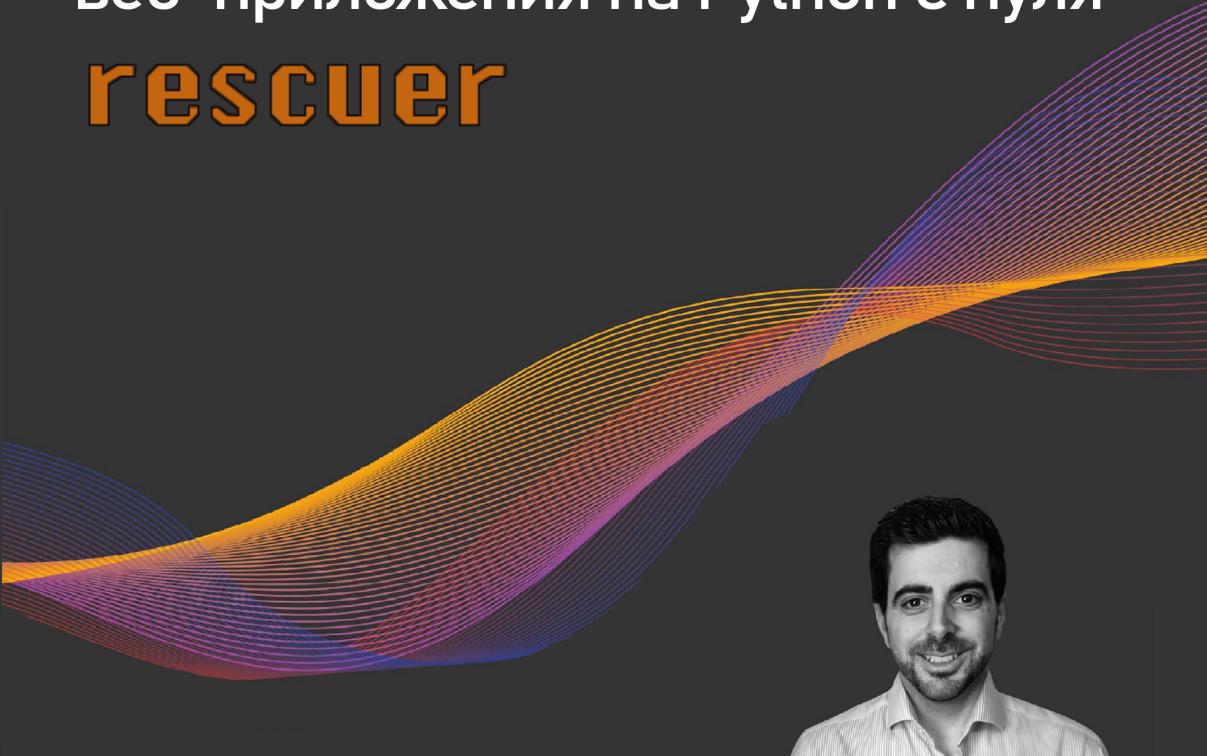


Django 4 в примерах

Разрабатывайте
мощные и надежные
веб-приложения на Python с нуля

rescuer



Антонио Меле

AMK
издательство

Книга охватывает многообразные аспекты создания веб-приложений с помощью самого популярного веб-фреймворка Django на языке Python. Изучив четыре проекта разной направленности (приложение для ведения блога и электронной коммерции, социальный веб-сайт, платформа электронного обучения), вы получите хорошее представление о том, как работает Django.

Прочитав книгу, вы:

- усвоите основы Django, включая модели, ORM-преобразователь, представления, шаблоны, URL-адреса, формы, аутентификацию, сигналы и промежуточные программные компоненты;
- реализуете аутентификацию с использованием учетных записей Facebook, Twitter и Google, настроите профили пользователей;
- разработаете каталог товаров и корзину покупок для онлайн-магазина;
- научитесь обрабатывать платежи с помощью платежного шлюза Stripe и управлять уведомлениями о платежах с помощью веб-перехватчиков;
- интегрируете в свой проект сторонние приложения Django.

Опираясь на изученный материал, вы сможете создавать полнофункциональные веб-приложения на Python с аутентификацией, системами управления контентом, RESTful API и прочими элементами.

Издание предназначено читателям с базовыми знаниями Python, а также программистам, переходящим на Django с других веб-фреймворков. Оно подойдет и тем, кто уже использует Django в своей работе и хочет расширить свои навыки. Для изучения материала необходимы базовый опыт работы с Python и знание HTML и JavaScript.

Антонио Меле – технический директор Nucor, лондонской финтех-компании, предоставляющей ведущую технологическую платформу для создания решений по управлению цифровыми активами. Разрабатывает проекты Django с 2006 года для клиентов из различных отраслей. Имеет степень магистра компьютерных наук в Университете Pontificia Comillas и владеет английским, испанским, немецким и китайским языками.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@aliens-kniga.ru

Packt

ДМК
издательство
www.dmk.ru

ISBN 978-5-93700-204-4



9 785937 002044 >

Антонио Меле

Django 4 в примерах

Antonio Melé

Django 4 By Example

**Build powerful and reliable Python
web applications from scratch**



BIRMINGHAM – MUMBAI

Антонио Меле

Django 4 в примерах

Разрабатывайте мощные и надежные
веб-приложения на Python с нуля



Москва, 2023

УДК 004.04
ББК 32.372
М47

Меле А.

M47 Django 4 в примерах / пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2023. – 800 с.: ил.

ISBN 978-5-93700-204-4

Книга охватывает многообразные аспекты создания веб-приложений с помощью самого популярного веб-фреймворка Django на языке Python. Изучив четыре проекта разной направленности (приложение для ведения блога и электронной коммерции, социальный веб-сайт, платформа электронного обучения), вы получите хорошее представление о том, как работает Django.

Издание предназначено читателям с базовыми знаниями Python, а также программистам, переходящим на Django с других веб-фреймворков. Оно подойдет и тем, кто уже использует Django в своей работе и хочет расширить свои навыки. Для изучения материала необходимы базовый опыт работы с Python и знание HTML и JavaScript.

УДК 004.04
ББК 32.372

Copyright ©Packt Publishing 2022. First published in the English language under the title Django 4 By Example - Fourth Edition – (9781801813051).

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 978-1-80181-305-1 (англ.)
ISBN 978-5-93700-204-4 (рус.)

© 2022 Packt Publishing
© Перевод, оформление, издание,
ДМК Пресс, 2023

Посвящается моей сестре

Содержание

От издательства	17
Вступительное слово	18
Об авторе	20
О рецензенте	21
Предисловие	22
Глава 1. Разработка приложения для ведения блога	28
Установка языка Python	29
Создание виртуальной среды Python	30
Установка веб-фреймворка Django	31
Установка Django с помощью pip	31
Новые функциональные возможности Django 4.....	32
Общий обзор веб-фреймворка Django.....	33
Главные компоненты веб-фреймворка	33
Архитектура Django	34
Создание первого проекта.....	35
Применение первоначальных миграций базы данных.....	36
Запуск и выполнение сервера разработки.....	37
Настроочные параметры проекта	39
Проекты и приложения.....	40
Создание приложения.....	41
Создание моделей данных блога	42
Создание модели поста.....	42
Добавление полей даты/времени.....	44
Определение предустановленного порядка сортировки	45
Добавление индекса базы данных	46
Активация приложения	47
Добавление поля статуса	47

Добавление взаимосвязи многие-к-одному	50
Создание и применение миграций.....	51
Создание сайта администрирования для моделей.....	54
Создание суперпользователя.....	54
Сайт администрирования	55
Добавление моделей на сайт администрирования	56
Адаптация внешнего вида моделей под конкретно-прикладную задачу	58
Работа с наборами запросов QuerySet и менеджерами	60
Создание объектов.....	61
Обновление объектов.....	62
Извлечение объектов	63
Применение метода filter()	63
Применение метода exclude().....	64
Применение метода order_by()	64
Удаление объектов.....	64
Когда вычисляются наборы запросов QuerySet	65
Создание модельных менеджеров	65
Разработка представлений списка и детальной информации	67
Создание представлений списка постов и детальной информации о посте	67
Применение функции сокращенного доступа get_object_or_404().....	68
Добавление шаблонов URL-адресов представлений.....	69
Создание шаблонов представлений.....	71
Создание базового шаблона.....	72
Создание шаблона списка постов	73
Доступ к приложению	74
Создание шаблона детальной информации о посте	74
Цикл запроса/ответа.....	75
Дополнительные ресурсы	76
Резюме	77
Присоединяйтесь к нам на Discord	78
Глава 2. Усовершенствование блога за счет продвинутых функциональностей	79
Использование канонических URL-адресов для моделей	80
Создание дружественных для поисковой оптимизации URL-адресов постов.....	82
Видоизменение шаблонов URL-адресов	84
Видоизменение представлений	85
Видоизменение канонического URL-адреса постов.....	86
Добавление постраничной разбивки.....	87
Добавление постраничной разбивки в представление списка постов	87
Создание шаблона постраничной разбивки	88
Обработка ошибок постраничной разбивки	91
Разработка представлений на основе классов	94

Зачем использовать представления на основе классов	95
Использование представления на основе класса для отображения списка постов	95
Рекомендация постов по электронной почте.....	97
Разработка форм с помощью Django	98
Работа с формами в представлениях.....	99
Отправка электронных писем с помощью Django.....	101
Отправка электронных писем в представлениях	106
Прорисовка форм в шаблонах	107
Создание системы комментариев	112
Разработка модели комментария	112
Добавление комментариев на сайт администрирования	114
Создание форм из моделей.....	116
Оперирование формами ModelForm в представлениях	116
Создание шаблонов комментарной формы	119
Добавление комментариев в представление детальной информации о посте	121
Добавление комментариев в шаблон детальной информации о посте	122
Дополнительные ресурсы	129
Резюме	130
 Глава 3. Расширение приложения для ведения блога.....	131
Добавление функциональности тегирования	132
Извлечение постов по сходству	141
Создание конкретно-прикладных шаблонных тегов и фильтров.....	146
Реализация конкретно-прикладных шаблонных тегов.....	147
Создание простого шаблонного тега	147
Создание шаблонного тега включения	150
Создание шаблонного тега, возвращающего набор запросов	152
Реализация конкретно-прикладных шаблонных фильтров	154
Создание шаблонного фильтра для поддержки синтаксиса Markdown	154
Добавление карты сайта	159
Создание новостных лент для постов блога	164
Добавление полнотекстового поиска в блог	171
Установка базы данных PostgreSQL	172
Создание базы данных PostgreSQL	173
Выгрузка существующих данных	174
Переключение базы данных в проекте	174
Загрузка данных в новую базу данных.....	176
Простые операции поиска.....	177
Поиск по нескольким полям	177
Разработка представления поиска.....	178
Выделение основ слов и ранжирование результатов.....	182
Выделение основ слов и удаление стоп-слов на разных языках.....	183
Взвешивание запросов	184
Поиск по триграммному сходству.....	185

Дополнительные ресурсы	186
Резюме	187
Глава 4. Разработка социального веб-сайта	188
Создание проекта социального веб-сайта	189
Запуск проекта социального веб-сайта.....	189
Использование поставляемого с Django фреймворка аутентификации.....	191
Создание представления входа в систему	192
Использование встроенных в Django представлений аутентификации.....	199
Представления входа и выхода.....	199
Представления смены пароля.....	205
Представление сброса пароля.....	208
Регистрация пользователей и профили пользователей	216
Регистрация пользователя	216
Расширение модели пользователя.....	223
Установка библиотеки Pillow и раздача медиафайлов	224
Создание миграций для модели профиля	225
Использование конкретно-прикладной модели пользователя	231
Использование фреймворка сообщений	231
Разработка конкретно-прикладного бэкенда аутентификации	235
Предотвращение использования существующего адреса электронной почты	238
Дополнительные ресурсы	239
Резюме	240
Глава 5. Реализация социальной аутентификации	241
Добавление социальной аутентификации на сайт	242
Обеспечение работы сервера разработки по протоколу HTTPS.....	245
Аутентификация с учетной записью Facebook.....	248
Аутентификация с учетной записью Twitter	256
Аутентификация с учетной записью Google.....	268
Создание профиля пользователей, регистрирующихся посредством социальной аутентификации	277
Дополнительные ресурсы	279
Резюме	280
Глава 6. Распространение контента на веб-сайте	281
Создание веб-сайта для управления визуальными закладками	282
Разработка модели изображения	282
Создание взаимосвязей многие-ко-многим	284
Регистрация модели изображения на сайте администрирования	285
Отправка контента с других сайтов	286
Очистка полей формы.....	287
Установка библиотеки requests.....	288

Переопределение метода save() класса ModelForm	288
Разработка букмаклера с помощью JavaScript	293
Создание представления детальной информации об изображениях.....	306
Создание миниатюр изображений с помощью easy-thumbnails	309
Добавление асинхронных действий с помощью JavaScript	312
Загрузка JavaScript в DOM.....	314
Защита от подделки межсайтовых HTTP-запросов на JavaScript.....	315
Выполнение HTTP-запросов с помощью JavaScript	317
Добавление бесконечной постраничной прокрутки в список изображений.....	323
Дополнительные ресурсы	330
Резюме	331
 Глава 7. Отслеживание действий пользователя	332
Разработка системы подписки.....	333
Формирование взаимосвязей многие-ко-многим с промежуточной моделью.....	333
Создание представлений списка и детальной информации для профилей пользователей	336
Добавление действий пользователя по подписке/отписке с помощью JavaScript	342
Разработка типового приложения для потока активности	345
Применение фреймворка contenttypes	346
Добавление обобщенных отношений в модели	347
Игнорирование повторных действий в потоке активности	351
Добавление действий пользователя в поток активности.....	352
Отображение потока активности	355
Оптимизация наборов запросов, предусматривающих связанные объекты	355
Применение метода select_related()	356
Применение метода prefetch_related()	357
Создание шаблонов действий.....	357
Использование сигналов для денормализации количественных данных	361
Работа с сигналами.....	361
Конфигурационные классы приложений	364
Использование меню отладочных инструментов Django.....	366
Установка меню отладочных инструментов Django.....	367
Панели меню отладочных инструментов Django	370
Команды меню отладочных инструментов Django	373
Подсчет просмотров изображений с помощью хранилища Redis.....	374
Установка платформы Docker	375
Установка хранилища Redis	375
Использование хранилища Redis вместе с Python	377
Хранение просмотров изображений в хранилище Redis	379
Хранение рейтинга в хранилище Redis.....	381
Следующие шаги с Redis	384
Дополнительные ресурсы	385
Резюме	385

Глава 8. Разработка интернет-магазина	387
Создание проекта интернет-магазина	388
Создание моделей каталога товаров	389
Регистрация моделей каталога на сайте администрирования.....	393
Формирование представлений каталога	395
Создание шаблонов каталога.....	397
Разработка корзины покупок.....	403
Использование сеансов Django.....	403
Настроочные параметры сеанса.....	404
Срок истечения сеанса	405
Хранение корзин покупок в сеансах	406
Создание представлений корзины покупок.....	410
Добавление товаров в корзину.....	411
Разработка шаблона отображения корзины	413
Добавление товаров в корзину.....	415
Обновление количества товаров в корзине	417
Создание процессора контекста для текущей корзины	418
Процессоры контекста	418
Установка корзины в контекст запроса	419
Регистрация заказов клиентов	421
Создание моделей заказа	422
Включение моделей заказа на сайт администрирования	424
Создание заказов клиентов.....	425
Асинхронные задания	431
Работа с асинхронными заданиями	431
Работники, очереди сообщений и брокеры сообщений	432
Использование Django с Celery и RabbitMQ.....	433
Отслеживание Celery с помощью инструмента Flower	440
Дополнительные ресурсы	443
Резюме	443
Глава 9. Управление платежами и заказами	444
Интеграция платежного шлюза.....	444
Создание учетной записи Stripe	445
Установка библиотеки Stripe.....	448
Добавление Stripe в проект	449
Формирование процесса платежа	450
Интеграция платежного инструмента Stripe Checkout.....	452
Тестирование процесса оформления заказа	459
Использование тестовых кредитных карт.....	461
Проверка платежной информации в информационной панели Stripe.....	463
Применение веб-перехватчиков для получения уведомлений о платежах.....	467
Создание конечной точки веб-перехватчика	467
Тестирование уведомлений веб-перехватчиков	472

Отсылки к платежам Stripe в заказах	475
Выход в прямой эфир.....	479
Экспорт заказов в CSV-файлы.....	480
Добавление конкретно-прикладных действий на сайт администрирования.....	480
Расширение сайта администрирования за счет конкретно-прикладных представлений.....	483
Динамическое генерирование счетов-фактур в формате PDF	488
Установка библиотеки WeasyPrint.....	489
Создание шаблона PDF	489
Прорисовка PDF-файлов.....	490
Отправка PDF-файлов по электронной почте.....	494
Дополнительные ресурсы	497
Резюме	498
Глава 10. Расширение магазина	499
Создание купонной системы	499
Разработка купонной модели	500
Применение купона к корзине.....	504
Применение купонов к заказам	512
Создание купонов для платежного инструмента Stripe Checkout	517
Добавление купонов в заказы на сайте администрации и в счета-фактуры в формате PDF.....	520
Разработка рекомендательного механизма	523
Рекомендация товаров на основе предыдущих покупок	524
Дополнительные ресурсы	532
Резюме	532
Глава 11. Добавление интернационализации в магазин	534
Интернационализация в Django	535
Настроочные параметры интернационализации и локализации.....	535
Команды управления интернационализацией.....	536
Установка инструментария gettext	536
Как добавлять переводы в проект Django	537
Как Django определяет текущий язык	537
Подготовка проекта к интернационализации	538
Перевод исходного кода Python.....	539
Стандартные переводы.....	540
Ленивые переводы	540
Переводы с переменными.....	540
Формы множественного числа в переводах	541
Перевод собственного исходного кода	541
Перевод шаблонов	545
Шаблонный тег { % trans %}.....	546

Шаблонный тег {% blocktrans %}.....	546
Перевод шаблонов магазина.....	547
Использование интерфейса перевода Rosetta.....	551
Нечеткие переводы.....	554
Шаблоны URL-адресов для интернационализации	554
Добавление префикса языка в шаблоны URL-адресов	555
Перевод шаблонов URL-адресов.....	556
Переключение языка сайта	560
Перевод моделей с помощью модуля django-parler	562
Установка модуля django-parler.....	562
Перевод полей моделей	563
Интеграция переводов на сайт администрирования	565
Создание миграций для переводов моделей	566
Использование переводов с ORM-преобразователем	569
Адаптация представлений под переводы.....	570
Локализация формата	572
Использование модуля django-localflavor для валидации полей формы	573
Дополнительные ресурсы	575
Резюме	576
 Глава 12. Разработка платформы электронного обучения	577
Настройка проекта электронного обучения	578
Раздача медиафайлов.....	579
Разработка моделей курса	580
Регистрация моделей на сайте администрирования	582
Использование фикстур с целью предоставления моделям	
первоначальных данных	583
Создание моделей полиморфного содержимого	586
Использование модельного наследования	587
Абстрактные модели.....	588
Наследование многотабличной модели	588
Прокси-модели	589
Создание моделей Content	589
Создание конкретно-прикладных модельных полей	592
Добавление упорядочивания в модули и объекты содержимого.....	594
Добавление представлений аутентификации	598
Добавление системы аутентификации	598
Создание шаблонов аутентификации	599
Дополнительные ресурсы	602
Резюме	603
 Глава 13. Создание системы управления контентом	604
Создание CMS	604
Создание представлений на основе классов	605

Использование примесей для представлений на основе классов.....	605
Работа с группами и разрешениями.....	608
Ограничение доступа к представлениям, основанным на классах	610
Управление модулями курса и их содержимым	616
Использование наборов форм для модулей курса	616
Добавление содержимого в модули курса	621
Управление модулями и их содержимым.....	627
Переупорядочивание модулей и их содержимого	632
Использование примесей из модуля django-braces	633
Дополнительные ресурсы	641
Резюме	641
Глава 14. Прорисовка и кеширование контента.....	642
Отображение курсов	643
Добавление регистрации студентов	648
Создание представления регистрации студентов	649
Зачисление на курсы	651
Доступ к содержимому курсов	655
Прорисовка разных типов содержимого.....	659
Использование кеш-фреймворка	661
Доступные кеш-бэкенды	662
Установка резидентного кеш-сервера Memcached.....	663
Установка образа Memcached платформы Docker	663
Установка привязки Python к Memcached.....	663
Настроочные параметры кеша	664
Добавление кеш-сервера Memcached в проект	664
Уровни кеша	665
Использование низкоуровневого API кеша.....	665
Проверка запросов к кешу с помощью меню отладочных инструментов Django Debug Toolbar	667
Кеширование на основе динамических данных.....	671
Кеширование фрагментов шаблона	672
Кеширование представлений	673
Использование сайтового кеша	674
Использование кеш-бэкенда Redis	675
Отслеживание сервера Redis с помощью приложения Django Redisboard.....	676
Дополнительные ресурсы	678
Резюме	679
Глава 15. Разработка API	680
Разработка RESTful API	681
Установка фреймворка Django REST framework.....	681
Определение сериализаторов.....	682

Что такое парсер и рендерер.....	683
Разработка представлений списка и детальной информации	684
Потребление API	686
Создание вложенных сериализаторов	688
Разработка конкретно-прикладных представлений API	690
Обработка аутентификации.....	691
Добавление разрешений в представления	692
Создание наборов представлений и маршрутизаторов	694
Добавление дополнительных действий в наборы представлений	696
Создание конкретно-прикладных разрешений	697
Сериализация содержимого курса	697
Потребление RESTful API.....	700
Дополнительные ресурсы	703
Резюме	704
 Глава 16. Разработка чат-сервера.....	705
Создание приложения для ведения чата.....	705
Реализация представления чат-комнаты	706
Реально-временной Django на основе Channels	709
Асинхронные приложения с использованием ASGI.....	710
Цикл запроса/ответа с использованием приложения Channels	710
Установка приложения-обертки Channels	712
Написание потребителя	714
Маршрутизация.....	716
Реализация WebSocket-клиента.....	717
Активирование канального слоя	723
Каналы и группы.....	724
Установление канального слоя с использованием Redis	724
Обновление потребителя с целью широковещательной рассылки сообщений	725
Добавление контекста в сообщения	730
Видоизменение потребителя с целью обеспечения полной асинхронности	733
Интеграция приложения для ведения чата с существующими представлениями	735
Дополнительные ресурсы	736
Резюме	737
 Глава 17. Выход в прямой эфир.....	738
Создание производственной среды	739
Управление настроечными параметрами для нескольких сред	739
Настрочные параметры локальной среды.....	740
Запуск локальной среды.....	741
Настройки производственной среды	741

Использование инструмента Docker Compose.....	743
Установка инструмента Docker Compose	743
Создание файла Dockerfile.....	744
Добавление требующихся пакетов Python.....	745
Создание файла Compose платформы Docker	746
Конфигурирование службы PostgreSQL	749
Применение миграции базы данных и создание суперпользователя	752
Конфигурирование службы Redis.....	753
Раздача Django через WSGI и NGINX	754
Использование сервера приложений uWSGI.....	755
Конфигурирование сервера приложений uWSGI.....	756
Использование веб-сервера NGINX	757
Конфигурирование веб-сервера NGINX	758
Использование хост-имени.....	760
Раздача статических и мультимедийных ресурсов.....	761
Сбор статических файлов.....	761
Раздача статических файлов с помощью веб-сервера NGINX	762
Обеспечение защиты сайта с помощью SSL/TLS	764
Проверка готовности проекта к работе в производственной среде	764
Конфигурирование проекта Django под SSL/TLS.....	765
Создание SSL/TLS-сертификата	767
Конфигурирование веб-сервера NGINX под использование SSL/TLS	767
Перенаправление HTTP-трафика на HTTPS.....	770
Использование Daphne для приложения Django Channels	771
Использование безопасных соединений для веб-сокетов	773
Включение веб-сервера Daphne в конфигурацию веб-сервера NGINX	773
Создание конкретно-прикладных промежуточных программных компонентов.....	777
Создание поддоменного промежуточного компонента.....	778
Раздача нескольких поддоменов с помощью веб-сервера NGINX	780
Реализация конкретно-прикладных команд управления.....	780
Дополнительные ресурсы	783
Резюме	784
Предметный указатель.....	786

От издательства

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 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 укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и 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.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступительное слово

Django: веб-фреймворк для перфекционистов, которые стараются придерживаться дедлайнов.

Мне нравится этот слоган, потому что бывает, что разработчики легко становятся жертвами перфекционизма, когда им приходится доставлять работоспособный исходный код точно в срок.

Есть много отличных веб-фреймворков, но иногда они требуют от разработчика слишком много, например правильно структурировать проект, отыскивать нужные плагины и элегантно использовать существующие абстракции.

Django снимает большую часть этой усталости от принятия решений и предоставляет вам гораздо больше. Но этот фреймворк также и большой, так что изучение его с нуля может оказаться непосильным.

Я изучил Django в 2017 году, в лоб, из необходимости, когда мы решили, что он будет ключевой технологией для нашей платформы программирования на Python ([CodeChalleng.es](#)). Я заставил себя изучить все тонкости, разрабатывая крупное практическое технологическое решение, которое с момента своего создания служило тысячам начинающих и опытных разработчиков Python.

Где-то в этом путешествии я подобрал раннюю редакцию данной книги. И она оказалось настоящей сокровищницей. Очень близкая нашим сердцам в Pybites, она обучает фреймворку Django, помогая **разрабатывать** интересные приложения для решения практических задач. Мало того, Антонио привносит в работу много реального опыта и знаний, что проявляется в том, как он реализует эти проекты.

И Антонио никогда не упускает возможности представить менее известные функциональности, например оптимизацию запросов к базе данных с помощью Postgres, полезные пакеты, такие как django-taggit, социальную аутентификацию с использованием различных платформ, (модельные) менеджеры, шаблонные теги включения и многое другое.

В нескольких главах этого нового издания он даже добавил дополнительные схемы, изображения и примечания и перешел с jQuery на ванильный JavaScript (ну, не приятно ли?!).

Данная книга не только подробно описывает Django, используя чистые примеры исходного кода, которые хорошо объяснены, но и освещает смежные технологии, которые необходимы любому разработчику Django: Django REST framework, django-debug-toolbar, frontend/JS и, последнее, но не менее важное, Docker.

Что еще более важно, вы найдете целый ряд нюансов, с которыми столкнетесь, и лучших образцов практики, которые вам понадобятся, чтобы стать эффективным разработчиком Django в профессиональной среде.

Найти такой многогранный ресурс, как этот, непросто, и я хочу поблагодарить Антонио за всю ту тяжелую работу, которую он постоянно прилагает, чтобы поддерживать его в актуальном состоянии.

Для меня как разработчика Python, который часто использует Django, книга «Django в примерах» стала моим путеводителем, незаменимым ресурсом, который я хочу иметь под рукой. Всякий раз, когда я возвращаюсь к этой книге, я узнаю что-то новое, даже после того, как прочитал ее несколько раз и использую Django уже целых пять лет.

Если вы отправитесь в это путешествие, то будьте готовы к тяжелой практической работе. Ведь это практическое руководство, так что заварите себе хороший кофе и приготовьтесь полностью погрузиться в кучу исходного кода Django! Но именно так нам лучше всего учиться, верно? :)

– *Боб Белдербос*, соучредитель Pybites

Об авторе

Антонио Меле – соучредитель и технический директор Numero, финтех-платформы, которая позволяет финансовым учреждениям строить, автоматизировать и масштабировать цифровые продукты для управления состояниями. Антонио также является техническим директором Exo Investing, цифровой инвестиционной платформы на базе искусственного интеллекта для рынка Великобритании.

Антонио разрабатывает проекты Django с 2006 года для клиентов из нескольких отраслей. В 2009 году Антонио основал Zenx IT, компанию-разработчик, специализирующуюся на разработке цифровых продуктов. Он работал техническим директором и технологическим консультантом во многих технологических стартапах и руководил коллективами разработчиков, создающими проекты для крупных цифровых компаний. Антонио получил степень магистра компьютерных наук в ICAI – Университете Понтификации Комильяс (Pontificia Comillas), в котором он руководит стартапами, находящимися на ранней стадии. Его отец вдохновил его на увлечение компьютерами и программированием.

О рецензенте

Асиф Сайфуддин – разработчик программного обеспечения из Бангладеш. У него десятилетний профессиональный опыт работы с Python и Django. Помимо работы с различными стартапами и клиентами, Асиф также вносит свой вклад в некоторые часто используемые пакеты Python и Django. За его заслуженный вклад в разработку с открытым исходным кодом ныне он является основным сопровождающим такого ПО, как Celery, oAuthLib, PyJWT и auditwheel. Он также является сопровождающим нескольких пакетов расширений Django Extensions и инструментария фреймворка Django REST framework. Кроме того, он является членом фонда **Django Software Foundation (DSF)** с правом голоса и членом фонда **Python Software Foundation (PSF)** с правом участия в разработке/управлении. Для многих молодых людей он является наставником в изучении Python и Django как в профессиональном, так и в личном плане.

*Особая благодарность **Карен Стингел** и **Исмиру Куллолли** за чтение и предоставление отзывов о книге с целью дальнейшего улучшения ее содержимого. Мы очень ценим вашу помощь!*

Предисловие

Django – это веб-фреймворк Python с открытым исходным кодом, который способствует быстрой разработке и чистому, прагматичному дизайну. Он снимает большую часть хлопот, связанных с веб-разработкой, и обеспечивает относительно плавную кривую обучения для начинающих программистов. Django следует философии Python «батарейки включены в комплект», поставляя богатый и разнообразный набор модулей, которые решают распространенные задачи веб-разработки. Простота Django в сочетании с его мощными функциональными возможностями делает его привлекательным как для начинающих, так и для опытных программистов. Django был разработан с учетом простоты, гибкости, надежности и масштабируемости.

В настоящее время Django используется бесчисленными стартапами и крупными организациями, такими как Instagram, Spotify, Pinterest, Udemy, Robinhood и Coursera. Не случайно, что в течение последних нескольких лет в ежегодном опросе разработчиков Stack Overflow разработчики по всему миру неизменно выбирали Django в качестве одного из самых любимых веб-фреймворков.

Эта книга проведет вас через весь процесс разработки профессиональных веб-приложений с помощью Django. Книга посвящена объяснению механизмов работы веб-фреймворка Django путем написания нескольких проектов с нуля. В данной книге содержатся не только наиболее важные аспекты веб-фреймворка, но и объясняется, как применять Django к самым разнообразным реальным ситуациям.

В ней не только рассказывается о Django, но и представлены другие популярные технологии, такие как база данных PostgreSQL, резидентное хранилище Redis, очередь заданий Celery, брокер сообщений RabbitMQ и кеш-сервер Memcached. По ходу чтения книги вы научитесь интегрировать указанные технологии в свои проекты Django, чтобы создавать продвинутые функциональности и разрабатывать сложные веб-приложения.

Книга «*Django 4 в примерах*» проведет вас по всему процессу разработки практических приложений, по ходу дела решая распространенные задачи и внедряя лучшие образцы практики, используя пошаговый подход, которому легко следовать.

Прочитав эту книгу, вы получите хорошее представление о том, как работает Django и как разрабатывать полноценные веб-приложения на Python.

Для кого эта книга предназначена

Данная книга должна послужить руководством для программистов, недавно приступивших к работе с Django. Она предназначена для разработчиков со

знанием Python, которые хотят изучать Django прагматичным образом. Вы можете быть абсолютным новичком в Django либо вы уже его немного знаете, но хотите извлечь из него максимальную пользу. Так или иначе, эта книга поможет вам освоить наиболее актуальные области веб-фреймворка, разрабатывая практические проекты с нуля. Вы должны быть знакомы с концепциями программирования, чтобы при чтении понимать излагаемый материал. В дополнение к базовым знаниям Python подразумевается некоторое предварительное знание HTML и JavaScript.

О чем эта книга рассказывает

Данная книга охватывает целый ряд тем разработки веб-приложений с помощью Django. Она поможет вам разработать четыре разных полнофункциональных веб-приложения, работа над которыми ведется на протяжении 17 глав:

- приложение для ведения блога (главы 1–3);
- веб-сайт по управлению визуальными закладками (главы 4–7);
- интернет-магазин (главы с 8 по 11);
- платформу электронного обучения (главы 12–17).

Каждая глава охватывает несколько функциональных возможностей Django.

Глава 1 «Разработка приложения для ведения блога» ознакомит с веб-фреймворком Django посредством создания приложения для ведения блога. Вы создадите базовые модели, представления, шаблоны и URL-адреса блога, чтобы отображать посты блога на страницах. Вы научитесь формировать наборы запросов QuerySet с помощью объектно-реляционного преобразователя Django (ORM) и сконфигурируете встроенный в Django сайт администрирования.

Глава 2 «Усовершенствование блога за счет продвинутых функциональностей» научит добавлять в свой блог постраничную разбивку и реализовывать представления на основе классов Django. Вы научитесь отправлять электронные письма с помощью Django, а также обрабатывать и моделировать формы. Вы также реализуете систему комментариев к постам блога.

Глава 3 «Расширение приложения для ведения блога» посвящена технике интегрирования сторонних приложений. В этой главе вы ознакомитесь с процессом создания системы тегирования и научитесь формировать сложные наборы запросов QuerySet, чтобы рекомендовать схожие посты. Здесь вы научитесь создавать собственные шаблонные теги и фильтры. Вы также узнаете, как использовать фреймворк карт веб-сайтов и создавать новостную RSS-ленту для своих постов. Вы завершите свое приложение для ведения блога, разработав поисковый механизм, в котором используются возможности полнотекстового поиска PostgreSQL.

Глава 4 «Разработка социального веб-сайта» посвящена объяснению техники разработки социального веб-сайта. Вы научитесь использовать встроенный в Django фреймворк аутентификации и расширите модель пользователя конкретно-прикладной моделью профиля. В этой главе вы научитесь

использовать фреймворк сообщений и разработаете конкретно-прикладной бэкенд аутентификации.

Глава 5 «*Реализация социальной аутентификации*» посвящена реализации социальной аутентификации с использованием учетных записей Google, Facebook и Twitter по стандарту OAuth 2 с помощью механизма Python Social Auth. Вы научитесь использовать расширения Django для работы сервера разработки по протоколу HTTPS и адаптировать конвейер социальной аутентификации под конкретно-прикладную задачу автоматизации создания профиля пользователя.

Глава 6 «*Распространение контента на веб-сайте*» научит технике трансформации социального приложения в веб-сайт по управлению визуальными закладками (CMS). Вы определите взаимосвязи многие-ко-многим в моделях и создадите букмаклет JavaScript, который будет интегрирован в ваш проект. В этой главе будет показано, как создавать миниатюры изображений. Вы также научитесь реализовывать асинхронные HTTP-запросы с использованием JavaScript и Django, и вы реализуете бесконечную постраничную прокрутку контента.

Глава 7 «*Отслеживание действий пользователя*» продемонстрирует технику разработки системы подписки для пользователей. Вы дополните свой веб-сайт по управлению визуальными закладками, создав приложение для слежения за потоками активности пользователей. Вы научитесь создавать обобщенные отношения между моделями и оптимизировать наборы запросов QuerySet, поработаете с сигналами и реализуете денормализацию. Вы научитесь использовать меню отладочных инструментов Django Debug Toolbar, чтобы получать соответствующую отладочную информацию. Наконец, интегрируете в свой проект быстрое хранилище данных Redis, чтобы вести подсчет просмотров изображений, и с помощью него создадите рейтинг наиболее просматриваемых изображений.

Глава 8 «*Разработка интернет-магазина*» посвящена обследованию техники создания интернет-магазина. Вы разработаете модели для каталога товаров и создадите корзину покупок, используя сеансы Django. Вы разработаете процессор контекста для корзины покупок и научитесь управлять заказами клиентов. В этой главе вы научитесь отправлять асинхронные уведомления с помощью очереди заданий Celery и брокера сообщений RabbitMQ. Вы также научитесь отслеживать Celery с помощью мониторингового инструмента Flower.

Глава 9 «*Управление платежами и заказами*» посвящена технике интегрирования платежного шлюза в интернет-магазин. Вы выполните интеграцию платежного инструмента Stripe Checkout и будете получать асинхронные уведомления о платежах в своем приложении. Вы реализуете конкретно-прикладные представления на сайте администрирования, а также адаптируете сайт администрирования под конкретно-прикладную задачу экспорта заказов в CSV-файлы. Вы также научитесь динамически создавать счета-фактуры в формате PDF.

Глава 10 «*Расширение магазина*» научит технике создания купонной системы для применения скидок к корзине покупок. Вы обновите интеграцию платежного инструмента Stripe Checkout, чтобы имплементировать купон-

ные скидки, и будете применять купоны к заказам. Вы будете использовать резидентное хранилище Redis для хранения товаров, которые обычно покупаются вместе, и применять эту информацию для разработки механизма рекомендации товаров.

Глава 11 «Добавление интернационализации в магазин» покажет, как добавлять интернационализацию в проект. Вы научитесь генерировать файлы перевода и управлять ими, а также переводить строковые литералы в исходном коде Python и шаблонах Django. Вы будете использовать приложение Rosetta, чтобы управлять переводами, и реализуете URL-адреса в зависимости от применяемого языка. Вы научитесь переводить поля моделей с помощью модуля `django-parler` и использовать переводы с помощью ORM-преобразователя. Наконец, создадите локализованное поле формы, используя модуль `django-localflavor`.

Глава 12 «Разработка платформы электронного обучения» проведет вас по процессу создания платформы электронного обучения. В ваш проект будут добавлены фикстуры, и вы создадите первоначальные модели для системы управления контентом. Вы будете использовать наследование моделей, чтобы создавать модели данных для полиморфного контента. Вы научитесь создавать конкретно-прикладные модельные поля, разработав поле для упорядочивания объектов. Вы также реализуете представления аутентификации для системы управления контентом.

Глава 13 «Создание системы управления контентом» научит технике создания системы управления контентом, используя представления на основе классов и примесных классов. Вы воспользуетесь встроенными в Django группами и системой разрешений, чтобы ограничивать доступ к представлениям, и реализуете наборы форм, дабы редактировать содержимое курсов. Вы также создадите функциональность перетаскивания, чтобы переупорядочивать модули курса и их содержимое с помощью JavaScript и Django.

Глава 14 «Прорисовка и кеширование контента» покажет, как реализовывать общедоступные представления для каталога курсов. Вы создадите систему регистрации студентов и будете управлять зачислением студентов на курсы. Вы напишете функциональность прорисовки различных типов контента курсовых модулей. Вы научитесь кешировать контент с помощью кеш-фреймворка Django и конфигурировать кеш-бэкенд Memcached и Redis под свой проект. Наконец, вы научитесь отслеживать Redis с помощью сайта администрирования.

Глава 15 «Разработка API» посвящена обследованию техники разработки RESTful API к своему проекту с помощью фреймворка Django REST framework. Вы научитесь создавать сериализаторы для моделей и конкретно-прикладные представления API. Вы будете оперировать аутентификацией по API и реализуете разрешения для представлений API. Научитесь разрабатывать наборы представлений API и маршрутизаторы. В этой главе также будет рассказано о методах использования API с помощью библиотеки `requests`.

Глава 16 «Разработка сервера чатов» посвящена технике применения приложения Django Channels с целью создания реально-временного чата-сервера для студентов. Вы научитесь реализовывать функциональности, которые опираются на асинхронную связь путем обмена данными по протоколу

WebSocket. Вы создадите WebSocket-потребителя с помощью Python и реализуете WebSocket-клиента с помощью JavaScript. Вы будете использовать хранилище Redis для настройки канального слоя и научитесь делать своего WebSocket-потребителя полностью асинхронным.

Глава 17 «Выход в прямой эфир» покажет, как создавать настроечные параметры для нескольких сред и устанавливать производственную среду на основе базы данных PostgreSQL, хранилища Redis, сервера приложения uWSGI, веб-сервера NGINX и асинхронного веб-сервера Daphne с помощью инструмента Docker Compose. Вы научитесь безопасно раздавать свой проект по протоколу HTTPS и использовать встроенный в Django фреймворк проверки системы. В этой главе также будет рассказано о методах разработки конкретно-прикладных промежуточных программных компонентов и конкретно-прикладных команд управления.

Что нужно, чтобы извлечь максимум пользы из этой книги

- Читатель должен обладать хорошими практическими знаниями Python.
- Читатель должен хорошо разбираться в HTML и JavaScript.
- Рекомендуется, чтобы читатель ознакомился с частями 1–3 практического руководства в официальной документации Django по адресу <https://docs.djangoproject.com/en/4.1/intro/tutorial01/>.

Используемые обозначения

В этой книге используется ряд текстовых обозначений.

ИсходныйКодВТексте: указывает слова исходного кода в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, имена путей, фиктивные URL-адреса, вводимые пользователем данные и дескрипторы Twitter. Например, «Отредактировать файл `models.py` приложения `shop`».

Блок исходного кода задается, как показано ниже:

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

admin.site.register(Post)
```

Когда мы хотим привлечь ваше внимание к определенной части блока исходного кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
INSTALLED_APPS = [
    'django.contrib.admin',
```

```
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'blog.apps.BlogConfig',
]
```

Любые данные на входе или на выходе из команды командой оболочки записываются, как показано ниже:

```
python manage.py runserver
```

Жирный шрифт: выделяет новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах пишутся в тексте следующим образом: «заполнить форму и нажать кнопку **Сохранить**».



Предупреждения или важные примечания помечаются так.



Советы и программные трюки выглядят так.

1

Разработка приложения для ведения блога

В этой книге вы научитесь разрабатывать профессиональные проекты Django. В данной главе будет продемонстрировано, как разрабатывать приложение Django, используя главные компоненты веб-фреймворка. Если Django у вас еще не установлен, то в первой части главы вы научитесь это делать.

Прежде чем приступить к первому проекту Django, давайте воспользуемся моментом, чтобы посмотреть, чему вы научитесь. В данной главе вы получите общий обзор веб-фреймворка. В ней вы ознакомитесь с различными самыми главными компонентами, служащими для создания полнофункционального веб-приложения: моделями, шаблонами, представлениями и URL-адресами. После ее прочтения вы будете иметь хорошее понимание того, как работает Django и как взаимодействуют различные компоненты веб-фреймворка.

В ней вы узнаете разницу между проектами и приложениями Django, а также ознакомитесь с наиболее важными настроочными параметрами Django. Вы разработаете простое приложение для ведения блога, которое позволит пользователям перемещаться по всем опубликованным постам и читать одиночные посты. Вы также создадите простой интерфейс администрирования, чтобы управлять постами и их публиковать. В следующих двух главах вы расширите приложение для ведения блога более продвинутыми функциональностями.

Данная глава должна послужить руководством по разработке полноценного приложения Django и дать представление о механизмах работы веб-фреймворка. Не беспокойтесь, если вы не понимаете всех аспектов веб-фреймворка. Различные компоненты веб-фреймворка будут подробно рассматриваться на протяжении всей этой книги.

В данной главе будут освещены следующие темы:

- установка Python;
- создание виртуальной среды Python;
- установка Django;

- создание и конфигурирование проекта Django;
- разработка приложения Django;
- конструирование моделей данных;
- создание и применение миграций моделей;
- создание сайта администрирования для моделей;
- работа с наборами запросов QuerySet и модельными менеджерами;
- формирование представлений, шаблонов и URL-адресов;
- понимание цикла «запрос/ответ» Django.

Установка языка Python

Django 4.1 поддерживает язык Python версий 3.8, 3.9 и 3.10. В приведенных в этой книге примерах мы будем использовать Python 3.10.6.

Если вы применяете Linux или macOS, то у вас, вероятно, Python уже установлен. Если же вы используете Windows, то на странице <https://www.python.org/downloads/windows/> можно скачать установщик Python.

Откройте оболочку командной строки своего компьютера. Если вы используете macOS, то откройте каталог /Applications/Utilities в файловом менеджере **Finder**, затем дважды кликните по **Terminal**. Если вы используете Windows, то откройте меню **Пуск** и в поле поиска наберите cmd. Затем кликните по приложению **командной строки**, чтобы его открыть.

Проверьте, что на вашем компьютере Python установлен, набрав следующую ниже команду в командной оболочке:

```
python
```

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

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

Если установленная версия Python ниже 3.10 или если Python на вашем компьютере не установлен, то скачайте Python 3.10.6 на странице <https://www.python.org/downloads/> и следуйте инструкциям по его установке. На странице скачиваний находится установщик Python для Windows, macOS и Linux.

На протяжении всей этой книги, когда в командной оболочке упоминается Python, мы будем использовать команду `python`, хотя в некоторых системах, возможно, потребуется применение команды `python3`. Если вы работаете на Linux или macOS, а в вашей системе используется Python 2, то для того чтобы задействовать установленную вами версию Python 3, вам нужно будет использовать команду `python3`.

В Windows команда `python` представляет исполняемый файл установки Python, которая используется по умолчанию, тогда как команда `py` – программу

быстрого запуска языка Python. Программа быстрого запуска языка Python для Windows была впервые представлена в Python версии 3.3. Она выясняет версии языка Python, которые были установлены на вашем компьютере, и автоматически переключается на последнюю версию. Если вы работаете с Windows, то рекомендуется заменить команду `python` командой `py`. Подробнее о программе быстрого запуска языка Python в Windows можно почитать на странице <https://docs.python.org/3/using/windows.html#launcher>.

Создание виртуальной среды Python

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

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

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

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

```
python -m venv my_env
```

В случае если ваша система идет в комплекте с Python 2 и вы установили Python 3, то не забудьте применить команду `python3` вместо `python`.

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

```
py -m venv my_env
```

При этом будет использоваться программа быстрого запуска Python в Windows.

Приведенная выше команда создаст среду Python в новом каталоге с именем `my_env/`. Любые библиотеки Python, которые устанавливаются вами, пока

ваша виртуальная среда является активной, будут помещаться в каталог `my_env/lib/python3.10/site-packages`.

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

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

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

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

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

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

Свою среду можно деактивировать в любое время с помощью команды `deactivate`. Более подробная информация о `venv` находится на странице <https://docs.python.org/3/library/venv.html>.

Установка веб-фреймворка Django

Если вы уже установили Django 4.1, то этот раздел можно пропустить и перейти непосредственно к разделу «Создание первого проекта».

Django поставляется в виде модуля Python, и, следовательно, его можно устанавливать в любой среде Python. Если вы еще не установили Django, то ниже приведено краткое руководство по его установке на компьютер.

Установка Django с помощью pip

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

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

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

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

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

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

Если вы получите результат 4.1.X, то, значит, Django был успешно установлен на вашем компьютере. Если вы получаете сообщение No module named Django, то, значит, Django на вашем компьютере не установлен. Если у вас возникли проблемы с установкой Django, то на странице <https://docs.djangoproject-project.com/en/4.1/intro/install/> можно уточнить различные опции установки.



Веб-фреймворк Django можно устанавливать различными способами. С вариантами опций установки можно ознакомиться на странице <https://docs.djangoproject-project.com/en/4.1/topics/install/>.

Все используемые в этой главе пакеты Python включены в файл requirements.txt в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды pip install -r requirements.txt.

Новые функциональные возможности Django 4

Django 4 вводит набор новых функциональных возможностей, включающий несколько обратно несовместимых изменений, в то же время объявляя о скорой замене других функциональностей и устранивая старые. Поскольку релиз Django 4 основан на времени, в нем нет кардинальных изменений, и приложения Django 3 легко мигрируются в версию 4.1. В отличие от релиза Django 3, куда впервые была включена поддержка интерфейса шлюза асинхронного сервера ASGI¹, в Django 4.0 добавлено несколько характерных особенностей, таких как функциональные ограничения по уникальности для моделей Django, встроенная поддержка кеширования данных с использованием сервера хранилища Redis, новая стандартная реализация часового пояса с применением стандартного пакета Python zoneinfo, новый механизм хеширования паролей scrypt, шаблонно-ориентированная прорисовка форм, а также другие новые второстепенные функциональности. В Django 4.0 отменяется поддержка Python 3.6 и 3.7. В нем также прекращается поддержка PostgreSQL 9.6, Oracle 12.2 и Oracle 18c. В Django 4.1 вводятся асинхронные обработчики представлений на основе классов, асинхронный интерфейс ORM-преобразователя, новая валидация модельных ограничений и новые шаблоны прорисовки форм. В версии 4.1 прекращается поддержка PostgreSQL 10 и MariaDB 10.2.

С полным списком изменений можно ознакомиться в примечаниях к релизу Django 4.0 на странице <https://docs.djangoproject-project.com/en/dev/releases/4.0/> и примечаниях к релизу Django 4.1 на странице <https://docs.djangoproject-project.com/en/4.1/releases/4.1/>.

¹ Англ. Asynchronous Server Gateway Interface. – Прим. перев.

Общий обзор веб-фреймворка Django

Django – это веб-фреймворк, состоящий из набора компонентов, которые решают распространенные задачи веб-разработки. Компоненты Django слабо сцеплены между собой, и поэтому ими можно управлять независимо, что помогает разделять обязанности разных слоев веб-фреймворка; слой базы данных ничего не знает о том, как данные отображаются на странице, система шаблонов ничего не знает о веб-запросах и т. д.

Django обеспечивает максимальную возможность реиспользования исходного кода, следуя принципу DRY¹. Django также способствует быстрой разработке и позволяет использовать меньше исходного кода, применяя динамические возможности языка Python, такие как интроспекция.

Подробнее о философии дизайна Django можно почитать на странице <https://docs.djangoproject.com/en/4.1/misc/design-philosophies/>.

Главные компоненты веб-фреймворка

Django подчиняется шаблону архитектурного дизайна MTV (Model-Template-View)². Он немного похож на хорошо известный шаблон архитектурного дизайна MVC (Model-View-Controller)³, где Template (Шаблон)⁴ действует как View (Представление), а сам веб-фреймворк действует как Controller (Контроллер).

Обязанности в шаблоне архитектурного дизайна MTV Django распределены следующим образом:

- **модель** – определяет логическую структуру данных и является обработчиком данных между базой данных и их представлением;
- **шаблон** – это слой представления. В Django используется система текстовых шаблонов, в которой хранится все, что браузер прорисовывает на страницах;
- **представление** – взаимодействует с базой данных через модель и передает данные в шаблон для их прорисовки и просмотра.

Сам веб-фреймворк выступает в качестве контроллера. Он отправляет запрос в надлежащее представление в соответствии с конфигурацией URL-адреса.

При разработке любого проекта Django вы всегда будете работать с моделями, представлениями, шаблонами и URL-адресами. В этой главе вы узнаете, как они сочетаются друг с другом.

¹ От англ. don't repeat yourself (не повторяйся). – Прим. перев.

² Модель–шаблон–представление. – Прим. перев.

³ Модель–представление–контроллер. – Прим. перев.

⁴ Синоним «трафарет». В рамках Django «шаблон» (или трафарет) представляет собой некую заготовку страницы или пустой бланк, в котором выделено несколько пустых полей различной формы, служащий для размножения документов. Шаблон накладывается на контекстные данные и прорисовывается. См. <https://ru.wikipedia.org/wiki/Трафарет>. – Прим. перев.

Архитектура Django

На рис. 1.1 показано, как Django обрабатывает запросы и как различные главные компоненты Django – модели, шаблоны, URL-адреса и представления – управляет циклом запроса/ответа.



Рис. 1.1. Архитектура Django

Вот как Django оперирует HTTP-запросами и генерирует ответы:

1. Веб-браузер запрашивает страницу по ее URL-адресу, и веб-сервер передает HTTP-запрос веб-фреймворку Django.
2. Django просматривает свои сконфигурированные шаблоны URL-адресов и останавливается на первом, который совпадает с запрошенным URL-адресом.
3. Django исполняет представление, соответствующее совпавшему шаблону URL-адреса.
4. Представление потенциально использует модели данных, чтобы извлекать информацию из базы данных.
5. Модели данных обеспечивают определение данных и их поведение. Они используются для запроса к базе данных.
6. Представление прорисовывает¹ шаблон (обычно с использованием HTML), чтобы отображать данные на странице, и возвращает их вместе с HTTP-ответом.

¹ Согласно документации Django под рендерингом понимается взятие промежуточного представления шаблона вместе с контекстом и его превращение в итоговый поток байтов, который может быть передан клиенту (например, браузеру). На техническом языке это *интерполяция* (т. е. заполнение) шаблона контекстными данными и возврат результатирующего строкового литерала, или *трансляция* данных в другой формат. В переводе при изложении тем, связанных с архитектурой MVP и шаблонами, используется термин «прорисовка» шаблона, а при обсуждении темы API – термин «трансляция» (см. <https://docs.djangoproject.com/en/stable/ref/template-response/#the-rendering-process>). – Прим. перев.

Мы вернемся к циклу запроса/ответа Django в конце этой главы в разделе «Цикл запроса/ответа».

В составе Django также имеются перехватчики¹, вызываемые внутри процесса запроса/ответа, которые называются промежуточными программными компонентами². Промежуточные программные компоненты были намеренно исключены из этой диаграммы ради простоты. Вы будете использовать промежуточные программные компоненты в различных примерах этой книги, а о том, как создавать конкретно-прикладной промежуточный программный компонент, вы узнаете в главе 17 «Выход в прямой эфир».

Создание первого проекта

Ваш первый проект Django будет состоять из приложения для ведения блога. Мы начнем с создания проекта Django и приложения Django для ведения блога. Затем создадим модели данных и синхронизируем их с базой данных.

Django предоставляет команду, которая позволяет создавать изначальную файловую структуру проекта. Выполните следующую ниже команду в командной оболочке:

```
django-admin startproject mysite
```

Она создаст проект Django с именем `mysite`.



Во избежание конфликтов имен следует избегать именования проектов по имени встроенных модулей Python или Django.

Давайте взглянем на сгенерированную структуру проекта:

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

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

¹ Син. зацепки, хуки. – Прим. перев.

² Англ. middleware. – Прим. перев.

- `manage.py`: это утилита командной строки, используемая для взаимодействия с проектом. Редактировать этот файл не требуется;
- `mysite/`: это пакет проекта на языке Python; пакет состоит из следующих ниже файлов:
 - `__init__.py`: пустой файл, который сообщает Python, что каталог `mysite` нужно трактовать как модуль Python;
 - `asgi.py`: конфигурация для выполнения проекта в качестве приложения, работающего по протоколу интерфейса шлюза асинхронного сервера (**ASGI**) с ASGI-совместимыми веб-серверами. ASGI – это новый стандарт Python для асинхронных веб-серверов и приложений;
 - `settings.py`: здесь указаны настроочные параметры и конфигурация проекта и содержатся изначальные параметры со значениями, используемыми по умолчанию;
 - `urls.py`: место, где располагаются ваши шаблоны URL-адресов. Каждый URL-адрес, который определен здесь, соотносится с представлением;
 - `wsgi.py`: конфигурация для выполнения проекта в качестве приложения, работающего по протоколу интерфейса шлюза веб-сервера (**WSGI**)¹ с WSGI-совместимыми веб-серверами.

Применение первоначальных миграций базы данных

Для того чтобы хранить данные, приложениям Django требуется база данных. Упомянутый выше файл `settings.py` содержит конфигурацию базы данных проекта в настроечном параметре `DATABASES`. Изначально конфигурацией предусматривается использование базы данных SQLite3, если не указана иная. SQLite идет в комплекте с Python 3 и может применяться в любом приложении Python. SQLite – это облегченная база данных, которую можно использовать с Django для разработки. Если вы планируете развернуть свое приложение в производственной среде, то вам следует использовать полнофункциональную базу данных, такую как PostgreSQL, MySQL или Oracle. Более подробная информация о совместной работе базы данных с Django содержится по адресу <https://docs.djangoproject.com/en/4.1/topics/install/#database-installation>.

Файл `settings.py` также содержит настроочный параметр `INSTALLED_APPS` со списком, содержащим распространенные приложения Django, которые добавляются в ваш проект по умолчанию. Мы рассмотрим эти приложения позже в разделе «Настроочные параметры проекта».

Приложения Django содержат модели данных, которые соотносятся с таблицами базы данных. В разделе «Создание моделей данных блога» вы создадите свои собственные модели. Для того чтобы завершить настройку проекта, необходимо создать таблицы, ассоциированные с моделями стандартных

¹ Англ. Web Server Gateway Interface. – Прим. перев.

приложений Django, включенных в состав параметра `INSTALLED_APPS`. Django поставляется вместе с системой, которая помогает управлять миграциями баз данных.

Откройте приглашение командной оболочки и выполните следующие ниже команды:

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

Вы увидите результат, который заканчивается следующими ниже строками:

```
Applying contenttypes.0001_initial... OK  
Applying auth.0001_initial... OK  
Applying admin.0001_initial... OK  
Applying admin.0002_logentry_remove_auto_add... OK  
Applying admin.0003_logentry_add_action_flag_choices... OK  
Applying contenttypes.0002_remove_content_type_name... OK  
Applying auth.0002_alter_permission_name_max_length... OK  
Applying auth.0003_alter_user_email_max_length... OK  
Applying auth.0004_alter_user_username_opts... OK  
Applying auth.0005_alter_user_last_login_null... OK  
Applying auth.0006_require_contenttypes_0002... OK  
Applying auth.0007_alter_validators_add_error_messages... OK  
Applying auth.0008_alter_user_username_max_length... OK  
Applying auth.0009_alter_user_last_name_max_length... OK  
Applying auth.0010_alter_group_name_max_length... OK  
Applying auth.0011_update_proxy_permissions... OK  
Applying auth.0012_alter_user_first_name_max_length... OK  
Applying sessions.0001_initial... OK
```

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

Подробнее о команде управления `migrate` можно узнать в разделе «Создание и применение миграций» данной главы.

Запуск и выполнение сервера разработки

Django идет в комплекте вместе с облегченным веб-сервером с целью быстрого выполнения вашего исходного кода без необходимости тратить время на настройку производственного сервера. Во время работы сервера разработки он непрерывно проверяет наличие изменений в исходном коде. Он автоматически перезагружается, освобождая от необходимости перезагружать его вручную после изменения кода. Однако есть случаи, когда он может не

замечать некоторые действия, такие как добавление новых файлов в проект, поэтому в подобных случаях приходится перезапускать сервер вручную.

Запустите сервер разработки, набрав следующую ниже команду в командной оболочке:

```
python manage.py runserver
```

Вы должны увидеть что-то вроде этого:

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

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



Рис. 1.2. Домашняя страница сервера разработки

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

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

Каждый HTTP-запрос регистрируется в консоли сервером разработки. Любая ошибка, возникающая во время работы сервера разработки, также будет появляться в консоли.

Сервер разработки можно выполнять на конкретно-прикладном хосте и порту либо сообщать Django, что нужно загружать определенный настроочный файл, как показано ниже:

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



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

Этот сервер предназначен только для разработки и не подходит для производственного использования. Для того чтобы развернуть Django в производственной среде, необходимо его выполнять как приложение на основе WSGI с использованием такого веб-сервера, как Apache, Gunicorn или uWSGI, или же как приложение на основе ASGI с использованием такого сервера, как Daphne или Uvicorn. Более подробная информация о том, как разворачивать Django с различными веб-серверами, находится на странице <https://docs.djangoproject.com/en/4.1/howto/deployment/wsgi/>.

В главе 17 «Выход в прямой эфир» объясняется техника настраивания производственной среды под проекты Django.

Настроечные параметры проекта

Давайте откроем файл `settings.py` и взглянем на конфигурацию проекта. Несколько настроечных параметров уже внесены в указанный файл веб-фреймворком Django, но это лишь часть всех имеющихся параметров. Все настроечные параметры и их значения, которые используются по умолчанию, можно увидеть на странице <https://docs.djangoproject.com/en/4.1/ref/settings/>.

Давайте рассмотрим некоторые настроечные параметры проекта.

- `DEBUG` – это булев параметр, который включает и выключает режим отладки проекта. Если его значение установлено равным `True`, то Django будет отображать подробные страницы ошибок в случаях, когда приложение выдает неперехваченное исключение. При переходе в производственную среду следует помнить о том, что необходимо устанавливать его значение равным `False`. Никогда не развертывайте свой сайт в производственной среде с включенной отладкой, поскольку вы предоставите конфиденциальные данные, связанные с проектом.
- `ALLOWED_HOSTS` не применяется при включенном режиме отладки или при выполнении тестов. При перенесении своего сайта в производственную среду и установке параметра `DEBUG` равным `False` в этот настроечный

параметр следует добавлять свои домен/хост, чтобы разрешить ему раздавать ваш сайт Django.

- INSTALLED_APPS – это параметр, который придется редактировать во всех проектах. Он сообщает Django о приложениях, которые для этого сайта являются активными. По умолчанию Django вставляет следующие ниже приложения:
 - django.contrib.admin: сайт администрирования;
 - django.contrib.auth: фреймворк аутентификации;
 - django.contrib.contenttypes: фреймворк типов контента;
 - django.contrib.sessions: фреймворк сеансов;
 - django.contrib.messages: фреймворк сообщений;
 - django.contrib.staticfiles: фреймворк управления статическими файлами.
- MIDDLEWARE – подлежащие исполнению промежуточные программные компоненты.
- ROOT_URLCONF указывает модуль Python, в котором определены шаблоны корневых URL-адресов приложения.
- DATABASES – словарь, содержащий настроечные параметры всех баз данных, которые будут использоваться в проекте. Всегда должна существовать база данных, которая будет использоваться по умолчанию. В стандартной конфигурации используется база данных SQLite3, если не указана иная.
- LANGUAGE_CODE определяет заранее заданный языковой код этого сайта Django.
- USE_TZ сообщает Django, что нужно активировать/деактивировать поддержку часовых поясов. Django поставляется вместе с поддержкой дат и времен с учетом часовых поясов. Этот настроечный параметр получает значение True при создании нового проекта с помощью команды управления startproject.

Не волнуйтесь, если вы многое не понимаете из того, что здесь видите. Подробнее о различных настроечных параметрах Django можно узнать в последующих главах.

Проекты и приложения

На протяжении всей этой книги вы снова и снова будете сталкиваться с терминами «проект» и «приложение». В Django проектом считается установленный веб-фреймворк Django с несколькими настроечными параметрами. Приложение – это группа моделей, шаблонов, URL-адресов и представлений. Приложения взаимодействуют с веб-фреймворком с целью обеспечения определенных функциональностей, и их можно реиспользовать в разных проектах. Проект можно трактовать как свой собственный веб-сайт, содержащий несколько приложений, таких как блог, вики или форум, который другие проекты Django тоже могут использовать.

На рис. 1.3 показана структура проекта Django.

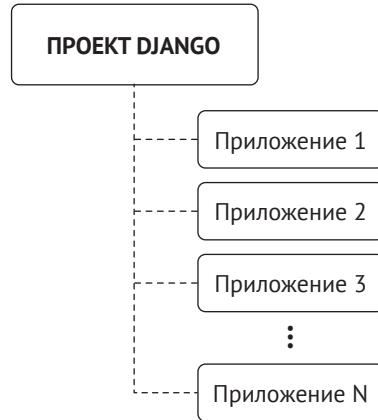


Рис. 1.3. Структура проекта/приложений Django

Создание приложения

Давайте создадим первое приложение Django. Мы разработаем приложение для ведения блога с нуля.

Выполните следующую ниже команду в командной оболочке из корневого каталога проекта:

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

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

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

Ниже приведено описание этих файлов:

- `__init__.py`: пустой файл, который сообщает Python, что каталог `blog` нужно трактовать как модуль Python;
- `admin.py`: здесь вы регистрируете модели, чтобы включать их в состав сайта администрирования – этот сайт используется optionalno, по вашему выбору;
- `apps.py`: содержит главную конфигурацию приложения `blog`;
- `migrations`: этот каталог будет содержать миграции базы данных приложения. Миграции позволяют Django отслеживать изменения модели

и соответствующим образом синхронизировать базу данных. Указанный каталог содержит пустой файл `__init__.py`;

- `models.py`: содержит относимые к приложению модели данных; все приложения Django должны иметь файл `models.py`, но его можно оставлять пустым;
- `tests.py`: здесь можно добавлять относимые к приложению тесты;
- `views.py`: здесь расположена логика приложения; каждое представление получает HTTP-запрос, обрабатывает его и возвращает ответ.

Когда структура приложения готова, можно приступать к разработке моделей данных блога.

Создание моделей данных блога

Напомним, что объект Python – это набор данных и методов. Класс – это концептуальная схема, которая объединяет данные и функциональности в единое целое. Создание нового класса влечет новый тип объекта, позволяя формировать экземпляры этого типа.

Модель Django – это источник информации и поведения данных. Она состоит из класса Python, который является подклассом `django.db.models.Model`. Каждой модели ставится в соответствие одна таблица базы данных, где каждый атрибут класса соотносится с полем базы данных. Когда вы будете создавать модель, Django будет предоставлять практический API, чтобы легко запрашивать объекты в базе данных.

Сначала мы определим модели баз данных для приложения `blog`. Затем сгенерируем для этих моделей миграции базы данных, чтобы создать соответствующие таблицы базы данных. При применении миграций Django будет создавать таблицу по каждой модели, определенной в файле `models.py` приложения.

Создание модели поста

Сначала мы определим модель `Post`, которая позволит хранить посты блога в базе данных.

Добавьте следующие ниже строки в файл `models.py` приложения `blog`. Новые строки выделены жирным шрифтом:

```
from django.db import models

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

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

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

- `title`: поле заголовка поста. Это поле с типом `CharField`, которое транслируется в столбец `VARCHAR` в базе данных SQL;
- `slug`: поле `SlugField`, которое транслируется в столбец `VARCHAR` в базе данных SQL. Слаг – это короткая метка, содержащая только буквы, цифры, знаки подчеркивания или дефисы. Пост с заголовком «*Django Reinhardt: A legend of Jazz*» мог бы содержать такой заголовок: «*djangoreinhardt-legend-jazz*». В главе 2 «Усовершенствование блога за счет продвинутых функциональностей» мы будем использовать поле `slug` для формирования красивых и дружественных для поисковой оптимизации URL-адресов постов блога;
- `body`: поле для хранения тела поста. Это поле с типом `TextField`, которое транслируется в столбец `Text` в базе данных SQL.

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



Если вы использовали Python 2.x, то обратите внимание, что в Python 3 все строковые литералы изначально считаются кодированными в Юникоде; поэтому мы используем только метод `__str__()`. Метод `__unicode__()` из Python 2.x устарел.

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



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

Django создаст столбец базы данных для каждого поля модели: `title`, `slug` и `body`. На рисунке хорошо видно, как каждый тип поля соответствует типу данных в базе данных.

Django по умолчанию добавляет поле автоматически увеличивающегося первичного ключа в каждую модель. Тип этого поля указывается в конфигурации каждого приложения либо глобально в настроочном параметре `DEFAULT_AUTO_FIELD`. При создании приложения командой `startapp` значение параметра `DEFAULT_AUTO_FIELD` по умолчанию имеет тип `BigAutoField`. Это 64-битное целое число, которое увеличивается автоматически в соответствии с доступными идентификаторами. Если не указывать первичный ключ своей модели, то Django будет добавлять это поле автоматически. В качестве первичного ключа можно также определить одно из полей модели, установив для него параметр `primary_key=True`.

Мы расширим модель `Post` дополнительными полями и поведением. После завершения мы синхронизируем ее с базой данных, создав миграцию в базе данных и применив ее.

Добавление полей даты/времени

Мы продолжим, добавив в модель `Post` различные поля даты/времени. Каждый пост будет публиковаться в определенную дату и время. Следовательно, необходимо иметь поле для хранения даты и времени публикации. Мы также хотим хранить дату и время создания объекта `Post` и его последнего изменения.

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

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

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

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

В модель `Post` были добавлены следующие ниже поля:

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

даются методом Django `timezone.now`. Обратите внимание, что для того, чтобы использовать этот метод, был импортирован модуль `timezone`. Метод `timezone.now` возвращает текущую дату/время в формате, зависящем от часового пояса. Его можно трактовать как версию стандартного метода Python `datetime.now` с учетом часового пояса;

- `created`: поле с типом `DateTimeField`. Оно будет использоваться для хранения даты и времени создания поста. При применении параметра `auto_now_add` дата будет сохраняться автоматически во время создания объекта;
- `updated`: поле с типом `DateTimeField`. Оно будет использоваться для хранения последней даты и времени обновления поста. При применении параметра `auto_now` дата будет обновляться автоматически во время сохранения объекта.

Определение предустановленного порядка сортировки

Посты блога обычно отображаются на странице в обратном хронологическом порядке (от самых новых к самым старым). В нашей модели мы определим заранее заданный порядок. Он будет применяться при извлечении объектов из базы данных, в случае если в запросе порядок не будет указан.

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

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

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

    class Meta:
        ordering = ['-publish']

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

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

указан конкретный порядок. Убывающий порядок задается с помощью дефиса перед именем поля: `-publish`. По умолчанию посты будут возвращаться в обратном хронологическом порядке.

Добавление индекса базы данных

Давайте определим индекс базы данных по полю `publish`. Индекс повысит производительность запросов, фильтрующих или упорядочивающих результаты по указанному полю. Мы ожидаем, что многие запросы извлекут преимущества из этого индекса, поскольку для упорядочивания результатов мы по умолчанию используем поле `publish`.

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

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

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

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

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

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



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

Дополнительная информация о том, как определять индексы в моделях, находится на странице <https://docs.djangoproject.com/en/4.1/ref/models/indexes/>.

Активация приложения

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

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

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

Класс `BlogConfig` – это конфигурация приложения. Теперь Django знает, что для этого проекта приложение является активным, и сможет загружать модели приложения.

Добавление поля статуса

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

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

```
from django.db import models  
from django.utils import timezone  
  
class Post(models.Model):  
  
    class Status(models.TextChoices):  
        DRAFT = 'DF', 'Draft'  
        PUBLISHED = 'PB', 'Published'
```

```

title = models.CharField(max_length=250)
slug = models.SlugField(max_length=250)
body = models.TextField()
publish = models.DateTimeField(default=timezone.now)
created = models.DateTimeField(auto_now=True)
updated = models.DateTimeField(auto_now=True)
status = models.CharField(max_length=2,
                         choices=Status.choices,
                         default=Status.DRAFT)

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

def __str__(self):
    return self.title

```

Мы определили перечисляемый класс `Status` путем подклассирования класса `models.TextChoices`. Доступными вариантами статуса поста являются `DRAFT` и `PUBLISHED`. Их соответствующими значениями выступают `DF` и `PB`, а их метками или читаемыми именами являются *Draft* и *Published*.

Django предоставляет перечисляемые типы, которые можно подклассировать, чтобы легко и просто определять варианты выбора. Они основаны на объекте `enum` стандартной библиотеки Python. Подробнее об `enum` можно почитать на странице <https://docs.python.org/3/library/enum.html>.

Перечисляемые типы Django имеют несколько видоизменений по сравнению с `enum`. Об этих различиях можно узнать по адресу <https://docs.djangoproject.com/en/4.1/ref/models/fields/#enumeration-types>.

Для того чтобы получать имеющиеся варианты, можно обращаться к вариантам статуса (`Post.Status.choices`), для того чтобы получать удобочитаемые имена – к меткам статуса (`Post.Status.labels`), и для того чтобы получать фактические значения вариантов – к значениям статуса (`Post.Status.values`).

В модель также было добавлено новое поле `status`, являющееся экземпляром типа `CharField`. Оно содержит параметр `choices`, чтобы ограничивать значение поля вариантами из `Status.choices`. Кроме того, применяя параметр `default`, задано значение поля, которое будет использоваться по умолчанию. В этом поле статус `DRAFT` используется в качестве предустановленного варианта, если не указан иной.



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

Давайте посмотрим, как взаимодействовать с вариантами статуса.

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

```
python manage.py shell
```

Затем наберите такие строки:

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

Вы получите варианты перечисления в формате пар значение–метка, по-добные показанным ниже:

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

Наберите следующую ниже строку:

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

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

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

Наберите следующую ниже строку:

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

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

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

Наберите такую строку:

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

Вы получите имена вариантов, как показано ниже:

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

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

Добавление взаимосвязи многие-к-одному

Посты всегда пишутся автором. В данном разделе будет создана взаимосвязь¹ между пользователями и постами, которая будет указывать на конкретных пользователей и написанные ими посты. Django идет в комплекте с фреймворком аутентификации, который ведет учетные записи пользователей. Встроенный в Django фреймворк аутентификации располагается в пакете `django.contrib.auth` и содержит модель `User` (Пользователь). Модель `User` будет применяться из указанного фреймворка аутентификации, чтобы создавать взаимосвязи между пользователями и постами.

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

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

class Post(models.Model):

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

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
author = models.ForeignKey(User,
on_delete=models.CASCADE,
related_name='blog_posts')

    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=2,
                             choices=Status.choices,
                             default=Status.DRAFT)

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

¹ Для справки: *отношение* (relation) в реляционной базе данных – это таблица, или сущность, состоящая из строк (кортежей) и различных атрибутов (столбцов или полей). *Взаимосвязь* (relationship) – это ассоциация между двумя сущностями, основанная на реляционной модели. См. <https://pediaa.com/what-is-the-difference-between-relation-and-relationship-in-dbms/>. – Прим. перев.

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

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

Параметр `on_delete` определяет поведение, которое следует применять при удалении объекта, на который есть ссылка. Это поведение не относится конкретно к Django; оно является стандартным для SQL. Использование ключевого слова `CASCADE` указывает на то, что при удалении пользователя, на которого есть ссылка, база данных также удалит все связанные с ним посты в блоге. Со всеми возможными опциями можно ознакомиться по адресу https://docs.djangoproject.com/en/4.1/ref/models/fields/#django.db.models.ForeignKey.on_delete.

Мы используем `related_name`, чтобы указывать имя обратной связи, от `User` к `Post`. Такой подход позволит легко обращаться к связанным объектам из объекта `User`, используя обозначение `user.blog_posts`. Подробнее об этом мы узнаем позже.

Django содержит разные типы полей, которые можно использовать для определения своих моделей. Все типы полей находятся на странице <https://docs.djangoproject.com/en/4.1/ref/models/fields/>.

Теперь модель `Post` завершена, и сейчас можно синхронизировать ее с базой данных. Но перед этим нужно активировать приложение `blog` в проекте Django.

Создание и применение миграций

Теперь, когда есть модель постов блога, необходимо создать соответствующую таблицу базы данных. Django идет в комплекте с системой миграции, которая отслеживает внесенные в модели изменения и позволяет их распространять по базе данных.

Команда `migrate` применяет миграции ко всем приложениям, перечисленным в `INSTALLED_APPS`. Она синхронизирует базу данных с текущими моделями и существующими миграциями.

Прежде всего необходимо создать первоначальную миграцию модели `Post`.

Выполните следующую ниже команду в командной оболочке из корневого каталога своего проекта:

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

Вы должны получить результат, аналогичный приведенному ниже:

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

Внутри каталога миграций приложения `blog` Django только что создал файл `0001_initial.py`. Эта миграция содержит инструкции SQL по созданию таблицы базы данных для модели `Post` и определения индекса базы данных для поля `publish`.

Можно взглянуть на содержимое файла, чтобы увидеть, как определяется миграция. Миграция задает зависимости от других миграций и операций, которые необходимо выполнить в базе данных, чтобы синхронизировать ее с изменениями модели.

Давайте взглянем на исходный код SQL, который Django исполнит в базе данных, чтобы создать таблицы вашей модели. Команда `sqlmigrate` принимает имена миграций и возвращает их SQL без его исполнения.

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

```
python manage.py sqlmigrate blog 0001
Результат должен выглядеть вот так:
BEGIN;
-- 
-- Create model Post
-- 

CREATE TABLE "blog_post" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(250) NOT NULL,
    "slug" varchar(250) NOT NULL,
    "body" text NOT NULL,
    "publish" datetime NOT NULL,
    "created" datetime NOT NULL,
    "updated" datetime NOT NULL,
    "status" varchar(10) NOT NULL,
    "author_id" integer NOT NULL REFERENCES "auth_user" ("id") DEFERRABLE
INITIALLY DEFERRED);
-- 
-- Create blog_post_publish_bb7600_idx on field(s) -publish of model post
-- 

CREATE INDEX "blog_post_publish_bb7600_idx" ON "blog_post" ("publish" DESC);
CREATE INDEX "blog_post_slug_b95473f2" ON "blog_post" ("slug");
CREATE INDEX "blog_post_author_id_dd7a8485" ON "blog_post" ("author_id");
COMMIT;
```

Точный результат зависит от используемой вами базы данных. Приведенный выше результат сгенерирован для SQLite. Из полученного результата видно,

что Django генерирует имена таблиц, комбинируя имя приложения с именем модели, обозначенной в нижнем регистре (`blog_post`). Кроме того, существует возможность указывать своей модели имя конкретно-прикладной базы данных. Это делается в Meta-классе модели при помощи атрибута `db_table`.

Django создает автоинкрементный столбец `id`, используемый в каждой модели в качестве первичного ключа, указав `primary_key=True` в одном из полей модели, но это поведение можно тоже переопределить. Столбец `id` состоит из автоматически увеличивающегося целого числа. Этот столбец соответствует полю `id`, которое добавляется в модель автоматически.

Создаются следующие три индекса базы данных:

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

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

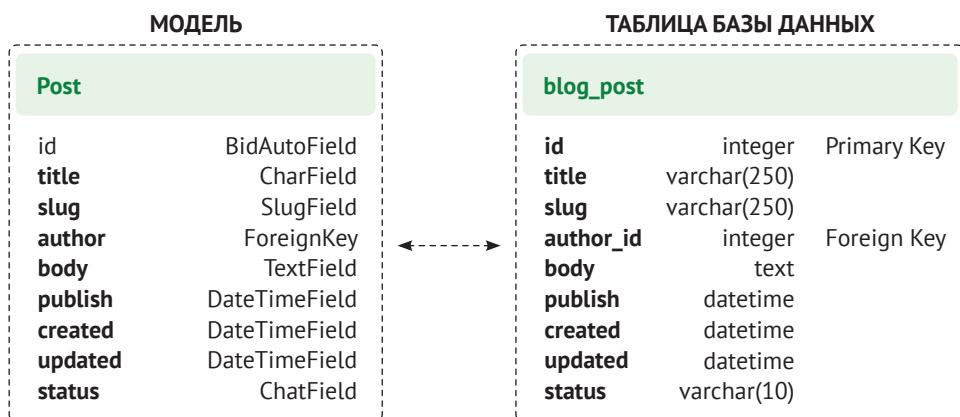


Рис. 1.5. Полное соответствие модели `Post` и таблицы базы данных

На рис. 1.5 показано, как поля модели соответствуют столбцам таблицы базы данных.

Давайте синхронизируем базу данных с новой моделью.

Примените следующую ниже команду в командной оболочке, чтобы воспользоваться существующими миграциями:

```
python manage.py migrate
```

Вы получите результат, который заканчивается следующей ниже строкой:

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

Мы только что применили миграции приложений, перечисленных в `INSTALLED_APPS`, включая приложение `blog`.

После применения миграций база данных отражает текущее состояние моделей.

Если вы внесете в файл `models.py` любые правки, чтобы добавить, удалить либо изменить поля существующих моделей, либо добавите новые модели, то вам придется создать новые миграции, снова применив команду `migrations`. Каждая миграция дает Django возможность отслеживать изменения модели. Затем нужно применить миграцию командой `migrate`, чтобы синхронизировать базу данных с моделями.

Создание сайта администрирования для моделей

Теперь, когда модель `Post` синхронизирована с базой данных, можно создать простой сайт администрирования, чтобы управлять постами блога.

Django идет в комплекте со встроенным интерфейсом администрирования, который широко используется для редактирования контента. Сайт Django формируется динамически путем чтения метаданных моделей и предоставления готового к работе интерфейса для редактирования контента. Его можно использовать прямо «из коробки», сконфигурировав его так, чтобы ваши модели отображались в нем в том виде, в котором вы хотите.

Приложение `django.contrib.admin` уже вставлено в настроочный параметр `INSTALLED_APPS`, поэтому добавлять его нет необходимости.

Создание суперпользователя

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

```
python manage.py createsuperuser
```

Вы увидите следующий ниже результат. Введите желаемое пользовательское имя (`username`)¹, адрес электронной почты и пароль, как показано:

¹ Для справки: пользовательское имя (`username`), также именуемое именем входа в систему (`login name`, `sign-in name`) – это уникальная последовательность символов, используемая для идентификации пользователя и разрешения доступа к компьютерной системе, компьютерной сети или онлайновой учетной записи. Не следует путать с настоящим именем пользователя (`first name`, `user's name` или `the name of the user`). – Прим. перев.

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

И вы увидите такое сообщение об успехе:

```
Superuser created successfully.
```

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

Сайт администрирования

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

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



Рис. 1.6. Экран входа на сайт администрирования

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

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



Рис. 1.7. Индексная страница сайта администрирования

Добавление моделей на сайт администрирования

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

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

admin.site.register(Post)
```

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

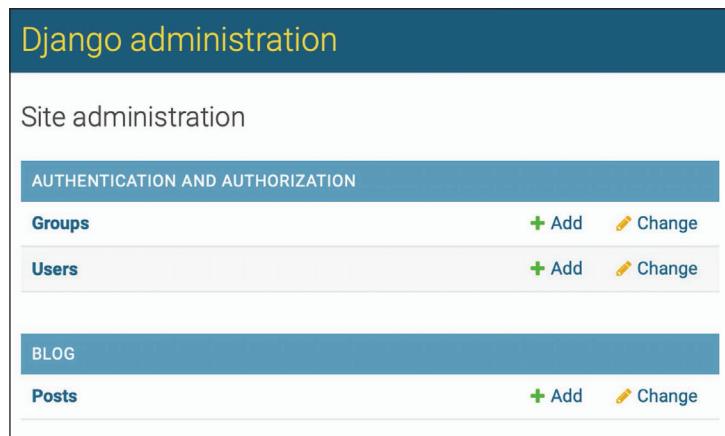


Рис. 1.8. Модель `Post` приложения `blog`, включенная в индексную страницу сайта администрирования

Все достаточно просто, не правда ли? При регистрации модели на сайте администрирования будет получен удобный интерфейс, генерированный путем интроспекции созданных разработчиком моделей, позволяющий простым способом выводить списки, редактировать, создавать и удалять объекты.

Кликните по ссылке **Add** (Добавить) напротив **Posts** (Посты), чтобы добавить новый пост. Вы увидите форму, которую Django генерировал для модели динамически, как показано на рис. 1.9.

The screenshot shows the 'Add post' form in the Django Admin interface. It includes fields for Title, Slug, Author (with a dropdown and '+' button), Body (a large text area), Publish (Date: 2022-01-01, Time: 23:39:19), and Status (Draft). At the bottom are buttons for Save and add another, Save and continue editing, and a large blue SAVE button.

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

Для каждого типа поля Django использует различные виджеты форм. Даже сложные поля, такие как поле `DateTimeField`, отображаются на странице с простым интерфейсом, таким как элемент выбора даты на языке JavaScript.

Заполните форму и кликните по кнопке **Save** (Сохранить). Вы будете перенаправлены на страницу списка постов с сообщением об успехе и только что созданным постом, как показано на рис. 1.10.

The screenshot shows the Django admin interface for the Post model. At the top, a green banner displays the message: "The post 'Who was Django Reinhardt?' was added successfully." Below this, the title "Select post to change" is displayed, followed by a "ADD POST +" button. A search bar and filter dropdown are present. The main list contains one item: "POST Who was Django Reinhardt?". A summary at the bottom indicates "1 post".

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

Адаптация внешнего вида моделей под конкретно-прикладную задачу

Теперь давайте посмотрим на способы адаптации сайта администрирования под конкретно-прикладную задачу.

Отредактируйте файл `admin.py` приложения `blog`, изменив его, как показано ниже. Новые строки выделены жирным шрифтом:

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

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

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

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

Давайте адаптируем модель `admin`, внеся в нее еще несколько опций.

Отредактируйте файл `admin.py` приложения `blog`, изменив его, как показано ниже. Новые строки выделены жирным шрифтом:

```

from django.contrib import admin
from .models import Post

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

```

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

TITLE	SLUG	AUTHOR	PUBLISH	STATUS
Who was Django Reinhardt?	who-was-django-reinhardt	admin	Jan 1, 2022, 11:39 p.m.	Draft

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

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

На странице появилась строка поиска. Это вызвано тем, что мы определили список полей, по которым можно выполнять поиск, используя атрибут `search_fields`. Чуть ниже строки поиска находятся навигационные ссылки для навигации по иерархии дат; это определено атрибутом `date_hierarchy`. Вы также видите, что по умолчанию посты упорядочены по столбцам **STATUS** (Статус) и **PUBLISH** (Опубликован). С помощью атрибута `ordering` были заданы критерии сортировки, которые будут использоваться по умолчанию.

Далее кликните по ссылке **ADD POST** (Добавить пост). Здесь вы тоже замените некоторые изменения. При вводе заголовка нового поста поле `slug` заполняется автоматически. Вы сообщили Django, что нужно предзаполнять поле `slug` данными, вводимыми в поле `title`, используя атрибут `prepopulated_fields`:

Add post	
Title:	Who was Django Reinhardt?
Slug:	who-was-django-reinhardt

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

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

🔍

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

Всего несколькими строками исходного кода мы адаптировали отображение модели на сайте администрирования. Адаптировать и расширять сайт администрирования можно огромным числом способов; подробнее об этом вы узнаете позже в этой книге.

Более подробная информация о сайте администрирования находится на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/admin/>.

Работа с наборами запросов QuerySet и менеджерами

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

Встроенный в Django объектно-реляционный преобразователь **ORM (object-relational mapper)** – это мощный API абстракции базы данных, кото-

рый позволяет легко создавать, извлекать, обновлять и удалять объекты¹. ORM-преобразователь дает возможность генерировать запросы на языке SQL, используя объектно-ориентированную парадигму Python. Его можно трактовать как способ взаимодействия с базой данных в Python'овском стиле вместо написания сырых SQL-запросов.

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

Напомним, что базу данных своего проекта можно определять в настроечном параметре DATABASES файла `settings.py` проекта. Django может работать с несколькими базами данных одновременно, при этом можно программировать маршрутизаторы баз данных, чтобы создавать конкретно-прикладные схемы маршрутизации данных.

После создания своих моделей данных Django предоставит бесплатный API для взаимодействия с ними. Справочный материал по моделям данных находится в официальной документации на странице <https://docs.djangoproject.com/en/4.1/ref/models/>.

Встроенный в Django ORM-преобразователь основан на итерируемых наборах запросов `QuerySet`. Итерируемый набор запросов `QuerySet` – это коллекция запросов к базе данных, предназначенных для извлечения объектов из базы данных. К наборам запросов можно применять фильтры, чтобы сужать результаты запросов на основе заданных параметров.

Создание объектов

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

```
python manage.py shell
```

Затем наберите следующие ниже строки:

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

¹ Объектно-реляционный преобразователь – это технология преобразования объектов в таблицы реляционной базы данных и наоборот. – Прим. перев.

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

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

Метод `get()` позволяет извлекать из базы данных только один объект. Обратите внимание, что этот метод ожидает результат, совпадающий с запросом. Если база данных не возвращает результатов, то указанный метод вызовет исключение `DoesNotExist`, а если база данных возвращает более одного результата, то он вызовет исключение `MultipleObjectsReturned`. Оба исключения являются атрибутами модельного класса, на котором выполняется запрос.

Затем мы создаем экземпляр класса `Post` с конкретно-прикладным заголовком, слагом и телом и задаем пользователя, которого мы ранее извлекли, в качестве автора поста:

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

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

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

```
post.save()
```

Приведенное выше действие за кулисами выполняет инструкцию SQL `INSERT`.

Сначала мы создали объект в памяти, а затем сохранили его в базе данных. Создавать объект и сохранять его в базе данных также можно одной операцией, используя метод `create()`. Это делается следующим образом:

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

Обновление объектов

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

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

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



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

Извлечение объектов

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

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

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

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

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

Применение метода `filter()`

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

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

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

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

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

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



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

Применение метода `exclude()`

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

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

Применение метода `order_by()`

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

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

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

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

Удаление объектов

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

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

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

Когда вычисляются наборы запросов QuerySet

Создание набора запросов QuerySet не требует каких-либо действий с базой данных до тех пор, пока он не будет вычислен. Наборы запросов обычно возвращают еще один невычисленный набор запросов. В наборе запросов можно конкатенировать столько фильтров, сколько потребуется, и база данных не будет затронута до тех пор, пока набор запросов не будет вычислен. При вычислении набора запросов он конвертируется в запрос на языке SQL к базе данных.

Наборы запросов QuerySet вычисляются только в следующих ниже случаях:

- при первом их прокручивании в цикле;
- при их нарезке, например `Post.objects.all()[:3]`;
- при их консервации в поток байтов или кешировании;
- при вызове на них функций `first()` или `len()`;
- при вызове на них функции `list()` в явной форме;
- при их проверке в операциях `bool()`, `or`, `and` или `if`.

Создание модельных менеджеров

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

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

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

Мы выберем второй метод, чтобы реализовать менеджер, который позволит извлекать посты, используя обозначение `Post.published.all()`.

Отредактируйте файл `models.py` приложения `blog`, добавив конкретно-прикладной менеджер, как показано ниже. Новые строки выделены жирным шрифтом:

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

```
# ...  
  
objects = models.Manager() # менеджер, применяемый по умолчанию  
published = PublishedManager() # конкретно-прикладной менеджер  
  
class Meta:  
    ordering = ['-publish']  
  
def __str__(self):  
    return self.title
```

Первый объявленный в модели менеджер становится менеджером, который используется по умолчанию. Для того чтобы указать другой такой менеджер, применяется `Meta`-атрибут `default_manager_name`. Если менеджер в модели не определен, то Django автоматически создает для нее стандартный менеджер `objects`. Если в своей модели вы объявляете какие-либо менеджеры, но также хотите сохранить менеджер `objects`, то вы должны добавить его в свою модель явным образом. В приведенном выше исходном коде мы добавили в модель `Post` стандартный менеджер `objects` и конкретно-прикладной менеджер `published`.

Метод `get_queryset()` менеджера возвращает набор запросов `QuerySet`, который будет выполнен. Мы переопределили этот метод, чтобы сформировать конкретно-прикладной набор запросов `QuerySet`, фильтрующий посты по их статусу и возвращающий поочередный набор запросов `QuerySet`, содержащий посты только со статусом `PUBLISHED`.

Теперь, когда мы определили для модели `Post` конкретно-прикладной менеджер, давайте его протестируем!

Следующей ниже командой снова запустите сервер разработки из командной оболочки:

```
python manage.py shell
```

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

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

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

Разработка представлений списка и детальной информации

Теперь, когда вы понимаете, как использовать ORM-преобразователь, вы готовы к разработке представлений приложения `blog`. Представление Django – это просто функция Python, которая получает веб-запрос и возвращает веб-ответ. Вся логика желаемого ответа находится внутри функции-представления.

Сначала в своем приложении нужно создать функции-представления, затем по каждому представлению сформировать шаблон URL-адреса и, наконец, создать шаблоны HTML, чтобы прорисовывать генерированные представлениями данные. Каждое представление будет прорисовывать шаблон, передавая ему переменные, и возвращать HTTP-ответ с прорисованным результатом.

Создание представлений списка постов и детальной информации о посте

Давайте начнем с создания представления списка постов на странице.

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

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

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

Это ваше самое первое представление Django. Представление `post_list` принимает объект `request` в качестве единственного параметра. Указанный параметр необходим для всех функций-представлений.

В данном представлении извлекаются все посты со статусом `PUBLISHED`, используя менеджер `published`, который мы создали ранее.

Наконец, мы используем функцию сокращенного доступа¹ `render()`, предоставляемую Django, чтобы прорисовать список постов заданным шаблоном. Указанная функция принимает объект `request`, путь к шаблону и контекстные

¹ В Django функция сокращенного доступа (`shortcut function`) – это вспомогательная функция, которая «хватывает» несколько уровней MVC. Другими словами, такая функция ради удобства привносит управляемое сопряжение. Кроме того, такими свойствами обладают и некоторые классы. – *Прим. перев.*

переменные, чтобы прорисовать данный шаблон. Она возвращает объект `HttpResponse` с прорисованным текстом (обычно исходным кодом HTML).

Функция сокращенного доступа `render()` учитывает контекст запроса, поэтому любая переменная, установленная процессорами контекста шаблона, доступна данному шаблону. Процессоры контекста шаблона – это просто **вызываемые объекты** (функции, методы и классы), которые назначают контекст переменным. Вы научитесь использовать процессоры контекста в главе 4 «Разработка социального веб-сайта».

Давайте создадим второе представление одиночного поста на странице. Добавьте следующую ниже функцию в файл `views.py`:

```
from django.http import HttpResponse

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

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

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

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

Применение функции сокращенного доступа `get_object_or_404()`

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

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

```
from django.shortcuts import render, get_object_or_404

# ...
```

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

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

Добавление шаблонов URL-адресов представлений

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

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

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

app_name = 'blog'

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

В приведенном выше исходном коде определяется именное пространство¹ приложения с помощью переменной `app_name`. Такой подход позволяет упорядочивать URL-адреса по приложениям и при обращении к ним использу-

¹ Также пространство имен. – Прим. перев.

зователь имя. С помощью функции `path()` определяются два разных шаблона. Первый шаблон URL-адреса не принимает никаких аргументов и соотносится с представлением `post_list`. Второй шаблон соотносится с представлением `post_detail` и принимает только один аргумент `id`, который совпадает с целым числом, заданным целым числом конвертора путей `int`.

Для захвата значений из URL-адреса используются угловые скобки. Любое значение, указанное в шаблоне URL-адреса как `<parameter>`, записывается в качестве строкового литерала. Для конкретного сопоставления и возврата целого числа используются конверторы путей, такие как `<int:year>`. Например, `<slug:post>` будет, в частности, совпадать со слагом (строковым литералом, который может содержать только буквы, цифры, подчеркивания или дефисы). Все предоставляемые веб-фреймворком Django конверторы путей можно посмотреть по адресу <https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters>.

Если функции `path()` и конверторов будет недостаточно, то вместо них можно использовать `re_path()`, чтобы определять сложные шаблоны URL-адресов с помощью регулярных выражений Python. Подробнее об определении шаблонов URL-адресов с помощью регулярных выражений можно узнать по адресу https://docs.djangoproject.com/en/4.1/ref/urls/#django.urls.re_path. Если вы раньше с регулярными выражениями не работали, то, возможно, вам сперва захочется взглянуть на руководство по регулярным выражениям на странице <https://docs.python.org/3/howto/regex.html>.



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

Далее необходимо вставить шаблоны URL-адресов приложения `blog` в главные шаблоны URL-адресов проекта.

Отредактируйте файл `urls.py`, расположенный внутри каталога `mysite` проекта, придав ему следующий вид. Новый исходный код выделен жирным шрифтом:

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

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

Новый шаблон URL-адреса, определенный с помощью функции `include`, ссылается на шаблоны URL-адресов, определенные в приложении `blog`, чтобы они были включены в рамки пути `blog/`. Указанные шаблоны вставляются в рамки именного пространства `blog`. Именные пространства должны

быть уникальными для всего проекта. Позже можно будет легко ссылаться на URL-запросы блога, используя именное пространство, за которым следует двоеточие, и имя URL-запроса, например `blog:post_list` и `blog:post_detail`. Подробнее о пространствах имен для URL-запросов можно узнать по адресу <https://docs.djangoproject.com/en/4.1/topics/http/urls/#url-namespaces>.

Создание шаблонов представлений

Вы создали представления и шаблоны URL-адресов для приложения `blog`. Шаблоны URL-адресов соотносят URL-адреса с представлениями, а те, в свою очередь, решают, какие данные будут возвращаться пользователю. Шаблоны определяют способ отображения данных; обычно они пишутся на HTML в сочетании с языком шаблонов Django. Более подробная информация о языке шаблонов Django находится на странице <https://docs.djangoproject.com/en/4.1/ref/templates/language/>.

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

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

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

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

Django обладает мощным языком шаблонов, который позволяет указывать внешний вид отображения данных. Он основан на *шаблонных тегах*, *шаблонных переменных* и *шаблонных фильтрах*¹:

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

¹ То есть теги, переменные и фильтры, относящиеся к конкретному шаблону. – Прим. перев.

Все встроенные шаблонные теги и фильтры можно посмотреть на странице <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/>.

Создание базового шаблона

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

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

Тег `{% load static %}` сообщает Django, что нужно загрузить статические шаблонные теги (`static`), предоставляемые приложением `django.contrib.staticfiles`, которое содержится в настроичном параметре `INSTALLED_APPS`. После их загрузки шаблонный тег `{% static %}` можно использовать во всем этом шаблоне. С помощью указанного шаблонного тега можно вставлять статические файлы, такие как файл `blog.css`, который находится в исходном коде данного примера в каталоге `static/` приложения `blog`. Скопируйте каталог `static/` из прилагаемого к этой главе исходного кода в то же место, где находится ваш проект, чтобы применить стили CSS к шаблонам. С содержимым каталога можно ознакомиться на странице <https://github.com/PacktPublishing/Django-4-by-Example/tree/master/Chapter01/mysite/blog/static>.

Вы видите, что присутствуют два тега `{% block %}`. Они сообщают Django, что нужно определить блок в отмеченной области. Шаблоны, которые наследуют от этого шаблона, могут заполнять блоки контентом. В приведенном выше исходном коде был определен блок под названием `title` и блок под названием `content`.

Создание шаблона списка постов

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

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

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

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

Шаблонный тег `{% extends %}` сообщает Django, что надо наследовать от шаблона `blog/base.html`. Затем заполняются блоки `title` и `content` базового шаблона. Посты прокручиваются в цикле, и их заголовок, дата, автор и тело отображаются на странице, включая ссылку в заголовке на подробный URL-адрес поста. URL-адрес формируется с использованием предоставляемого веб-фреймворком Django шаблонного тега `{% url %}`.

Этот шаблонный тег позволяет формировать URL-адреса динамически по их имени. Мы используем `blog:post_detail`, чтобы ссылаться на URL-адрес `post_detail` в именном пространстве `blog`. Мы передаем необходимый параметр `post.id`, чтобы сформировать URL-адрес для каждого поста.



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

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

Доступ к приложению

Откройте командную оболочку и исполните следующую ниже команду, чтобы запустить сервер разработки:

```
python manage.py runserver
```

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



Рис. 1.14. Страница представления списка постов

Создание шаблона детальной информации о посте

Далее отредактируйте файл `post/detail.html`:

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

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

{% block content %}
<h1>{{ post.title }}</h1>


Published {{ post.publish }} by {{ post.author }}


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

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



Рис. 1.15. Страница представления детальной информации о посте

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

Цикл запроса/ответа

Давайте рассмотрим цикл запроса/ответа Django, воспользуясь приложением, которое мы разработали. Следующая ниже схема показывает упрощенный пример того, как Django обрабатывает HTTP-запросы и генерирует HTTP-ответы (рис. 1.16).

Рассмотрим процесс запроса/ответа Django.

1. Веб-браузер запрашивает страницу по ее URL-адресу, например <https://domain.com/blog/33/>. Веб-сервер получает HTTP-запрос и передает его Django.
2. Django пробегает по всем шаблонам URL-адресов, определенным в конфигурации шаблонов URL-адресов. Он проверяет каждый шаблон на соответствие заданному пути URL-адреса в порядке их появления и останавливается на первом, который совпадает с запрошенным URL-адресом. В данном случае шаблон /blog/<id>/ соответствует пути /blog/33/.
3. Django импортирует представление совпавшего шаблона URL-адреса и исполняет его, передавая экземпляр класса `HttpRequest` и именованные либо позиционные аргументы. Представление использует модели, чтобы извлечь информацию из базы данных. С помощью встроенного в Django ORM-преобразователя наборы запросов `QuerySets` транслируются в SQL и исполняются в базе данных.
4. В представлении используется функция `render()`, которая прорисовывает шаблон HTML, передав в него объект `Post` в качестве контекстной переменной.
5. Прорисованный контент возвращается представлением в виде объекта `HttpResponse`, по умолчанию с типом контента `text/html`.

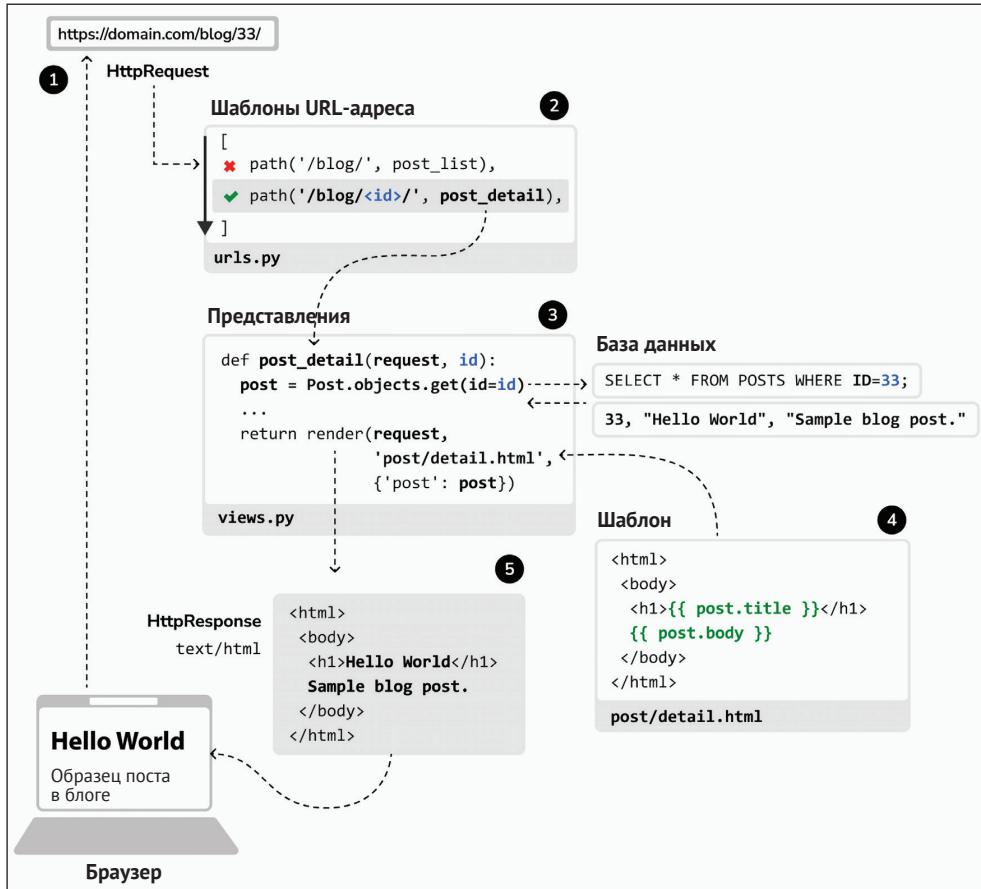


Рис. 1.16. Цикл запроса/ответа Django

Указанную схему всегда можно использовать в качестве базового ориентира в отношении того, как Django обрабатывает запросы. В целях простоты данная схема не содержит промежуточные программные компоненты Django. Вы будете использовать промежуточные программные компоненты в различных примерах этой книги, а о том, как создавать конкретно-прикладной промежуточный программный компонент, вы узнаете в главе 17 «Выход в прямой эфир».

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter01>.

- Библиотека Python `venv` для виртуальных сред: <https://docs.python.org/3/library/venv.html>.
- Опции установки Django: <https://docs.djangoproject.com/en/4.1/topics/install/>.
- Примечания к релизу Django 4.0: <https://docs.djangoproject.com/en/dev/releases/4.0/>.
- Примечания к релизу Django 4.1: <https://docs.djangoproject.com/en/4.1/releases/4.1/>.
- Философия дизайна Django: <https://docs.djangoproject.com/en/dev/misc/design-philosophies/>.
- Справочный материал по модельным полям Django: <https://docs.djangoproject.com/en/4.1/ref/models/fields/>.
- Справочный материал по модельным индексам Django: <https://docs.djangoproject.com/en/4.1/ref/models/indexes/>.
- Поддержка перечислений со стороны Python: <https://docs.python.org/3/library/enum.html>.
- Перечисляемые типы моделей Django: <https://docs.djangoproject.com/en/4.1/ref/models/fields/#enumeration-types>.
- Справочный материал по настроенным параметрам Django: <https://docs.djangoproject.com/en/4.1/ref/settings/>.
- Встроенный в Django сайт администрирования: <https://docs.djangoproject.com/en/4.1/ref/contrib/admin/>.
- Составление запросов с помощью встроенного в Django ORM-программиста: <https://docs.djangoproject.com/en/4.1/topics/db/queries/>.
- Встроенный в Django диспетчер URL-адресов: <https://docs.djangoproject.com/en/4.1/topics/http/urls/>.
- Встроенные в Django утилиты-рэзерверы URL-адресов: <https://docs.djangoproject.com/en/4.1/ref/urlresolvers/>.
- Язык шаблонов Django: <https://docs.djangoproject.com/en/4.1/ref/templates/language/>.
- Встроенные шаблонные теги и фильтры: <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/>.
- Статические файлы для исходного кода к этой главе: <https://github.com/PacktPublishing/Django-4-by-Example/tree/master/Chapter01/mysite/blog/static>.

Резюме

В этой главе вы изучили основы веб-фреймворка Django, создав простое приложение для ведения блога. Вы разработали модели данных и применили миграции к базе данных. Вы также создали представления, шаблоны и URL-адреса для своего блога.

В следующей главе вы научитесь создавать канонические URL-адреса для моделей и формировать дружественные для поисковой оптимизации URL-адреса постов блога. Вы также научитесь реализовывать постраничную раз-

бивку объектов и разрабатывать представления на основе классов, реализуете формы Django, чтобы ваши пользователи могли рекомендовать посты по электронной почте и их комментировать.

Присоединяйтесь к нам на Discord

Читайте эту книгу вместе с другими пользователями и автором.

Задавайте вопросы, предлагайте решения другим читателям, общайтесь с автором через сеансы «Спроси меня о чем угодно» и многое другое. Отсканируйте QR-код или пройдите по ссылке, чтобы присоединиться к книжному сообществу.

<https://packt.link/django>



2

Усовершенствование блога за счет продвинутых функциональностей

В предыдущей главе мы ознакомились с главными компонентами Django, разработав простое приложение для ведения блога. Мы создали простое приложение `blog`, используя представления, шаблоны и URL-адреса. В этой главе мы расширим функциональности приложения `blog` за счет функций, которые в настоящее время можно найти на многих блоговых платформах.

В данной главе будут рассмотрены следующие темы:

- использование канонических URL-адресов для моделей;
- создание дружественных для поисковой оптимизации URL-адресов постов;
- добавление постраничной разбивки в представление списка постов;
- разработка представлений на основе классов;
- отправка электронных писем с помощью Django;
- использование форм Django, позволяющих делиться постами по электронной почте;
- добавление комментариев к постам с использованием форм из моделей.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter02>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Использование канонических URL-адресов для моделей

На веб-сайте могут быть разные страницы, отображающие один и тот же контент. В нашем приложении изначальная часть контента по каждому посту отображается как на странице списка постов, так и на странице детальной информации о посте. Канонический URL-адрес – это предпочтительный URL-адрес ресурса. Его можно представить как URL-адрес наиболее репрезентативной страницы с конкретным контентом. На сайте могут быть разные страницы, которые показывают посты, но есть один URL-адрес, который используется в качестве главного URL-адреса поста. Канонические URL-адреса позволяют указывать URL-адрес мастер-копии страницы. Django дает возможность в своих собственных моделях реализовывать метод `get_absolute_url()`, который возвращает канонический URL-адрес объекта.

Мы будем использовать URL-адрес `post_detail`, определенный в шаблонах URL-адресов приложения, чтобы формировать канонический URL-адрес для объектов `Post`. Django предоставляет различные функции-резольверы `URL-filters`¹, которые позволяют формировать URL-адреса динамически, используя их имя и любые требуемые параметры. Мы будем использовать функцию-утилиту `reverse()`² модуля `django.urls`.

Отредактируйте файл `models.py` приложения `blog`, импортировав функцию `reverse()` и добавив метод `get_absolute_url()` в модель `Post`, как показано ниже. Новый исходный код выделен жирным шрифтом:

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

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

class Post(models.Model):

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

¹ Резольвер URL-адресов – это программная утилита или функция, которая конвертирует логический адрес или метаданные в физический URL-адрес целевых данных. – *Прим. перев.*

² URL-адрес, или URL-указатель (от англ. Uniform Resource Locator, аббр. URL, т. е. Унифицированный указатель ресурса), указывает на ресурс в сети. Функция `reverse` в Django выполняет обратное действие и используется для отыскания URL-адреса / URL-указателя заданного ресурса. – *Прим. перев.*

```
PUBLISHED = 'PB', 'Published'

title = models.CharField(max_length=250)
slug = models.SlugField(max_length=250)
author = models.ForeignKey(User,
                           on_delete=models.CASCADE,
                           related_name='blog_posts')
body = models.TextField()
publish = models.DateTimeField(default=timezone.now)
created = models.DateTimeField(auto_now_add=True)
updated = models.DateTimeField(auto_now=True)
status = models.CharField(max_length=2,
                          choices=Status.choices,
                          default=Status.DRAFT)

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

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse('blog:post_detail',
                      args=[self.id])
```

Функция `reverse()` будет формировать URL-адрес динамически, применяя имя URL-адреса, определенное в шаблонах URL-адресов. Мы использовали именное пространство `blog`, за которым следуют двоеточие и URL-адрес `post_detail`. Напомним, что именное пространство `blog` определяется в главном файле `urls.py` проекта при вставке шаблонов URL-адресов из `blog.urls`. URL-адрес `post_detail` определен в файле `urls.py` приложения `blog`. Результирующий строковый литерал, `blog:post_detail`, можно использовать глобально в проекте, чтобы ссылаться на URL-адрес детальной информации о посте. Этот URL-адрес имеет обязательный параметр – `id` извлекаемого поста блога. Идентификатор `id` объекта `Post` был включен в качестве позиционного аргумента, используя параметр `args=[self.id]`.

Подробнее о функциях-резольверах URL-адресов можно узнать на странице <https://docs.djangoproject.com/en/4.1/ref/urlresolvers/>.

Давайте заменим URL-адреса детальной информации о посте в шаблонах новым методом `get_absolute_url()`.

Отредактируйте файл `blog/post/list.html`, заменив строку

```
<a href="{% url 'blog:post_detail' post.id %}">
```

строкой

```
<a href="{{ post.get_absolute_url }}">
```

Теперь файл `blog/post/list.html` должен выглядеть 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|truncatetwords:30|linebreaks }}
    {% endfor %}
{% endblock %}
```

Откройте оболочку и исполните следующую ниже команду, чтобы запустить сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере. Ссылки на одиночные посты блога по-прежнему должны работать. Теперь Django формирует их, используя метод `get_absolute_url()` модели Post.

Создание дружественных для поисковой оптимизации URL-адресов постов

Канонический URL-адрес представления детальной информации о посте блога в настоящее время выглядит как `/blog/1/`. Мы изменим шаблон URL-адреса, чтобы формировать дружественные для поисковой оптимизации URL-адреса постов. В целях формирования URL-адресов одиночных постов

мы будем использовать дату публикации `publish` и значения `slug`. Присоединив даты, мы приведем URL-адрес детальной информации о посте к следующему виду: `/blog/2022/1/1/who-was-django-reinhardt/`. Мы предоставим поисковым механизмам дружественные для индексации URL-адреса, содержащие как заголовок, так и дату поста.

Для того чтобы получить одиночные посты с комбинацией даты публикации и слага, необходимо обеспечить, чтобы ни одну запись невозможно было сохранить в базе данных с тем же значением поля `slug` и поля `publish`, что и у существующего поста. Мы предотвратим хранение в модели `Post` дублирующихся записей, определив, что слаги являются уникальными для даты публикации поста.

Отредактируйте файл `models.py`, добавив следующий ниже параметр `unique_for_date` в поле `slug` модели `Post`:

```
class Post(models.Model):
    # ...
    slug = models.SlugField(max_length=250,
                           unique_for_date='publish')
    # ...
```

Теперь при использовании параметра `unique_for_date` поле `slug` должно быть уникальным для даты, сохраненной в поле `publish`. Обратите внимание, что поле `publish` является экземпляром класса `DateTimeField`, но проверка на уникальность значений будет выполняться только по дате (не по времени). Django будет предотвращать сохранение нового поста с тем же именем, что и у существующего поста на заданную дату публикации. В результате мы обеспечили уникальность слагов для даты публикации, поэтому теперь можно извлекать одиночные посты по полям `publish` и `slug`.

Мы изменили модели, поэтому давайте создадим миграции. Обратите внимание, что параметр `unique_for_date` не соблюдается на уровне базы данных, поэтому миграция базы данных не требуется. Между тем миграции в Django используются для отслеживания всех изменений модели. Мы создадим миграцию только для того, чтобы привести миграции в соответствие с текущим состоянием модели.

Выполните следующую ниже команду в командной оболочке:

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

Вы должны получить следующий ниже результат:

```
Migrations for 'blog':
  blog/migrations/0002_alter_post_slug.py
    - Alter field slug on post
```

Django только что создал файл `0002_alter_post_slug.py` внутри каталога `migrations` приложения `blog`.

Выполните следующую ниже команду в командной оболочке, чтобы применить существующие миграции:

```
python manage.py migrate
```

Вы получите результат, который заканчивается такой строкой:

```
Applying blog.0002_alter_post_slug... OK
```

Django будет считать, что все миграции были применены и модели синхронизированы. В базе данных не будет выполнено никаких действий, поскольку параметр `unique_for_date` не применяется на уровне базы данных.

Видоизменение шаблонов URL-адресов

Давайте видоизменим шаблоны URL-адресов, чтобы использовать дату публикации и слаг для URL-адреса детальной информации о посте.

Отредактируйте файл `urls.py` приложения `blog`, заменив строку

```
path('<int:id>', views.post_detail, name='post_detail'),
```

строками

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

Теперь файл `urls.py` должен выглядеть следующим образом:

```
from django.urls import path  
from . import views  
  
app_name = 'blog'  
  
urlpatterns = [  
    # представления поста  
    path('', views.post_list, name='post_list'),  
    path('<int:year>/<int:month>/<int:day>/<slug:post>/' ,  
         views.post_detail,
```

```
    name='post_detail'),  
]
```

Шаблон URL-адреса представления `post_detail` принимает следующие ниже аргументы:

- `year`: требуется целое число;
- `month`: требуется целое число;
- `day`: требуется целое число;
- `post`: требуется слаг (строка, содержащая только буквы, цифры, знаки подчеркивания или дефисы).

Конвертор пути `int` используется для параметров `year`, `month` и `day`, тогда как конвертор пути `slug` применяется для параметра `post`. В предыдущей главе вы узнали о конверторах путей. Все предоставляемые фреймворком Django конверторы путей можно посмотреть на странице <https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters>.

Видоизменение представлений

Теперь необходимо видоизменить параметры представления `post_detail`, чтобы они соответствовали новым параметрам URL-адреса, и использовать их для извлечения соответствующего объекта `Post`.

Откройте файл `views.py` и отредактируйте представление `post_detail`, как показано ниже:

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

Мы видоизменили представление `post_detail`, чтобы использовать аргументы `year`, `month`, `day` и `post` и извлекать опубликованный пост с заданным слагом и датой публикации. Ранее, добавив в поле `slug` значение параметра `unique_for_date='publish'` модели `Post`, мы обеспечили, чтобы был только один пост со слагом на заданную дату. Таким образом, используя дату и слаг, можно извлекать одиночные посты.

Видоизменение канонического URL-адреса постов

Также необходимо видоизменить параметры канонического URL-адреса для постов блога, чтобы они сочетались с новыми параметрами URL-адреса.

Откройте файл `models.py` приложения `blog` и отредактируйте метод `get_absolute_url()`, как показано ниже:

```
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])
```

Запустите сервер разработки, набрав следующую ниже команду в командной оболочке:

```
python manage.py runserver
```

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



Рис. 2.1. Страница представления детальной информации о посте

Взгляните на URL-адрес – он должен выглядеть как `/blog/2022/1/1/who-was-django-reinhardt/`. Вы разработали дружественные для поисковой оптимизации URL-адреса постов блога.

Добавление постраничной разбивки

Когда вы начнете добавлять контент в свой блог, вы сможете легко хранить десятки и даже сотни постов в своей базе данных. Возможно, вы захотите разделить список постов на несколько страниц, не отображая все записи на одной странице, и вставить навигационные ссылки на разные страницы. Эта функциональность называется постраничной разбивкой, и ее можно найти почти в каждом веб-приложении, которое показывает длинные списки элементов.

Например, Google использует постраничную разбивку с целью распределения результатов поиска по нескольким страницам. На рис. 2.2 показаны постранично разбитые ссылки Google на страницы результатов поиска:

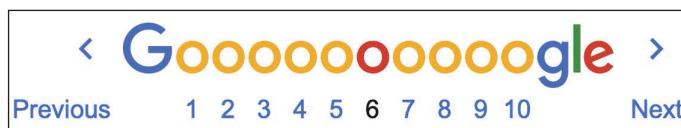


Рис. 2.2. Постранично разбитые ссылки Google
на страницы результатов поиска

В Django есть встроенный класс постраничной разбивки, который позволяет легко управлять постранично разбитыми данными, при этом имеется возможность определять число объектов, которое необходимое возвращать в расчете на страницу, и извлекать записи, соответствующие запрошенной пользователем странице.

Добавление постраничной разбивки в представление списка постов

Отредактируйте файл `views.py` приложения `blog`, импортировав класс Django `Paginator` и видоизменив представление `post_list`, как показано ниже:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator

def post_list(request):
    post_list = Post.published.all()
    # Постраничная разбивка с 3 постами на страницу
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    posts = paginator.page(page_number)
```

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

Давайте рассмотрим новый исходный код, который был добавлен в представление.

1. Мы создаем экземпляр класса `Paginator` с числом объектов, возвращаемых в расчете на страницу. Мы будем отображать по три поста на страницу.
2. Мы извлекаем HTTP GET-параметр `page` и сохраняем его в переменной `page_number`. Этот параметр содержит запрошенный номер страницы. Если параметра `page` нет в GET-параметрах запроса, то мы используем стандартное значение 1, чтобы загрузить первую страницу результатов.
3. Мы получаем объекты для желаемой страницы, вызывая метод `page()` класса `Paginator`. Этот метод возвращает объект `Page`, который хранится в переменной `posts`.
4. Мы передаем номер страницы и объект `posts` в шаблон.

Создание шаблона постраничной разбивки

Далее необходимо создать навигацию по страницам, чтобы пользователи имели возможность просматривать разные страницы. Мы создадим шаблон отображения постранично разбитых ссылок и сделаем его типовым, чтобы иметь возможность реиспользовать шаблон для постраничной разбивки любого объекта на веб-сайте.

Внутри каталога `templates/` создайте новый файл и назовите его `pagination.html`. Добавьте в файл следующий ниже исходный код HTML:

```
<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>
```

Это типовой шаблон постраничной разбивки. Предполагается, что данный шаблон будет иметь в контексте объект `Page`, чтобы прорисовывать преды-

дущую и следующую ссылки, а также отображать текущую страницу и общее число страниц результатов.

Давайте вернемся к шаблону `blog/post/list.html` и разместим шаблон `pagination.html` в нижней части блока `{% content %}`, как показано ниже:

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

Шаблонный тег `{% include %}` загружает данный шаблон и прорисовывает его с использованием текущего контекста шаблона. Ключевое слово `with` используется для того, чтобы передавать дополнительные контекстные переменные в шаблон. Для прорисовки в шаблоне постраничной разбивки используется переменная `page`, при этом объект `Page`, который мы передаем из представления в шаблон, называется `posts`. Мы используем выражение `with page=posts`, чтобы передавать переменную, ожидаемую шаблоном постраничной разбивки. Описанному методу можно следовать для применения шаблона постраничной разбивки для любого типа объекта.

Запустите сервер разработки, набрав следующую ниже команду в командной оболочке:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/blog/post/` в своем браузере и используйте сайт администрирования, чтобы создать в общей сложности четыре разных поста. Проверьте, чтобы у всех этих постов был установлен статус **Published**.

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере. Вы должны увидеть первые три поста в обратном хронологическом порядке, а затем навигационные ссылки в нижней части списка постов, как показано ниже:

My Blog

Notes on Duke Ellington
Published Jan. 3, 2022, 1:19 p.m. by admin
Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...

Who was Miles Davis?
Published Jan. 2, 2022, 1:18 p.m. by admin
Miles Davis was an American trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.

Who was Django Reinhardt?
Published Jan. 1, 2022, 11:59 p.m. by admin
Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...

Page 1 of 2. [Next](#)

Рис. 2.3. Страница списка постов с постраничной разбивкой ссылок внизу

Если кликнуть по **Next** (Далее), то можно увидеть последний пост. URL-адрес второй страницы содержит GET-параметр ?page=2. Указанный параметр используется представлением для загрузки запрошенной страницы результатов с использованием постраничного разбивщика.

My Blog

Another post
Published Jan. 1, 2022, 11:57 p.m. by admin
Post body.

[Previous](#) [Page 2 of 2.](#)

Рис. 2.4. Вторая страница результатов

Отлично! Ссылки на постраничную разбивку работают так, как и ожидались.

Обработка ошибок постраничной разбивки

Теперь, когда постраничная разбивка работает, в представление можно добавить обработку исключений, вызванных ошибками постраничной разбивки. Параметр `page`, используемый представлением для извлечения заданной страницы, потенциально может применяться с неправильными значениями, такими как несуществующие номера страниц или строковое значение, которое нельзя использовать в качестве номера страницы. Мы выполним соответствующую обработку ошибок для таких случаев.

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/?page=3` в своем браузере. Вы должны увидеть следующую ниже страницу с ошибкой:

EmptyPage at /blog/

That page contains no results

```
Request Method: GET
Request URL: http://127.0.0.1:8000/blog/?page=3
Django Version: 4.1
Exception Type: EmptyPage
Exception Value: That page contains no results
Exception Location: /Users/amele/Documents/env/dbe4/lib/python3.10/site-packages/django/core/paginator.py, line 57, in validate_number
Raised during: blog.views.post_list
Python Executable: /Users/amele/Documents/env/dbe4/bin/python
Python Version: 3.10.6
```

Рис. 2.5. Страница с ошибкой `EmptyPage`

При извлечении страницы 3 объект `Paginator` выдает исключение `EmptyPage`, поскольку она находится вне диапазона.

Подлежащих отображению результатов нет. Давайте обработаем эту ошибку в представлении.

Отредактируйте файл `views.py` приложения `blog`, добавив необходимую инструкцию импорта и видоизменив представление `post_list`, как показано ниже:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator, EmptyPage

def post_list(request):
    post_list = Post.published.all()
    # Постстраничная разбивка с 3 постами на страницу
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    try:
        posts = paginator.page(page_number)
    except EmptyPage:
        # Если page_number находится вне диапазона, то
        # выдать последнюю страницу
```

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

Мы добавили блок `try` и `except`, чтобы при извлечении страницы управлять исключением `EmptyPage`. Если запрошенная страница находится вне диапазона, то мы возвращаем последнюю страницу результатов. Мы получаем общее число страниц посредством `paginator.num_pages`. Общее число страниц совпадает с номером последней страницы.

Снова пройдите по URL-адресу `http://127.0.0.1:8000/blog/?page=3` в своем браузере. Теперь исключение управляется представлением, и последняя страница результатов возвращается, как показано ниже.



Рис. 2.6. Последняя страница результатов

Данное представление также должно обрабатывать случай, когда в параметре `page` передается нечто отличное от целого числа.

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/?page=asdf` в своем браузере. Вы должны увидеть следующую ниже страницу ошибки:



Рис. 2.7. Страница ошибки PageNotAnInteger

В этом случае при извлечении страницы asdf объект Paginator выдаст исключение `PageNotAnInteger`, поскольку номера страниц могут быть только целыми числами. Давайте обработаем эту ошибку в представлении.

Отредактируйте файл `views.py` приложения `blog`, добавив необходимую инструкцию импорта и видоизменив представление `post_list`, как показано ниже:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator, EmptyPage,\n    PageNotAnInteger
def post_list(request):
    post_list = Post.published.all()
    # Постраничная разбивка с 3 постами на страницу
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page')
    try:
        posts = paginator.page(page_number)
    except PageNotAnInteger:
        # Если page_number не целое число, то
        # выдать первую страницу
        posts = paginator.page(1)
    except EmptyPage:
        # Если page_number находится вне диапазона, то
        # выдать последнюю страницу результатов
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                  {'posts': posts})
```

Мы добавили новый блок `except`, чтобы при извлечении страницы управлять исключением `PageNotAnInteger`. Если запрошенная страница не является целым числом, то мы возвращаем первую страницу результатов.

Снова пройдите по URL-адресу `http://127.0.0.1:8000/blog/?page=asdf` в своем браузере. Теперь исключение обрабатывается представлением, и первая страница результатов возвращается, как показано ниже на рис. 2.8.

Теперь постраничная разбивка постов блога полностью реализована.

Подробнее о классе `Paginator` можно узнать на странице <https://docs.djangoproject.com/en/4.1/ref/paginator/>.

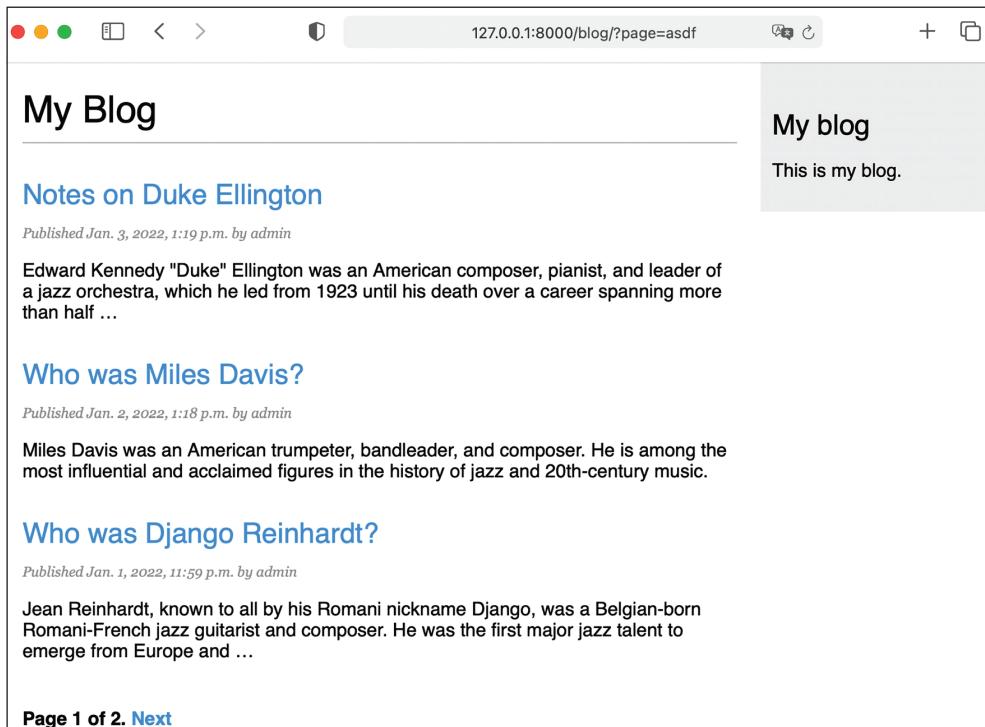


Рис. 2.8. Первая страница результатов

Разработка представлений на основе классов

Мы разработали приложение `blog`, используя представления на основе функций. Такие представления просты и мощны, но Django также позволяет разрабатывать представления с использованием классов.

Представления на основе классов являются альтернативным функциям способом реализации представлений как объектов Python. Поскольку представление – это функция, которая принимает веб-запрос и возвращает веб-ответ, то существует возможность определять представления как методы класса. Django предоставляет базовые классы-представления, которые можно использовать для реализации своих собственных представлений. Все они наследуют от класса `View`, который служит для диспетчеризации HTTP-методов и других распространенных функциональностей.

Зачем использовать представления на основе классов

Представления на основе классов обладают некоторыми преимуществами по сравнению с представлениями на основе функций, которые удобны для конкретных случаев использования. Представления на основе классов позволяют:

- организовывать исходный код, относящийся к HTTP-методам, таким как GET, POST или PUT, в отдельные методы, не используя ветвление по условию;
- использовать множественное наследование, чтобы создавать реиспользуемые классы-представления (также именуемые примесями, примечательными классами или миксинами).

Использование представления на основе класса для отображения списка постов

В целях понимания того, как писать представления на основе классов, мы создадим новое представление на основе класса, эквивалентное представлению `post_list`. Мы создадим класс, который будет наследовать от предлагаемого веб-фреймворком Django типового представления `ListView`. Представление `ListView` позволяет перечислять объекты любого типа.

Отредактируйте файл `views.py` приложения `blog`, добавив следующий ниже исходный код:

```
from django.views.generic import ListView

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

Представление `PostListView` похоже на разработанное ранее представление `post_list`. Мы имплементировали представление, основанное на классе, которое наследует от класса `ListView`, при этом определив его со следующими атрибутами:

- атрибут `queryset` используется для того, чтобы иметь конкретно-прикладной набор запросов `QuerySet`, не извлекая все объекты. Вместо определения атрибута `queryset` мы могли бы указать `model=Post`, и Django сформировал бы для нас типовой набор запросов `Post.objects.all()`;
- контекстная переменная `posts` используется для результатов запроса. Если не указано имя контекстного объекта `context_object_name`, то по умолчанию используется переменная `object_list`;
- в атрибуте `paginate_by` задается постраничная разбивка результатов с возвратом трех объектов на страницу;
- конкретно-прикладной шаблон используется для прорисовки страницы шаблоном `template_name`. Если шаблон не задан, то по умолчанию `ListView` будет использовать `blog/post_list.html`.

Теперь отредактируйте файл `urls.py` приложения `blog`, закомментировав предыдущий шаблон URL-адреса `post_list` и добавив новый шаблон URL-адреса, используя класс `PostListView`, как показано ниже:

```
urlpatterns = [
    # представления поста
    # 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 того чтобы постраничная разбивка продолжала работать, необходимо использовать правильный объект страницы, который передается в шаблон. Встроенное в Django типовое представление `ListView` передает запрошенную страницу в переменную с именем `page_obj`. В связи с этим необходимо соответствующим образом отредактировать шаблон `post/list.html`, чтобы вставить разбивщика, используя правильную переменную, как показано ниже:

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

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере и проверьте, чтобы ссылки с постраничной разбивкой работали должным образом. Поведение постранично разбитых ссылок должно быть таким же, как и в предыдущем представлении `post_list`.

Обработка исключений в этом случае немного отличается. Если попытаться загрузить страницу вне диапазона или передать нецелочисленное значение в параметре `page`, то представление вернет HTTP-ответ с кодом состояния, равным 404 (Страница не найдена), как показано ниже.

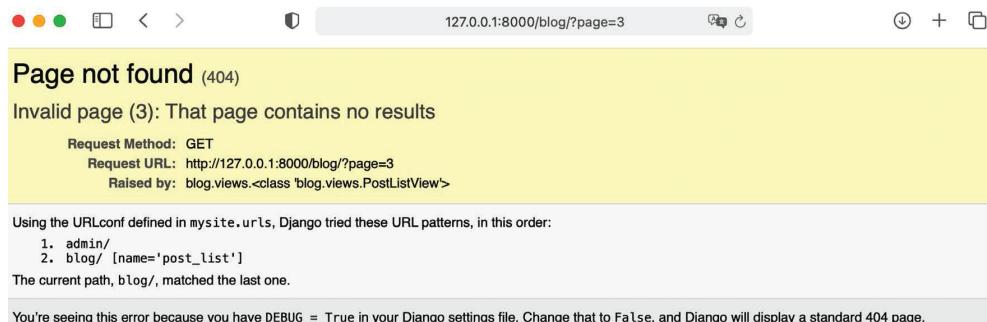


Рис. 2.9. HTTP-ответ с кодом состояния 404 (Страница не найдена)

Обработка исключений, которая возвращает HTTP-ответ с кодом состояния 404, предусмотрена типовым представлением `ListView`.

Это простой пример того, как писать представления на основе классов. Подробнее о представлениях на основе классов можно узнать в главе 13 «Создание системы управления контентом» и последующих главах.

Введение в представления на основе классов можно почитать на странице <https://docs.djangoproject.com/en/4.1/topics/class-based-views/intro/>.

Рекомендация постов по электронной почте

Теперь мы научимся создавать формы и отправлять электронные письма с помощью Django. Мы предоставим пользователям возможность делиться постами блога с другими, отправляя рекомендуемые посты по электронной почте.

Найдите минутку, чтобы подумать о том, как можно бы использовать представления, URL-адреса и шаблоны для создания этой функциональности, используя то, что вы узнали в предыдущей главе.

Для того чтобы дать пользователям возможность делиться постами по электронной почте, необходимо:

- создать форму, в которой пользователи должны заполнить свое имя, адрес электронной почты, адрес электронной почты получателя и опциональные комментарии;
- создать представление в файле `views.py`, которое обрабатывает опубликованные данные и отправляет электронное письмо;
- добавить шаблон URL-адреса нового представления в файл `urls.py` приложения `blog`;
- создать шаблон отображения формы.

Разработка форм с помощью Django

Давайте начнем с разработки формы, позволяющей делиться постами. Django имеет встроенный фреймворк форм, который позволяет легко создавать формы. Фреймворк форм упрощает определение полей формы, указывает их внешний вид на странице и способы валидации ими входных данных. Встроенный в Django фреймворк форм предлагает гибкий способ прорисовки форм в исходном коде HTML и оперирования данными.

Django поставляется с двумя базовыми классами для разработки форм:

- `Form`: позволяет компоновать стандартные формы путем определения полей и валидаций;
- `ModelForm`: позволяет компоновать формы, привязанные к экземплярам модели. Он предоставляет все функциональности базового класса `Form`, но поля формы можно объявлять явным образом или автоматически генерировать из полей модели. Форму можно использовать для создания либо редактирования экземпляров модели.

Сначала внутри каталога приложения `blog` создайте файл `forms.py` и добавьте в него следующий ниже исходный код:

```
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)
```

Мы определили первую форму Django. Форма `EmailPostForm` наследует от базового класса `Form`. Мы используем различные типы полей, чтобы выполнить валидацию данных в соответствии с ними.



Формы могут находиться в любом месте проекта Django. По традиции их помещают внутри файла `forms.py` в каждом приложении.

Форма содержит следующие ниже поля:

- `name`: экземпляр класса `CharField` с максимальной длиной 25 символов, который будет использоваться для имени человека, отправляющего пост;
- `email`: экземпляр класса `EmailField`. Здесь используется адрес электронной почты человека, отправившего рекомендуемый пост;
- `to`: экземпляр класса `EmailField`. Здесь используется адрес электронной почты получателя, который будет получать электронное письмо с рекомендуемым постом;
- `comments`: экземпляр класса `CharField`. Он используется для комментариев, которые будут вставляться в электронное письмо с рекомендуемым постом. Это поле сделано опциональным путем установки `required` равным значению `False`, при этом был задан конкретно-прикладной виджет прорисовки поля.

У каждого типа поля есть заранее заданный виджет, который определяет то, как поле прорисовывается в исходном коде HTML. Поле `name` является экземпляром класса `CharField`. Поле этого типа прорисовывается как HTML-элемент `<input type="text">`. Заранее заданный виджет можно переопределять посредством атрибута `widget`. В поле `comments` используется виджет `Textarea`, чтобы отображать его как HTML-элемент `<textarea>` вместо используемого по умолчанию элемента `<input>`.

Валидация полей также зависит от типа полей. Например, поля `email` и `to` являются полями типа `EmailField`. Для обоих полей требуется валидный адрес электронной почты; в противном случае валидация поля вызовет исключение `forms.ValidationError`, и форма не пройдет валидацию. При валидации полей формы также принимаются во внимание и другие параметры, такие как поле `name`, имеющее максимальную длину 25, или поле `comments`, являющееся опциональным.

Это лишь некоторые типы полей, которые Django предоставляет для форм. Список всех имеющихся типов полей находится на странице <https://docs.djangoproject.com/en/4.1/ref/forms/fields/>.

Работа с формами в представлениях

Мы определили форму для рекомендации постов по электронной почте. Теперь требуется представление, чтобы создавать экземпляр формы и работать с передачей формы на обработку.

Отредактируйте файл `views.py` приложения `blog`, добавив следующий ниже исходный код:

```

from .forms import EmailPostForm

def post_share(request, post_id):
    # Извлечь пост по идентификатору id
    post = get_object_or_404(Post,
                            id=post_id,
                            status=Post.Status.PUBLISHED)
    if request.method == 'POST':
        # Форма была передана на обработку
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Поля формы успешно прошли валидацию
            cd = form.cleaned_data
            # ... отправить электронное письмо
    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form': form})

```

Мы определили представление `post_share`, которое в качестве параметров принимает объект `request` и переменную `post_id`. Мы используем функцию сокращенного доступа `get_object_or_404()`, чтобы извлечь опубликованный пост по его `id`.

Одно и то же представление используется как для отображения изначальной формы на странице, так и для обработки представленных для валидации данных. HTTP-метод `request` позволяет различать случаи, когда форма передается на обработку. Запрос GET будет указывать на то, что пользователю должна быть отображена пустая форма, а запрос POST – на то, что форма передается на обработку. Булево выражение `request.method == 'POST'` используется для того, чтобы проводить различие между этими двумя сценариями.

Ниже описывается процесс отображения формы на странице и работы с передачей формы на обработку.

- Когда страница загружается в первый раз, представление получает запрос GET. В этом случае создается новый экземпляр класса `EmailPostForm`, который сохраняется в переменной `form`. Указанный экземпляр формы будет использоваться для отображения пустой формы в шаблоне:

```
form = EmailPostForm()
```

- Когда пользователь заполняет форму и передает ее методом POST на обработку, создается экземпляр формы с использованием переданных данных, содержащихся в `request.POST`:

```

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

```

3. После этого переданные данные валидируются методом `is_valid()` формы. Указанный метод проверяет допустимость введенных в форму данных и возвращает значение `True`, если все поля содержат валидные данные. Если какое-либо поле содержит невалидные данные, то `is_valid()` возвращает значение `False`. Список ошибок валидации можно получить посредством `form.errors`.
4. Если форма невалидна, то форма снова прорисовывается в шаблоне, включая переданные данные. Ошибки валидации будут отображены в шаблоне.
5. Если форма валидна, то валидированные данные извлекаются посредством `form.cleaned_data`. Указанный атрибут представляет собой словарь полей формы и их значений.



Если данные формы не проходят валидацию, то `cleaned_data` будет содержать только валидные поля.

Мы реализовали представление отображения формы на странице и передачи формы на обработку. Теперь мы научимся отправлять электронные письма с помощью Django и затем добавим эту функциональность в представление `post_share`.

Отправка электронных писем с помощью Django

Отправка электронных писем в Django очень проста. Для того чтобы отправлять электронные письма с помощью Django, необходимо иметь локальный **SMTP-сервер**¹ (работающий по простому протоколу передачи почты) либо обращаться к внешнему SMTP-серверу, например к своему поставщику услуг электронной почты.

Следующие ниже настроечные параметры позволяют определять конфигурацию SMTP, чтобы отправлять электронные письма с помощью Django:

- `EMAIL_HOST`: хост SMTP-сервера; по умолчанию используется `localhost`;
- `EMAIL_PORT`: SMTP-порт; по умолчанию равен 25;
- `EMAIL_HOST_USER`: пользовательское имя для SMTP-сервера;
- `EMAIL_HOST_PASSWORD`: пароль для SMTP-сервера;
- `EMAIL_USE_TLS`: следует ли использовать защищенное соединение транспортного слоя (**TLS**)²;
- `EMAIL_USE_SSL`: следует ли использовать неявное защищенное соединение TLS.

В этом примере мы будем использовать SMTP-сервер Google со стандартной учетной записью Gmail.

¹ Англ. Simple Mail Transfer Protocol. – Прим. перев.

² Англ. Transport Layer Security. – Прим. перев.

Если у вас есть учетная запись Gmail, то отредактируйте файл `settings.py` проекта, добавив следующий ниже исходный код:

```
# Конфигурация сервера электронной почты
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = ''
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Замените `your_account@gmail.com` своей реальной учетной записью Gmail. Если у вас нет учетной записи Gmail, то можете использовать конфигурацию SMTP-сервера своего поставщика услуг электронной почты.

Вместо Gmail также можно использовать профессиональный масштабируемый почтовый сервис, который позволяет отправлять электронные письма по протоколу SMTP, используя ваш собственный домен. Например, SendGrid (<https://sendgrid.com/>) или простой почтовый сервис Amazon (<https://aws.amazon.com/ses/>). Оба сервиса потребуют подтверждения домена и учетных записей электронной почты отправителя и предоставят учетные данные SMTP для отправки электронных писем. Приложения Django `django-sendgrid` и `django-ses` упрощают задачу добавления сервисов SendGrid или Amazon SES в свой проект. Инструкции по установке `django-sendgrid` находятся на странице <https://github.com/sklarsa/django-sendgrid-v5>, инструкции по установке `django-ses` расположены на странице <https://github.com/django-ses/django-ses>.

Если вы не можете использовать SMTP-сервер, то можно сообщить Django, что нужно писать электронные письма в консоль, добавив в файл `settings.py` следующий ниже настроечный параметр:

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

Используя этот параметр, Django будет выводить все электронные письма в оболочку, не отправляя их. Это бывает очень удобно при тестировании своего приложения без SMTP-сервера.

Завершая конфигурирование Gmail, необходимо ввести пароль для SMTP-сервера. Поскольку Google использует двухэтапный процесс верификации и дополнительные меры безопасности, вы не сможете использовать пароль своей учетной записи Google напрямую. Вместо этого Google позволяет создавать конкретно-прикладные пароли в вашем аккаунте. Пароль приложения – это 16-значный код доступа, который дает менее защищенному приложению или устройству разрешение на доступ к вашей учетной записи Google.

Пройдите по URL-адресу <https://myaccount.google.com/> в своем браузере. В левом меню выберите пункт **Security** (Безопасность). Вы увидите следующий ниже экран:



Рис. 2.10. Вход в Google для учетных записей Google

В разделе **Signing in to Google** (Вход в Google) кликните по **App passwords** (Пароли приложений). Если вы не видите пароли приложений, то, возможно, для вашей учетной записи не настроена двухэтапная верификация, ваша учетная запись является учетной записью организации, а не стандартной учетной записью Gmail, либо вы задействовали расширенную защиту Google. Проверьте, чтобы использовалась стандартная учетная запись Gmail и была активирована двухэтапная верификация своей учетной записи Google. Более подробная информация находится на странице <https://support.google.com/accounts/answer/185833>.

При нажатии на **App passwords** вы увидите следующий ниже экран:

A screenshot of the 'App passwords' generation form. At the top, there's a back arrow and the text 'App passwords'. Below that, a explanatory text: 'App passwords let you sign in to your Google Account from apps on devices that don't support 2-Step Verification. You'll only need to enter it once so you don't need to remember it.' followed by a 'Learn more' link. The main form area has two dropdown menus: 'Select app' (with options: Mail, Calendar, Contacts, YouTube, Other (Custom name), and 'Other (Custom name)' is selected) and 'Select device' (with a dropdown arrow). To the right of the device dropdown is a 'GENERATE' button.

Рис. 2.11. Форма для генерирования нового пароля приложения Google

В ниспадающем списке **Select app** (Выбрать приложение) выберите **Other** (Другое).

Затем введите имя **Blog** и кликните по кнопке **GENERATE** (Сгенерировать), как показано ниже:



Рис. 2.12. Форма для генерирования нового пароля приложения Google

Новый пароль будет сгенерирован и выведен на страницу, как показано ниже:

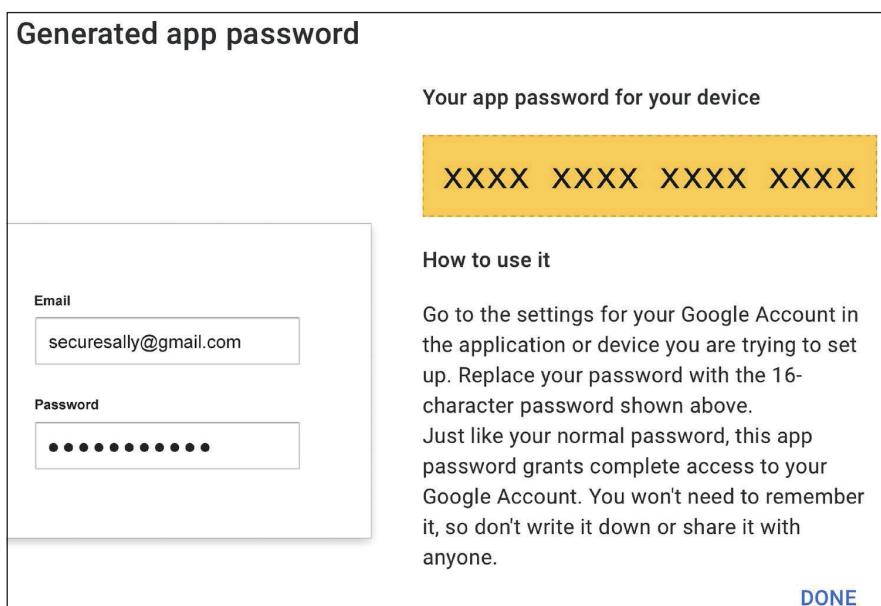


Рис. 2.13. Сгенерированный пароль приложения Google

Скопируйте сгенерированный пароль приложения.

Отредактируйте файл `settings.py` проекта, добавив пароль приложения в настроечный параметр `EMAIL_HOST_PASSWORD`, как показано ниже:

```
# Конфигурация сервера электронной почты
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = 'xxxxxxxxxxxxxxxxx'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Откройте оболочку Python, выполнив следующую ниже команду в командной строке системной оболочки:

```
python manage.py shell
```

Исполните следующий ниже исходный код в оболочке Python:

```
>>> 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 тему, сообщение, отправителя и список получателей в качестве требуемых аргументов. Установливая optionalный аргумент `fail_silently=False`, мы сообщаем ей, что если электронное письмо невозможно отправить, нужно вызывать исключение. Если результат, который вы видите, равен 1, значит, ваше электронное письмо было успешно отправлено.

Проверьте свой почтовый ящик. Вы должны были получить электронное письмо:

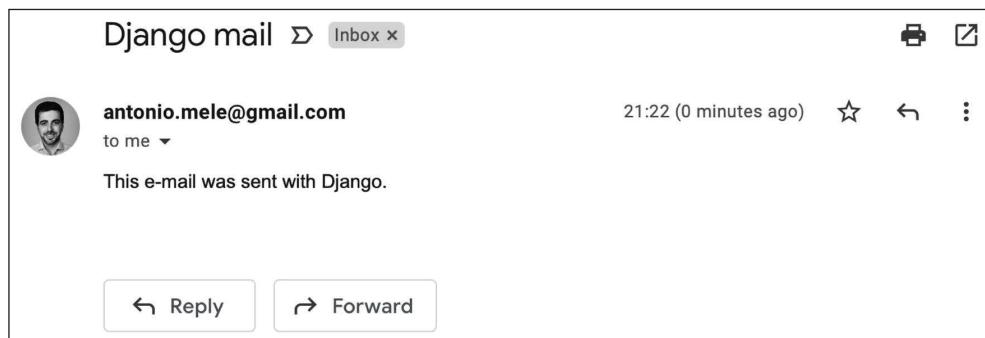


Рис. 2.14. Отправленное тестовое электронное письмо отображается в Gmail

Вы только что отправили свое первое электронное письмо с помощью Django! Более подробная информация об отправке электронных писем с помощью Django находится на странице <https://docs.djangoproject.com/en/4.1/topics/email/>.

Давайте добавим эту функциональность в представление post_share.

Отправка электронных писем в представлениях

Отредактируйте представление post_share в файле views.py приложения blog, как показано ниже:

```
from django.core.mail import send_mail

def post_share(request, post_id):
    # Извлечь пост по его идентификатору id
    post = get_object_or_404(Post,
                            id=post_id,
                            status=Post.Status.PUBLISHED)

    sent = False

    if request.method == 'POST':
        # Форма была передана на обработку
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Поля формы успешно прошли валидацию
            cd = form.cleaned_data
            post_url = request.build_absolute_uri(
                post.get_absolute_url())
            subject = f'{cd["name"]} recommends you read " \
                      f'{post.title}"'
            message = f'Read {post.title} at {post_url}\n\n" \
                      f'{cd["name"]}'s comments: {cd["comments"]}'
            send_mail(subject, message, 'your_account@gmail.com',
                      [cd['to']])
            sent = True
    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form': form,
                                                    'sent': sent})
```

Если вы используете SMTP-сервер, а не почтовый бэкенд¹ console.EmailBackend, то замените your_account@gmail.com своей реальной учетной записью электронной почты.

¹ Син. серверная часть веб-приложения. – Прим. перев.

В приведенном выше исходном коде мы объявили переменную `sent` с изначальным значением `False`. Мы задаем этой переменной значение `True` после отправки электронного письма. Позже мы будем использовать переменную `sent` в шаблоне отображения сообщения об успехе при успешной передаче формы.

Поскольку ссылка на пост должна вставляться в электронное письмо, мы получаем абсолютный путь к посту, используя его метод `get_absolute_url()`. Мы используем этот путь на входе в метод `request.build_absolute_uri()`, чтобы сформировать полный URL-адрес, включая HTTP-схему и хост-имя (`hostname`)¹.

Мы создаем тему и текст сообщения электронного письма, используя очищенные данные валидированной формы. Наконец, мы отправляем электронное письмо на адрес электронной почты, указанный в поле `to` (Кому) формы.

Теперь, когда представление `post_share` завершено, для него необходимо добавить новый шаблон URL-адреса.

Откройте файл `urls.py` приложения `blog` и добавьте шаблон URL-адреса `post_share`, как показано ниже:

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

app_name = 'blog'

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

Прорисовка форм в шаблонах

После того как была создана форма, запрограммировано представление и добавлен шаблон URL-адреса, не хватает только одного – шаблона представления.

Внутри каталога `blog/templates/blog/post/` создайте новый файл и назовите его `share.html`.

Добавьте следующий ниже исходный код в новый шаблон `share.html`:

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

¹ Син. сетевое имя, имя узла. – Прим. перев.

```
{% 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 method="post">
    {{ form.as_p }}
    {{ csrf_token }}
    <input type="submit" value="Send e-mail">
</form>
{% endif %}
{% endblock %}
```

Это шаблон, который используется для отображения формы, служащей для того, чтобы делиться постом по электронной почте, и для отображения успешного сообщения после отправки электронного письма. Различие между обоими случаями проводится с помощью тега `{% if sent %}`.

Для того чтобы отобразить форму, мы определили HTML-элемент `form`, указав, что форма должна быть передана методом POST:

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

Экземпляр формы вставлен с помощью тега `{{ form.as_p }}`. При этом веб-фреймворку Django сообщается, что нужно прорисовывать поля формы, используя абзацные HTML-элементы `<p>` с применением метода `as_p`. Кроме того, форму можно было бы прорисовывать в виде неупорядоченного списка с использованием метода `as_ul` или в виде HTML-таблицы с применением метода `as_table`. Еще одним вариантом является прорисовка каждого поля путем прокручивания полей формы в цикле, как в следующем ниже примере:

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

Здесь добавлен шаблонный тег `{% csrf_token %}`. Указанный тег вводит скрытое поле с автоматически генерированным токеном во избежание атак

по подделке межсайтовых запросов (**CSRF**)¹. Такие атаки заключаются в том, что вредоносный веб-сайт или программа выполняют нежелательные для пользователя действия на сайте. Более подробная информация о подделке межсайтовых запросов находится на странице <https://owasp.org/www-community/attacks/csrf>.

Шаблонный тег `{% csrf_token %}` генерирует скрытое поле, которое прописывается следующим образом:

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



По умолчанию Django проверяет наличие токена CSRF во всех запросах методом POST. Тег `csrf_token` следует вставлять во все формы, передаваемые на обработку методом POST.

Отредактируйте шаблон `blog/post/detail.html`, придав ему следующий вид:

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

Здесь была добавлена ссылка на URL-адрес `post_share`. URL-адрес формируется динамически с помощью предоставляемого веб-фреймворком Django шаблонного тега `{% url %}`. При этом используются именное пространство `blog` и URL-адрес `post_share`. Для того чтобы сформировать URL-адрес, `id` поста передается в качестве параметра.

Откройте приглашение командной оболочки и исполните следующую ниже команду, чтобы запустить сервер разработки:

```
python manage.py runserver
```

¹ Англ. Cross-Site Request Forgery. – Прим. перев.

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере и кликните по заголовку любого поста, чтобы просмотреть страницу детальной информации о посте.

Под телом поста вы должны увидеть ссылку, которую вы только что добавили, как показано на рис. 2.15.

Notes on Duke Ellington

Published Jan. 3, 2022, 1:19 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half a century.

[Share this post](#)

My blog
This is my blog.

Рис. 2.15. Страница детальной информации о посте, включая ссылку, чтобы поделиться постом

Кликните по **Share this post** (Поделиться этим постом), и вы должны увидеть страницу, включая форму, позволяющую делиться этим постом по электронной почте, как показано ниже:

Share "Notes on Duke Ellington" by e-mail

Name:

Email:

To:

Comments:

SEND E-MAIL

My blog
This is my blog.

Рис. 2.16. Страница, позволяющая делиться постом по электронной почте

Стили CSS формы включены в пример исходного кода и находятся в файле `static/css/blog.css`. При нажатии кнопки **SEND E-MAIL** (Отправить электронное письмо) форма передается на обработку и затем валидируется. Если

все поля содержат валидные данные, то вы получите сообщение об успехе, как показано ниже:

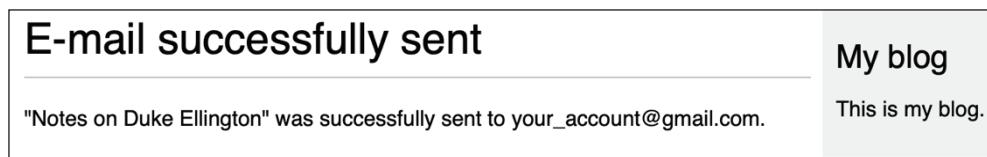


Рис. 2.17. Сообщение об успехе для поста, отправленного по электронной почте

Отправьте пост на свой собственный адрес электронной почты и проверьте свой почтовый ящик. Полученное вами электронное письмо должно выглядеть следующим образом:

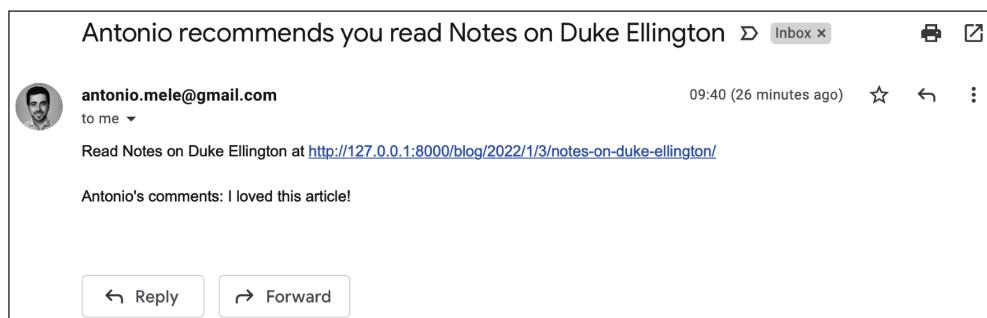


Рис. 2.18. Отправленное тестовое электронное письмо отображается в Gmail

Если передать форму на обработку с невалидными данными, то форма будет прорисована снова, включая все ошибки валидации (рис. 2.19).

Большинство современных браузеров не позволят передавать форму на обработку с пустыми или ошибочными полями. Это вызвано тем, что перед передачей формы на обработку браузер проверяет поля на основе их атрибутов. В этом случае форма не будет передана на обработку, и браузер отобразит сообщение об ошибке у неправильных полей. Для того чтобы протестировать валидацию формы Django с использованием современного браузера, можно пропустить валидацию формы браузером, добавив атрибут `novalidate` в элемент HTML `<form>`. Например, `<form method="post" novalidate>`. Этот атрибут можно добавлять, чтобы запрещать браузеру валидировать поля и тестировать свою собственную валидацию формы. После завершения тестирования удалите атрибут `novalidate`, чтобы вернуть валидацию формы браузером.

Теперь функциональность, позволяющая делиться постами по электронной почте, завершена. Более подробная информация о работе с формами находится на странице <https://docs.djangoproject.com/en/4.1/topics/forms/>.

Share "Notes on Duke Ellington" by e-mail

Name:
Antonio

• Enter a valid email address.

Email:
Invalid

• This field is required.

To:

Comments:

SEND E-MAIL

Рис. 2.19. Форма для отправки поста,
отображающая ошибки недопустимости данных

Создание системы комментариев

Мы продолжим работу над расширением приложения для ведения блога, разработав систему комментариев, которая позволит пользователям комментировать посты. Для того чтобы разработать такую систему, понадобится:

- модель комментария, чтобы хранить комментарии пользователей к постам;
- форма, которая позволяет пользователям передавать комментарии на обработку и управляет валидацией данных;
- представление, которое обрабатывает форму и сохраняет новый комментарий в базе данных;
- список комментариев и форма, чтобы добавлять новый комментарий, который может быть вставлен в шаблон детальной информации о посте.

Разработка модели комментария

Давайте начнем с разработки модели для хранения комментариев пользователей к постам.

Откройте файл `models.py` приложения `blog` и добавьте в него следующий ниже исходный 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']
        indexes = [
            models.Index(fields=['created']),
        ]

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

Это модель `Comment`. Поле `ForeignKey` было добавлено для того, чтобы связать каждый комментарий с одним постом. Указанная взаимосвязь многие-к-одному определена в модели `Comment`, потому что каждый комментарий будет делаться к одному посту, и каждый пост может содержать несколько комментариев.

Атрибут `related_name` позволяет назначать имя атрибуту, который используется для связи от ассоциированного объекта назад к нему. Пост комментарного объекта можно извлекать посредством `comment.post` и все комментарии, ассоциированные с объектом-постом, – посредством `post.comments.all()`. Если атрибут `related_name` не определен, то Django будет использовать имя модели в нижнем регистре, за которым следует `_set` (то есть `comment_set`), чтобы именовать взаимосвязь ассоциированного объекта с объектом модели, в которой эта взаимосвязь была определена.

Подробнее о взаимосвязях многие-к-одному можно узнать на странице https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_one/.

Мы определили булево поле `active`, чтобы управлять статусом комментариев. Данное поле позволит деактивировать неуместные комментарии вручную с помощью сайта администрирования. Мы используем параметр `default=True`, чтобы указать, что по умолчанию все комментарии активны.

Мы определили поле `created`, чтобы хранить дату и время создания комментария. Используя `auto_now_add`, дата будет сохраняться автоматически при создании объекта. В `Meta`-класс модели был добавлен атрибут `ordering = ['created']`, чтобы по умолчанию сортировать комментарии в хронологическом порядке и индексировать поля `created` в возрастающем порядке.

В результате этого будет повышена производительность операций поиска в базе данных и упорядочивания результатов с использованием поля `created`.

Разработанная модель `Comment` не синхронизирована с базой данных, и поэтому необходимо сгенерировать новую миграцию в базе данных, чтобы создать соответствующую таблицу базы данных.

Выполните следующую ниже команду из командной оболочки:

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

Вы должны увидеть следующий ниже результат:

```
Migrations for 'blog':  
  blog/migrations/0003_comment.py  
    - Create model Comment
```

Django сгенерировал файл `0003_comment.py` внутри каталога `migrations/` приложения `blog`. Теперь необходимо создать соответствующую схему базы данных и применить изменения к базе данных.

Выполните следующую ниже команду, чтобы применить существующие миграции:

```
python manage.py migrate
```

Вы получите результат, который содержит следующую ниже строку:

```
Applying blog.0003_comment... OK
```

Миграция была применена, и в базе данных была создана таблица `blog_comment`.

Добавление комментариев на сайт администрирования

Далее мы добавим новую модель на сайт администрирования, чтобы управлять комментариями через простой интерфейс.

Откройте файл `admin.py` приложения `blog`, импортируйте модель `Comment` и добавьте следующее:

```
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']
```

Откройте командную оболочку и исполните следующую ниже команду, чтобы запустить сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/> в своем браузере. Вы должны увидеть, что новая модель была вставлена в раздел **BLOG**, как показано на рис. 2.20.

The screenshot shows the Django admin interface for the 'BLOG' application. At the top, there are two main categories: 'Comments' and 'Posts'. Each category has a green '+' icon labeled 'Add' and a pencil icon labeled 'Change'. The 'Comments' category is currently selected, indicated by a blue background.

Рис. 2.20. Модели приложения для ведения блога на индексной странице сайта администрирования

Теперь модель зарегистрирована на сайте администрирования. В строке **Comments** (Комментарии) кликните по **Add** (Добавить). Вы увидите форму для добавления нового комментария:

The screenshot shows the 'Add comment' form. It consists of several input fields: 'Post' (with a dropdown menu and a '+' icon), 'Name' (text input), 'Email' (text input), 'Body' (large text area), and a checked checkbox for 'Active'. At the bottom, there are three buttons: 'Save and add another', 'Save and continue editing', and a dark blue 'SAVE' button.

Рис. 2.21. Форма для добавления нового комментария на сайте администрирования

Теперь появилась возможность управлять экземплярами комментариев с помощью сайта администрирования.

Создание форм из моделей

Далее необходимо скомпоновать форму, позволяющую пользователям комментировать посты блога. Напомним, что в Django есть два базовых класса, которые можно использовать для создания форм: `Form` и `ModelForm`. Мы использовали класс `Form`, чтобы предоставлять пользователям возможность делиться постами по электронной почте. Теперь мы будем использовать `ModelForm`, чтобы воспользоваться преимуществами существующей модели `Comment` и скомпоновать для нее форму динамически.

Отредактируйте файл `forms.py` приложения `blog`, добавив следующие ниже строки:

```
from .models import Comment

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

Для того чтобы создать форму из модели, надо в `Meta`-классе формы просто указать модель, для которой следует скомпоновать форму. Django проведет интроспекцию модели и динамически скомпонует соответствующую форму.

Каждому типу поля модели соответствует заранее заданный тип поля формы. Атрибуты полей модели учитываются при валидации формы. По умолчанию Django создает поле формы для каждого содержащегося в модели поля. Однако, используя атрибут `fields`, можно сообщать поля, которые следует включать в форму, либо, используя атрибут `exclude`, сообщать поля, которые следует исключать, задавая поля в явном виде. В форме `CommentForm` мы включили поля `name`, `email` и `body` в явном виде. Это единственные поля, которые будут включены в форму.

Более подробная информация о создании форм из моделей находится на странице <https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/>.

Оперирование формами `ModelForm` в представлениях

Для того чтобы делиться постами по электронной почте, мы использовали одно и то же представление, которое служило как для отображения формы, так и для управления ее передачей на обработку. Мы использовали HTTP-метод, чтобы проводить различие между обоими случаями, `GET`, чтобы ото-

бражать форму на странице, и POST, чтобы передавать ее на обработку. В этом случае мы добавим комментарную форму на страницу детальной информации о посте и разработаем отдельное представление, которое посвящено передаче формы на обработку. Новое обрабатывающее форму представление позволит пользователю возвращаться к представлению детальной информации о посте, после того как комментарий будет сохранен в базе данных.

Отредактируйте файл views.py приложения blog, добавив следующий ниже исходный код:

```
from django.shortcuts import render, get_object_or_404, redirect
from .models import Post, Comment
from django.core.paginator import Paginator, EmptyPage,\n                                         PageNotAnInteger
from django.views.generic import ListView
from .forms import EmailPostForm, CommentForm
from django.core.mail import send_mail
from django.views.decorators.http import require_POST

# ...

@require_POST
def post_comment(request, post_id):
    post = get_object_or_404(Post,
                           id=post_id,
                           status=Post.Status.PUBLISHED)
    comment = None
    # Комментарий был отправлен
    form = CommentForm(data=request.POST)
    if form.is_valid():
        # Создать объект класса Comment, не сохраняя его в базе данных
        comment = form.save(commit=False)
        # Назначить пост комментарию
        comment.post = post
        # Сохранить комментарий в базе данных
        comment.save()
    return render(request, 'blog/post/comment.html',
                  {'post': post,
                   'form': form,
                   'comment': comment})
```

Мы определили представление post_comment, которое принимает объект request и переменную post_id в качестве параметров. Мы будем использовать это представление, чтобы управлять передачей поста на обработку. Мы ожидаем, что форма будет передаваться с использованием HTTP-метода POST. Мы используем предоставляемый веб-фреймворком Django декоратор require_POST, чтобы разрешить запросы методом POST только для этого представления. Django позволяет ограничивать разрешенные для представлений

HTTP-методы. Если пытаться обращаться к представлению посредством любого другого HTTP-метода, то Django будет выдавать ошибку HTTP 405 (Метод не разрешен).

В этом представлении реализованы следующие ниже действия.

1. По `id` поста извлекается опубликованный пост, используя функцию сокращенного доступа `get_object_or_404()`.
2. Определяется переменная `comment` с изначальным значением `None`. Указанная переменная будет использоваться для хранения комментарного объекта при его создании.
3. Создается экземпляр формы, используя переданные на обработку POST-данные, и проводится их валидация методом `is_valid()`. Если форма невалидна, то шаблон прорисовывается с ошибками валидации.
4. Если форма валидна, то создается новый объект `Comment`, вызывая метод `save()` формы, и назначается переменной `new_comment`, как показано ниже:

```
comment = form.save(commit=False)
```

5. Метод `save()` создает экземпляр модели, к которой форма привязана, и сохраняет его в базе данных. Если вызывать его, используя `commit=False`, то экземпляр модели создается, но не сохраняется в базе данных. Такой подход позволяет видоизменять объект перед его окончательным сохранением.



Метод `save()` доступен для `ModelForm`, но не для экземпляров класса `Form`, поскольку они не привязаны ни к одной модели.

6. Пост назначается созданному комментарию:

```
comment.post = post
```

7. Новый комментарий создается в базе данных путем вызова его метода `save()`:

```
comment.save()
```

8. Прорисовывается шаблон `blog/post/comment.html`, передавая объекты `post`, `form` и `comment` в контекст шаблона. Этот шаблон еще не существует; мы создадим его позже.

Давайте создадим шаблон URL-адреса этого представления.

Отредактируйте файл `urls.py` приложения `blog`, добавив следующий ниже шаблон URL-адреса:

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

app_name = 'blog'

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

Мы реализовали представление, чтобы управлять передачей комментариев на обработку и соответствующими им URL-адресами. Давайте создадим необходимые шаблоны.

Создание шаблонов комментарной формы

Мы создадим шаблон комментарной формы, которая будет использоваться в двух местах:

- в шаблоне детальной информации о посте, ассоциированном с представлением `post_detail`, чтобы пользователи могли публиковать комментарии;
- в шаблоне комментария к посту, ассоциированном с представлением `post_comment`, чтобы отображать форму снова, если в форме есть какие-либо ошибки.

Мы создадим шаблон формы и будем использовать шаблонный тег `{% include %}`, чтобы вставлять его в два других шаблона.

Внутри каталога `templates/blog/post/` создайте новый каталог `includes/`. В этот каталог добавьте новый файл и назовите его `comment_form.html`.

Файловая структура должна выглядеть следующим образом:

```
templates/
  blog/
    post/
      includes/
        comment_form.html
        detail.html
```

```
list.html
share.html
```

Отредактируйте новый шаблон `blog/post/includes/comment_form.html`, добавив следующий ниже исходный код:

```
<h2>Add a new comment</h2>
<form action="{% url "blog:post_comment" post.id %}" method="post">
    {{ form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Add comment"></p>
</form>
```

В указанном шаблоне мы динамически формируем URL-адрес `action` HTML-элемента `<form>`, используя шаблонный тег `{% url %}`. Мы формируем URL-адрес представления `post_comment`, которое будет обрабатывать форму. Мы отображаем форму, прорисованную абзацами HTML, и вставляем тег `{% csrf_token %}`, чтобы защититься от CSRF, поскольку данная форма будет передаваться на обработку методом POST.

Внутри каталога `templates/blog/post/` создайте новый файл приложения `blog` и назовите его `comment.html`.

Теперь файловая структура должна выглядеть следующим образом:

```
templates/
blog/
post/
    includes/
        comment_form.html
    comment.html
    detail.html
    list.html
    share.html
```

Отредактируйте новый шаблон `blog/post/comment.html`, добавив следующий ниже исходный код:

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

{% block title %}Add a comment{% endblock %}

{% block content %}
    {% if comment %}
        <h2>Your comment has been added.</h2>
        <p><a href="{{ post.get_absolute_url }}>Back to the post</a></p>
    {% else %}
        {% include "blog/post/includes/comment_form.html" %}
```

```
{% endif %}  
{% endblock %}
```

Это шаблон представления комментариев к посту. В данном представлении мы ожидаем, что форма будет передаваться на обработку методом POST. Шаблон охватывает два разных сценария:

- если переданные данные формы валидны, то переменная `comment` будет содержать созданный объект `comment`, и на страницу будет выведено сообщение об успехе;
- если переданные данные формы невалидны, то переменной `comment` будет назначено значение `None`. В этом случае мы отобразим комментарийную форму. Для вставки созданного ранее шаблона `comment_form.html` используется шаблонный тег `{% include %}`.

Добавление комментариев в представление детальной информации о посте

Откройте файл `views.py` приложения `blog` и отредактируйте представление `post_detail`, как показано ниже:

```
def post_detail(request, year, month, day, post):  
    post = get_object_or_404(Post,  
                           status=Post.Status.PUBLISHED,  
                           slug=post,  
                           publish__year=year,  
                           publish__month=month,  
                           publish__day=day)  
  
    # Список активных комментариев к этому посту  
    comments = post.comments.filter(active=True)  
    # Форма для комментирования пользователями  
    form = CommentForm()  
    return render(request,  
                 'blog/post/detail.html',  
                 {'post': post,  
                  'comments': comments,  
                  'form': form})
```

Давайте рассмотрим исходный код, который мы добавили в представление `post_detail`:

- мы добавили набор запросов `QuerySet`, чтобы извлекать все активные комментарии к посту, как показано ниже:

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

- этот набор запросов сформирован с использованием объекта `post`. Вместо того чтобы формировать набор запросов для комментарий модели напрямую, мы используем объект `post`, чтобы извлекать связанные объекты `Comment`. Мы применяем менеджер `comments` для ранее определенных в модели `Comment` связанных с `Comment` объектов, используя атрибут `related_name` поля `ForeignKey` в модели `Post`;
- мы также создали экземпляры формы для комментария посредством инструкции `form = CommentForm()`.

Добавление комментариев в шаблон детальной информации о посте

Далее необходимо отредактировать шаблон `blog/post/detail.html`, чтобы реализовать следующее:

- показывать общее число комментариев к посту;
- показывать список комментариев;
- показывать форму для добавления пользователями новых комментариев.

Мы начнем с добавления общего числа комментариев к посту.

Отредактируйте шаблон `blog/post/detail.html`, внеся в него изменения, как показано ниже:

```
{% 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 }}
    <p>
        <a href="{% url "blog:post_share" post.id %}">
            Share this post
        </a>
    </p>
    {% with comments.count as total_comments %}
        <h2>
            {{ total_comments }} comment{{ total_comments|pluralize }}
        </h2>
    {% endwith %}
{% endblock %}
```

В указанном шаблоне мы используем Django ORM-преобразователь, применяя набор запросов `comments.count()`. Обратите внимание, что на языке шаблонов Django для вызова методов круглые скобки не используются. Тег `{% with %}` позволяет присваивать значение новой переменной, которая будет доступна в шаблоне до тех пор, пока не появится тег `{% endwith %}`.



Шаблонный тег `{% with %}` полезен тем, что он позволяет избегать многократного обращения к базе данных или к дорогостоящим методам.

Мы используем шаблонный фильтр `pluralize`, чтобы отображать суффикс множественного числа для слова `comment`, в зависимости от значения `total_comments`. Шаблонные фильтры на входе принимают значение переменной, к которой они применяются, и на выходе возвращают вычисленное значение. Подробнее о шаблонных фильтрах мы узнаем в главе 3 «Расширение приложения для ведения блога».

Шаблонный фильтр `pluralize` возвращает строковый литерал с буквой «`s`», если значение отличается от `1`. Приведенный выше текст будет прорисовываться как `0 comments`, `1 comment` или `N comments`, в зависимости от числа активных комментариев к посту.

Теперь давайте добавим список активных комментариев в шаблон детальной информации о посте.

Отредактируйте шаблон `blog/post/detail.html`, внеся в него следующие ниже изменения:

```
{% 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 }}
<p>
    <a href="{% url "blog:post_share" post.id %}">
        Share this post
    </a>
</p>
{% with comments.count as total_comments %}
<h2>
    {{ total_comments }} comment{{ total_comments|pluralize }}
</h2>
```

```

{%- endwith %}
{% 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.</p>
{% endfor %}
{% endblock %}

```

Мы добавили шаблонный тег `{% for %}`, чтобы прокручивать комментарии к посту в цикле. Если список комментариев пуст, то выводится сообщение, информирующее пользователей о том, что комментариев к этому посту нет. Комментарии прокручиваются в цикле посредством переменной `{{ forloop.counter }}`, которая обновляет счетчик цикла на каждой итерации. По каждому посту мы показываем имя пользователя, который его опубликовал, дату и текст комментария.

Наконец, давайте добавим форму комментария в шаблон.

Отредактируйте шаблон `blog/post/detail.html`, вставив шаблон комментарной формы, как показано ниже:

```

{% 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 }}
<p>
  <a href="{% url "blog:post_share" post.id %}">
    Share this post
  </a>
</p>
{% with comments.count as total_comments %}
<h2>
  {{ total_comments }} comment{{ total_comments|pluralize }}
</h2>

```

```
{% endwith %}  
{% 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.</p>  
  {% endfor %}  
  {% include "blog/post/includes/comment_form.html" %}  
  {% endblock %}
```

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере и кликните по заголовку поста, чтобы взглянуть на страницу подробного описания поста. Вы увидите что-то вроде рис. 2.22:

Notes on Duke Ellington

Published Jan. 3, 2022, 1:19 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half a century.

[Share this post](#)

0 comments

There are no comments yet.

Add a new comment

Name:

Email:

Body:

[ADD COMMENT](#)

My blog

This is my blog.

Рис. 2.22. Страница детальной информации о посте, содержащая форму для добавления комментария

Заполните форму валидными данными и кликните по **Add comment** (Добавить комментарий). Вы должны увидеть следующую ниже страницу:



Рис. 2.23. Страница успешного добавления комментария

Кликните по ссылке **Back to the post** (Вернуться к посту). Вы должны быть перенаправлены обратно на страницу детальной информации о посте и увидеть комментарий, который вы только что добавили, как показано ниже:

The screenshot shows a post titled "Notes on Duke Ellington". It includes a timestamp ("Published Jan. 3, 2022, 1:19 p.m. by admin"), a bio about Duke Ellington, a "Share this post" link, and a section for "1 comment". A single comment from "Antonio" is displayed: "I didn't know that!". Below the post, there's a form for "Add a new comment" with fields for Name, Email, and Body, and a blue "ADD COMMENT" button.

Рис. 2.24. Страница детальной информации о посте, включая комментарий

Добавьте еще один комментарий в этот пост. Комментарии должны располагаться под содержимым поста в хронологическом порядке, как показано ниже:

2 comments

Comment 1 by Antonio Jan. 3, 2022, 7:58 p.m.
I didn't know that!

Comment 2 by Bienvenida Jan. 3, 2022, 9:13 p.m.
I really like this article.

Рис. 2.25. Список комментариев на странице детальной информации о посте

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/blog/comment/> в своем браузере. Вы увидите страницу администрирования со списком созданных вами комментариев. Например, как на рис. 2.26.

Select comment to change

Search

Action: — Go 0 of 2 selected

<input type="checkbox"/>	NAME	EMAIL	POST	CREATED	ACTIVE
<input type="checkbox"/>	Antonio	test_account@gmail.com	Notes on Duke Ellington	Jan. 3, 2022, 7:58 p.m.	<input checked="" type="checkbox"/>
<input type="checkbox"/>	Bienvenida	test_account2@gmail.com	Notes on Duke Ellington	Jan. 3, 2022, 9:13 p.m.	<input checked="" type="checkbox"/>

2 comments

Рис. 2.26. Список комментариев на сайте администрирования

Кликните по заголовку одного из постов, чтобы его отредактировать. Снимите флажок **Active** (Активен), как показано ниже, и кликните по кнопке **Save** (Сохранить):

Change comment

Comment by Antonio on Notes on Duke Ellington

Post: Notes on Duke Ellington

Name: Antonio

Email: test_account@gmail.com

Body: I didn't know that!

Active

Delete **Save and add another** **Save and continue editing** **SAVE**

Рис. 2.27. Редактирование комментария на сайте администрирования

Вы будете перенаправлены на список комментариев. В столбце **Active** отобразится неактивный значок комментария, как показано на рис. 2.28:

The comment "Comment by Antonio on Notes on Duke Ellington" was changed successfully.

Select comment to change

Action: Go 0 of 2 selected

<input type="checkbox"/>	NAME	EMAIL	POST	CREATED	ACTIVE
<input type="checkbox"/>	Antonio	test_account@gmail.com	Notes on Duke Ellington	Jan. 3, 2022, 7:58 p.m.	
<input type="checkbox"/>	Bienvenida	test_account2@gmail.com	Notes on Duke Ellington	Jan. 3, 2022, 9:13 p.m.	

2 comments

Рис. 2.28. Активные/неактивные комментарии на сайте администрирования

Если вернуться к представлению детальной информации о посте, то можно заметить, что неактивный комментарий больше не отображается и не учитывается при подсчете общего числа активных комментариев к посту:

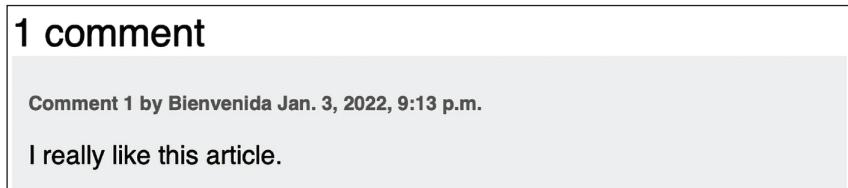


Рис. 2.29. Один активный комментарий, отображаемый на странице детальной информации о посте

Благодаря полю **Active** можно деактивировать неуместные комментарии и не показывать их в своих постах.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter02>.
- Функции-утилиты для URL-адресов: <https://docs.djangoproject.com/en/4.1/ref/urlresolvers/>.
- Конверторы путей URL-адресов: <https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters>.
- Встроенный в Django класс постраничной разбивки: <https://docs.djangoproject.com/en/4.1/ref/paginator/>.
- Введение в представления на основе классов: <https://docs.djangoproject.com/en/4.1/topics/class-based-views/intro/>.
- Отправка электронных писем с помощью Django: <https://docs.djangoproject.com/en/4.1/topics/email/>.
- Типы полей формы в Django: <https://docs.djangoproject.com/en/4.1/ref/forms/fields/>.
- Работа с формами: <https://docs.djangoproject.com/en/4.1/topics/forms/>.
- Создание форм из моделей: <https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/>.
- Модельные взаимосвязи многие-к-одному: https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_one/.

Резюме

В данной главе вы научились определять для моделей канонические URL-адреса. Вы создали дружественные для поисковой оптимизации URL-адреса постов блога и реализовали постраничную разбивку объектов для списка постов. Вы также научились работать с формами Django и моделировать формы. Вы создали систему рекомендации постов по электронной почте и систему комментариев для своего блога.

В следующей главе вы создадите в своем блоге систему тегирования. Вы научитесь формировать сложные наборы запросов, чтобы извлекать объекты по сходству. Вы освоите создание конкретно-прикладных шаблонных тегов и фильтров. Вы также разработаете конкретно-прикладную карту сайта и новостную ленту для постов блога и реализуете функциональность полно-текстового поиска постов.

3

Расширение приложения для ведения блога

В предыдущей главе были рассмотрены основы форм и создание комментарийной системы. Вы также научились отправлять электронные письма с помощью Django. В этой главе вы расширите свое приложение для ведения блога другими популярными используемыми на блоговых платформах функциональными возможностями, такими как тегирование, рекомендация схожих постов, предоставление читателям новостной RSS-ленты и поиск постов. В ходе разработки этих функциональностей вы узнаете о новых компонентах и функциональностях Django.

В данной главе будут рассмотрены следующие темы:

- интегрирование сторонних приложений;
- использование приложения `django-taggit` для реализации системы тегирования;
- формирование сложных наборов запросов для рекомендации схожих постов;
- создание конкретно-прикладных шаблонных тегов и фильтров для показа на боковой панели списка последних постов и постов, получивших наибольшее число комментариев;
- создание карты сайта с использованием фреймворка карт сайтов;
- формирование новостной RSS-ленты с использованием фреймворка синдицированных новостных лент;
- установка базы данных PostgreSQL;
- реализация полнотекстового поискового механизма с использованием Django и PostgreSQL.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter03>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по уста-

новке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Добавление функциональности тегирования

Очень распространенной функциональностью в блогах является категоризация постов посредством тегов. Теги позволяют классифицировать контент неиерархическим образом, используя простые ключевые слова. Тег – это просто метка или ключевое слово, которое можно назначать постам. Мы создадим систему тегирования, интегрировав в проект стороннее приложение Django по тегированию.

`django-taggit` – это приспособленное для реиспользования приложение, которое в первую очередь предлагает модель `Tag` и менеджер для удобного добавления тегов в любую модель. Исходный код приложения доступен для просмотра на странице <https://github.com/jazzband/django-taggit>.

Сначала необходимо установить приложение `django-taggit` с помощью `pip`, выполнив следующую ниже команду:

```
pip install django-taggit==3.0.0
```

Затем откройте файл `settings.py` проекта `mysite` и добавьте `taggit` в настроечный параметр `INSTALLED_APPS`, как показано ниже:

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

Откройте файл `models.py` приложения `blog` и добавьте предоставляемый приложением `django-taggit` менеджер `TaggableManager` в модель `Post`, используя следующий ниже исходный код:

```
from taggit.managers import TaggableManager

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

Менеджер `tags` позволит добавлять, извлекать и удалять теги из объектов `Post`.

Приведенная ниже схема показывает модели данных, определенные в приложении `django-taggit` для создания тегов и хранения схожих тегированных объектов:



Рис. 3.1. Модель Tag приложения django-taggit

Модель `Tag` используется для хранения тегов. Она содержит поля `name` и `slug`.

Модель `TaggedItem` используется для хранения схожих тегированных объектов. В ней есть поле `ForeignKey` для схожего объекта `Tag`. Она содержит внешний ключ (`ForeignKey`) к объекту `ContentType` и целочисленное поле (`IntegerField`) для хранения соответствующего `id` тегированного объекта. Поля `content_type` и `object_id` в сочетании образуют обобщенное отношение с любой моделью в проекте. Такой подход позволяет создавать взаимосвязи между экземпляром модели `Tag` и экземпляром любой другой модели своих собственных приложений. Об обобщенных отношениях вы узнаете в главе 7 «Отслеживание действий пользователя».

Выполните следующую ниже команду в командной оболочке, чтобы создать миграцию изменений в модели:

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

Вы должны получить такой результат:

```
Migrations for 'blog':
  blog/migrations/0004_post_tags.py
    - Add field tags to post
```

Теперь выполните следующую ниже команду, чтобы создать необходимые таблицы базы данных для моделей приложения `django-taggit` и синхронизировать изменения в своей модели:

```
python manage.py migrate
```

Вы увидите результат, говорящий о том, что миграции были применены, как показано ниже:

```
Applying taggit.0001_initial... OK
Applying taggit.0002_auto_20150616_2121... OK
```

```
Applying taggit.0003_taggeditem_add_unique_index... OK
Applying taggit.0004_alter_taggeditem_content_type_alter_taggeditem_tag... OK
Applying taggit.0005_auto_20220424_2025... OK
Applying blog.0004_post_tags... OK
```

Теперь база данных синхронизирована с моделями `taggit`, и можно начать использовать функциональности приложения `django-taggit`.

Давайте сейчас проинспектируем способы применения менеджера `tags`.

Откройте оболочку Django, выполнив следующую ниже команду в командной строке системной оболочки:

```
python manage.py shell
```

Выполните следующий ниже исходный код, чтобы получить один из постов (пост с ИД 1):

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

Затем добавьте в него несколько тегов и извлеките его теги, чтобы проверить успешность их добавления:

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

Наконец, удалите тег и еще раз проверьте список тегов:

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

Добавлять, извлекать или удалять теги из модели с помощью определенного нами менеджера действительно легко.

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/taggit/tag/` в своем браузере.

Вы увидите страницу администрирования со списком объектов Tag приложения taggit:

The screenshot shows a Django admin page titled "Select tag to change". At the top right is a button labeled "ADD TAG +". Below it is a search bar with a magnifying glass icon and a "Search" button. Underneath is a table header with columns "NAME" and "SLUG". The table contains three rows of data:

NAME	SLUG
django	django
jazz	jazz
music	music

Below the table, the text "3 tags" is displayed. At the bottom left of the table area, there is a link "Action: —— Go 0 of 3 selected".

Рис. 3.2. Представление списка с изменениями тегов на сайте администрирования

Кликните по тегу jazz. Вы увидите следующее:

The screenshot shows a "Change tag" page for the tag "jazz". At the top right is a "HISTORY" button. The main form has two fields: "Name:" with "jazz" and "Slug:" with "jazz". Below the form is a section titled "TAGGED ITEMS" containing a list: "Tagged item: Who was Django Reinhardt? tagged with jazz". To the right of this list is a "Delete" checkbox. Further down are fields for "Content type:" (set to "blog | post") and "Object ID:" (set to "1").

Рис. 3.3. Поле схожих тегов объекта Post

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/blog/post/1/change/>, чтобы отредактировать пост с ИД 1.

Вы увидите, что теперь в посты вставлено новое поле **Tags**, как показано ниже, в котором можно легко редактировать теги:



Рис. 3.4. Поле схожих тегов объекта Post

Сейчас необходимо отредактировать свои посты в блоге, чтобы отобразить теги.

Откройте шаблон `blog/post/list.html` и добавьте в него следующий ниже исходный код HTML, выделенный жирным шрифтом:

```
{% 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="tags">Tags: {{ post.tags.all|join:", " }}</p>
<p class="date">
Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|truncatewords:30|linebreaks }}
{% endfor %}
{% include "pagination.html" with page=page_obj %}
{% endblock %}
```

Шаблонный фильтр `join` работает так же, как метод Python `string.join()`, чтобы конкатенировать элементы с заданной строкой.

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере. Под заголовком каждого поста вы должны увидеть список тегов:

Who was Django Reinhardt?

Tags: music, jazz

Published Jan. 1, 2022, 11:59 p.m. by admin

Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...

Рис. 3.5. Элемент списка постов, включающий схожие теги

Далее мы отредактируем представление `post_list`, чтобы пользователи имели возможность отображать список всех постов, помеченных конкретным тегом.

Откройте `views.py` файл приложения `blog`, импортируйте модель `Tag` из приложения `django-taggit` и измените представление `post_list`, как показано ниже, чтобы при необходимости фильтровать посты по тегу. Новый исходный код выделен жирным шрифтом:

```
from taggit.models import Tag

def post_list(request, tag_slug=None):
    post_list = Post.published.all()
    tag = None
    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        post_list = post_list.filter(tags__in=[tag])
    # Постраничная разбивка с 3 постами на страницу
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    try:
        posts = paginator.page(page_number)
    except PageNotAnInteger:
        # Если page_number не целое число, то
        # выдать первую страницу
        posts = paginator.page(1)
    except EmptyPage:
        # Если page_number находится вне диапазона, то
        # выдать последнюю страницу результатов
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                  {'posts': posts,
                   'tag': tag})
```

Теперь представление `post_list` работает следующим образом.

1. Представление принимает опциональный параметр `tag_slug`, значение которого по умолчанию равно `None`. Этот параметр будет передан в URL-адресе.
2. Внутри указанного представления формируется изначальный набор запросов, извлекающий все опубликованные посты, и если имеется слаг данного тега, то берется объект `Tag` с данным слагом, используя функцию сокращенного доступа `get_object_or_404()`.
3. Затем список постов фильтруется по постам, которые содержат данный тег. Поскольку здесь используется взаимосвязь многие-ко-многим, необходимо фильтровать записи по тегам, содержащимся в заданном списке, который в данном случае содержит только один элемент. Здесь используется операция `__in` поиска по полю. Взаимосвязи многие-ко-многим возникают, когда несколько объектов модели ассоциированы с несколькими объектами другой модели. В нашем приложении пост может иметь несколько тегов, и тег может быть связан с несколькими постами. О методах создания взаимосвязи многие-ко-многим вы узнаете в главе 6 «Распространение контента на веб-сайте». Подробнее о взаимосвязях многие-ко-многим можно узнать на странице https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/.
4. Наконец, теперь функция `render()` передает новую переменную `tag` в шаблон.

Напомним, что итерируемые наборы запросов `QuerySet` являются ленивыми. Наборы запросов, служащие для извлечения постов, будут оцениваться только при прокручивании списка постов в цикле во время прорисовки шаблона.

Откройте файл `urls.py` приложения `blog`, закомментируйте основанный на классе шаблон URL-адреса `PostListView` и раскомментируйте представление `post_list`, как показано ниже:

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

Добавьте следующий ниже дополнительный шаблон URL-адреса, чтобы отображать список постов по тегу:

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

Как вы видите, оба шаблона указывают на одно и то же представление, но у них разные имена. Первый шаблон будет вызывать представление `post_list` без каких-либо опциональных параметров, тогда как второй шаблон будет вызывать это представление с параметром `tag_slug`. Конвертер путей `slug` используется для сочетания с параметром, представленным строковым литералом в нижнем регистре с буквами или цифрами ASCII, а также символами дефиса и подчеркивания.

Теперь файл urls.py приложения blog должен выглядеть следующим образом:

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

app_name = 'blog'

urlpatterns = [
    # представления поста
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
    path('tag/<slug:tag_slug>/',
        views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>',
        views.post_detail,
        name='post_detail'),
    path('<int:post_id>/share/',
        views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
        views.post_comment, name='post_comment'),
]
```

Поскольку используется представление post_list, отредактируйте шаблон blog/post/list.html, видоизменив постраничную разбивку, чтобы использовать объект posts:

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

Добавьте в шаблон blog/post/list.html следующие ниже строки, выделенные жирным шрифтом:

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

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

{% block content %}
<h1>My Blog</h1>
{% if tag %}
<h2>Posts tagged with "{{ tag.name }}"</h2>
{% endif %}
{% for post in posts %}
<h2>
<a href="{{ post.get_absolute_url }}>
{{ post.title }}
</a>
```

```

</h2>
<p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
<p class="date">
    Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|truncatewords:30|linebreaks }}
{% endfor %}
{% include "pagination.html" with page=posts %}
{% endblock %}

```

Если пользователь зайдет в блог, то он увидит список всех постов. Если он будет фильтровать по постам, помеченным конкретным тегом, то увидит тег, по которому он фильтрует.

Теперь отредактируйте шаблон `blog/post/list.html`, изменив вид отображения тегов, как показано ниже. Новые строки выделены жирным шрифтом:

```

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

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

{% block content %}
    <h1>My Blog</h1>
    {% if tag %}
        <h2>Posts tagged with "{{ tag.name }}"</h2>
    {% endif %}
    {% for post in posts %}
        <h2>
            <a href="{{ post.get_absolute_url }}">
                {{ post.title }}
            </a>
        </h2>
        <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>
        <p class="date">
            Published {{ post.publish }} by {{ post.author }}
        </p>
        {{ post.body|truncatewords:30|linebreaks }}
    {% endfor %}
    {% include "pagination.html" with page=posts %}
{% endblock %}

```

В приведенном выше исходном коде прокручиваются в цикле все теги поста, отображающие конкретно-прикладную ссылку на URL-адрес, чтобы фильтровать посты по этому тегу. URL-адрес формируется с помощью тега `{% url "blog:post_list_by_tag" tag.slug %}`, используя имя URL-адреса и тег `slug` в качестве его параметра. Теги отделяются запятыми.

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/tag/jazz/` в своем браузере, и вы увидите список постов, отфильтрованных по этому тегу, примерно как показано ниже:

The screenshot shows a blog interface titled 'My Blog'. On the right, there's a sidebar with the title 'My blog' and the subtitle 'This is my blog.' Below the sidebar, the main content area displays a post titled 'Posts tagged with "jazz"'. The first post in the list is 'Who was Django Reinhardt?' with the tag 'Tags: music , jazz'. Below the post, it says 'Published Jan. 1, 2022, 11:59 p.m. by admin'. A snippet of the post content follows: 'Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...'. At the bottom of the content area, it says 'Page 1 of 1.'

Рис. 3.6. Пост, отфильтрованный по тегу «jazz»

Извлечение постов по сходству

Теперь, когда мы реализовали тегирование постов блога, с помощью тегов можно делать много интересных вещей. Теги позволяют классифицировать посты неиерархическим образом. Посты на схожие темы будут иметь несколько общих тегов. Мы разработаем функциональность отображения схожих постов по числу имеющихся у них общих тегов. При таком подходе при чтении пользователем поста ему можно будет предлагать почитать другие родственные посты.

Для того чтобы получить посты, схожие с конкретным постом, необходимо выполнить следующие действия:

- 1) извлечь все теги текущего поста;
- 2) получить все посты, помеченные любым из этих тегов;
- 3) исключить текущий пост из этого списка, чтобы не рекомендовать тот же самый пост;
- 4) упорядочить результаты по числу общих тегов, которое есть у текущего поста;

- 5) в случае двух или более постов с одинаковым числом тегов рекомендовать самый последний пост;
- 6) ограничить запрос числом постов, которые вы хотите рекомендовать.

Эти шаги транслируются в сложный набор запросов QuerySet, который вставляется в представление post_detail.

Откройте файл views.py приложения blog и добавьте следующую ниже инструкцию импорта в верхнюю его часть:

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

Это функция агрегирования Count из Django ORM-преобразователя. Данная функция позволит выполнять агрегированный подсчет тегов. Модуль django.db.models содержит следующие ниже функции агрегирования:

- Avg: среднее значение;
- Max: максимальное значение;
- Min: минимальное значение;
- Count: общее количество объектов.

Об агрегировании можно узнать на странице <https://docs.djangoproject.com/en/4.1/topics/db/aggregation/>.

Откройте файл views.py приложения blog и добавьте следующие ниже строки в представление post_detail. Новые строки выделены жирным шрифтом:

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                           status=Post.Status.PUBLISHED,
                           slug=post,
                           publish__year=year,
                           publish__month=month,
                           publish__day=day)
    # Список активных комментариев к этому посту
    comments = post.comments.filter(active=True)

    # Форма для комментариев пользователей
    form = CommentForm()

    # Список схожих постов
    post_tags_ids = post.tags.values_list('id', flat=True)
    similar_posts = Post.published.filter(tags__in=post_tags_ids)\n        .exclude(id=post.id)
    similar_posts = similar_posts.annotate(same_tags=Count('tags'))\n        .order_by('-same_tags', '-publish')[:4]

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

```
'comments': comments,
'form': form,
'similar_posts': similar_posts})
```

Приведенный выше исходный код делает следующее:

- 1) извлекается Python'овский список идентификаторов тегов текущего поста. Набор запросов `QuerySet values_list()` возвращает кортежи со значениями заданных полей. Ему передается параметр `flat=True`, чтобы получить одиночные значения, такие как `[1, 2, 3, ...]`, а не одноэлементные кортежи, такие как `[(1,), (2,), (3,) ...]`;
- 2) берутся все посты, содержащие любой из этих тегов, за исключением текущего поста;
- 3) применяется функция агрегирования `Count`. Ее работа – генерировать вычисляемое поле – `same_tags`, – которое содержит число тегов, общих со всеми запрошенными тегами;
- 4) результат упорядочивается по числу общих тегов (в убывающем порядке) и по `publish`, чтобы сначала отображать последние посты для постов с одинаковым числом общих тегов. Результат нарезается, чтобы получить только первые четыре поста;
- 5) объект `similar_posts` передается в контекстный словарь для функции `render()`.

Теперь отредактируйте шаблон `blog/post/detail.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

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

<h2>Similar posts</h2>
{% for post in similar_posts %}
<p>
    <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
</p>
{% empty %}
```

```

There are no similar posts yet.
{% endfor %}

{% with comments.count as total_comments %}
<h2>
{{ total_comments }} comment{{ total_comments|pluralize }}
</h2>
{% endwith %}
{% 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 %}
{% include "blog/post/includes/comment_form.html" %}
{% endblock %}

```

Страница детальной информации о посте должна выглядеть, как показано ниже:

Who was Django Reinhardt?

Published Jan. 1, 2022, 11:59 p.m. by admin

Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and remains the most significant.

[Share this post](#)

Similar posts

There are no similar posts yet.

Рис. 3.7. Страница детальной информации о посте, включающая список схожих постов

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/blog/post/> в своем браузере, отредактируйте пост, у которого нет тегов, добавив теги `music` и `jazz`, как показано ниже:

Who was Miles Davis?

Title: Who was Miles Davis?

Slug: who-was-miles-davis

Author: 1 Q admin

Body: Miles Davis was an American trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.

Publish:

Date: 2022-01-02 Today |

Time: 13:18:11 Now |

Note: You are 2 hours ahead of server time.

Status: Published

Tags: jazz, music

A comma-separated list of tags.

Рис. 3.8. Добавление тегов «jazz» и «music» в пост

Отредактируйте еще один пост, добавив тег jazz, как показано ниже:

Notes on Duke Ellington

Title: Notes on Duke Ellington

Slug: notes-on-duke-ellington

Author: 1 Q admin

Body: Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half a century.

Publish:

Date: 2022-01-03 Today |

Time: 13:19:33 Now |

Note: You are 2 hours ahead of server time.

Status: Published

Tags: jazz

A comma-separated list of tags.

Рис. 3.9. Добавление тега «jazz» в пост

Теперь страница детальной информации о первом посте должна выглядеть, как показано ниже:

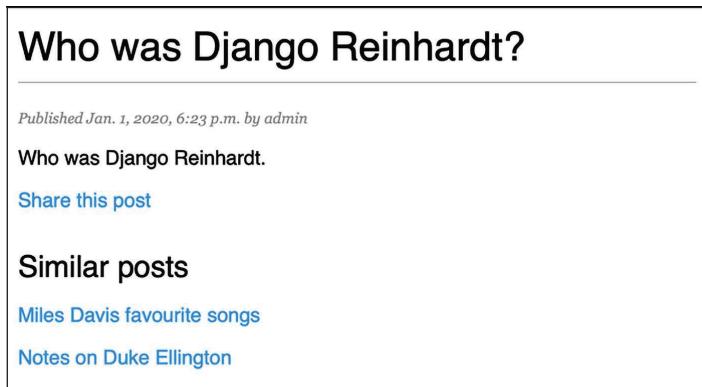


Рис. 3.10. Страница детальной информации о посте, включающая список схожих постов

Посты, рекомендованные в разделе **Similar posts** (Схожие посты), появляются в убывающем порядке в зависимости от числа общих тегов с изначальным постом.

Теперь появилась возможность успешно рекомендовать читателям схожие посты. Приложение django-taggit также содержит менеджер `similar_objects()`, который можно использовать для извлечения объектов на основе общих тегов. Со всеми менеджерами приложения django-taggit можно ознакомиться на странице <https://django-taggit.readthedocs.io/en/latest/api.html>.

Кроме того, список тегов можно добавить и в шаблон детальной информации о посте, точно таким же образом, как это было сделано в шаблоне `blog/post/list.html`.

Создание конкретно-прикладных шаблонных тегов и фильтров

Django предлагает разнообразный набор встроенных шаблонных тегов, таких как `{% if %}` или `{% block %}`. В главе 1 «Разработка приложения для ведения блога» и главе 2 «Усовершенствование блога за счет продвинутых функциональностей» использовались разные шаблонные теги. Полный справочный материал по встроенным шаблонным тегам и фильтрам находится на странице <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/>.

Django также позволяет создавать свои собственные шаблонные теги для выполнения действий, адаптированных под конкретно-прикладные зада-

чи. Конкретно-прикладные шаблонные теги бывают весьма кстати, когда появляется необходимость добавлять в свои шаблоны функциональность, которая не охватывается ключевым набором шаблонных тегов Django. Это может быть тег для исполнения набора запросов QuerySet или любой обработки на стороне сервера, которую вы хотите реиспользовать в шаблонах. Например, можно было бы разработать шаблонный тег для отображения списка последних постов, опубликованных в блоге. Этот список можно было бы вставлять в боковую панель, чтобы он всегда был виден, независимо от обрабатывающего запрос представления.

Реализация конкретно-прикладных шаблонных тегов

Django предоставляет следующие вспомогательные функции, которые позволяют легко создавать шаблонные теги:

- `simple_tag`: обрабатывает предоставленные данные и возвращает строковый литерал;
- `inclusion_tag`: обрабатывает предоставленные данные и возвращает прорисованный шаблон.

Шаблонные теги должны находиться внутри приложений Django.

Внутри каталога приложения `blog` создайте новый каталог, назовите его `templatetags` и добавьте в него пустой файл `__init__.py`. Создайте еще один файл в той же папке и назовите его `blog_tags.py`. Файловая структура приложения `blog` должна выглядеть, как показано ниже:

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

Очень важно то, как файл называется. Вы будете использовать имя этого модуля для загрузки тегов в шаблоны.

Создание простого шаблонного тега

Давайте начнем с создания простого тега, чтобы получать общее число опубликованных в блоге постов.

Отредактируйте только что созданный вами файл `templatetags/blog_tags.py`, добавив следующий ниже исходный код:

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

register = template.Library()

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

Мы создали простой шаблонный тег, который возвращает число опубликованных в блоге постов.

Для того чтобы быть допустимой библиотекой тегов, в каждом содержащем шаблонные теги модуле должна быть определена переменная с именем `register`. Эта переменная является экземпляром класса `template.Library`, и она используется для регистрации шаблонных тегов и фильтров приложения.

В приведенном выше исходном коде тег `total_posts` был определен с помощью простой функции Python. В функцию был добавлен декоратор `@register.simple_tag`, чтобы зарегистрировать ее как простой тег. Django будет использовать имя функции в качестве имени тега. Если есть потребность зарегистрировать ее под другим именем, то это можно сделать, указав атрибут `name`, например `@register.simple_tag(name='my_tag')`.



После добавления нового модуля шаблонных тегов потребуется перезапустить сервер разработки, чтобы новые теги и фильтры стали доступны для использования в шаблонах.

Прежде чем использовать конкретно-прикладные шаблонные теги, необходимо сделать их доступными для шаблона с помощью тега `{% load %}`. Как упоминалось ранее, нужно использовать имя модуля Python, содержащего ваши шаблонные теги и фильтры.

Отредактируйте шаблон `blog/templates/base.html`, добавив тег `{% load blog_tags %}` в верхней его части, чтобы загрузить свой модуль шаблонных тегов. Затем примените созданный вами тег для отображения общего числа постов, как показано ниже. Новые строки выделены жирным шрифтом:

```
{% 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>
```

Для того чтобы отслеживать добавленные в проект новые файлы, нужно перезапустить сервер. Остановите сервер разработки с помощью **Ctrl+C** и запустите его снова, используя следующую ниже команду:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/blog/> в своем браузере. На боковой панели вы должны увидеть общее число постов сайта, как показано ниже:



Рис. 3.11. Общее число опубликованных постов, вставленных в боковую панель

Если вы увидите следующее ниже сообщение об ошибке, то весьма вероятно, что вы не перезапустили сервер разработки.

```
TemplateSyntaxError at /blog/2022/1/1/who-was-django-reinhardt/  
'blog_tags' is not a registered tag library. Must be one of:  
admin_list  
admin_modify  
admin_urls  
cache  
i18n  
l10n  
log  
static  
tz
```

Рис. 3.12. Сообщение об ошибке, когда библиотека шаблонных тегов не зарегистрирована

Шаблонные теги позволяют обрабатывать любые данные и добавлять их в любой шаблон независимо от исполняемого представления. При этом для отображения результатов в своих шаблонах можно выполнять наборы запросов или обрабатывать любые данные.

Создание шаблонного тега включения

Мы создадим еще один тег, чтобы отображать последние посты на боковой панели блога. На этот раз мы реализуем тег включения. Используя тег включения, можно отображать шаблон с контекстными переменными, возвращаемыми вашим шаблонным тегом.

Отредактируйте файл `templatetags/blog_tags.py`, добавив следующий ниже исходный код:

```
@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}
```

В приведенном выше исходном коде мы зарегистрировали шаблонный тег, применяя декоратор `@register.inclusion_tag`. Используя `blog/post/latest_posts.html`, был указан шаблон, который будет прорисовываться возвращенными значениями. Шаблонный тег будет принимать optionalный параметр `count`, который по умолчанию равен 5. Этот параметр позволяет задавать число отображаемых постов. Данная переменная используется для того, чтобы ограничивать результаты запроса `Post.published.order_by('-publish')[count]`.

Обратите внимание, что приведенная выше функция возвращает не простое значение, а словарь переменных. Теги включения должны возвращать словарь значений, который используется в качестве контекста для прорисовки заданного шаблона. Только созданный шаблонный тег позволяет задавать optionalное число отображаемых постов как `{% show_latest_posts 3 %}`.

Теперь создайте новый файл шаблона в разделе `blog/post/` и назовите его `latest_posts.html`.

Отредактируйте новый шаблон `blog/post/latest_posts.html`, добавив следующий ниже исходный код:

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

В приведенном выше исходном коде отображается неупорядоченный список постов, используя возвращаемую вашим шаблонным тегом переменную `latest_posts`. Теперь отредактируйте шаблон `blog/base.html`, добавив новый шаблонный тег, чтобы отображать последние три поста, как показано ниже. Новые строки выделены жирным шрифтом:

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

Здесь вызывается шаблонный тег, передающий число отображаемых постов, и шаблон прорисовывается прямо на месте с заданным контекстом.

Затем вернитесь в свой браузер и обновите страницу. Теперь боковая панель должна выглядеть следующим образом:



Рис. 3.13. Боковая панель блога,
включая последние опубликованные посты

Создание шаблонного тега, возвращающего набор запросов

Наконец, мы создадим простой шаблонный тег, который возвращает значение. Мы сохраним результат в реиспользуемой переменной, не выводя его напрямую. Мы создадим тег, чтобы отображать посты с наибольшим числом комментариев.

Отредактируйте файл `templatetags/blog_tags.py`, добавив следующую ниже инструкцию импорта и шаблонный тег:

```
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 приведенном выше шаблонном теге с помощью функции `annotate()` формируется набор запросов `QuerySet`, чтобы агрегировать общее число комментариев к каждому посту. Функция агрегирования `Count` используется для сохранения количества комментариев в вычисляемом поле `total_comments` по каждому объекту `Post`. Набор запросов `QuerySet` упорядочивается по вычисляемому полю в убывающем порядке. Также предоставляется опциональная переменная `count`, чтобы ограничивать общее число возвращаемых объектов.

В дополнение к `Count` Django предлагает функции агрегирования `Avg`, `Max`, `Min` и `Sum`. Подробнее о функциях агрегирования можно почитать на странице <https://docs.djangoproject.com/en/4.1/topics/db/aggregation/>.

Далее отредактируйте шаблон `blog/base.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
{% 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.
        </p>
```

```
I've written {% total_posts %} posts so far.  
</p>  
<h3>Latest posts</h3>  
{% show_latest_posts 3 %}  
<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>  
</div>  
</body>  
</html>
```

В приведенном выше исходном коде результат сохраняется в конкретно-прикладной переменной, используя аргумент `as`, за которым следует имя переменной. В качестве шаблонного тега используется `{% get_most_commented_posts as most_commented_posts %}`, чтобы сохранить результат шаблонного тега в новой переменной с именем `most_commented_posts`. Затем возвращенные посты отображаются, используя HTML-элемент в виде неупорядоченного списка.

Теперь откройте свой браузер и обновите страницу, чтобы увидеть итоговый результат. Он должен выглядеть следующим образом:

The screenshot shows a blog application interface. On the left, there's a main content area with three posts listed:

- Notes on Duke Ellington**
Tags: jazz
Published Jan. 3, 2022, 1:19 p.m. by admin
Text: Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...
- Who was Miles Davis?**
Tags: music , jazz
Published Jan. 2, 2022, 1:18 p.m. by admin
Text: Miles Davis was an American trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.
- Who was Django Reinhardt?**
Tags: music , jazz
Published Jan. 1, 2022, 11:59 p.m. by admin
Text: Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...

On the right side, there's a sidebar with the following sections:

- My blog**
Text: This is my blog. I've written 4 posts so far.
- Latest posts**
 - Notes on Duke Ellington
 - Who was Miles Davis?
 - Who was Django Reinhardt?
- Most commented posts**
 - Notes on Duke Ellington
 - Who was Django Reinhardt?
 - Another post
 - Who was Miles Davis?

At the bottom left of the sidebar, there's a link: **Page 1 of 2. Next**.

Рис. 3.14. Представление списка постов, включая полную боковую панель с последними и наиболее прокомментированными постами

Теперь у вас есть четкое понимание того, как разрабатывать конкретно-прикладные шаблонные теги. Подробнее о них можно почитать на странице <https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/>.

Реализация конкретно-прикладных шаблонных фильтров

Django имеет целый ряд встроенных шаблонных фильтров, которые позволяют изменять переменные в шаблонах. Это функции Python, которые принимают один или два параметра, значение переменной, к которой применяется фильтр, и optionalный аргумент. Они возвращают значение, которое может быть отображено или обработано еще одним фильтром.

Фильтр записывается как {{ variable|my_filter }}. Фильтры с аргументом записываются как {{ variable|my_filter:"foo" }}. Например, фильтр capfirst можно использовать для приведения первого символа значения в верхний регистр, например {{ value|capfirst }}. Если значение равно django, то на выходе будет Django. К переменной можно применять столько фильтров, сколько потребуется, например {{ variable|filter1|filter2 }}, и каждый фильтр будет применен к результату, сгенерированному предыдущим фильтром.

Со списком встроенных шаблонных фильтров можно ознакомиться по адресу <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#built-in-filter-reference>.

Создание шаблонного фильтра для поддержки синтаксиса Markdown

Мы создадим конкретно-прикладной фильтр, который позволит использовать синтаксис упрощенной разметки Markdown в постах блога, а затем в шаблонах конвертировать текст поста в HTML.

Markdown – это синтаксис форматирования обычного текста, который очень прост в использовании и предназначен для конвертирования в HTML. Посты можно писать, используя простой синтаксис Markdown, и получать автоматическую конвертацию контента в исходный код HTML. Изучить синтаксис Markdown намного проще, чем изучить HTML. Используя Markdown, можно сделать так, чтобы другие не сведущие в технологиях участники легко писали посты для вашего блога. С основами формата Markdown можно ознакомиться на странице <https://daringfireball.net/projects/markdown/basics>.

Сначала установите модуль Python markdown посредством pip, используя следующую ниже команду в командной оболочке:

```
pip install markdown==3.4.1
```

Затем отредактируйте файл `templatetags/blog_tags.py`, вставив следующий ниже исходный код:

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

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

Шаблонные фильтры регистрируются таким же образом, как и шаблонные теги. Во избежание конфликта имен между именем функции и модулем `markdown` мы дали функции имя `markdown_format`, а фильтру – имя `markdown` для использования в шаблонах, в частности `{{ variable|markdown }}`.

Django экранирует генерируемый фильтрами исходный код HTML; символы HTML-сущностей заменяются их кодированными в HTML символами. Например, `<p>` преобразовывается в `&ltp&gt` (символ *меньше*, символ *p*, символ *больше*).

Мы используем предоставляемую веб-фреймворком Django функцию `mark_safe`, чтобы помечать результат как безопасный для прорисовки в шаблоне исходный код HTML. По умолчанию Django не будет доверять никакому исходному коду HTML и будет экранировать его перед его вставкой в результат. Единственными исключениями являются переменные, которые помечены как безопасные, чтобы тем самым избежать экранирования. Такое поведение не дает Django выводить потенциально опасный исходный код HTML и позволяет создавать исключения, дабы возвращать безопасный исходный код HTML.

Отредактируйте шаблон `blog/post/detail.html`, добавив следующий ниже новый исходный код, выделенный жирным шрифтом:

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

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

{% block content %}
<h1>{{ post.title }}</h1>
<p class="date">
    Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|markdown }}
<p>
    <a href="{% url "blog:post_share" post.id %}">
        Share this post
    </a>
</p>
<h2>Similar posts</h2>
{% for post in similar_posts %}
    <p>
```

```

<a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
</p>
{% empty %}
    There are no similar posts yet.
{% endfor %}

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

{% include "blog/post/includes/comment_form.html" %}
{% endblock %}

```

Фильтр `linebreaks` шаблонной переменной `{{ post.body }}` был заменен фильтром `markdown`. Данный фильтр не только преобразовывает разрывы строк в теги `<p>`, но и конвертирует форматирование Markdown в HTML.

Отредактируйте шаблон `blog/post/list.html`, добавив следующий ниже новый исходный код, выделенный жирным шрифтом:

```

{% extends "blog/base.html" %}
{% load blog_tags %}

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

{% block content %}
    <h1>My Blog</h1>
    {% if tag %}
        <h2>Posts tagged with "{{ tag.name }}"</h2>
    {% endif %}
    {% for post in posts %}

```

```
<h2>
    <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
    </a>
</h2>
<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>
<p class="date">
    Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|markdown|truncatewords_html:30 }}
{% endfor %}
{% include "pagination.html" with page=posts %}
{% endblock }
```

В шаблонную переменную `{{ post.body }}` был добавлен новый фильтр `markdown`. Этот фильтр преобразовывает контент в формате Markdown в формат HTML. Поэтому мы заменили приведенный выше фильтр `truncatewords` фильтром `truncatewords_html`. Данный фильтр усекает строку после определенного числа слов, избегая незакрытых HTML-тегов.

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/admin/blog/post/add/` в своем браузере и создайте новый пост со следующим ниже 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 website](<https://www.djangoproject.com/>).

Форма должна выглядеть следующим образом:

Add post

Title: Markdown post

Slug: markdown-post

Author: 1

Body:

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 website](https://www.djangoproject.com/).

Publish:

Date: 2022-01-22 Today |

Time: 09:30:04 Now |

Note: You are 2 hours ahead of server time.

Status: Draft

Tags: markdown

A comma-separated list of tags.

Рис. 3.15. Пост с контентом в формате Markdown, прорисовываемом в формат HTML

Пройдите по URL-адресу <http://127.0.0.1:8000/blog/> в своем браузере и посмотрите, как прорисовывается новый пост. Вы должны увидеть следующий ниже результат:

My Blog

Markdown post

Tags: markdown

Published Jan. 22, 2022, 9:30 a.m. by admin

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 website ...](#)

Рис. 3.16. Пост с контентом в формате Markdown, прорисованном как HTML

На рис. 3.16 хорошо видно, что конкретно-прикладные шаблонные фильтры очень удобны для адаптации форматирования под конкретно-прикладную задачу. Более подробная информация о конкретно-прикладных фильтрах находится по адресу <https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/#writing-custom-template-filters>.

Добавление карты сайта

Django идет в комплекте с фреймворком карт сайтов, который позволяет динамически создавать карты для своего сайта. Карта сайта – это XML-файл, который сообщает поисковым системам о страницах веб-сайта, их релевантность и частоту их обновления. Использование карты сделает сайт более заметным в рейтинге поисковых систем, поскольку она помогает поисковым роботам индексировать содержимое сайта.

Фреймворк карт сайтов Django зависит от приложения `django.contrib.sites`, которое позволяет ассоциировать объекты с теми или иными веб-сайтами, работающими вместе с вашим проектом. Это удобно, когда вы хотите управлять несколькими сайтами, используя один проект Django. Для установки фреймворка карт сайтов необходимо активировать приложения `sites` и `sitemaps` в своем проекте.

Отредактируйте файл `settings.py` проекта, добавив `django.contrib.sites` и `django.contrib.sitemaps` в настроочный параметр `INSTALLED_APPS`. Кроме того, определите новый настроочный параметр для ИД сайта, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
# ...  
  
SITE_ID = 1  
  
# Определение приложения  
  
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'blog.apps.BlogConfig',  
    'taggit',  
    ''django.contrib.sites'',  
    ''django.contrib.sitemaps'',  
]  
]
```

Теперь запустите следующую ниже команду из командной оболочки, чтобы создать таблицы приложения `Django sites` в базе данных:

```
python manage.py migrate
```

Вы должны увидеть результат, содержащий такие строки:

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

Теперь приложение `sites` синхронизировано с базой данных.

Затем внутри каталога своего приложения `blog` создайте новый файл и назовите его `sitemaps.py`. Откройте файл и добавьте в него следующий ниже исходный код:

```
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 определили конкретно-прикладную карту сайта, унаследовав класс `Sitemap` модуля `sitemaps`. Атрибуты `changefreq` и `priority` указывают частоту изменения страниц постов и их релевантность на веб-сайте (максимальное значение равно 1).

Метод `items()` возвращает набор запросов `QuerySet` объектов, подлежащих включению в эту карту сайта. По умолчанию Django вызывает метод `get_absolute_url()` по каждому объекту, чтобы получить его URL-адрес. Напомним, что мы применили этот метод в главе 2 «Усовершенствование блога за счет продвинутых функциональностей», чтобы формировать канонический URL-адрес постов. Если нужно указать URL-адрес каждого объекта, то в класс `sitemap` можно добавить метод `location`.

Метод `lastmod` получает каждый возвращаемый методом `items()` объект и возвращает время последнего изменения объекта.

Атрибуты `changefreq` и `priority` могут быть либо методами, либо атрибутами. С полным справочным материалом по картам сайтов можно ознакомиться в официальной документации Django на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/sitemaps/>.

Мы создали карту сайта. Теперь необходимо создать для него URL-адрес.

Отредактируйте главный файл `urls.py` проекта `mysite`, добавив карту сайта, как показано ниже. Новые строки выделены жирным шрифтом:

```
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 приведенном выше исходном коде были вставлены необходимые инструкции импорта и определен словарь `sitemaps`. На сайте может быть определено несколько карт. Мы определили шаблон URL-адреса, который совпадает с шаблоном `sitemap.xml` и в котором используется встроенное в Django представление `sitemap`. Словарь `sitemaps` передается в представление `sitemap`.

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/sitemap.xml` в своем браузере. Вы увидите результат в формате XML, включающий все опубликованные посты, как показано ниже:

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
  xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <url>
    <loc>http://example.com/blog/2022/1/22/markdown-post/</loc>
    <lastmod>2022-01-22</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/3/notes-on-duke-ellington/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/2/who-was-miles-davis/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/1/who-was-django-reinhardt/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/1/another-post/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
</urlset>
```

URL-адрес каждого объекта `Post` формируется путем вызова его метода `get_absolute_url()`.

Атрибут `lastmod` соответствует полю даты `updated` поста, как было указано в карте сайта, а атрибуты `changefreq` и `priority` также взяты из класса `PostSitemap`.

Для формирования URL-адресов используется домен example.com. Этот домен получен из хранящегося в базе данных объекта Site. Указанный объект был создан по умолчанию, когда вы синхронизировали фреймворк сайтов со своей базой данных. Подробнее о фреймворке сайтов можно почитать на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/sites/>.

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/sites/site/> в своем браузере. Вы должны увидеть что-то вроде этого:

DOMAIN NAME	DISPLAY NAME
<input type="checkbox"/> example.com	example.com

Рис. 3.17. Представление списка на сайте администрирования для модели Site фреймворка сайтов

На рис. 3.17 показано представление списка на сайте администрирования для фреймворка сайтов. Здесь можно задать домен или хост, который будет использоваться фреймворком сайтов и зависящими от него приложениями. Для того чтобы сгенерировать URL-адреса, существующие в локальной среде, измените доменное имя на localhost:8000, как показано на рис. 3.18, и сохраните его:

Domain name:	localhost:8000
Display name:	localhost:8000

Рис. 3.18. Представление редактирования на сайте администрирования для модели Site фреймворка сайтов

Снова пройдите по URL-адресу <http://127.0.0.1:8000/sitemap.xml> в своем браузере. Теперь в отображаемых в ленте URL-адресах будет использовать-

ся новое хост-имя, и они будут выглядеть вот так: `http://localhost:8000/blog/2022/1/22/markdown-post/`. Теперь ссылки доступны в локальной среде. В производственной среде, чтобы иметь возможность генерировать абсолютные URL-адреса, придется использовать домен своего веб-сайта.

Создание новостных лент для постов блога

В Django есть встроенный фреймворк синдицированных новостных лент, который можно использовать для динамического генерирования новостных RSS- или Atom-лент, по форме похожих на создание карт сайта с использованием фреймворка сайтов. Новостная веб-лента – это формат данных (обычно XML), который предоставляет пользователям самый последний обновленный контент. Пользователи могут подписываться на новостную ленту с помощью агрегатора новостных лент, то есть программного обеспечения, которое используется для чтения новостных лент и получения уведомлений о новом контенте.

Внутри каталога своего приложения `blog` создайте новый файл и назовите его `feeds.py`. Добавьте в него следующие ниже строки:

```
import markdown
from django.contrib.syndication.views import Feed
from django.template.defaultfilters import truncatewords_html
from django.urls import reverse_lazy
from .models import Post

class LatestPostsFeed(Feed):
    title = 'My blog'
    link = reverse_lazy('blog:post_list')
    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_html(markdown.markdown(item.body), 30)

    def item_pubdate(self, item):
        return item.publish
```

В приведенном выше исходном коде мы определили новостную ленту, создав подкласс класса Feed фреймворка синдицированных новостных лент. Атрибуты `title`, `link` и `description` соответствуют элементам RSS `<title>`, `<link>` и `<description>` в указанном порядке.

Функция-утилита `reverse_lazy()` используется для того, чтобы генерировать URL-адрес для атрибута `link`. Метод `reverse()` позволяет формировать URL-адреса по их имени и передавать опциональные параметры. Мы использовали `reverse()` в главе 2 «Усовершенствование блога за счет продвинутых функциональностей».

Функция-утилита `reverse_lazy()` представляет собой лениво вычисляемую версию `reverse()`. Она позволяет использовать обратный URL-адрес до того, как конфигурация URL-адреса проекта будет загружена.

Метод `items()` извлекает включаемые в новостную ленту объекты. Мы извлекаем последние пять опубликованных постов, которые затем будут включены в новостную ленту.

Методы `item_title()`, `item_description()` и `item_pubdate()` будут получать каждый возвращаемый методом `items()` объект и возвращать заголовок, описание и дату публикации по каждому элементу.

В методе `item_description()` используется функция `markdown()`, чтобы конвертировать контент в формате Markdown в формат HTML, и функция шаблонного фильтра `truncatewords_html()`, чтобы сокращать описание постов после 30 слов, избегая незакрытых HTML-тегов.

Теперь отредактируйте файл `blog/urls.py`, импортировав класс `LatestPostsFeed` и создав экземпляр новостной ленты в новом шаблоне URL-адреса, как показано ниже. Новые строки выделены жирным шрифтом:

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

app_name = 'blog'

urlpatterns = [
    # представления поста
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
    path('tag/<slug:tag_slug>/',
        views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>',
        views.post_detail,
        name='post_detail'),
    path('<int:post_id>/share/',
        views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
        views.post_comment, name='post_comment'),
```

```
    path('feed/', LatestPostsFeed(), name='post_feed'),  
]
```

Пройдите по URL-адресу <http://127.0.0.1:8000/blog/feed/> в своем браузере. Теперь вы должны увидеть новостную RSS-ленту, содержащую последние пять постов блога:

```
<?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, 2 Jan 2020 09:56:40 +0000</lastBuildDate>  
    <item>  
      <title>Who was Django Reinhardt?</title>  
      <link>http://localhost:8000/blog/2020/1/2/who-was-django-  
reinhardt/</link>  
      <description>Who was Django Reinhardt.</description>  
      <guid>http://localhost:8000/blog/2020/1/2/who-was-django-  
reinhardt/</guid>  
    </item>  
    ...  
  </channel>  
</rss>
```

Если вы используете Chrome, то увидите исходный код XML. Если же вы используете Safari, то он попросит вас установить программу чтения новостных RSS-лент.

Давайте установим RSS-клиента для рабочего стола, чтобы просматривать новостную RSS-ленту с удобным интерфейсом. Мы будем использовать многоплатформенный RSS-ридер Fluent Reader.

Скачайте Fluent Reader для Linux, macOS или Windows с <https://github.com/yang991178/fluent-reader/releases>.

Установите Fluent Reader и откройте его. Вы увидите следующий ниже экран:



Рис. 3.19. *Fluent Reader* без источников новостных RSS-лент

В правом верхнем углу окна кликните по значку настроек. Вы увидите экран добавления источников новостных RSS-лент, подобный следующему ниже:



Рис. 3.20. Добавление новостной RSS-ленты в *Fluent Reader*

Введите `http://127.0.0.1:8000/blog/feed/` в поле **Add source** (Добавить источник) и кликните по кнопке **Add** (Добавить).

В таблице под формой вы увидите новую запись с новостной RSS-лентой блога, примерно в таком виде:

The screenshot shows the 'Sources' tab in the Fluent Reader application. At the top, there are tabs for 'Sources', 'Groups', 'Rules', 'Service', 'Preferences', and 'About'. Below the tabs, there's a section titled 'OPML File' with 'Import' and 'Export' buttons. A 'Add source' button is present, along with a text input field containing the URL 'http://127.0.0.1:8000/blog/feed/'. A table below lists the added source with columns for 'Name' and 'URL'. The table row for 'My blog' shows 'Name' as 'My blog' and 'URL' as 'http://127.0.0.1:8000/blog/feed/'.

Name	URL
My blog	http://127.0.0.1:8000/blog/feed/

Рис. 3.21. Источники новостных RSS-лент в Fluent Reader

Теперь вернитесь к главному экрану Fluent Reader. Вы должны увидеть посты, включенные в новостную RSS-ленту блога, как показано ниже:

The screenshot shows the main screen of Fluent Reader with several news items listed. At the top, there are navigation icons for search, refresh, and settings. Below the icons, there are five news cards. Each card includes a small image, the source name 'My blog', the publication time 'now', and a title. The titles and snippets are as follows:

- Markdown post**: 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 website ...
- Notes on Duke Ellington**: Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...
- Who was Miles Davis?**: Miles Davis was an American trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.
- Who was Django Reinhardt?**: Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...
- Another post**: Post body.

Рис. 3.22. Новостная RSS-лента блога в Fluent Reader

Кликните по посту, чтобы увидеть описание:

The screenshot shows a mobile-style interface for a blog post. At the top, it says "My blog". Below that is a title "Notes on Duke Ellington". Under the title is the date and time "1/3/2022, 2:19:33 PM". The main content is a truncated version of the post: "Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...". To the right of the content are several icons: a circle, a star, a grid, a globe, and three dots.

Рис. 3.23. Описание поста в Fluent Reader

Кликните по третьему значку в правом верхнем углу окна, чтобы загрузить полный контент страницы поста:

The screenshot shows the full content of the blog post. It includes the title "Notes on Duke Ellington", the date "1/3/2022, 2:19:33 PM", and the author information "Published Jan. 3, 2022, 1:19 p.m. by admin". The main content is the full text: "Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half a century." Below the content is a blue link "Share this post". A section titled "Similar posts" lists two other posts: "Who was Miles Davis?" and "Who was Django Reinhardt?". There is also a section for "1 comment" and a button to "Add a new comment". The top right corner of the screen shows the same set of icons as in the previous screenshot: circle, star, grid, globe, and three dots, with the grid icon being highlighted.

Рис. 3.24. Полный контент поста в Fluent Reader

Последним шагом будет добавление ссылки на подписку на новостную RSS-ленту в боковую панель блога.

Откройте шаблон `blog/base.html` и добавьте в него следующий ниже исходный код, выделенный жирным шрифтом:

```
{% 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>
        <p>
            <a href="{% url "blog:post_feed" %}">
                Subscribe to my RSS feed
            </a>
        </p>
        <h3>Latest posts</h3>
        {% show_latest_posts 3 %}
        <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>
    </div>
</body>
</html>
```

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере и взгляните на боковую панель. Новая ссылка приведет пользователей к новостной ленте блога:

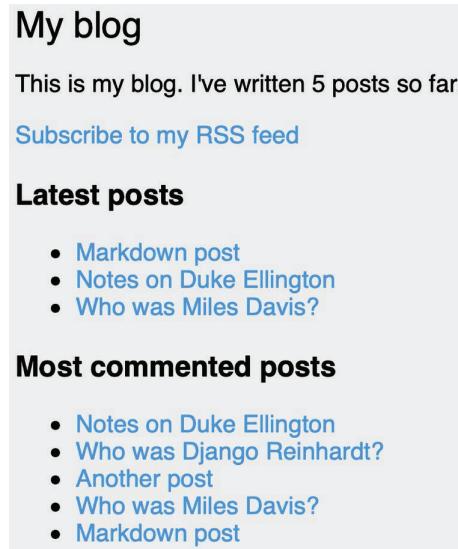


Рис. 3.25. Ссылка на подписку на новостную RSS-ленту, добавленная на боковую панель

Подробнее о встроеннем в Django фреймворке синдицированных новостных лент можно почитать на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/syndication/>.

Добавление полнотекстового поиска в блог

Далее мы добавим в блог возможности поиска. Поиск в базе данных на основе вводимых пользователем данных находит широкое применение в веб-приложениях. Встроенный в Django ORM-преобразователь позволяет выполнять простые операции сопоставления, используя, например, фильтр `contains` (или его нечувствительную к регистру версию `icontains`). Следующий ниже запрос можно использовать, чтобы найти посты, содержащие слово `framework` в их теле:

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

Однако если вы хотите выполнять сложные операции поиска, извлекая результаты по сходству или взвешивая термины на основе частоты их встречаемости в тексте или степени важности различных полей (например, релевантности термина, появляющегося в заголовке по сравнению с его появлением

теле), то необходимо задействовать полнотекстовый поисковый механизм. При рассмотрении больших блоков текста формирование запросов с операциями над цепочками символов становится недостаточным. Полнотекстовый поиск сравнивает фактические слова с сохраненным контентом, пытаясь удовлетворить критериям поиска.

Django предоставляет мощную функциональность поиска, построенную поверх функциональных возможностей реляционной базы данных PostgreSQL по полнотекстовому поиску. Модуль `django.contrib.postgres` предоставляет функциональности, предлагаемые базой данных PostgreSQL, которые не являются общими для других поддерживаемых веб-фреймворком Django баз данных. О поддержке принятого в PostgreSQL полнотекстового поиска можно узнать на странице <https://www.postgresql.org/docs/14/textsearch.html>.



Хотя веб-фреймворк Django не зависит от базы данных, он предоставляет модуль, который поддерживает часть богатого набора функциональных возможностей, предлагаемого базой данных PostgreSQL, которого нет в других поддерживаемых Django базах данных.

Установка базы данных PostgreSQL

В настоящее время в проекте `mysite` используется база данных SQLite. Поддержка полнотекстового поиска SQLite ограничена, и Django не поддерживает его прямо «из коробки». Однако PostgreSQL подходит для полнотекстового поиска гораздо лучше, и мы можем воспользоваться модулем `django.contrib.postgres`, чтобы задействовать возможности полнотекстового поиска PostgreSQL. Мы выполним миграции данных из SQLite в PostgreSQL, чтобы применить ее функциональные возможности полнотекстового поиска.



База данных SQLite приемлема для целей разработки. Однако для производственной среды вам понадобится более мощная база данных, такая как PostgreSQL, MariaDB, MySQL или Oracle.

Скачайте установщик PostgreSQL для macOS или Windows на странице <https://www.postgresql.org/download/>.

На той же странице вы найдете инструкции по установке PostgreSQL в различных дистрибутивах Linux. Следуйте инструкциям на веб-сайте по установке и запуску PostgreSQL.

Если вы используете macOS и решили установить PostgreSQL с помощью `Postgres.app`, то вам нужно будет настроить переменную `$PATH`, чтобы применить инструменты командной строки, как описано на странице <https://postgresapp.com/documentation/cli-tools.html>.

Вам также необходимо установить PostgreSQL-адаптер `psycopg2` для Python. Выполните следующую ниже команду в командной оболочке, чтобы его установить:

```
pip install psycopg2-binary==2.9.3
```

Создание базы данных PostgreSQL

Давайте создадим пользователя базы данных PostgreSQL. Мы будем использовать интерфейс PostgreSQL на основе терминала `psql`. Войдите в терминал PostgreSQL, выполнив следующую ниже команду в командной оболочке:

```
psql
```

Вы увидите следующий ниже результат:

```
psql (14.2)
Type "help" for help.
```

Введите следующую ниже команду, чтобы создать пользователя, который может создавать базы данных:

```
CREATE USER blog WITH PASSWORD 'xxxxxx';
```

Замените `xxxxxx` на желаемый пароль и выполните команду. Вы увидите такой результат:

```
CREATE ROLE
```

Пользователь был создан. Теперь давайте создадим базу данных `blog` и передадим права на владение этой базой данных только что созданному пользователю.

Исполните следующую ниже команду:

```
CREATE DATABASE blog OWNER blog ENCODING 'UTF8';
```

Этой командой мы сообщаем PostgreSQL, что нужно создать базу данных с именем `blog`, мы передаем право на владение базой данных `blog` ее пользователю, которого мы создали ранее, и указываем, что для новой базы данных должна использоваться кодировка UTF8. Вы увидите следующий ниже результат:

```
CREATE DATABASE
```

Мы успешно создали пользователя PostgreSQL и базу данных.

Выгрузка существующих данных

Перед переключением на другую базу данных необходимо выгрузить существующие в проекте Django данные из базы данных SQLite. Мы экспортируем данные, переключаем базу данных проекта на PostgreSQL и импортируем данные в новую базу данных.

Django предлагает простой способ загрузки и выгрузки данных из базы данных в файлы, которые называются **фикастурами**. Django поддерживает фикстуры в форматах JSON, XML или YAML. Мы собираемся создать фикстуру со всеми данными, содержащимися в базе данных.

Команда `dumpdata` выгружает данные из базы данных в стандартный вывод, по умолчанию сериализованный в формате JSON. Результирующая структура данных включает информацию о модели и ее полях, позволяя Django загружать ее в базу данных.

Выгружаемые данные можно ограничивать моделями приложения, предоставив команде имена приложений либо указав отдельные модели для вывода данных, используя формат `app.Model`. Также можно указывать формат, используя флаг `--format`. По умолчанию `dumpdata` выводит сериализованные данные в стандартный вывод. Однако, используя флаг `--output`, можно указать выходной файл. Флаг `--indent` позволяет указывать отступ. Дополнительную информацию об опциях команды `dumpdata` можно получить, выполнив команду `python manage.py dumpdata --help`.

Исполните следующую ниже команду из командной оболочки:

```
python manage.py dumpdata --indent=2 --output=mysite_data.json
```

Вы увидите результат, аналогичный приведенному ниже:

```
[.....]
```

Все существующие данные были экспортированы в формат JSON в новый файл с именем `mysite_data.json`. Содержимое файла можно просмотреть, чтобы увидеть структуру JSON, которая включает в себя разнообразные объекты данных различных моделей установленных вами приложений. Если при выполнении команды вы получаете ошибку кодировки, то включите флаг `-Xutf8`, как показано ниже, чтобы активировать режим Python UTF-8:

```
python -Xutf8 manage.py dumpdata --indent=2 --output=mysite_data.json
```

Теперь мы переключим базу данных в проекте Django, а затем импортируем данные в новую базу данных.

Переключение базы данных в проекте

Отредактируйте файл `settings.py` проекта, видоизменив настроечный параметр `DATABASES` и придав ему следующий вид. Новый исходный код выделен жирным шрифтом:

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

Замените `xxxxxx` паролем, который вы использовали при создании пользователя PostgreSQL. Новая база данных пуста.

Выполните следующую ниже команду, чтобы применить все миграции к новой базе данных PostgreSQL:

```
python manage.py migrate
```

Вы увидите результат, включая все миграции, которые были применены, как показано ниже:

```
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions, sites,
taggit
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002.Alter_permission_name_max_length... OK
  Applying auth.0003_Alter_user_email_max_length... OK
  Applying auth.0004_Alter_user_username_opts... OK
  Applying auth.0005_Alter_user_last_login_null... OK
  Applying auth.0006_Require_contenttypes_0002... OK
  Applying auth.0007_Alter_validators_add_error_messages... OK
  Applying auth.0008_Alter_user_username_max_length... OK
  Applying auth.0009_Alter_user_last_name_max_length... OK
  Applying auth.0010_Alter_group_name_max_length... OK
  Applying auth.0011_Update_proxy_permissions... OK
  Applying auth.0012_Alter_user_first_name_max_length... OK
  Applying taggit.0001_initial... OK
  Applying taggit.0002_auto_20150616_2121... OK
  Applying taggit.0003_taggeditem_add_unique_index... OK
  Applying blog.0001_initial... OK
  Applying blog.0002_Alter_post_slug... OK
  Applying blog.0003_comment... OK
  Applying blog.0004_post_tags... OK
  Applying sessions.0001_initial... OK
```

```
Applying sites.0001_initial... OK
Applying sites.0002_alter_domain_unique... OK
Applying taggit.0004_alter_taggeditem_content_type_alter_taggeditem_tag... OK
Applying taggit.0005_auto_20220424_2025... OK
```

Загрузка данных в новую базу данных

Выполните следующую ниже команду, чтобы загрузить данные в базу данных PostgreSQL:

```
python manage.py loaddata mysite_data.json
```

Вы увидите следующий ниже результат:

```
Installed 104 object(s) from 1 fixture(s)
```

Число объектов может отличаться в зависимости от пользователей, постов, комментариев и других объектов, которые были созданы в базе данных ранее.

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/blog/post/> в своем браузере, чтобы убедиться, что все записи были загружены в новую базу данных. Вы должны увидеть все посты, как показано ниже:

Select post to change					
<input type="text"/> <input type="button" value="Search"/>					
« 2022 January 1 January 2 January 3 January 22 »					
Action:	<input type="button" value="-----"/>	<input type="button" value="Go"/>	0 of 5 selected		
<input type="checkbox"/>	TITLE	SLUG	AUTHOR	PUBLISH	2 ▲ STATUS 1 ▲
<input type="checkbox"/>	Another post	another-post	admin	Jan. 1, 2022, 11:57 p.m.	Published
<input type="checkbox"/>	Who was Django Reinhardt?	who-was-django-reinhardt	admin	Jan. 1, 2022, 11:59 p.m.	Published
<input type="checkbox"/>	Who was Miles Davis?	who-was-miles-davis	admin	Jan. 2, 2022, 1:18 p.m.	Published
<input type="checkbox"/>	Notes on Duke Ellington	notes-on-duke-ellington	admin	Jan. 3, 2022, 1:19 p.m.	Published
<input type="checkbox"/>	Markdown post	markdown-post	admin	Jan. 22, 2022, 9:30 a.m.	Published

Рис. 3.26. Список постов на сайте администрирования

Простые операции поиска

Отредактируйте файл `settings.py` проекта, добавив `django.contrib.postgres` в настроечный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'blog.apps.BLogConfig',  
    'taggit',  
    'django.contrib.sites',  
    'django.contrib.sitemaps',  
    'django.contrib.postgres',  
]
```

Откройте оболочку Django, выполнив следующую ниже команду в системной командной оболочке:

```
python manage.py shell
```

Теперь можно выполнять поиск по одному полю, используя операцию `search` набора запросов `QuerySet`.

Выполните следующий ниже исходный код в оболочке Python:

```
>>> from blog.models import Post  
>>> Post.objects.filter(title__search='django')  
<QuerySet [<Post: Who was Django Reinhardt?>]>
```

В этом запросе PostgreSQL используется для того, чтобы создать поисковый вектор для поля `body` и поисковый запрос из термина `django`. Результаты получаются путем сопоставления запроса с вектором.

Поиск по нескольким полям

Возможно, вам захочется выполнить поиск по нескольким полям. В этом случае необходимо определить объект `SearchVector`. Давайте сформируем вектор, который позволит выполнять поиск по полям `title` и `body` модели `Post`.

Выполните следующий ниже исходный код в оболочке Python:

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

```
>>> Post.objects.annotate(
...     search=SearchVector('title', 'body'),
... ).filter(search='django')
<QuerySet [<Post: Markdown post>, <Post: Who was Django Reinhardt?>]>
```

Используя `annotate` и определяя `SearchVector` с обоими полями, предоставляется функциональность сопоставления запроса как с заголовком, так и с телом постов.



Полнотекстовый поиск – это интенсивный процесс. Если вы выполняете поиск среди нескольких сотен строк, то вам следует определить функциональный индекс, который сопоставляется с используемым поисковым индексом. Django предоставляет поле `SearchVectorField` для ваших моделей. Подробнее об этом можно почитать по адресу <https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/#performance>.

Разработка представления поиска

Теперь вы создадите конкретно-прикладное представление, позволяющее пользователям выполнять поиск постов. Прежде всего понадобится форма для поиска. Отредактируйте файл `forms.py` приложения `blog`, добавив следующую ниже форму:

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

Поле запроса будет использоваться для того, чтобы давать пользователям возможность вводить поисковые запросы. Отредактируйте файл `views.py` приложения `blog`, добавив следующий ниже исходный 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)
```

```
query = form.cleaned_data['query']
results = Post.published.annotate(
    search=SearchVector('title', 'body'),
).filter(search=query)

return render(request,
    'blog/post/search.html',
    {'form': form,
     'query': query,
     'results': results})
```

В приведенном выше представлении сначала создается экземпляр формы `SearchForm`. Для проверки того, что форма была передана на обработку, в словаре `request.GET` отыскивается параметр `query`. Форма отправляется методом GET, а не методом POST, чтобы результирующий URL-адрес содержал параметр `query` и им было легко делиться. После передачи формы на обработку создается ее экземпляр, используя переданные данные GET, и проверяется валидность данных формы. Если форма валидна, то с помощью конкретно-прикладного экземпляра `SearchVector`, сформированного с использованием полей `title` и `body`, выполняется поиск опубликованных постов.

Теперь представление поиска готово и необходимо создать шаблон отображения формы и результатов при выполнении пользователем поиска.

Внутри каталога `templates/blog/post/` создайте новый файл, назовите его `search.html` и добавьте в него следующий ниже исходный код:

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

{% 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|markdown|truncatewords_html:12 }}
{% empty %}
    <p>There are no results for your query.</p>
{% endfor %}
{% endif %}
```

```
{% endfor %}  
<p><a href="{% url "blog:post_search" %}">Search again</a></p>  
{% else %}  
    <h1>Search for posts</h1>  
    <form method="get">  
        {{ form.as_p }}  
        <input type="submit" value="Search">  
    </form>  
{% endif %}  
{% endblock %}
```

Как и в представлении поиска, по наличию параметра `query` определяется, что форма была передана на обработку. Перед передачей запроса мы отображаем форму и кнопку передачи формы. После передачи формы поиска на обработку отображается выполненный запрос, общее число результатов и список постов, совпадающих с поисковым запросом.

Наконец, отредактируйте файл `urls.py` приложения `blog`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [  
    # представления поста  
    path('', views.post_list, name='post_list'),  
    # path('', views.PostListView.as_view(), name='post_list'),  
    path('tag/<slug:tag_slug>',  
        views.post_list, name='post_list_by_tag'),  
    path('<int:year>/<int:month>/<int:day>/<slug:post>',  
        views.post_detail,  
        name='post_detail'),  
    path('<int:post_id>/share/',  
        views.post_share, name='post_share'),  
    path('<int:post_id>/comment/',  
        views.post_comment, name='post_comment'),  
    path('feed/', LatestPostsFeed(), name='post_feed'),  
    path('search/', views.post_search, name='post_search'),  
]
```

Далее пройдите по URL-адресу <http://127.0.0.1:8000/blog/search/> в своем браузере. Вы должны увидеть следующую ниже форму для поиска:

The screenshot shows a search interface. On the left, there is a text input field labeled "Query:" with a placeholder "Search posts..." and a blue "SEARCH" button below it. To the right, under the heading "My blog", is the text "This is my blog. I've written 5 posts so far." Below this is a link "Subscribe to my RSS feed". Under "Latest posts", there is a list of three items: "Markdown post", "Notes on Duke Ellington", and "Who was Miles Davis?". Under "Most commented posts", there is a list of five items: "Notes on Duke Ellington", "Who was Django Reinhardt?", "Another post", "Who was Miles Davis?", and "Markdown post".

Рис. 3.27. Форма с полем запроса для поиска постов

Ведите запрос и кликните по кнопке **SEARCH** (Найти). Вы увидите результаты поискового запроса, как показано ниже:

The screenshot shows the search results for the query "jazz". The title is "Posts containing \"jazz\"". It says "Found 3 results". The results are: "Notes on Duke Ellington" (with a description of Edward Kennedy "Duke" Ellington), "Who was Miles Davis?" (with a description of Miles Davis), and "Who was Django Reinhardt?" (with a description of Jean Reinhardt, known as Django). To the right, under "My blog", is the text "This is my blog. I've written 5 posts so far." Below this is a link "Subscribe to my RSS feed". Under "Latest posts", there is a list of three items: "Markdown post", "Notes on Duke Ellington", and "Who was Miles Davis?". Under "Most commented posts", there is a list of five items: "Notes on Duke Ellington", "Who was Django Reinhardt?", "Another post", "Who was Miles Davis?", and "Markdown post".

Рис. 3.28. Результаты поиска термина «jazz»

Поздравляю! Вы создали базовый поисковый механизм для своего блога.

Выделение основ слов и ранжирование результатов

Выделение основы слова, или стемминг, – это процесс приведения слов к их словообразовательной базе, основанию или корневой форме. Стемминг используется поисковыми системами для редуцирования индексированных слов до их основы и для того, чтобы иметь возможность сопоставлять изменяемые или производные слова. Например, слова *music*, *musical* и *musicality* поисковая система может считать похожими словами. В процессе выделения основы каждый поисковый токен нормализуется в лексему, или единицу лексического значения, которая лежит в основе набора слов, связанных посредством флексии. При создании поискового запроса слова *music*, *musical* и *musicality* будут преобразованы в *music*.

Django предоставляет класс `SearchQuery`, чтобы транслировать термины в объект поискового запроса. По умолчанию термины пропускаются через базовые алгоритмы выделения основ слов, что помогает получать более оптимальные совпадения.

Поисковый механизм PostgreSQL также удаляет стоп-слова, такие как *a*, *the*, *on* и *of*. Стоп-слова – это коллекция часто используемых слов в языке. Они удаляются при создании поискового запроса, поскольку появляются слишком часто, чтобы быть релевантными для поисковых запросов. Список стоп-слов, используемых PostgreSQL для английского языка, находится на странице <https://github.com/postgres/postgres/blob/master/src/backend/snowball/stopwords/english.stop>.

Мы также хотим упорядочивать результаты по релевантности. PostgreSQL предоставляет функцию ранжирования, которая упорядочивает результаты на основе частоты появления терминов запроса и степени их близости друг к другу.

Отредактируйте файл `views.py` приложения `blog`, добавив следующую ниже инструкцию импорта:

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

Затем отредактируйте представление `post_search`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
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']
```

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

return render(request,
    'blog/post/search.html',
    {'form': form,
     'query': query,
     'results': results})
```

В приведенном выше исходном коде создается объект `SearchQuery`, по нему фильтруются результаты, и для упорядочивания результатов по релевантности используется `SearchRank`.

Теперь можно перейти по адресу `http://127.0.0.1:8000/blog/search/` в своем браузере и протестировать различные поисковые запросы, чтобы проверить выделение основ слов и ранжирование. Ниже приведен пример ранжирования по числу появлений слова `django` в заголовке и теле постов:

The screenshot shows a search results page for the term "django". The main content area displays two posts: one about Jean Reinhardt and another post formatted with Markdown. The sidebar includes links to the blog's RSS feed and a list of latest posts.

Posts containing "django"

Found 2 results

Who was Django Reinhardt?
Jean Reinhardt, known to all by his Romani nickname Django, was a ...

Markdown post
This is a post formatted with markdown
This is emphasized and this ...

Search again

My blog
This is my blog. I've written 5 posts so far.
[Subscribe to my RSS feed](#)

Latest posts

- [Markdown post](#)
- [Notes on Duke Ellington](#)
- [Who was Miles Davis?](#)

Most commented posts

- [Notes on Duke Ellington](#)
- [Who was Django Reinhardt?](#)
- [Another post](#)
- [Who was Miles Davis?](#)
- [Markdown post](#)

Рис. 3.29. Результаты поиска по термину «django»

Выделение основ слов и удаление стоп-слов на разных языках

Объекты `SearchVector` и `SearchQuery` можно настроить под исполнение процедур выделения основ слов и удаления стоп-слов на любом языке. Для того чтобы использовать другую конфигурацию поиска, в `SearchVector` и `SearchQuery` передается атрибут `config`. Такой подход позволяет использовать раз-

ные языковые парсеры и словари. В следующем ниже примере выделяются основы слов и удаляются стоп-слова на испанском языке:

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

Используемый в PostgreSQL словарь испанских стоп-слов находится на странице <https://github.com/postgres/postgres/blob/master/src/backend/snowball/stopwords/spanish.stop>.

Взвешивание запросов

Влияние конкретных векторов можно усиливать таким образом, чтобы им придавался больший вес при упорядочивании результатов по релевантности. Например, взвешивание можно использовать, чтобы придавать большую релевантность постам, которые сочетаются по заголовку, а не по содержимому.

Отредактируйте файл `views.py` приложения `blog`, видоизменив представление `post_search`, как показано ниже. Новый исходный код выделен жирным шрифтом:

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

    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

В приведенном выше исходном коде к векторам поиска, сформированным с использованием полей `title` и `body`, применяются разные веса. По умолчанию веса таковы: D, C, B и A, и они относятся соответственно к числам 0.1, 0.2, 0.4 и 1.0. Вес 1.0 применяется к вектору поиска `title` (A), и вес 0.4 – к вектору `body` (B). Совпадения с заголовком будут преобладать над совпадениями с содержимым тела поста. Результаты фильтруются, чтобы отображать только те, у которых ранг выше 0.3.

Поиск по триграммному сходству

Еще одним подходом к поиску является триграммное сходство. Триграмма – это группа из трех следующих друг за другом символов. Сходство двух строковых литералов можно измерять, подсчитывая число общих для них триграмм. Во многих языках данный подход оказывается очень эффективным при измерении сходства слов.

Для того чтобы использовать триграммы в PostgreSQL, сначала необходимо установить расширение `pg_trgm`. Исполните следующую ниже команду в командной оболочке, чтобы подсоединиться к своей базе данных:

```
psql blog
```

Затем исполните следующую ниже команду, чтобы установить расширение `pg_trgm`:

```
CREATE EXTENSION pg_trgm;
```

Вы получите такой результат:

```
CREATE EXTENSION
```

Давайте отредактируем представление и видоизменим его под триграммный поиск.

Отредактируйте файл `views.py` приложения `blog`, добавив следующую ниже инструкцию импорта:

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

Затем видоизмените представление `post_search`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
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.published.annotate(
        similarity=TrigramSimilarity('title', query),
    ).filter(similarity__gt=0.1).order_by('-similarity')

return render(request,
              'blog/post/search.html',
              {'form': form,
               'query': query,
               'results': results})

```

Пройдите по URL-адресу <http://127.0.0.1:8000/blog/search/> в своем браузере и протестируйте различные варианты триграммного поиска. В следующем ниже примере показана гипотетическая опечатка в термине django, показаны результаты поиска термина yango:

The screenshot shows a search results page for the term "yango". The main content area displays a single result: "Who was Django Reinhardt?". Below the result is a link to "Search again". To the right, there's a sidebar with "My blog" information, a link to "Subscribe to my RSS feed", and sections for "Latest posts" and "Most commented posts", each listing five items.

Posts containing "yango"	
Found 1 result	
Who was Django Reinhardt?	
Jean Reinhardt, known to all by his Romani nickname Django, was a ...	
Search again	

My blog

This is my blog. I've written 5 posts so far.

[Subscribe to my RSS feed](#)

Latest posts

- [Markdown post](#)
- [Notes on Duke Ellington](#)
- [Who was Miles Davis?](#)

Most commented posts

- [Notes on Duke Ellington](#)
- [Who was Django Reinhardt?](#)
- [Another post](#)
- [Who was Miles Davis?](#)
- [Markdown post](#)

Рис. 3.30. Результаты поиска термина «yango»

В приложение для ведения блога был добавлен мощный поисковый механизм.

Более подробная информация о полнотекстовом поиске находится на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/>.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter03>.

- Приложение Django-taggit: <https://github.com/jazzband/django-taggit>.
- ORM-менеджеры приложения Django-taggit: <https://django-taggit.readthedocs.io/en/latest/api.html>.
- Взаимосвязи многие-ко-многим: https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/.
- Встроенные в Django функции агрегирования: <https://docs.djangoproject.com/en/4.1/topics/db/aggregation/>.
- Встроенные шаблонные теги и фильтры: <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/>.
- Написание конкретно-прикладных шаблонных тегов: <https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/>.
- Справочный материал по формату Markdown: <https://daringfireball.net/projects/markdown/basics>.
- Встроенный в Django фреймворк карт сайтов: <https://docs.djangoproject.com/en/4.1/ref/contrib/sitemaps/>.
- Встроенный в Django фреймворк сайтов Django: <https://docs.djangoproject.com/en/4.1/ref/contrib/sites/>.
- Встроенный в Django фреймворк синдицированных новостных лент: <https://docs.djangoproject.com/en/4.1/ref/contrib/syndication/>.
- Скачиваемые файлы PostgreSQL: <https://www.postgresql.org/download/>.
- Возможности полнотекстового поиска в PostgreSQL: <https://www.postgresql.org/docs/14/textsearch.html>.
- Поддержка со стороны Django полнотекстового поиска PostgreSQL: <https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/>.

Резюме

В этой главе вы реализовали систему тегирования, интегрировав стороннее приложение в свой проект. Вы генерировали рекомендуемые посты, используя сложные наборы запросов QuerySet. Вы также научились создавать конкретно-прикладные шаблонные теги и фильтры Django, чтобы обеспечивать шаблонам конкретно-прикладные функциональности. Создали карту сайта, чтобы поисковые системы имели возможность сканировать ваш сайт, и новостную RSS-ленту, дабы пользователи могли подписываться на ваш блог. Затем вы разработали в своем блоге поисковый механизм, используя полнотекстовый поисковый механизм PostgreSQL.

В следующей главе вы научитесь разрабатывать социальный веб-сайт с использованием встроенного в Django фреймворка аутентификации и реализовывать функциональности с применением учетных записей пользователей и конкретно-прикладные профили пользователей.

4

Разработка социального веб-сайта

В предыдущей главе вы научились реализовывать систему тегирования и рекомендовать схожие посты. Вы реализовали конкретно-прикладные шаблонные теги и фильтры. Вы также научились создавать карты сайта и новостные ленты для сайта и разработали полнотекстовый поисковый механизм с использованием базы данных PostgreSQL.

В этой главе вы научитесь разрабатывать функциональности с использованием учетной записи пользователя с целью создания социального сайта, включая регистрацию пользователя, управление паролем, редактирование профиля и аутентификацию. В последующих главах на этом сайте будут реализованы социальные функциональности, позволяющие пользователям деляться изображениями и взаимодействовать друг с другом. Пользователи смогут делать закладку на любое изображение в интернете и деляться им с другими пользователями. Они также смогут видеть внутриплатформенную активность пользователей, на которых они подписаны, и отмечать понравившиеся / не понравившиеся им изображения.

В этой главе будут рассмотрены следующие темы:

- создание представления входа в систему;
- использование встроенного в Django фреймворка аутентификации;
- создание шаблонов представлений входа в систему и выхода из системы, смены и сброса пароля;
- расширение модели пользователя с помощью конкретно-прикладной модели профиля;
- создание представлений регистрации пользователей;
- конфигурирование проекта под закачивание медиафайлов на сайт;
- использование фреймворка сообщений;
- разработка конкретно-прикладного бэкенда аутентификации;
- запрет на использование пользователями существующей электронной почты.

Давайте начнем с создания нового проекта.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter04>.

Все используемые в данной главе пакеты Python включены в файл requirements.txt в исходном коде к этой главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды pip install -r requirements.txt.

Создание проекта социального веб-сайта

Мы собираемся создать социальное приложение, и оно позволит пользователям делиться изображениями, которые они находят в интернете. В этом проекте нам потребуется разработать следующие элементы:

- система аутентификации, позволяющая пользователям регистрироваться, входить в систему, редактировать свой профиль, менять или сбрасывать пароль;
- система подписки, позволяющая пользователям подписываться друг на друга на сайте;
- функциональность отображения выложенных изображений и система, позволяющая пользователям делиться изображениями с любого веб-сайта;
- поток активности, который позволяет пользователям видеть контент, закачанный на сайт людьми, на которых они подписаны.

В этой главе будет рассмотрен первый пункт данного списка.

Запуск проекта социального веб-сайта

Откройте терминал и примените следующие ниже команды, чтобы создать виртуальную среду проекта:

```
mkdir env  
python -m venv env/bookmarks
```

Если вы используете Linux или macOS, то для активации виртуальной среды выполните следующую ниже команду:

```
source env/bookmarks/bin/activate
```

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

```
.\env\bookmarks\Scripts\activate
```

В командной оболочке будет отображена активная виртуальная среда, как показано ниже:

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

Следующей ниже командой установите Django в своей виртуальной среде:

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

Выполните следующую ниже команду, чтобы создать новый проект:

```
django-admin startproject bookmarks
```

Первоначальная структура проекта создана. С помощью следующих ниже команд войдите в каталог проекта и создайте новое приложение с именем account:

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

Напомним, что новое приложение добавляется в проект путем добавления имени приложения в настроечный параметр INSTALLED_APPS файла settings.py.

Отредактируйте файл settings.py, добавив в список INSTALLED_APPS следующую ниже строку, выделенную жирным шрифтом, перед всеми другими установленными приложениями:

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

Django просматривает шаблоны в каталогах template приложений в порядке их появления в настроечном параметре INSTALLED_APPS. Приложение django.contrib.admin содержит стандартные шаблоны аутентификации, которые мы будем переопределять в приложении account. Помещая приложение первым в настроечном параметре INSTALLED_APPS, мы обеспечиваем, что по умолчанию будут использоваться конкретно-прикладные шаблоны аутентификации, а не шаблоны аутентификации, содержащиеся в django.contrib.admin.

Выполните следующую ниже команду, чтобы синхронизировать базу данных с моделями стандартных приложений, включенных в настроечный параметр INSTALLED_APPS:

```
python manage.py migrate
```

Вы увидите, что будут применены все изначальные миграции базы данных Django. Далее в проект будет встроена система аутентификации, используя поставляемый с Django фреймворк аутентификации.

Использование поставляемого с Django фреймворка аутентификации

Django поставляется вместе со встроенным фреймворком аутентификации, который способен оперировать аутентификацией пользователей, сессиями, разрешениями и группами пользователей. Система аутентификации содержит представления обычных действий пользователя, таких как вход в систему, выход из системы, смена пароля и сброс пароля.

Фреймворк аутентификации находится в приложении `django.contrib.auth` и используется другими пакетами Django `contrib`. Напомним, что мы уже применяли фреймворк аутентификации в главе 1 «Разработка приложения для ведения блога», чтобы создать суперпользователя для приложения `blog` с целью доступа к сайту администрирования.

При создании нового проекта Django с помощью команды `startproject` фреймворк аутентификации вставляется в стандартные настройки проекта. Он состоит из приложения `django.contrib.auth` и следующих ниже двух классов промежуточных программных компонентов, которые находятся в настроечном параметре `MIDDLEWARE` проекта:

- `AuthenticationMiddleware`: ассоциирует пользователей с запросами с помощью сессий;
- `SessionMiddleware`: оперирует текущим сеансом во всех запросах.

Промежуточные программные компоненты состоят из классов с методами, которые исполняются глобально в фазе запроса или ответа. Классы промежуточных программных компонентов будут использоваться несколько раз на протяжении этой книги, а о том, как создавать конкретно-прикладной промежуточный программный компонент, вы узнаете в главе 17 «Выход в прямой эфир».

Фреймворк аутентификации также содержит следующие ниже модели, которые определены в `django.contrib.auth.models`:

- `User`: модель пользователя с базовыми полями; главные поля этой модели таковы: `username`, `password`, `email`, `first_name`, `last_name` и `is_active`;
- `Group`: модель группы для отнесения пользователей к категориям;
- `Permission`: флаги для пользователей или групп для выполнения ими определенных действий.

Фреймворк аутентификации также содержит стандартные представления и формы для аутентификации, которые будут использоваться позже.

Создание представления входа в систему

Мы начнем этот раздел с использования встроенного в Django фреймворка аутентификации с целью предоставления пользователям возможности входить в систему. Мы создадим представление, которое будет выполнять следующие ниже действия по входу пользователя в систему:

- показывать пользователю форму для входа в систему;
- получать пользовательское имя и пароль, предоставляемые пользователем при передаче формы на обработку;
- аутентифицировать пользователя по данным, хранящимся в базе данных;
- проверять, что пользователь активен;
- регистрировать вход пользователя в систему и начинать сеанс аутентификации.

Мы начнем с компоновки формы для входа в систему.

Внутри каталога приложения account создайте новый файл `forms.py` и добавьте в него следующие ниже строки:

```
from django import forms

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

Приведенная выше форма будет использоваться для аутентификации пользователей по базе данных. Обратите внимание, что для прорисовки HTML-элемента `password` используется виджет `PasswordInput`. Такой подход позволит вставлять `type="password"` в HTML, чтобы браузер воспринимал его как ввод пароля.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код:

```
from django.http import HttpResponseRedirect
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:
                login(request, user)
                return HttpResponseRedirect('/accounts/loggedin')
            else:
                return HttpResponseRedirect('/accounts/login')
    else:
        form = LoginForm()
    return render(request, 'account/login.html', {'form': form})
```

```
if user.is_active:  
    login(request, user)  
    return HttpResponseRedirect('Authenticated successfully')  
else:  
    return HttpResponseRedirect('Disabled account')  
else:  
    return HttpResponseRedirect('Invalid login')  
  
else:  
    form = LoginForm()  
return render(request, 'account/login.html', {'form': form})
```

Базовое представление входа в систему делает следующее.

При вызове представления `user_login` с запросом методом GET посредством инструкции `form = LoginForm()` создается экземпляр новой формы входа. Затем эта форма передается в шаблон.

Когда пользователь передает форму методом POST, выполняются следующие ниже действия:

- посредством инструкции `form = LoginForm(request.POST)` создается экземпляр формы с переданными данными;
- форма валидируется методом `form.is_valid()`. Если она невалидна, то ошибки формы будут выведены позже в шаблоне (например, если пользователь не заполнил одно из полей);
- если переданные на обработку данные валидны, то пользователь аутентифицируется по базе данных методом `authenticate()`. Указанный метод принимает объект `request`, параметры `username` и `password` и возвращает объект `User`, если пользователь был успешно аутентифицирован, либо `None` в противном случае. Если пользователь не был успешно аутентифицирован, то возвращается сырой ответ `HttpResponse` с сообщением **Invalid login** (Недопустимый логин);
- если пользователь успешно аутентифицирован, то статус пользователя проверяется путем обращения к атрибуту `is_active`. Указанный атрибут принадлежит модели `User` веб-фреймворка Django. Если пользователь не активен, то возвращается `HttpResponse` с сообщением **Disabled account** (Отключенная учетная запись);
- если пользователь активен, то он входит в систему. Пользователь задается в сеансе путем вызова метода `login()`. При этом возвращается сообщение **Authenticated successfully** (Аутентификация прошла успешно).



Обратите внимание на разницу между `authenticate()` и `login():authenticate()`: `authenticate()` проверяет учетные данные пользователя и возвращает объект `User`, если они правильны; `login()` задает пользователя в текущем сеансе.

Теперь мы создадим шаблон URL-адреса этого представления.

Внутри каталога приложения `account` создайте новый файл `urls.py` и добавьте в него следующий ниже исходный код:

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

urlpatterns = [
    path('login/', views.user_login, name='login'),
]
```

Отредактируйте расположенный в каталоге проекта `bookmarks` главный файл `urls.py`, импортировав `include` и добавив шаблоны URL-адресов приложения `account`, как показано ниже. Новый исходный код выделен жирным шрифтом:

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

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

Теперь появляется возможность обращаться к представлению входа в систему по URL-адресу.

Давайте создадим шаблон этого представления. Поскольку в проекте еще шаблонов нет, мы начнем с создания базового шаблона, который будет расширен шаблоном входа в систему.

Внутри каталога приложения `account` создайте следующие ниже файлы и каталоги:

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

Отредактируйте шаблон `base.html`, добавив следующий ниже исходный код:

```
{% load static %}
<!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 веб-сайта. Как и в предыдущем проекте, вставьте в базовый шаблон стили CSS. Статические файлы CSS находятся в приложении к данной главе исходном коде. Скопируйте каталог `static/` приложения `account` из исходного кода главы в то же место в вашем проекте, чтобы иметь возможность использовать статические файлы. Содержимое каталога находится на странице <https://github.com/PacktPublishing/Django-4-by-Example/tree/master/Chapter04/bookmarks/account/static>.

В базовом шаблоне определяется блок `title` и блок `content`. Расширяющие его шаблоны заполняют эти блоки контентом.

Давайте заполним шаблон формы входа в систему.

Откройте шаблон `account/login.html` и добавьте в него следующий ниже исходный код:

```
{% 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 method="post">
        {{ form.as_p }}
        {% csrf_token %}
        <p><input type="submit" value="Log in"></p>
    </form>
```

Приведенный выше шаблон содержит форму, экземпляр которой создается в представлении. Поскольку эта форма будет передаваться на обработку методом POST, вставляется и шаблонный тег `{% csrf_token %}`, чтобы защищаться от подделки межсайтовых запросов (CSRF). Вы узнали о защите от CSRF в главе 2 «Усовершенствование блога за счет продвинутых функциональностей».

В базе данных пока пользователей нет. Сначала необходимо создать суперпользователя, чтобы получить доступ к сайту администрирования и управлять другими пользователями.

Исполните следующую ниже команду в командной оболочке:

```
python manage.py createsuperuser
```

Вы увидите следующее ниже сообщение. Введите желаемое пользовательское имя, электронную почту и пароль, как показано ниже:

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

После этого вы увидите такое сообщение об успехе:

```
Superuser created successfully.
```

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/` в своем браузере. Войдите на сайт администрирования, используя учетные данные только что созданного вами пользователя. Вы увидите встроенный в Django сайт администрирования, включая модели User и Group фреймворка аутентификации. Он будет выглядеть, как показано ниже:

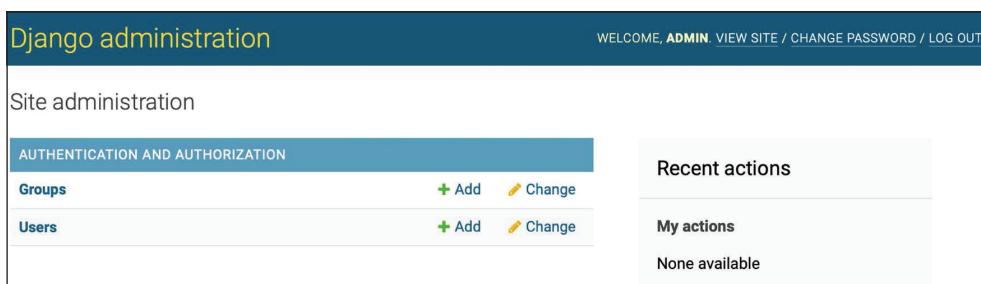


Рис. 4.1. Индексная страница встроенного в Django сайта администрирования, включая Users и Groups

В строке **Users** (Пользователи) кликните по ссылке **Add** (Добавить). Создайте нового пользователя с помощью сайта администрации, как показано ниже:

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

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.

Password confirmation: Enter the same password as before, for verification.

Save and add another **Save and continue editing** **SAVE**

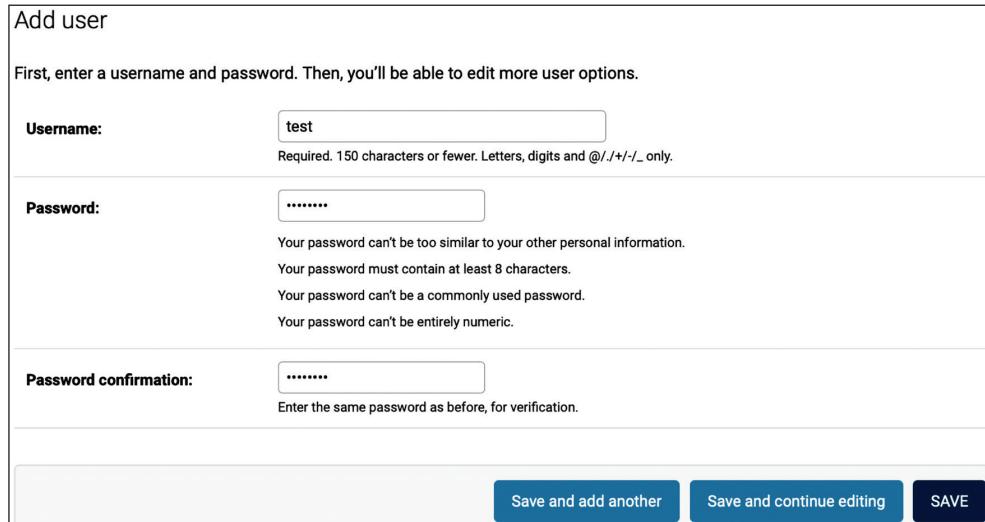


Рис. 4.2. Форма Add user (Добавить пользователя) на сайте администрирования

Ведите данные пользователя и кликните по кнопке **SAVE** (Сохранить), чтобы сохранить нового пользователя в базе данных.

Затем в разделе **Personal info** (Личная информация) заполните поля **First name** (Имя), **Last name** (Фамилия) и **Email address** (Адрес электронной почты), как показано ниже, и кликните по кнопке **Save** (Сохранить), чтобы сохранить изменения:

Personal info

First name:

Last name:

Email address:



Рис. 4.3. Форма редактирования пользователя на сайте администрирования

Пройдите по URL-адресу <http://127.0.0.1:8000/account/login/> в своем браузере. Вы должны увидеть прорисованный шаблон, включая форму входа в систему:

The screenshot shows a web browser window with the URL `127.0.0.1:8000/account/login/`. The title bar says "Bookmarks". The main content area has a heading "Log-in". Below it, a message says "Please, use the following form to log-in:". There are two input fields: one for "Username" and one for "Password", both of which are currently empty and grayed out. At the bottom is a green "LOG IN" button.

Рис. 4.4. Страница входа пользователя в систему

Введите недопустимые учетные данные и передайте форму на обработку. Вы должны получить следующий ответ **Invalid login** (Недопустимый логин):



Рис. 4.5. Ответ в виде простого текста о недопустимом логине

Введите допустимые учетные данные; вы получите следующий ниже ответ **Authenticated successfully** (Аутентификация прошла успешно):



Рис. 4.6. Ответ в виде простого текста об успешной аутентификации

Вы научились аутентифицировать пользователей и создавать свое собственное представление аутентификации. Хотя можно разрабатывать свои собственные представления аутентификации, Django имеет готовые и слушающие для этой цели представления, которые можно задействовать в своих проектах.

Использование встроенных в Django представлений аутентификации

Во встроенном Django фреймворке аутентификации содержится несколько форм и представлений, которые можно использовать сразу же. Созданное представление входа является хорошим упражнением на понимание процесса аутентификации пользователей в Django. Однако в большинстве случаев можно использовать стандартные представления аутентификации.

Django предоставляет следующие представления на основе классов для работы с аутентификацией. Все они расположены в `django.contrib.auth.views`:

- `LoginView`: оперирует формой входа и регистрирует вход пользователя;
- `LogoutView`: регистрирует выход пользователя.

Django предоставляет следующие ниже представления для оперирования сменой пароля:

- `PasswordChangeView`: оперирует формой для смены пароля пользователя;
- `PasswordChangeDoneView`: представление страницы об успехе, на которую пользователь перенаправляется после успешной смены пароля.

Django также содержит следующие ниже представления, позволяющие пользователям сбрасывать свой пароль:

- `PasswordResetView`: позволяет пользователям сбрасывать свой пароль. Генерирует одноразовую ссылку с токеном и отправляет ее на электронный ящик пользователя;
- `PasswordResetDoneView`: сообщает пользователям, что им было отправлено электронное письмо, содержащее ссылку на сброс пароля;
- `PasswordResetConfirmView`: позволяет пользователям устанавливать новый пароль;
- `PasswordResetCompleteView`: представление страницы об успехе, на которую пользователь перенаправляется после успешного сброса пароля.

Описанные выше представления сэкономят время при разработке любого веб-приложения с использованием учетных записей пользователей. В указанных представлениях используются стандартные значения, которые можно переопределить, например расположение прорисовываемого шаблона или используемая в представлении форма.

Более подробную информацию о встроенных представлениях аутентификации можно получить по адресу <https://docs.djangoproject.com/en/4.1/topics/auth/default/#all-authentication-views>.

Представления входа и выхода

Отредактируйте файл `urls.py` приложения `account`, добавив исходный код, выделенный жирным шрифтом:

```

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

urlpatterns = [
    # предыдущий url входа
    # path('login/', views.user_login, name='login'),

    # url-адреса входа и выхода
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]

```

В приведенном выше исходном коде закомментирован шаблон URL-адреса созданного ранее представления `user_login`. Теперь будет использоваться представление `LoginView` из фреймворка аутентификации. Также был добавлен шаблон URL-адреса представления `LogoutView`.

Создайте новый каталог внутри каталога `templates/` приложения `account` и назовите его `registration`. Это стандартный путь, по которому представления аутентификации будут обращаться к шаблонам аутентификации, если не указан иной.

Модуль `django.contrib.admin` вставляет используемые на сайте администрирования шаблоны аутентификации, например шаблон входа. Поместив приложение `account` в начало настроечного параметра `INSTALLED_APPS` при конфигурировании проекта, мы обеспечили, чтобы вместо шаблонов аутентификации, определенных в любом другом приложении, Django использовал наши шаблоны аутентификации.

Внутри каталога `templates/registration/` создайте новый файл, назовите его `login.html` и добавьте в него следующий ниже исходный 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 входа очень похож на созданный ранее. По умолчанию в Django используется форма `AuthenticationForm`, расположенная в `django.contrib.auth.forms`. Эта форма пытается аутентифицировать пользователя и выдает ошибку валидации, если вход оказался безуспешным. В шаблоне используется тег `{% if form.errors %}`, чтобы выполнять проверку на ошибочность предоставленных учетных данных.

Мы добавили скрытый HTML-элемент `<input>`, чтобы передавать на обработку значение переменной `next`. Если передать параметр с именем `next` в запрос, например обращаясь по `http://127.0.0.1:8000/account/login/?next=/account/`, то эта переменная передается представлению входа.

Следующим параметром должен быть URL-адрес. Если этот параметр задан, то встроенное в Django представление входа будет перенаправлять пользователя на указанный URL-адрес после успешного входа.

Теперь внутри каталога `templates/registration/` создайте шаблон `logged_out.html` и придайте ему следующий вид:

```
{% 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 %}
```

Это шаблон, который будет отображаться после выхода пользователя.

Мы добавили шаблоны URL-адресов и шаблоны представлений входа и выхода. Теперь пользователи могут входить и выходить, используя встроенные в Django представления аутентификации.

Сейчас мы создадим новое представление с целью отображения информационной панели при входе пользователей в свои учетные записи.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код:

```
from django.contrib.auth.decorators import login_required
```

```
@login_required
def dashboard(request):
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard'})
```

Мы создали представление `dashboard` и применили к нему декоратор `login_required` из фреймворка аутентификации. Декоратор `login_required` проверяет аутентификацию текущего пользователя.

Если пользователь аутентифицирован, то оно исполняет декорированное представление; если пользователь не аутентифицирован, то оно перенаправляет пользователя на URL-адрес входа с изначально запрошенным URL-адресом в качестве GET-параметра с именем `next`.

При таком подходе представление входа перенаправляет пользователей на URL-адрес, к которому они пытались обратиться после успешного входа. Напомним, что с этой целью в шаблон входа был добавлен скрытый HTML-элемент `<input>` с именем `next`.

Мы также определили переменную `section`. Эта переменная будет использоваться для подсвечивания текущего раздела в главном меню сайта.

Далее необходимо создать шаблон представления `dashboard`.

Внутри каталога `templates/account/` создайте новый файл и назовите его `dashboard.html`. Добавьте в него следующий ниже исходный код:

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

{% block title %}Dashboard{% endblock %}

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

Отредактируйте файл `urls.py` приложения `account`, добавив следующий ниже шаблон URL-адреса представления. Новый исходный код выделен жирным шрифтом:

```
urlpatterns = [
    # предыдущий url-адрес входа
    # path('login/', views.user_login, name='login'),

    # url-адреса входа и выхода
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

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

Отредактируйте файл `settings.py` проекта, добавив следующий ниже исходный код:

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

Мы определили следующие настроочные параметры:

- `LOGIN_REDIRECT_URL`: сообщает Django URL-адрес, на который следует перенаправлять пользователя после успешного входа, если в запросе нет параметра `next`;
- `LOGIN_URL`: URL-адрес, на который следует перенаправлять пользователя, чтобы зарегистрировать его вход (например, представления, в которых используется декоратор `login_required`);
- `LOGOUT_URL`: URL-адрес, на который следует перенаправлять пользователя, чтобы зарегистрировать его выход.

В шаблонах URL-адресов были использованы имена URL-адресов, которые были определены ранее с помощью атрибута `name` функции `path()`. Вместо имен URL-адресов в этих настроочных параметрах также можно использовать жестко привязанные URL-адреса.

Давайте подведем итоги того, что мы сделали к этому моменту:

- в проект были добавлены встроенные во фреймворк аутентификации представления входа и выхода;
- для обоих представлений были созданы конкретно-прикладные шаблоны и определено простое представление информационной панели, куда перенаправлять пользователей после их входа;
- наконец, добавлены настроочные параметры, чтобы Django использовал эти URL-адреса по умолчанию.

Теперь надо добавить ссылки на страницы входа и выхода в базовый шаблон. Для этого необходимо выяснить, вошел ли текущий пользователь в систему или нет, чтобы отображать надлежащую ссылку, соответствующую каждому случаю. Текущий пользователь задается в объекте `HttpRequest` промежуточным компонентом аутентификации. К нему можно обращаться через `request.user`. Объект `User` находится в запросе, даже если пользователь не аутентифицирован. Не прошедший аутентификацию пользователь задается в запросе как экземпляр `AnonymousUser`. Самый лучший способ проверить аутентификацию текущего пользователя – обратиться к доступному только для чтения атрибуту `is_authenticated`.

Отредактируйте шаблон `templates/base.html`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
{% load static %}  
<!DOCTYPE html>  
<html>  
<head>  
<title>{% block title %}{% endblock %}</title>
```

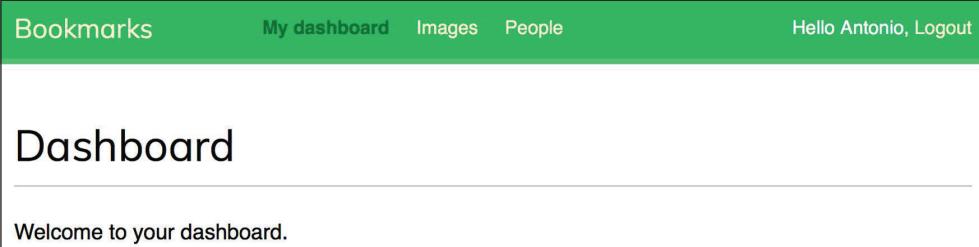
```

<link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
    <div id="header">
        <span class="logo">Bookmarks</span>
        {% 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|default:request.user.username }},
                <a href="{% url "logout" %}">Logout</a>
            {% else %}
                <a href="{% url "login" %}">Log-in</a>
            {% endif %}
        </span>
    </div>
    <div id="content">
        {% block content %}
        {% endblock %}
    </div>
</body>
</html>

```

Меню сайта отображается только для аутентифицированных пользователей. При этом проверяется переменная `section`, чтобы добавить атрибут `selected` CSS-класса в списоковый пункт `` меню; указанный пункт меню относится к текущему разделу. Благодаря этому соответствующий текущему разделу пункт меню будет выделяться с помощью CSS. Если пользователь аутентифицирован, то отображается настоящее имя пользователя и ссылка на страницу выхода; в противном случае отображается ссылка на страницу входа. Если настоящее имя пользователя является пустым, то вместо него отображается пользовательское имя (`username`) при помощи `request.user.first_name|default:request.user.username`.

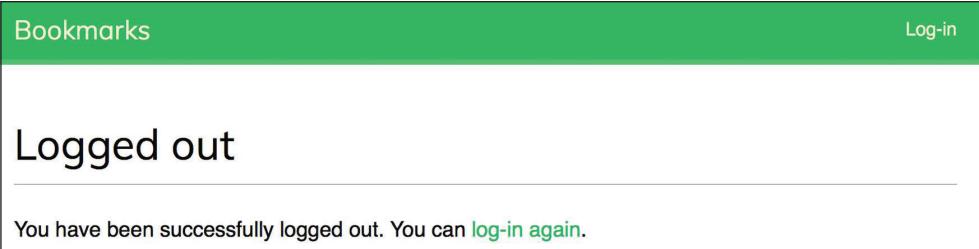
Пройдите по URL-адресу `http://127.0.0.1:8000/account/login/` в своем браузере. Вы должны увидеть страницу входа. Введите допустимое пользовательское имя и пароль и кликните по кнопке **Log-in** (Войти). Вы должны увидеть следующий ниже экран:



The screenshot shows the Django admin interface. At the top, there's a green header bar with the following navigation items: 'Bookmarks', 'My dashboard' (which is highlighted in blue), 'Images', 'People', and 'Hello Antonio, Logout'. Below the header, the main content area has a title 'Dashboard' and a message 'Welcome to your dashboard.'

Рис. 4.7. Страница информационной панели

Пункт меню **My dashboard** (Моя информационная панель) выделен с помощью CSS, так как имеет класс `selected`. Поскольку пользователь аутентифицирован, его имя отображается в правой части заголовка. Кликните по ссылке **Logout** (Выйти). Вы должны увидеть следующую ниже страницу:



The screenshot shows a simple page with a green header bar containing 'Bookmarks' and 'Log-in'. The main content area displays the message 'Logged out' and 'You have been successfully logged out. You can [log-in again](#)'.

Рис. 4.8. Страница информации о выходе из системы

На этой странице видно, что пользователь вышел из системы, и, следовательно, меню сайта не отображается. Теперь отображаемая в правой части заголовка ссылка называется **Log-in** (Войти).



Если вместо своей собственной страницы Logged out (Зарегистрирован выход) вы видите страницу Logged out сайта администрирования, то следует обратиться к настроенному параметру `INSTALLED_APPS` проекта и проверить, чтобы приложение `django.contrib.admin` располагалось после приложения `account`. Оба приложения содержат шаблоны выхода, которые расположены по одному и тому же относительному пути. Загрузчик шаблонов Django будет прокручивать различные приложения в списке `INSTALLED_APPS` и использовать первый найденный шаблон.

Представления смены пароля

Далее необходимо обеспечить пользователям возможность менять свой пароль после входа в систему. Мы выполним интеграцию встроенных в Django представлений аутентификации, предназначенных для смены пароля.

Откройте файл urls.py приложения account и добавьте следующие ниже шаблоны URL-адресов, выделенные жирным шрифтом:

```
urlpatterns = [
    # предыдущий url-адрес входа
    # path('login/', views.user_login, name='login'),
    # url-адреса входа и выхода
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # url-адреса 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'),

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

Представление PasswordChangeView будет работать с формой для смены пароля, а представление PasswordChangeDoneView будет отображать сообщение об успехе, после того как пользователь успешно сменит свой пароль. Давайте создадим шаблон каждого представления.

Добавьте новый файл в каталог templates/registration/ приложения account и назовите его password_change_form.html. Добавьте в него следующий ниже исходный код:

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

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

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

Шаблон password_change_form.html содержит форму для смены пароля. Теперь в том же каталоге создайте еще один файл и назовите его password_change_done.html. Добавьте в него следующий ниже исходный 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 только сообщение об успехе, которое будет отображаться, когда пользователь успешно сменит свой пароль.

Пройдите по URL-адресу `http://127.0.0.1:8000/account/password-change/` в своем браузере. Если вы не вошли, то браузер перенаправит на страницу **Log-in** (Войти). После успешной аутентификации вы увидите следующую ниже страницу смены пароля:

The screenshot shows a web browser window with a green header bar. The header contains navigation links: 'Bookmarks', 'My dashboard', 'Images', 'People', and 'Hello Antonio, Logout'. Below the header, the main content area has a title 'Change your password' and a sub-instruction 'Use the form below to change your password.' There are four input fields for password information, each represented by a light gray rectangular box. Below these fields is a bulleted list of password requirements. At the bottom of the form is a large green button labeled 'CHANGE'.

```
Bookmarks My dashboard Images People Hello Antonio, Logout

Change your password

Use the form below to change your password.

Old password:
[Redacted input field]

New password:
[Redacted input field]

• 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:
[Redacted input field]

CHANGE
```

Рис. 4.9. Форма для смены пароля

Заполните форму текущим паролем и новым паролем и кликните по кнопке **CHANGE** (Сменить).

Вы увидите следующую ниже страницу успешной смены пароля:



Рис. 4.10. Страница успешной смены пароля

Выйдите и снова войдите, используя новый пароль, чтобы убедиться, что все работает так, как и ожидалось.

Представление сброса пароля

Отредактируйте файл `urls.py` приложения `account`, добавив следующие ниже шаблоны URL-адресов, выделенные жирным шрифтом:

```
urlpatterns = [
    # path('login/', views.user_login, name='login'),

    # url-адреса входа и выхода
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # url-адреса 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'),

    # url-адреса сброса 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('password-reset/<uidb64>/<token>/',
      auth_views.PasswordResetConfirmView.as_view(),
      name='password_reset_confirm'),
path('password-reset/complete/',
      auth_views.PasswordResetCompleteView.as_view(),
      name='password_reset_complete'),

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

Добавьте новый файл в каталог `templates/registration/` приложения `account` и назовите его `password_reset_form.html`. Добавьте в него следующий ниже исходный код:

```
{% 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 method="post">
{{ form.as_p }}
<p><input type="submit" value="Send e-mail"></p>
{{ csrf_token }}
</form>
{% endblock %}
```

Теперь в том же каталоге создайте еще один файл и назовите его `password_reset_email.html`. Добавьте в него следующий ниже исходный код¹:

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

Шаблон `password_reset_email.html` будет использоваться для отображения отправляемого пользователям электронного письма, чтобы сбросить свой пароль. Оно содержит токен сброса, который генерируется представлением.

В том же каталоге создайте еще один файл и назовите его `password_reset_done.html`. Добавьте в него следующий ниже исходный код:

¹ Кто-то сделал запрос на сброс пароля электронной почты {}. Пройдите по ссылке {}. Ваше пользовательское имя, если вы забыли: {}. – Прим. перев.

```
{% 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 %}
```

В том же каталоге создайте еще один шаблон и назовите его `password_reset_confirm.html`. Добавьте в него следующий ниже исходный код:

```
{% 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 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 %}
```

В этом шаблоне мы подтверждаем валидность/невалидность ссылки на сброс пароля, проверяя переменную `validlink`. Представление `PasswordResetConfirmView` проверяет валидность указанного в URL-адресе токена и передает переменную `validlink` в шаблон. Если ссылка валидна, то отображается форма для сброса пароля пользователя. Пользователи могут устанавливать новый пароль, только если у них есть валидная ссылка на сброс пароля.

Создайте еще один шаблон и назовите его `password_reset_complete.html`. Введите в него следующий ниже исходный код:

```
{% 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 шаблон `registration/login.html` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

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

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/account/login/` в своем браузере. На странице входа должна появиться ссылка на страницу сброса пароля, как показано ниже:

Bookmarks

Log-in

Log-in

Please, use the following form to log-in:

Username:

Password:

LOG-IN

[Forgotten your password?](#)

Рис. 4.11. Страница входа, включающая ссылку на страницу сброса пароля

Кликните по ссылке **Forgotten your password?** (Забыли свой пароль?). Вы увидите следующую ниже страницу:

Bookmarks

Log-in

Forgotten your password?

Enter your e-mail address to obtain a new password.

Email:

SEND E-MAIL

Рис. 4.12. Форма для восстановления пароля

На данном этапе нужно добавить конфигурацию простого протокола передачи почты (**SMTP**) в файл `settings.py` проекта, чтобы Django мог отправлять электронные письма. В главе 2 «Усовершенствование блога за счет продвинутых функциональностей» вы научились добавлять в проект настроечные па-

раметры электронной почты. Однако во время разработки имеется возможность конфигурировать Django таким образом, чтобы он писал электронные письма в стандартный вывод, а не отправлял их через SMTP-сервер. Django предоставляет почтовый бэкенд, позволяющий писать письма в консоль.

Отредактируйте файл `settings.py` проекта, добавив следующую ниже строку:

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

Настроечный параметр `EMAIL_BACKEND` указывает класс, который будет использоваться для отправки электронной почты.

Вернитесь в браузер, введите адрес электронной почты существующего пользователя и кликните по кнопке **SEND E-MAIL** (Отправить почту). Вы должны увидеть следующую ниже страницу:



Рис. 4.13. Страница отправки сообщения электронной почты о сбросе пароля

Загляните в командную оболочку, в которой работает сервер разработки. Вы увидите сгенерированное электронное письмо, как показано ниже:

```
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: test@gmail.com
Date: Mon, 10 Jan 2022 19:05:18 -0000
Message-ID: <162896791878.58862.14771487060402279558@MBP-amele.local>
```

```
Someone asked for password reset for email test@gmail.com. Follow the link below:
http://127.0.0.1:8000/account/password-reset/MQ/ardx0u-
b4973cfa2c70d652a190e79054bc479a/
Your username, in case you've forgotten: test
```

Письмо прорисовывается с помощью созданного ранее шаблона `password_reset_email.html`. URL-адрес сброса пароля содержит токен, который Django генерировал динамически.

Скопируйте URL-адрес из письма, который должен выглядеть примерно так: <http://127.0.0.1:8000/account/password-reset/MQ/argdx0u-b4973cfa2c-70d652a190e79054bc479a/>, и откройте его в браузере. Вы должны увидеть следующую ниже страницу:

The screenshot shows a web page titled "Reset your password". At the top, there is a green header bar with the word "Bookmarks" on the left and "Log-in" on the right. Below the header, the main content area has a white background. It starts with the text "Please enter your new password twice:". Underneath this, there are two input fields labeled "New password:" and "New password confirmation:", both represented by light gray rectangular boxes. To the right of these fields is a vertical list of four bullet points: "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 of the form is a green rectangular button with the white text "CHANGE MY PASSWORD".

Рис. 4.14. Форма для сброса пароля

На странице установки нового пароля используется шаблон `password_reset_confirm.html`. Введите новое парольное слово и кликните по кнопке **CHANGE MY PASSWORD** (Сменить пароль). Django создаст новый хешированный пароль и сохранит его в базе данных. Вы увидите следующую ниже страницу с сообщением об успехе:

The screenshot shows a web page titled "Password set". At the top, there is a green header bar with the word "Bookmarks" on the left and "Log-in" on the right. Below the header, the main content area has a white background. It contains a single line of text: "Your password has been set. You can [log in now](#)".

Рис. 4.15. Страница успешного сброса пароля

Теперь можно снова войти в учетную запись пользователя с новым паролем.

Каждый токен для установки нового пароля можно использовать только один раз. Если вы откроете полученную ссылку еще раз, то получите сообщение о том, что токен недействителен.

Мы только что интегрировали в проект представления, встроенные в фреймворк аутентификации. Эти представления подходят для большинства случаев. Однако если требуется другое поведение, то существует возможность создавать свои собственные представления.

Django предоставляет шаблоны URL-адресов представлений аутентификации, которые эквивалентны тем, которые мы только что создали. Мы заменим указанные шаблоны аутентификации на те, которые имеются в Django.

Закомментируйте шаблоны URL-адресов аутентификации, которые были добавлены в файл `urls.py` приложения `account`, и вместо них вставьте `django.contrib.auth.urls`, как показано ниже. Новый исходный код выделен жирным шрифтом:

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

urlpatterns = [
    # предыдущий url-адрес входа
    # path('login/', views.user_login, name='login'),

    # url-адреса входа и выхода
    # path('login/', auth_views.LoginView.as_view(), name='login'),
    # path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # url-адреса 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'),

    # url-адреса сброса 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('password-reset/<uidb64>/<token>/',
    #     auth_views.PasswordResetConfirmView.as_view(),
    #     name='password_reset_confirm'),
    # path('password-reset/complete/',
    #     auth_views.PasswordResetCompleteView.as_view()),
```

```
#     name='password_reset_complete'),  
  
    path('', include('django.contrib.auth.urls')),  
    path('', views.dashboard, name='dashboard'),  
]
```

Встроенные шаблоны URL-адресов аутентификации можно посмотреть на странице <https://github.com/django/django/blob/stable/4.0.x/django/contrib/auth/urls.py>.

Теперь в проект были добавлены все необходимые представления аутентификации. Далее мы реализуем регистрацию пользователей.

Регистрация пользователей и профили пользователей

Теперь пользователи сайта могут входить в систему и выходить из системы, менять и сбрасывать пароль. Однако еще необходимо скомпоновать представление, позволяющее посетителям создавать учетную запись пользователя.

Регистрация пользователя

Давайте создадим простое представление регистрации пользователей в системе. Сначала нужно создать форму, чтобы пользователь мог вводить пользовательское имя, свое настоящее имя и пароль.

Отредактируйте файл `forms.py`, расположенный в каталоге приложения `account`, добавив в него следующие ниже строки, выделенные жирным шрифтом:

```
from django import forms  
from django.contrib.auth.models import User  
  
class LoginForm(forms.Form):  
    username = forms.CharField()  
    password = forms.CharField(widget=forms.PasswordInput)  
  
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']
```

Здесь была создана модельная форма для модели пользователя. Данная форма содержит поля `username`, `first_name` и `email` модели `User`. Указанные поля будут валидироваться в соответствии с проверками на валидность соответствующих полей модели. Например, если пользователь выберет пользовательское имя, которое уже существует, то он получит ошибку валидации, поскольку пользовательское имя – это поле, определенное с использованием `unique=True`.

Далее были добавлены два дополнительных поля – `password` и `password2`, – для того чтобы пользователи могли устанавливать пароль и повторять его. Давайте добавим валидацию полей, чтобы проверить, что оба пароля одинаковы.

Отредактируйте файл `forms.py` в приложении учетной записи, добавив в класс `UserRegistrationForm` следующий ниже метод `clean_password2()`. Новый исходный код выделен жирным шрифтом:

```
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 определили метод `clean_password2()`, чтобы сравнивать второй пароль с первым и выдавать ошибки валидации, если пароли не совпадают. Этот метод исполняется, когда форма проходит валидацию путем вызова ее метода `is_valid()`. Метод `clean_<fieldname>()` можно предоставлять любому полю формы, чтобы очищать значение или вызывать ошибку валидации формы для конкретного поля. Формы также содержат общий метод `clean()`, чтобы валидировать всю форму целиком, что бывает удобно при валидации полей, которые зависят друг от друга. В данном случае вместо переопределения метода `clean()` формы мы используем специфическую для поля валидацию `clean_password2()`. Такой подход позволяет избегать переопределения других специфических для полей проверок, которые `ModelForm` получает из ограничений, установленных в модели (например, валидация уникальности пользовательского имени `username`).

Django также предоставляет форму `UserCreationForm`, которая находится в `django.contrib.auth.forms` и очень похожа на созданную нами ранее.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.contrib.auth import authenticate, login
from django.contrib.auth.decorators import login_required
from .forms import LoginForm, UserRegistrationForm

# ...

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Создать новый объект пользователя,
            # но пока не сохранять его
            new_user = user_form.save(commit=False)
            # Установить выбранный пароль
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Сохранить объект User
            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})
```

Представление создания учетных записей пользователей выглядит довольно просто. В целях безопасности вместо сохранения введенного пользователем необработанного пароля мы используем метод `set_password()` модели `User`. Данный метод хеширует пароль перед его сохранением в базе данных.

Django не хранит открытые текстовые пароли; вместо этого он хранит хешированные пароли. Хеширование – это процесс преобразования заданного ключа в другое значение. Хеш-функция используется для генерирования значения фиксированной длины в соответствии с математическим алгоритмом. Благодаря хешированию паролей с помощью безопасных алгорит-

мов Django гарантирует, что для взлома хранящихся в базе данных паролей пользователей потребуется огромное количество вычислительного времени.

Для хранения всех паролей по умолчанию используется алгоритм хеширования PBKDF2 с хешем SHA256. Однако Django не только поддерживает проверку существующих паролей, хешированных с помощью PBKDF2, но и проверку сохраненных паролей, хешированных другими алгоритмами, такими как PBKDF2SHA1, argon2, bcrypt и scrypt.

Настроечный параметр `PASSWORD_HASHERS` определяет хешеры паролей, поддерживаемых проектом Django. Ниже приведен стандартный список хешеров паролей `PASSWORD_HASHERS`:

```
PASSWORD_HASHERS = [  
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',  
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',  
    'django.contrib.auth.hashers.Argon2PasswordHasher',  
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',  
    'django.contrib.auth.hashers.ScryptPasswordHasher',  
]
```

Для хеширования всех паролей Django применяет первый элемент списка, в данном случае `PBKDF2PasswordHasher`. Остальные хешеры используются для проверки существующих паролей.



В Django 4.0 был введен хешер `scrypt`. Он более безопасен и рекомендуется чаще по сравнению с `PBKDF2`. Однако хешер `PBKDF2` все еще используется по умолчанию, так как `scrypt` требует наличия криптографической библиотеки `OpenSSL 1.1+` и больше памяти.

Подробнее о том, как Django хранит пароли, и о задействованных хешерах паролей можно узнать на странице <https://docs.djangoproject.com/en/4.1/topics/auth/passwords/>.

Теперь отредактируйте файл `urls.py` приложения `account`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [  
    # ...  
  
    path('', include('django.contrib.auth.urls')),  
    path('', views.dashboard, name='dashboard'),  
    path('register/', views.register, name='register'),  
]
```

Наконец, внутри каталога `templates/account/template` приложения `account` создайте новый шаблон, назовите его `register.html` и придайте ему следующий вид:

```
{% 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 method="post">
{{ user_form.as_p }}
{% csrf_token %}
<p><input type="submit" value="Create my account"></p>
</form>
{% endblock %}
```

В том же каталоге создайте дополнительный файл шаблона и назовите его `register_done.html`. Добавьте в него следующий ниже исходный код:

```
{% 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 %}
```

Пройдите по URL-адресу `http://127.0.0.1:8000/account/register/` в своем браузере. Вы увидите созданную вами страницу регистрации:

The screenshot shows a registration form titled "Create an account". The form includes fields for "Username", "First name", "Email address", "Password", and "Repeat password". A "CREATE MY ACCOUNT" button is at the bottom. The "Username" field has a placeholder "Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.".

Bookmarks	Log-in
<h2>Create an account</h2>	
Please, sign up using the following form:	
Username:	<input type="text"/>
Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.	
First name:	<input type="text"/>
Email address:	<input type="text"/>
Password:	<input type="password"/>
Repeat password:	<input type="password"/>
CREATE MY ACCOUNT	

Рис. 4.16. Форма для создания учетной записи

Заполните данные нового пользователя и кликните по кнопке **CREATE MY ACCOUNT** (Создать учетную запись).

Если все поля валидны, то пользователь будет создан, и вы увидите следующее ниже сообщение об успехе:

The screenshot shows a success message "Welcome Paloma!" and a note stating that the account was successfully created and log in is available.

Bookmarks	Log-in
<h2>Welcome Paloma!</h2>	
Your account has been successfully created. Now you can log in .	

Рис. 4.17. Страница успешного создания учетной записи

Кликните по ссылке входа и введите свое пользовательское имя и пароль, чтобы убедиться, что вы можете получить доступ к только что созданной учетной записи.

Давайте добавим ссылку на регистрацию в шаблон входа. Откройте шаблон `registration/login.html` и найдите следующую ниже строку:

```
<p>Please, use the following form to log-in:</p>
```

Замените ее такими строками:

```
<p>
    Please, use the following form to log-in.
    If you don't have an account
    <a href="{% url "register" %}">register here</a>.
</p>
```

Пройдите по URL-адресу `http://127.0.0.1:8000/account/login/` в своем браузере. Теперь страница должна выглядеть, как показано ниже:

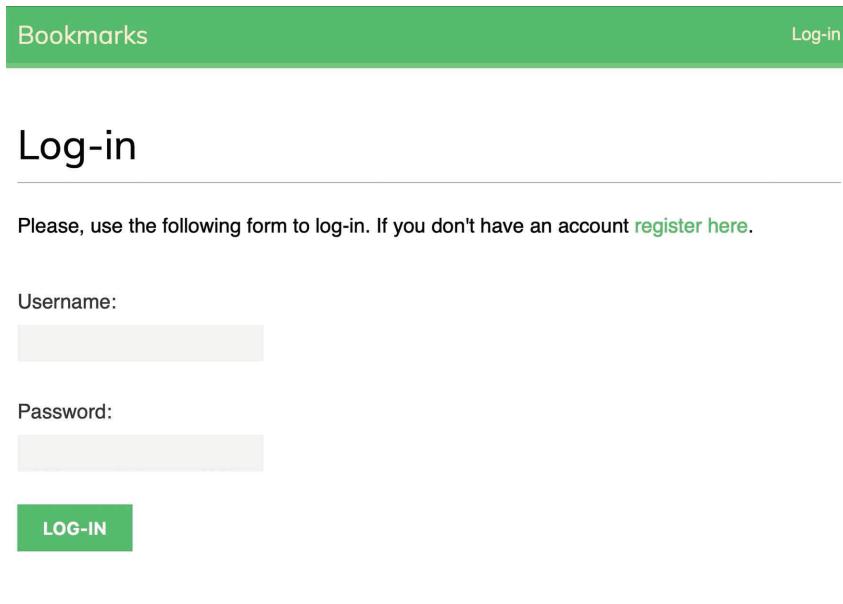


Рис. 4.18. Страница входа, в том числе ссылка на регистрацию

Мы сделали страницу регистрации доступной со страницы входа.

Расширение модели пользователя

Во время работы с учетными записями пользователей вы обнаружите, что модель User фреймворка аутентификации подходит для большинства распространенных случаев. Однако стандартная модель User идет в комплекте с ограниченным набором полей. Возможно, вы захотите расширить ее дополнительной информацией, которая будет релевантна для вашего приложения.

Простым способом расширения модели User является создание модели профиля, которая содержит взаимосвязи один-к-одному со встроенной в Django моделью User и любые дополнительные поля. Взаимосвязь один-к-одному похожа на поле ForeignKey с параметром unique=True. Обратной стороной взаимосвязи является неявная взаимосвязь один-к-одному со связанный моделью вместо менеджера для нескольких элементов. С каждой стороны взаимосвязи имеется доступ к одному связанному объекту.

Отредактируйте файл models.py приложения account, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
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 f'Profile of {self.user.username}'
```



В целях поддержания обобщенного характера исходного кода следует использовать метод get_user_model(), который позволяет извлекать модель пользователя и настроенный параметр AUTH_USER_MODEL, чтобы ссылаться на него при определении связи модели с моделью пользователя, не ссылаясь на модель пользователя auth напрямую. Более подробную информацию об этом можно почитать по адресу https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#django.contrib.auth.get_user_model.

Профиль пользователя будет содержать дату рождения пользователя и его фотографию.

Поле user со взаимосвязью один-к-одному будет использоваться для ассоциирования профилей с пользователями. С помощью параметра on_delete=models.CASCADE мы принудительно удаляем связанный объект Profile при удалении объекта User.

Поле `date_of_birth` является экземпляром класса `DateField`. Мы сделали это поле опциональным посредством `blank=True`, и мы разрешаем нулевые значения посредством `null=True`.

Поле `photo` является экземпляром класса `ImageField`. Мы сделали это поле опциональным посредством `blank=True`. Класс `ImageField` управляет хранением файлов изображений. Он проверяет, что предоставленный файл является валидным изображением, сохраняет файл изображения в каталоге, указанном параметром `upload_to`, и сохраняет относительный путь к файлу в связанном поле базы данных. Класс `ImageField` по умолчанию транслируется в столбец `VARCHAR(100)` в базе данных. Если значение оставлено пустым, то будет сохранена пустая строка.

Установка библиотеки Pillow и раздача медиафайлов

Для работы с изображениями необходимо установить библиотеку `Pillow`. Библиотека `Pillow` – это де-факто стандартная библиотека, предназначенная для обработки изображений на Python. В ней поддерживаются многочисленные форматы изображений и предоставляются мощные функции обработки изображений. Библиотека `Pillow` требуется веб-фреймворку `Django` для работы с изображениями в классе `ImageField`.

Установите `Pillow`, выполнив следующую ниже команду из командной оболочки:

```
pip install Pillow==9.2.0
```

Отредактируйте файл `settings.py` проекта, добавив следующие ниже строки:

```
MEDIA_URL = 'media/'  
MEDIA_ROOT = BASE_DIR / 'media'
```

Эти настройки обеспечат `Django` возможность управлять загружанием файлов на сайт и раздачей медиафайлов. `MEDIA_URL` – это базовый URL-адрес, используемый для раздачи медиафайлов, загруженных пользователями на сайт. `MEDIA_ROOT` – это локальный путь, где они находятся. Пути и URL-адреса файлов формируются динамически посредством добавления к ним пути проекта или URL-адреса медиафайлов в качестве префикса с целью переносимости.

Теперь отредактируйте главный файл `urls.py` проекта `bookmarks`, видоизменив исходный код, как показано ниже. Новые строки выделены жирным шрифтом:

```
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)
```

Была добавлена вспомогательная функция `static()`, чтобы раздавать медиафайлы с помощью сервера разработки во время разработки (то есть когда настроочный параметр `DEBUG` задан равным `True`).



Вспомогательная функция `static()` подходит только для разработки, но не для использования в производстве. Django очень неэффективен при раздаче статических файлов. Никогда не раздавайте статические файлы с помощью Django в производственной среде. В главе 17 «Выход в прямой эфир» вы научитесь раздавать статические файлы в производственной среде.

Создание миграций для модели профиля

Откройте оболочку и выполните следующую ниже команду, чтобы создать миграции базы данных для новой модели:

```
python manage.py makemigrations
```

Вы получите такой результат:

```
Migrations for 'account':
  account/migrations/0001_initial.py
    - Create model Profile
```

Затем следующей ниже командой командной оболочки синхронизируйте базу данных:

```
python manage.py migrate
```

Вы увидите результат, включающий такую строку:

```
Applying account.0001_initial... OK
```

Отредактируйте файл `admin.py` приложения `account`, чтобы зарегистрировать модель `Profile` на сайте администрирования, добавив исходный код, выделенный жирным шрифтом:

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

@admin.register(Profile)
class ProfileAdmin(admin.ModelAdmin):
    list_display = ['user', 'date_of_birth', 'photo']
    raw_id_fields = ['user']
```

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/> в своем браузере. Сейчас вы должны увидеть модель `Profile` на сайте администрирования проекта, как показано ниже:

The screenshot shows a screenshot of the Django Admin interface. At the top, there's a blue header bar with the word 'ACCOUNT' in white. Below it, a white section has the word 'Profiles' in blue on the left and two buttons on the right: a green '+' icon labeled 'Add' and a yellow pencil icon labeled 'Change'. There are some very faint, illegible text elements below the main header.

Рис. 4.19. Блок **ACCOUNT** (Учетная запись)
на индексной странице сайта администрирования

Кликните по ссылке **Add** (Добавить) в строке **Profiles** (Профили). Вы увидите следующую ниже форму для добавления нового профиля:

The screenshot shows a 'Add profile' form. It has three main input fields: 'User' (with a search icon), 'Date of birth' (with a date input field, a 'Today' link, and a calendar icon), and 'Photo' (with a 'Choose File' button and a message 'no file selected'). Below the photo field is a note: 'Note: You are 2 hours ahead of server time.'

Рис. 4.20. Форма для добавления профиля

Создайте объект `Profile` вручную для каждого существующего пользователя в базе данных.

Далее мы предоставим пользователям возможность редактировать свои профили на сайте.

Отредактируйте файл `forms.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
# ...
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 таковы:

- `UserEditForm` позволит пользователям редактировать свое имя, фамилию и адрес электронной почты, которые являются атрибутами встроенной в Django модели `User`;
- `ProfileEditForm` позволит пользователям редактировать данные профиля, сохраненные в конкретно-прикладной модели `Profile`. Пользователи смогут редактировать дату своего рождения и закачивать изображение на сайт в качестве фотоснимка профиля.

Отредактируйте файл `views.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
# ...
from .models import Profile

# ...

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Создать новый объект пользователя,
            # но пока не сохранять его
```

```

new_user = user_form.save(commit=False)
# Установить выбранный пароль
new_user.set_password(
    user_form.cleaned_data['password'])
# Сохранить объект User
new_user.save()
# Создать профиль пользователя
Profile.objects.create(user=new_user)
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})

```

При регистрации пользователей в системе будет создаваться объект `Profile`, который будет ассоциирован с созданным объектом `User`.

Теперь мы предоставим пользователям возможность редактировать свои профили.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```

from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.contrib.auth import authenticate, login
from django.contrib.auth.decorators import login_required
from .forms import LoginForm, UserRegistrationForm, \
    UserEditForm, ProfileEditForm
from .models import Profile

# ...

@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 добавили новое представление `edit`, чтобы пользователи могли редактировать свою личную информацию. Мы добавили в него декоратор `login_required`, поскольку только аутентифицированные пользователи могут редактировать свои профили. В этом представлении используются две модельные формы: `UserEditForm` для хранения данных во встроенной модели `User` и `ProfileEditForm` для хранения дополнительных персональных данных в конкретно-прикладной модели `Profile`. В целях валидации переданных данных вызывается метод `is_valid()` обеих форм. Если обе формы содержат валидные данные, то обе формы сохраняются путем вызова метода `save()`, чтобы обновить соответствующие объекты в базе данных.

Добавьте следующий ниже шаблон URL-адреса в файл `urls.py` приложения `account`:

```
urlpatterns = [
    ...
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    path('edit/', views.edit, name='edit'),
]
```

Наконец, создайте шаблон данного представления, разместив его в каталоге `templates/account/`, и назовите этот шаблон `edit.html`. Добавьте в него следующий ниже исходный код:

```
{% 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 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 %}
```

В приведенном выше исходном коде был добавлен в HTML-элемент `<form>` атрибут `enctype="multipart/form-data"`, чтобы обеспечить закачивание файлов на сайт. Мы используем HTML-форму для передачи форм `user_form` и `profile_form` на обработку.

Откройте URL-адрес `http://127.0.0.1:8000/account/register/` и зарегистрируйте нового пользователя. Затем войдите под новым пользователем и откройте URL-адрес `http://127.0.0.1:8000/account/edit/`. Вы должны увидеть следующую ниже страницу:

The screenshot shows a web application interface for editing a user profile. At the top, there is a green header bar with navigation links: 'Bookmarks', 'My dashboard', 'Images', 'People', and 'Hello Paloma, Logout'. Below the header, the main content area has a title 'Edit your account'. A sub-instruction reads 'You can edit your account using the following form:'. The form fields include: 'First name:' with the value 'Paloma'; 'Last name:' with the value 'Melé'; 'Email address:' with the value 'paloma@zenxit.com'; 'Date of birth:' with the value '1981-04-14'; 'Photo:' with a file input field showing 'Choose File no file selected'; and a large green 'SAVE CHANGES' button at the bottom.

Рис. 4.21. Форма для редактирования профиля

Теперь можно добавлять информацию о профиле и сохранять изменения.

Далее мы отредактируем шаблон информационной панели, вставив в него ссылки на страницы редактирования профиля и смены пароля.

Откройте шаблон `templates/account/dashboard.html` и добавьте следующие ниже строки, выделенные жирным шрифтом:

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

```
{% block content %}  
  <h1>Dashboard</h1>  
  <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>  
{% endblock %}
```

Теперь пользователи могут обращаться к форме редактирования своего профиля из информационной панели. Пройдите по URL-адресу <http://127.0.0.1:8000/account/> в своем браузере и проверьте новую ссылку на редактирование профиля пользователя. Теперь информационная панель должна выглядеть следующим образом:

Dashboard

Welcome to your dashboard. You can [edit your profile](#) or [change your password](#).

Рис. 4.22. Содержимое страницы информационной панели, включающей ссылки на редактирование профиля и смены пароля

Использование конкретно-прикладной модели пользователя

Django также предлагает способ замены модели `User` на конкретно-прикладную модель. Класс `User` должен наследовать от класса `AbstractUser` веб-фреймворка Django, который предоставляет полную реализацию стандартного пользователя в виде абстрактной модели. Подробнее об этом методе можно прочитать по адресу <https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#substituting-a-custom-user-model>.

Использование конкретно-прикладной модели пользователя будет придавать больше гибкости, но оно также может приводить к более сложной интеграции с подключаемыми приложениями, которые взаимодействуют с моделью пользователя `auth` веб-фреймворка Django напрямую.

Использование фреймворка сообщений

При взаимодействии пользователей с платформой нередко возникает потребность информировать их о результате определенных действий. Django имеет встроенный фреймворк сообщений, который позволяет показывать пользователям одноразовые уведомления.

Фреймворк сообщений находится в `django.contrib.messages` и вставляется в стандартный список `INSTALLED_APPS` в файле `settings.py` при создании новых проектов с помощью команды `python manage.py startproject`. В параметре

MIDDLEWARE файла настроек параметров `settings.py` также содержатся промежуточные программные компоненты `django.contrib.messages.middleware.MessageMiddleware`.

Фреймворк сообщений предоставляет пользователям простой способ добавления сообщений. По умолчанию сообщения хранятся в cookie-файле (отступая к сеансовому хранению), и они отображаются и очищаются при следующем запросе от пользователя. Фреймворк сообщений можно использовать в своих представлениях, импортируя модуль `messages` и добавляя новые сообщения с помощью простых методов сокращенного доступа, как показано ниже:

```
from django.contrib import messages
messages.error(request, 'Something went wrong')
```

Новые сообщения можно создавать, используя метод `add_message()` или любой метод сокращенного доступа, приведенный ниже:

- `success()`: сообщения об успехе, отображаемые при успешном выполнении действия;
- `info()`: информационные сообщения;
- `warning()`: сбой еще не произошел, но он, возможно, неизбежен;
- `error()`: действие не было успешным либо произошел сбой;
- `debug()`: отладочные сообщения, которые будут удалены либо проигнорированы в производственной среде.

Давайте добавим сообщения в проект. Фреймворк сообщений применяется к проекту глобально. Для отображения всех имеющихся для клиента сообщений мы будем использовать базовый шаблон. Такой подход позволит уведомлять клиента о результатах любого действия на любой странице.

Откройте шаблон `templates/base.html` приложения `account` и добавьте в него следующий ниже исходный код, выделенный жирным шрифтом:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
    <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
    <div id="header">
        ...
    </div>
    {% if messages %}
        <ul class="messages">
            {% for message in messages %}
                <li class="{{ message.tags }}>
                    {{ message|safe }}
            {% endfor %}
        </ul>
    {% endif %}
</body>
</html>
```

```
<a href="#" class="close">x</a>
</li>
{% endfor %}
</ul>
{% endif %}
<div id="content">
    {% block content %}
    {% endblock %}
</div>
</body>
</html>
```

В фреймворке сообщений содержится процессор контекста под названием `django.contrib.messages.context_processors.messages`, который добавляет переменную `messages` в контекст запроса. Он находится в списке `context_processors` в настроичном параметре `TEMPLATES` проекта. Переменная `messages` используется в шаблонах для отображения пользователю всех существующих сообщений.



Процессор контекста – это функция Python, которая принимает объект `request` в качестве аргумента и возвращает словарь, который добавляется в контекст запроса. Вы научитесь создавать свои собственные процессоры контекста в главе 8 «Разработка интернет-магазина».

Давайте видоизменим представление редактирования, чтобы использовать фреймворк сообщений.

Отредактируйте файл `views.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
# ...
from django.contrib import messages

# ...

@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()
messages.success(request, 'Profile updated \'\\
                           \'successfully')
else:
    messages.error(request, 'Error updating your profile')
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})

```

Сообщение об успехе генерируется, когда пользователи успешно обновляют свой профиль. Если в какой-либо из форм содержатся невалидные данные, то вместо него генерируется сообщение об ошибке.

Пройдите по URL-адресу <http://127.0.0.1:8000/account/edit/> в своем браузере и отредактируйте профиль пользователя. При успешном обновлении профиля вы должны увидеть следующее ниже сообщение:



Рис. 4.23. Сообщение об успешно отредактированном профиле

Введите недействительную дату в поле **Date of birth** (Дата рождения) и снова передайте форму на обработку. Вы должны увидеть следующее сообщение:

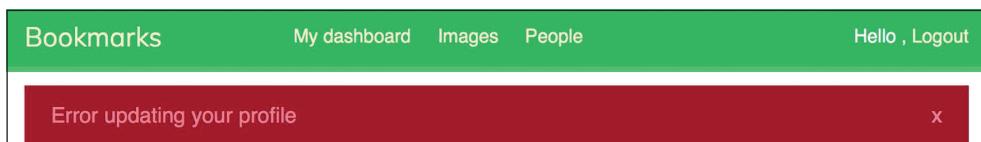


Рис. 4.24. Сообщение об ошибке обновления профиля

Сообщения, которые информируют пользователей о результатах их действий, генерируются по-настоящему просто, причем сообщения можно легко добавлять и в другие представления.

Подробнее о фреймворке сообщений можно узнать на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/messages/>.

Теперь, когда мы разработали всю функциональность, связанную с аутентификацией пользователей и редактированием профиля, мы углубимся в адаптацию процедуры аутентификации под конкретно-прикладную задачу. Мы научимся разрабатывать конкретно-прикладной бэкенд аутентификации, чтобы пользователи имели возможность входить в систему, используя свой адрес электронной почты.

Разработка конкретно-прикладного бэкенда аутентификации

Django позволяет аутентифицировать пользователей по различным источникам. Настроочный параметр `AUTHENTICATION_BACKENDS` содержит список доступных в проекте бэкендов аутентификации. По умолчанию этот настроочный параметр имеет следующее значение:

```
['django.contrib.auth.backends.ModelBackend'].
```

Применяемый по умолчанию бэкенд `ModelBackend` аутентифицирует пользователей по базе данных, используя модель `User` из `django.contrib.auth`. Такой подход приемлем для большинства веб-проектов. Однако еще есть возможность создавать конкретно-прикладные бэкенды, чтобы аутентифицировать пользователей по другим источникам, например по каталогу на основе протокола облегченного доступа к каталогам (**LDAP**)¹ или любой другой системе.

Более подробную информацию об адаптации процедуры аутентификации под конкретно-прикладную задачу можно почитать по адресу <https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#other-authentication-sources>.

При использовании функции `authenticate()` из `django.contrib.auth` Django пытается по очереди аутентифицировать пользователя на каждом из бэкендов, определенных в `AUTHENTICATION_BACKENDS`, до тех пор, пока один из них не аутентифицирует пользователя успешно. Пользователь не будет аутентифицирован, только если всем бэкенду не удалось выполнить аутентификацию.

Django предоставляет простой способ определения своих собственных бэкендов аутентификации. Бэкенд аутентификации – это класс, который предоставляет следующие два метода:

- `authenticate()`: принимает объект `request` и учетные данные пользователя в качестве параметров. Он возвращает объект `user`, соответствующий этим учетным данным, если учетные данные валидны, либо `None` в противном случае. Параметр `request` – это объект `HttpRequest`, либо `None`, если он не передан функции `authenticate()`;

¹ Англ. Lightweight Directory Access Protocol. – Прим. перев.

- `get_user()`: принимает ИД пользователя в качестве параметра и возвращает объект `User`.

Конкретно-прикладной бэкенд аутентификации создается так же просто, как и класс Python, реализующий оба метода. Давайте создадим бэкенд аутентификации, который позволит пользователям аутентифицироваться в системе, используя адрес электронной почты вместо пользовательского имени (`username`).

Внутри каталога приложения `account` создайте новый файл и назовите его `authentication.py`. Добавьте в него следующий ниже исходный код:

```
from django.contrib.auth.models import User

class EmailAuthBackend:
    """
    Аутентифицировать посредством адреса электронной почты.
    """

    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, User.MultipleObjectsReturned):
            return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

Приведенный выше исходный код является простым бэкендом аутентификации. Метод `authenticate()` получает объект `request` и опциональные параметры `username` и `password`. Мы могли бы использовать другие параметры, но мы используем `username` и `password`, чтобы этот бэкенд сразу же заработал с представлениями из фреймворка аутентификации. Показанный выше исходный код работает следующим образом:

- `authenticate()`: извлекается пользователь с данным адресом электронной почты, а пароль проверяется посредством встроенного метода `check_password()` модели пользователя. Указанный метод хеширует пароль, чтобы сравнить данный пароль с паролем, хранящимся в базе данных. Отлавливаются два разных исключения, относящихся к набору запросов `QuerySet`: `DoesNotExist` и `MultipleObjectsReturned`. Исключение

`DoesNotExist` возникает, если пользователь с данным адресом электронной почты не найден. Исключение `MultipleObjectsReturned` возникает, если найдено несколько пользователей с одним и тем же адресом электронной почты. Позже мы видоизменим представления регистрации и редактирования, чтобы предотвратить использование пользователями существующего адреса электронной почты;

- `get_user()`: пользователь извлекается по его ИД, указанному в параметре `user_id`. Django использует аутентифицировавший пользователя бэкенд, чтобы извлечь объект `User` на время сеанса пользователя. `pk` (сокращение от **primary key**) является уникальным идентификатором каждой записи в базе данных. Каждая модель Django имеет поле, которое служит ее первичным ключом. По умолчанию первичным ключом является автоматически генерируемое поле `id`. Во встроенным в Django ORM-преобразователе первичный ключ тоже может называться `pk`. Более подробная информация об автоматических полях первичного ключа находится по адресу <https://docs.djangoproject.com/en/4.1/topics/db/models/#automatic-primary-key-fields>.

Отредактируйте файл `settings.py` проекта, добавив следующий ниже исходный код:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
]
```

В данном настроечном параметре мы оставляем стандартный `ModelBackend`, который используется для аутентификации с помощью пользовательского имени и пароля, и вставляем наш собственный бэкенд аутентификации с применением электронной почты `EmailAuthBackend`.

Пройдите по URL-адресу `http://127.0.0.1:8000/account/login/` в своем браузере. Напомним, что Django будет пытаться аутентифицировать пользователя на каждом бэкенде, поэтому теперь вы должны иметь возможность беспрепятственно входить в систему, используя свое пользовательское имя либо учетную запись электронной почты.

Учетные данные пользователя будут проверены с помощью `ModelBackend`, а если пользователь не возвращен, то учетные данные будут проверены с помощью `EmailAuthBackend`.



Порядок следования бэкендов, перечисленных в настроечном параметре `AUTHENTICATION_BACKENDS`, имеет значение. Если одни и те же учетные данные валидны для нескольких бэкендов, то Django остановится на первом бэкенде, который успешно аутентифицирует пользователя.

Предотвращение использования существующего адреса электронной почты

Модель `User` в фреймворке аутентификации не препятствует созданию пользователей с одинаковым адресом электронной почты. Если две или более учетных записей пользователей имеют один и тот же адрес электронной почты, то мы не сможем определить, кто из пользователей проходит аутентификацию. Теперь, когда пользователи могут входить в систему, используя свой адрес электронной почты, мы должны предотвратить регистрацию пользователей с существующим адресом электронной почты.

Сейчас мы изменим форму для регистрации пользователей, чтобы предотвратить регистрацию нескольких пользователей с одним и тем же адресом электронной почты.

Отредактируйте файл `forms.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом, в класс `UserRegistrationForm`:

```
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']

    def clean_email(self):
        data = self.cleaned_data['email']
        if User.objects.filter(email=data).exists():
            raise forms.ValidationError('Email already in use.')
        return data
```

Мы добавили валидацию поля электронной почты, которая не позволяет пользователям регистрироваться с уже существующим адресом электронной почты. Мы формируем набор запросов `QuerySet`, чтобы свериться, нет ли существующих пользователей с одинаковым адресом электронной почты. Мы проверяем наличие результатов посредством метода `exists()`. Метод `exists()` возвращает `True`, если набор запросов `QuerySet` содержит какие-либо результаты, и `False` в противном случае.

Теперь добавьте в класс `UserEditForm` следующие ниже строки, выделенные жирным шрифтом:

```
class UserEditForm(forms.ModelForm):

    class Meta:
        model = User
        fields = ['first_name', 'last_name', 'email']

    def clean_email(self):
        data = self.cleaned_data['email']
        qs = User.objects.exclude(id=self.instance.id) \
            .filter(email=data)
        if qs.exists():
            raise forms.ValidationError(' Email already in use.')
        return data
```

В данном случае мы добавили валидацию поля `email`, чтобы пользователи не могли изменять свой бывший адрес электронной почты на существующий адрес электронной почты другого пользователя. Мы исключаем текущего пользователя из набора запросов. В противном случае текущий адрес электронной почты пользователя будет считаться существующим адресом электронной почты, и форма не пройдет валидацию.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter04>.
- Встроенные представления аутентификации: <https://docs.djangoproject.com/en/4.1/topics/auth/default/#all-authentication-views>.
- Шаблоны URL-адресов аутентификации: <https://github.com/django/django/blob/stable/3.0.x/django/contrib/auth/urls.py>.
- Как Django управляет паролями и имеющимися хешерами паролей: <https://docs.djangoproject.com/en/4.1/topics/auth/passwords/>.
- Типовая модель пользователя и метод `get_user_model()`: https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#django.contrib.auth.get_user_model.
- Использование конкретно-прикладной модели пользователя: <https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#substituting-a-custom-user-model>.
- Фреймворк сообщений: <https://docs.djangoproject.com/en/4.1/ref/contrib/messages/>.

- Конкретно-прикладные источники аутентификации: <https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#other-authentication-sources>.
- Автоматические поля первичных ключей: <https://docs.djangoproject.com/en/4.1/topics/db/models/#automatic-primary-key-fields>.

Резюме

В этой главе вы научились разрабатывать систему аутентификации для своего сайта. Вы реализовали все необходимые представления для регистрации, входа в систему и выхода из системы, редактирования и сброса пароля. Вы разработали модель для конкретно-прикладных профилей пользователей и создали конкретно-прикладной бэкенд аутентификации, позволяющий пользователям входить в систему, используя свой адрес электронной почты.

В следующей главе вы научитесь реализовывать социальную аутентификацию на сайте с помощью механизма Python Social Auth. Пользователи смогут аутентифицироваться с использованием своих учетных записей Google, Facebook или Twitter. Вы также научитесь раздавать сервер разработки по протоколу HTTPS с помощью пакета расширений Django Extensions. Вы адаптируете конвейер аутентификации под конкретно-прикладную задачу автоматического создания профилей пользователей.

5

Реализация социальной аутентификации

В предыдущей главе вы встроили регистрацию и аутентификацию пользователей в свой сайт. Вы реализовали функциональности смены, сброса и восстановления пароля, а также научились создавать конкретно-прикладную модель профиля своих пользователей.

В этой главе вы добавите социальную аутентификацию с использованием учетных записей Facebook, Google и Twitter на свой сайт. Вы будете использовать механизм Django Social Auth, чтобы реализовать социальную аутентификацию с помощью стандартного протокола авторизации под названием OAuth 2.0. Вы также видоизмените конвейер социальной аутентификации, чтобы автоматически создавать профиль для новых пользователей.

В данной главе будут рассмотрены следующие темы:

- добавление социальной аутентификации с помощью модуля Python Social Auth;
- установка пакета расширений Django Extensions;
- запуск сервера разработки по HTTPS;
- добавление аутентификации с учетной записью Facebook;
- добавление аутентификации с учетной записью Twitter;
- добавление аутентификации с учетной записью Google;
- создание профиля для пользователей, которые регистрируются с помощью социальной аутентификации.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter05>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Добавление социальной аутентификации на сайт

Социальная аутентификация – это широко используемая функциональность, которая дает возможность пользователям проходить аутентификацию с помощью существующей учетной записи поставщика услуг, используя схему единого входа в систему (**SSO**)¹. Процесс аутентификации позволяет пользователям аутентифицироваться в системе, используя существующую учетную запись в социальных сервисах, таких как Google. В данном разделе мы добавим социальную аутентификацию в системе с использованием учетных записей Facebook, Twitter и Google.

В целях реализации социальной аутентификации мы будем использовать протокол OAuth 2.0, являющийся промышленным стандартом для авторизации. OAuth расшифровывается как Open Authorization (Открытая авторизация). Стандарт OAuth 2.0 предназначен для того, чтобы предоставить веб-сайту или приложению возможность получать доступ к ресурсам, размещенным на серверах другими веб-приложениями, от имени пользователя. Протокол OAuth 2.0 используется в Facebook, Twitter и Google для аутентификации и авторизации.

Python Social Auth – это модуль Python, который упрощает процесс добавления социальной аутентификации в системе. Используя этот модуль, можно предоставлять пользователям возможность входить на ваш сайт, используя свои учетные записи из других сервисов. Исходный код этого модуля находится на странице <https://github.com/python-social-auth/social-app-django>.

Указанный модуль идет в комплекте с бэкендами аутентификации для различных фреймворков Python, включая Django. Для того чтобы установить упомянутый пакет Django из Git-репозитория проекта, откройте консоль и выполните следующую ниже команду:

```
git+https://github.com/python-social-auth/social-app-django.  
git@20fabcd7bd9a8a41910bc5c8ed1bd6ef2263b328
```

Она позволит установить библиотеку Python Social Auth из коммита GitHub, который работает с Django 4.1. На момент написания этой книги последний релиз библиотеки Python Social Auth был не совместим с Django 4.1, но более новый совместимый релиз, возможно, был уже опубликован.

Затем добавьте `social_django` в настроечный параметр `INSTALLED_APPS` в файле `settings.py` проекта, как показано ниже:

```
INSTALLED_APPS = [  
    # ...  
    'social_django',  
]
```

¹ Англ. Single Sign-on. – Прим. перев.

Это приложение, которое используется по умолчанию и служит для добавления библиотеки Python Social Auth в проекты Django. Теперь выполните следующую ниже команду, чтобы синхронизировать модели Python Social Auth с вашей базой данных:

```
python manage.py migrate
```

Вы должны увидеть, что миграции используемого по умолчанию приложения применяются, как показано ниже:

```
Applying social_django.0001_initial... OK
Applying social_django.0002_add_related_name... OK
...
Applying social_django.0011_alter_id_fields... OK
```

Пакет Python Social Auth содержит бэкенды аутентификации для многочисленных сервисов. Список всех доступных бэкендов находится на странице <https://python-social-auth.readthedocs.io/en/latest/backends/index.html#supported-backends>.

Мы добавим социальную аутентификацию в проект, позволяющую пользователям аутентифицироваться с помощью бэкендов Facebook, Twitter и Google.

Сначала необходимо добавить в проект шаблоны URL-адресов социального входа.

Откройте главный файл `urls.py` проекта `bookmarks` и вставьте в него шаблоны URL-адресов `social_django`, как показано ниже. Новые строки выделены жирным шрифтом:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/',
        include('social_django.urls', namespace='social')),
]
```

В настоящее время наше веб-приложение доступно через IP-адрес локального хоста, закрепленного за адресом `127.0.0.1` или с использованием локального сетевого имени `localhost`. Некоторые социальные сервисы не позволяют перенаправлять пользователей на `127.0.0.1` или `localhost` после успешной аутентификации; для перенаправления по URL-адресам они ожидают получения доменного имени. Доменное имя необходимо использовать прежде всего для того, чтобы социальная аутентификация работала. К счастью, есть возможность имитировать раздачу сайта под доменным именем на локальной машине.

Найдите файл `hosts` вашей машины. Если вы используете Linux или macOS, то файл `hosts` находится в `/etc/hosts`. Если же вы используете Windows, то файл `hosts` находится в `C:\Windows\System32\Drivers\etc\hosts`.

Отредактируйте файл `hosts` вашей машины, добавив следующую ниже строку:

```
127.0.0.1 mysite.com
```

Этим вашему компьютеру сообщается, что хост-имя `mysite.com` указывает на вашу собственную машину.

Давайте проверим, что ассоциация хост-имен сработала. Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://mysite.com:8000/account/login/` в своем браузере. Вы увидите следующую ниже ошибку:

DisallowedHost at /account/login/

Invalid HTTP_HOST header: 'mysite.com:8000'. You may need to add 'mysite.com' to ALLOWED_HOSTS.

Рис. 5.1. Сообщение о недопустимом заголовке хоста

Django контролирует могущие раздавать приложение хосты с помощью настроичного параметра `ALLOWED_HOSTS`. Эта мера безопасности служит для предотвращения атак на HTTP-заголовки хостов. Django будет разрешать раздавать приложение только тем хостам, которые включены в этот список.

Подробнее о разрешенных в Django хостах (настроичный параметр `ALLOWED_HOSTS`) можно узнать по адресу <https://docs.djangoproject.com/en/4.1/ref/settings/#allowed-hosts>.

Отредактируйте файл `settings.py` проекта, видоизменив настроичный параметр `ALLOWED_HOSTS`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
ALLOWED_HOSTS = ['mysite.com', 'localhost', '127.0.0.1']
```

Кроме хоста `mysite.com`, мы в явном виде включили `localhost` и `127.0.0.1`. Такой подход позволяет получать доступ к сайту через `localhost` и `127.0.0.1`. Это поведение принято в Django по умолчанию, когда параметр `DEBUG` равен `True` и параметр `ALLOWED_HOSTS` пуст.

Снова пройдите по URL-адресу `http://mysite.com:8000/account/login/` в своем браузере. Теперь вместо ошибки вы должны увидеть страницу входа в систему.

Обеспечение работы сервера разработки по протоколу HTTPS

Некоторые методы социальной аутентификации, которые мы будем использовать далее, требуют защищенного соединения по безопасному протоколу передачи данных (**HTTPS**)¹. Криптографический протокол защищенного соединения транспортного слоя (**TLS**)² – это стандарт раздачи веб-сайтов через защищенное соединение. Предшественником TLS является криптографический протокол слоя защищенных сокетов (**SSL**)³.

Хотя SSL уже и устарел, во многих библиотеках и онлайновой документации можно отыскать ссылки на термины TLS и SSL. Встроенный в Django сервер разработки не может раздавать ваш сайт по HTTPS, поскольку для этого он не предназначен. В целях тестирования функциональности социальной аутентификации, раздающей сайт по HTTPS, мы будем использовать расширение `RunServerPlus` из пакета расширений Django Extensions. Django Extensions – это сторонняя коллекция конкретно-прикладных расширений для Django. Обратите внимание, что его никогда не следует использовать для раздачи вашего сайта в реальной среде; это всего лишь сервер разработки.

Примените следующую ниже команду, что установить пакет расширений Django Extensions:

```
pip install git+https://github.com/django-extensions/django-extensions.  
git@25a41d8a3ecb24c009c5f4cac6010a091a3c91c8
```

Она позволит установить пакет Django Extensions из коммита GitHub, в который включена поддержка Django 4.1. На момент написания этой книги последний релиз пакета Django Extensions не совместим с Django 4.1, но, возможно, уже был опубликован более новый совместимый релиз.

Вам потребуется установить библиотеку Werkzeug, которая содержит слой отладчика, необходимый для расширения `RunServerPlus` из пакета Django Extensions. Примените следующую ниже команду, чтобы установить библиотеку Werkzeug:

```
pip install werkzeug==2.2.2
```

Наконец, примените следующую ниже команду, чтобы установить библиотеку `pyOpenSSL`, которая необходима для использования функциональности SSL/TLS расширения `RunServerPlus`:

```
pip install pyOpenSSL==22.0.0
```

Отредактируйте файл `settings.py` проекта, добавив пакет Django Extensions в настроечный параметр `INSTALLED_APPS`, как показано ниже:

¹ Англ. HyperText Transfer Protocol Secure. – *Прим. перев.*

² Англ. Transport Layer Security. – *Прим. перев.*

³ Англ. Secure Sockets Layer. – *Прим. перев.*

```
INSTALLED_APPS = [
    # ...
    'django_extensions',
]
```

Теперь примените предоставляемую пакетом Django Extensions команду управления `runserver_plus`, чтобы запустить сервер разработки, как показано ниже:

```
python manage.py runserver_plus --cert-file cert.crt
```

Мы указали команде `runserver_plus` имя файла SSL/TLS-сертификата. Django Extensions автоматически генерирует ключ и сертификат.

Пройдите по URL-адресу `https://mysite.com:8000/account/login/` в своем браузере. Теперь вы получаете доступ к своему сайту по HTTPS. Обратите внимание, что теперь вместо `http://` мы используем `https://`.

Ваш браузер выдаст предупреждение об опасности, так как вы используете самогенерируемый сертификат, а не тот сертификат, которому доверяет Центр сертификации (CA)¹.

Если вы работаете с Google Chrome, то увидите следующий ниже экран:

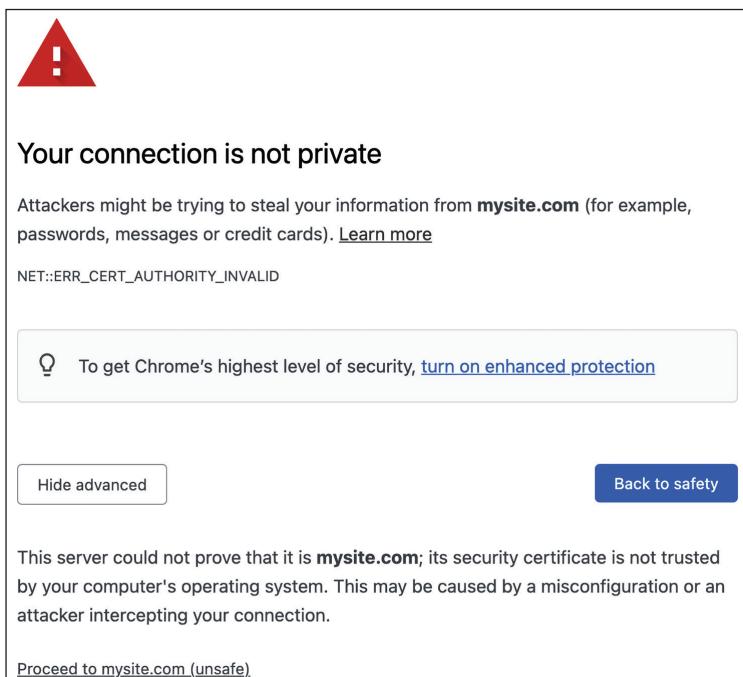


Рис. 5.2. Ошибка безопасности в Google Chrome

¹ Англ. Certification Authority. – Прим. перев.

В этом случае кликните по **Advanced** (Дополнительно) и затем кликните по **Proceed to 127.0.0.1 (unsafe)** (Перейти по 127.0.0.1 (небезопасно)).

Если вы используете Safari, то увидите следующий ниже экран:

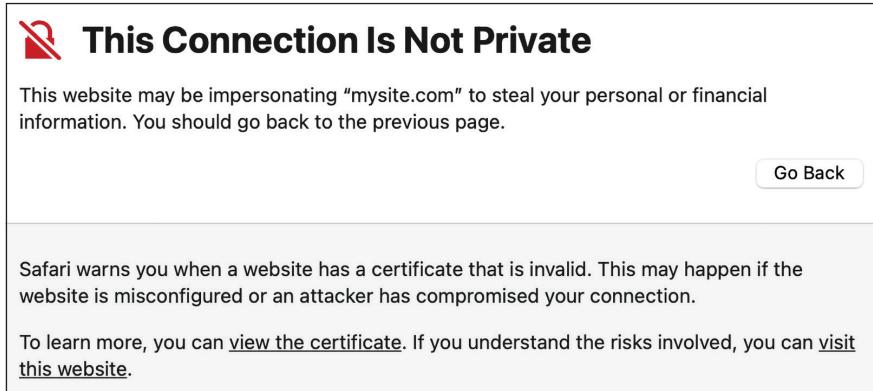


Рис. 5.3. Ошибка безопасности в Safari

В этом случае кликните по **Show details** (Показать детальную информацию), а затем кликните по **visit this website** (посетить этот веб-сайт).

Если вы используете Microsoft Edge, то увидите следующий ниже экран:

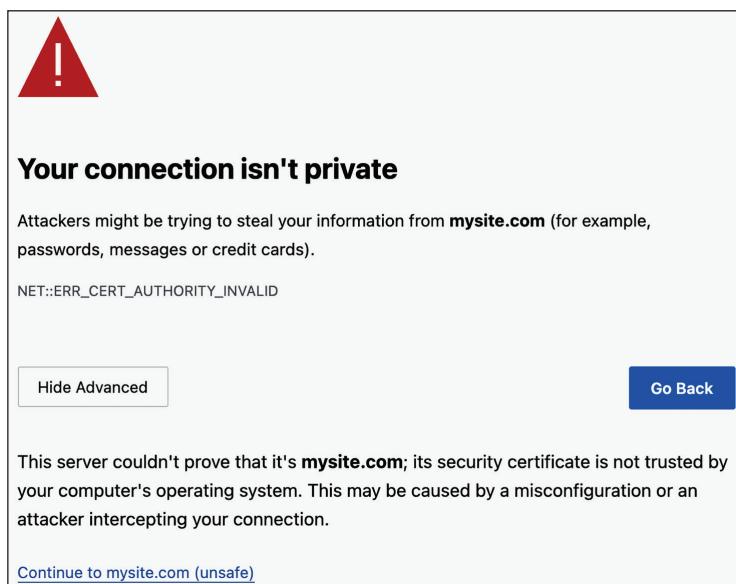


Рис. 5.4. Ошибка безопасности в Microsoft Edge

В этом случае кликните по кнопке **Advanced** (Дополнительно), а затем на **Continue to mysite.com (unsafe)** (Продолжить на mysite.com (небезопасно)).

Если вы используете любой другой браузер, то зайдите в отображаемую вашим браузером расширенную информацию и примите самозаверенный сертификат, чтобы ваш браузер доверял сертификату.

Вы увидите, что URL-адрес начинается с `https://`, и в некоторых случаях еще сможете наблюдать значок замка, который указывает на безопасность соединения. Некоторые браузеры могут отображать неработающий значок замка, поскольку вы используете самозаверенный сертификат вместо доверяемого. Для наших тестов это не будет проблемой:

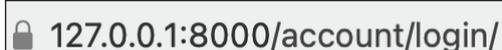


Рис. 5.5. URL-адрес со значком защищенного соединения



Пакет Django Extensions содержит целый ряд других интересных инструментов и возможностей. Более подробная информация об этом пакете находится на странице <https://django-extensions.readthedocs.io/en/latest/>.

Теперь вы можете раздавать свой сайт по HTTPS во время разработки с целью тестирования социальной аутентификации с использованием учетных записей Facebook, Twitter и Google.

Аутентификация с учетной записью Facebook

Для проведения социальной аутентификации с использованием учетной записи Facebook с целью входа на ваш сайт добавьте следующую ниже строку, выделенную жирным шрифтом, в настроечный параметр AUTHENTICATION_BACKENDS в файле `settings.py` проекта:

```
AUTHENTICATION_BACKENDS = [  
    'django.contrib.auth.backends.ModelBackend',  
    'account.authentication.EmailAuthBackend',  
    ''social_core.backends.facebook.FacebookOAuth2',  
]
```

В этом случае понадобится учетная запись Facebook для разработчика, и нужно будет создать новое приложение Facebook.

Пройдите по URL-адресу <https://developers.facebook.com/apps/> в своем браузере. После создания учетной записи разработчика вы увидите сайт со следующим заголовком:

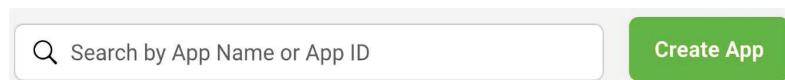


Рис. 5.6. Заголовок портала Facebook для разработчика

Кликните по кнопке **Create App** (Создать приложение). Вы увидите следующую ниже форму для выбора типа приложения:

The screenshot shows the "Create an App" wizard. On the left, there's a sidebar with "Type" selected (indicated by a blue background) and "Details" below it. The main area is titled "Select an app type" with the sub-instruction "The app type can't be changed after your app is created. [Learn more](#)". Below this, there are six categories, each with an icon and a brief description. The "Consumer" category is highlighted with a blue border and has a checked radio button next to its icon. The other categories are: "Business" (briefcase icon), "Games" (game controller icon), "Gaming" (gaming controller icon), "Workplace" (office chair icon), and "None" (cube icon). At the bottom right of the main area is a "Next" button.

Type	Description	Status
Business	Create or manage business assets like Pages, Events, Groups, Ads, Messenger, WhatsApp and Instagram Graph API using the available business permissions, features and products.	<input type="radio"/>
Consumer	Connect consumer products, and permissions, like Facebook Login and Instagram Basic Display to your app.	<input checked="" type="radio"/>
Games	Create an HTML5 game hosted on Facebook.	<input type="radio"/>
Gaming	Connect an off-platform game to Facebook Login.	<input type="radio"/>
Workplace	Create enterprise tools for Workplace from Meta.	<input type="radio"/>
None	Create an app with combinations of consumer and business permissions and products.	<input type="radio"/>

Рис. 5.7. Форма для создания приложения Facebook с выбором типа приложения

В разделе **Select an app type** (Выберите тип приложения) выберите **Consumer** (Потребитель) и кликните по кнопке **Next** (Далее).

Вы увидите следующую ниже форму для создания нового приложения:

Add details

Display name
This is the app name associated with your app ID. You can change this later.

App Contact Email
This email address is used to contact you about potential policy violations, app restrictions or steps to recover the app if it's been deleted or compromised.

Business Account · Optional
To access certain permissions or features, apps need to be connected to a Business Account.

By proceeding, you agree to the [Facebook Platform Terms](#) and [Developer Policies](#).

Рис. 5.8. Форма Facebook с детальной информацией о приложении

Введите **Bookmarks** (Закладки) в качестве **Display name** (Отображаемого имени), добавьте контактный адрес электронной почты и кликните по кнопке **Create App** (Создать приложение).

Вы увидите информационную панель нового приложения. На ней отобразятся различные службы, которые можно сконфигурировать для приложения. Найдите поле **Facebook Login** (Вход в Facebook) и кликните по **Set Up** (Настроить):

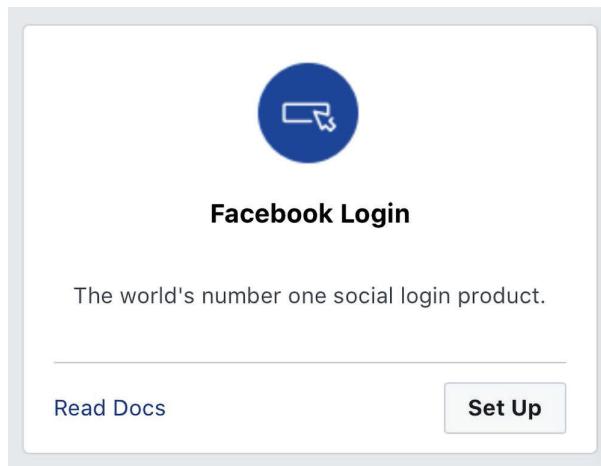


Рис. 5.9. Блок продуктов входа в Facebook

Вам будет предложено выбрать платформу, как показано ниже:



Рис. 5.10. Выбор платформы для входа в Facebook

Выберите **Web** в качестве платформы. Вы увидите следующую ниже форму:

1. Tell Us about Your Website

Tell us what the URL of your site is.

Site URL

https://mysite.com:8000/

Save

Continue

Рис. 5.11. Конфигурация веб-платформы для входа в Facebook

Введите `https://mysite.com:8000/` в поле **Site URL** (URL-адрес сайта) и кликните по кнопке **Save** (Сохранить). Затем кликните по кнопке **Continue** (Продолжить). Остальную часть процесса быстрого запуска можно пропустить.

В левом меню кликните по **Settings** (Настройки), а затем на **Basic** (Базовые), как показано ниже:

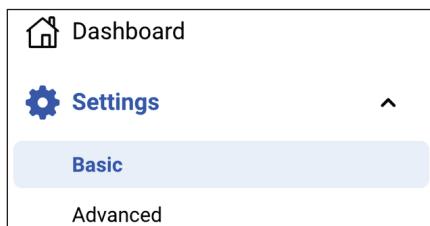


Рис. 5.12. Боковое меню портала Facebook для разработчика

Вы увидите форму с данными, аналогичными следующим ниже:

App ID	App Secret
312115414132251	•••••••••• Show
Display Name	Namespace
Bookmarks	

Рис. 5.13. Детальная информация о приложении Facebook

Скопируйте ключи **App ID** (ИД приложения) и **App Secret** (Секрет приложения) и добавьте их в файл `settings.py` своего проекта, как показано ниже:

```
SOCIAL_AUTH_FACEBOOK_KEY = 'XXX' # ИД приложения Facebook
SOCIAL_AUTH_FACEBOOK_SECRET = 'XXX' # Секрет приложения Facebook
```

Настроечный параметр `SOCIAL_AUTH_FACEBOOK_SCOPE` можно опционально определить с дополнительными разрешениями, которые вы хотите запрашивать у пользователей Facebook:

```
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email']
```

Теперь вернитесь на портал Facebook для разработчика и кликните по **Settings** (Настройки). Добавьте `mysite.com` в раздел **App Domains** (Домены приложения), как показано ниже:

App Domains
mysite.com X

Рис. 5.14. Домены, разрешенные для приложения Facebook

Вы должны ввести публичный URL-адрес для **Privacy Policy URL** (URL-адрес политики конфиденциальности) и еще один URL-адрес для **User Data Deletion Instructions URL** (URL-адрес инструкций по удалению данных пользователя). Ниже приведен пример использования URL-адреса страницы Википедии для политики конфиденциальности. Обратите внимание, что необходимо использовать валидный URL-адрес:

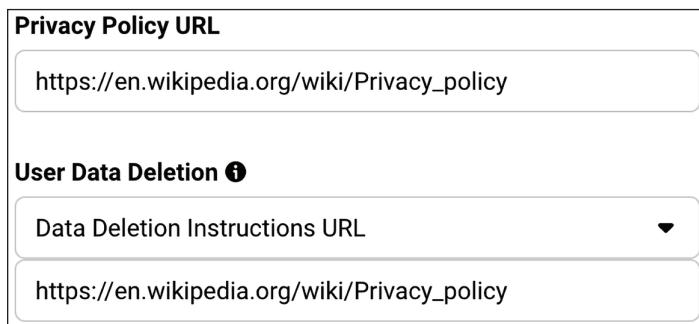


Рис. 5.15. URL-адреса политики конфиденциальности и инструкций по удалению данных пользователя для приложения Facebook

Кликните по кнопке **Save Changes** (Сохранить изменения). Затем в левом меню под продуктами кликните по **Facebook Login** (Вход в Facebook) и потом **Settings** (Настройки), как показано здесь:



Рис. 5.16. Меню входа в Facebook

Проверьте, чтобы были активными только следующие ниже настройки:

- **Client OAuth Login** (Вход через OAuth для клиентов);
- **Web OAuth Login** (Вход через OAuth для веб-приложений);
- **Enforce HTTPS** (Обеспечить работу по HTTPS);
- **Embedded Browser OAuth Login** (Вход через OAuth для встроенного браузера);
- **Used Strict Mode for Redirect URIs** (Использование строгого режима для URI-идентификаторов перенаправления).

Введите адрес `https://mysite.com:8000/social-auth/complete/facebook/` в поле **Valid OAuth Redirect URIs** (OAuth-допустимые URI-идентификаторы перенаправления). Подборка должна выглядеть следующим образом:

Client OAuth Settings

Client OAuth Login (Yes): Enables the standard OAuth client token flow. Secure your application and prevent abuse by locking down which token redirect URLs are allowed with the options below. Disable globally if not used. [?]

Web OAuth Login (Yes): Enables web-based Client OAuth Login. [?]

Enforce HTTPS (Yes): Enforce the use of HTTPS for Redirect URLs and the JavaScript SDK. Strongly recommended. [?]

Force Web OAuth Reauthentication (No): When on, prompts people to enter their Facebook password in order to log in on the web. [?]

Embedded Browser OAuth Login (Yes): Enable webview Redirect URLs for Client OAuth Login. [?]

Use Strict Mode for Redirect URIs (Yes): Only allow redirects that exactly match the Valid OAuth Redirect URIs. Strongly recommended. [?]

Valid OAuth Redirect URIs: A manually specified redirect_uri used with Login on the web must exactly match one of the URIs listed here. This list is also used by the JavaScript SDK for in-app browsers that suppress popups. [?]

`https://mysite.com:8000/social-auth/complete/facebook/`

Login from Devices (No): Enables the OAuth client login flow for devices like a smart TV [?]

Login with the JavaScript SDK (No): Enables Login and signed-in functionality with the JavaScript SDK. [?]

Рис. 5.17. Клиентские настройки OAuth для входа в Facebook

Откройте шаблон `registration/login.html` приложения `account` и добавьте в нижнюю часть блока `content` следующий ниже исходный код, выделенный жирным шрифтом:

```
{% block content %}

...
<div class="social">
  <ul>
    <li class="facebook">
      <a href="{% url "social:begin" "facebook" %}">
        Sign in with Facebook
      </a>
    </li>
  </ul>
</div>
{% endblock %}
```

Примените предоставляемую пакетом Django Extensions команду управления `runserver_plus`, чтобы запустить сервер разработки, как показано ниже:

```
python manage.py runserver_plus --cert-file cert.crt
```

Пройдите по URL-адресу `https://mysite.com:8000/account/login/` в своем браузере. Теперь страница входа будет выглядеть следующим образом:

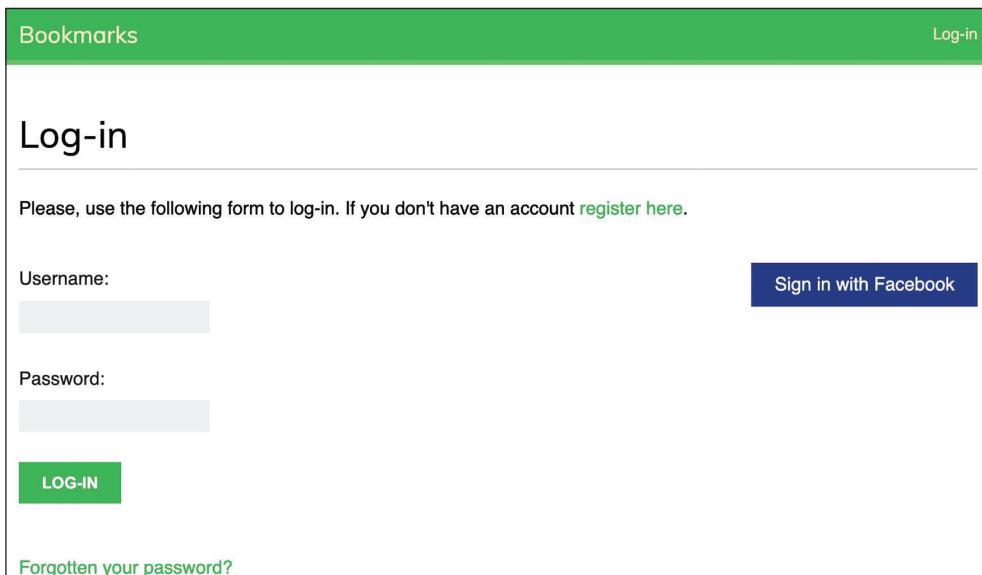


Рис. 5.18. Страница входа, включая кнопку для аутентификации через Facebook

Кликните по кнопке **Sign in with Facebook** (Войти с учетной записью Facebook). Вы будете перенаправлены в Facebook и увидите модальное диалоговое окно с предложением разрешить приложению *Bookmarks* (Закладки) доступ к вашему публичному профилю Facebook.

Вы увидите предупреждение, указывающее на необходимость передать приложение на проверку входа. Кликните по кнопке **Continue as ...** (Продолжить как ...).

Ваш вход будет зарегистрирован, и вы будете перенаправлены на страницу информационной панели своего сайта. Напомним, что вы задали этот URL-адрес в настроичном параметре `LOGIN_REDIRECT_URL`. Как видите, социальная аутентификация добавляется на сайт довольно просто.



Рис. 5.19. Модальное диалоговое окно Facebook для предоставления разрешений приложению

Аутентификация с учетной записью Twitter

Для проведения социальной аутентификации с использованием учетной записи Twitter добавьте следующую ниже строку, выделенную жирным шрифтом, в настроечный параметр AUTHENTICATION_BACKENDS в файле `settings.py` проекта:

```
AUTHENTICATION_BACKENDS = [  
    'django.contrib.auth.backends.ModelBackend',  
    'account.authentication.EmailAuthBackend',  
    'social_core.backends.facebook.FacebookOAuth2'  
    ''social_core.backends.twitter.TwitterOAuth',  
]
```

В этом случае потребуется учетная запись Twitter для разработчика. Пройдите по URL-адресу <https://developer.twitter.com/> в своем браузере и кликните по кнопке **Sign up** (Зарегистрироваться).

После создания учетной записи Twitter для разработчика зайдите на информационную панель портала для разработчиков по адресу <https://developer.twitter.com/en/portal/dashboard>. Информационная панель должна выглядеть следующим образом:



Рис. 5.20. Информационная панель портала Twitter для разработчиков

Кликните по кнопке **Create Project** (Создать проект). Вы увидите такой экран:



Рис. 5.21. Экран создания проекта Twitter – Имя проекта

Введите Bookmarks (Закладки) в поле **Project name** (Имя проекта) и кликните по кнопке **Next** (Далее). Вы увидите следующий ниже экран:



Рис. 5.22. Экран создания проекта в Twitter – Вариант использования

В разделе **Use case** (Вариант использования) выберите **Exploring the API** (Обследование API) и кликните по кнопке **Next** (Далее). Можно выбрать любой другой вариант использования; это не повлияет на конфигурацию. Затем вы увидите такое окно:



Рис. 5.23. Экран создания проекта Twitter – Описание проекта

Ведите краткое описание проекта и кликните по кнопке **Next** (Далее). Проект создан, и вы увидите следующий ниже экран:



Рис. 5.24. Конфигурация приложения Twitter

Далее мы создадим новое приложение. Кликните по кнопке **Create new** (Создать новое). Вы увидите следующий ниже экран для конфигурирования нового приложения:

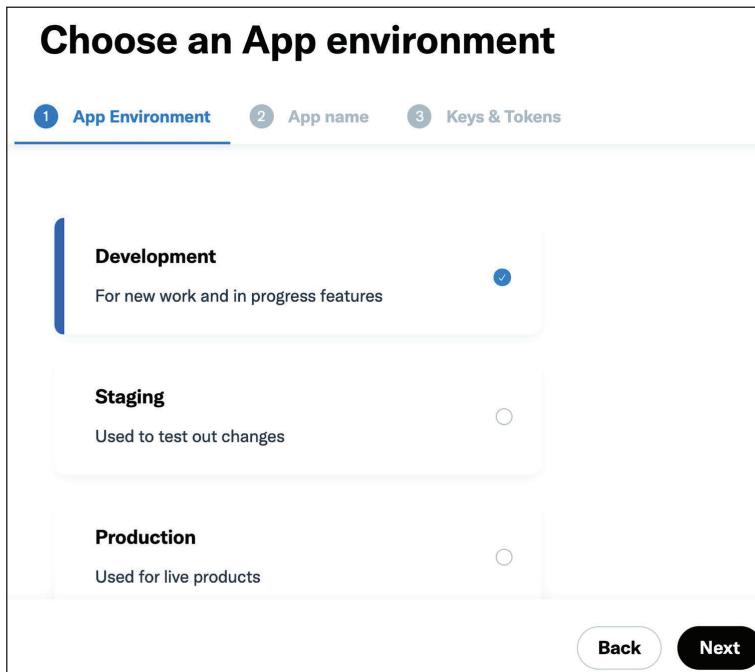


Рис. 5.25. Конфигурация приложения Twitter – выбор среды

В разделе **App Environment** (Среда приложения) выберите **Development** (Разработка) и кликните по кнопке **Next** (Далее). Мы создаем среду разработки для приложения. Вы увидите следующее ниже окно:

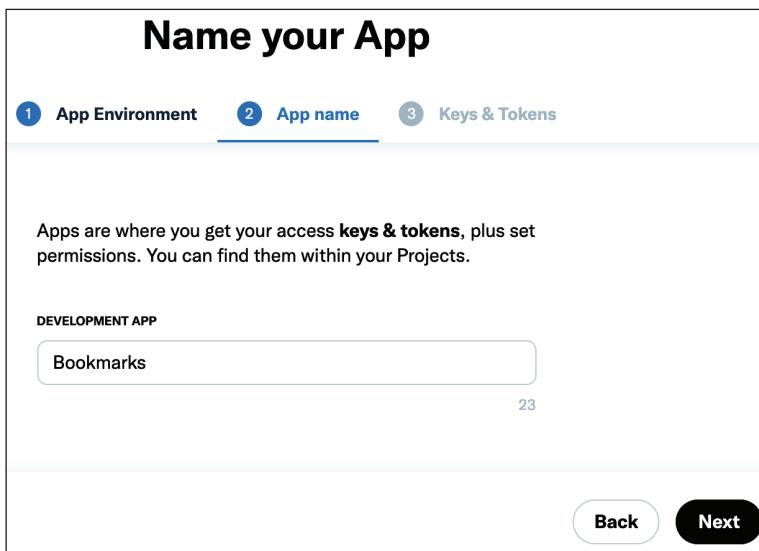


Рис. 5.26. Конфигурация приложения Twitter – Имя приложения

В разделе **App name** (Имя приложения) введите **Bookmarks** (Закладки), за которым следует суффикс. Twitter не позволит использовать имя существующего в рамках Twitter приложения разработчика, поэтому необходимо ввести имя, доступное для использования. Кликните по кнопке **Next** (Далее). Если имя, которое вы пытаетесь использовать для своего приложения, уже занято, то Twitter выдаст ошибку.

После выбора доступного имени вы увидите следующий ниже экран:

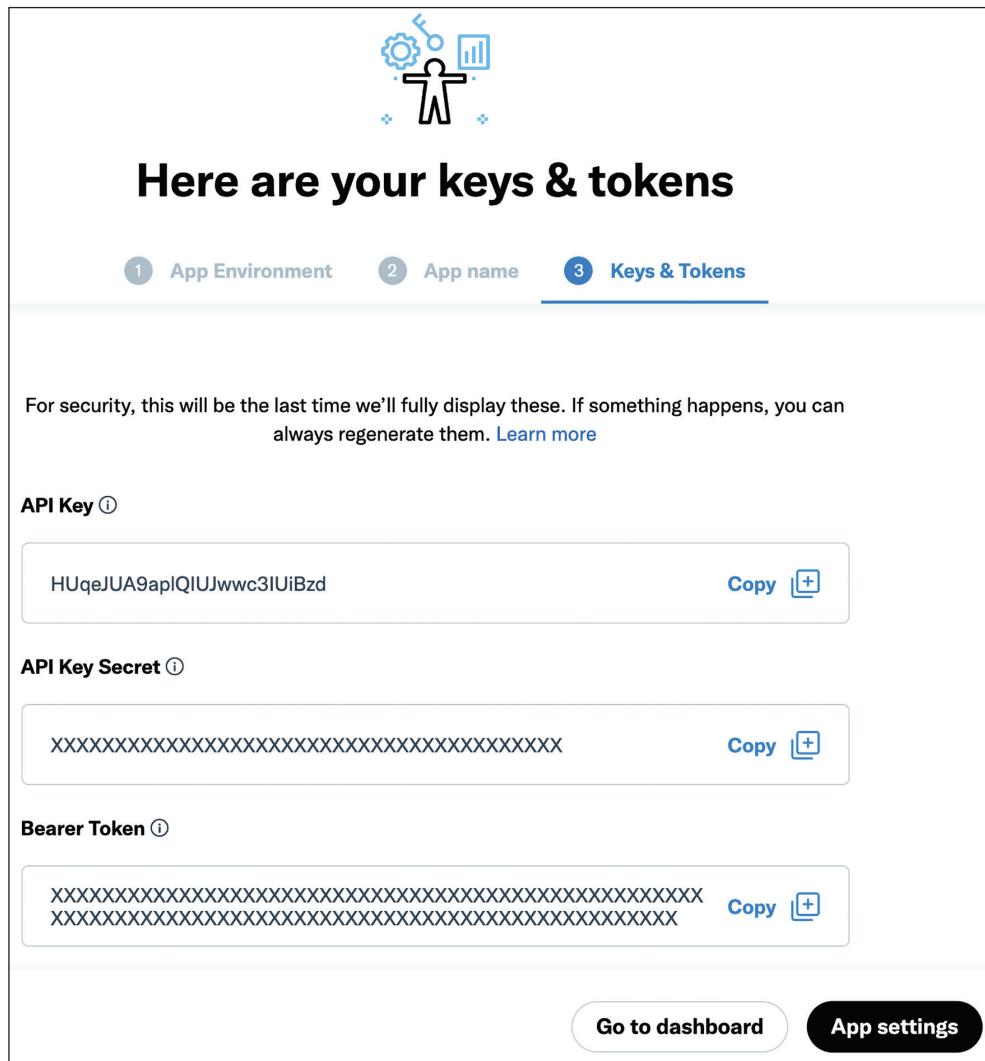


Рис. 5.27. Конфигурация приложения Twitter – сгенерированные ключи API

Скопируйте **API Key** (Ключ API) и **API Key Secret** (Секрет ключа API) в следующие ниже настроечные параметры в файле `settings.py` проекта:

```
SOCIAL_AUTH_TWITTER_KEY = 'XXX' # Ключ API Twitter  
SOCIAL_AUTH_TWITTER_SECRET = 'XXX' # Секрет API Twitter
```

Затем кликните по **App settings** (Настройки приложения). Вы увидите экран, включающий следующий ниже раздел:

User authentication settings

Authentication not set up

OAuth 2.0 and OAuth 1.0a are authentication methods that allow users to sign in to your App with Twitter. They also allow your App to make specific requests on behalf of authenticated users. You can turn on one, or both methods.

Set up

Рис. 5.28. Настройка аутентификации пользователя приложения Twitter

В разделе **User authentication settings** (Настроочные параметры аутентификации пользователя) кликните по **Set up** (Настроить). Вы увидите следующий ниже экран:

User authentication settings

OAuth 2.0 and OAuth 1.0a are authentication methods that allow users to sign in to your App with Twitter. They also allow your App to make specific requests on behalf of authenticated users. You can turn on one, or both methods. [Read the docs](#)

OAuth 2.0 NEW



- Can be used with the Twitter API v2 only
- Allows you to pick specific scopes (also known as, permissions)

OAuth 1.0a



- Can be used with Twitter API v1.1 and v2
- Uses broad authorization with coarse scopes

Рис. 5.29. Активация OAuth 2.0 в приложении Twitter

Активируйте опцию **OAuth 2.0**. Это версия OAuth, которую мы будем использовать. Затем в разделе **OAuth 2.0 Settings** (Настроочные параметры OAuth 2.0) выберите **Web App** (Веб-приложение) в поле **Type of App** (Тип приложения), как показано ниже:



Рис. 5.30. Настроочные параметры OAuth 2.0 приложения Twitter

В разделе **General Authentication Settings** (Общие настроочные параметры аутентификации) введите следующую ниже детальную информацию о своем приложении:

- **Callback URI / Redirect URL** (URI-идентификатор обратного вызова / URL-адрес перенаправления): <https://mysite.com:8000/social-auth/complete/twitter/>;
- **Website URL** (URL-адрес веб-сайта): <https://mysite.com:8000/>.

Настроочные параметры должны выглядеть, как показано ниже:



Рис. 5.31. Конфигурация URL-адресов для аутентификации через Twitter

Кликните по кнопке **Save** (Сохранить). Теперь вы увидите следующее ниже окно с указанием **Client ID** (ИД клиента) и **Client Secret** (Секрет клиента):

OAuth 2.0 Client ID and Client Secret

Think of your Client ID and Client Secret as the user name and password that allow you to use OAuth 2.0 as an authentication method.

For security, this will be the last time we'll fully display the Client Secret. If something happens, you can always regenerate it. [Read the docs](#)

Client ID ⓘ

Mlh4cWg2R1ZNZUdSY2ZqUVNWckE6MTpjaQ

Copy 

Client Secret ⓘ

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Copy 

Done

Рис. 5.32. ИД клиента и секрет клиента приложения Twitter

Для аутентификации клиента они не понадобятся, поскольку вместо них вы будете использовать **API Key** (Ключ API) и **API Key Secret** (Секрет ключа API). Однако можно скопировать и сохранить **Client Secret** (Секрет клиента) в надежном месте. Кликните по кнопке **Done** (Готово).

Вы увидите еще одно напоминание о необходимости сохранить секрет клиента:



Рис. 5.33. Напоминание о секрете клиента Twitter

Кликните по **Yes, I saved it** (Да, я его сохранил). Теперь вы увидите, что аутентификация OAuth 2.0 была включена, как на следующем ниже экране:

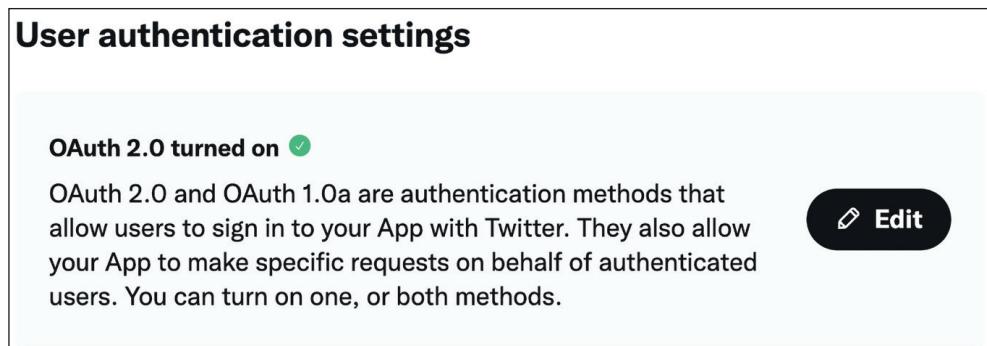


Рис. 5.34. Настроочные параметры аутентификации приложения через Twitter

Теперь отредактируйте шаблон `registration/login.html`, добавив в элемент `` следующий ниже исходный код, выделенный жирным шрифтом:

```
<ul>
    <li class="facebook">
        <a href="{% url "social:begin" "facebook" %}">
            Sign in with Facebook
        </a>
    </li>
    <li class="twitter">
        <a href="{% url "social:begin" "twitter" %}">
            Sign in with Twitter
        </a>
    </li>
</ul>
```

Примените предоставляемую пакетом Django Extensions команду управления `runserver_plus`, чтобы запустить сервер разработки, как показано ниже:

```
python manage.py runserver_plus --cert-file cert.crt
```

Пройдите по URL-адресу `https://mysite.com:8000/account/login/` в своем браузере. Теперь страница входа в систему будет выглядеть следующим образом:

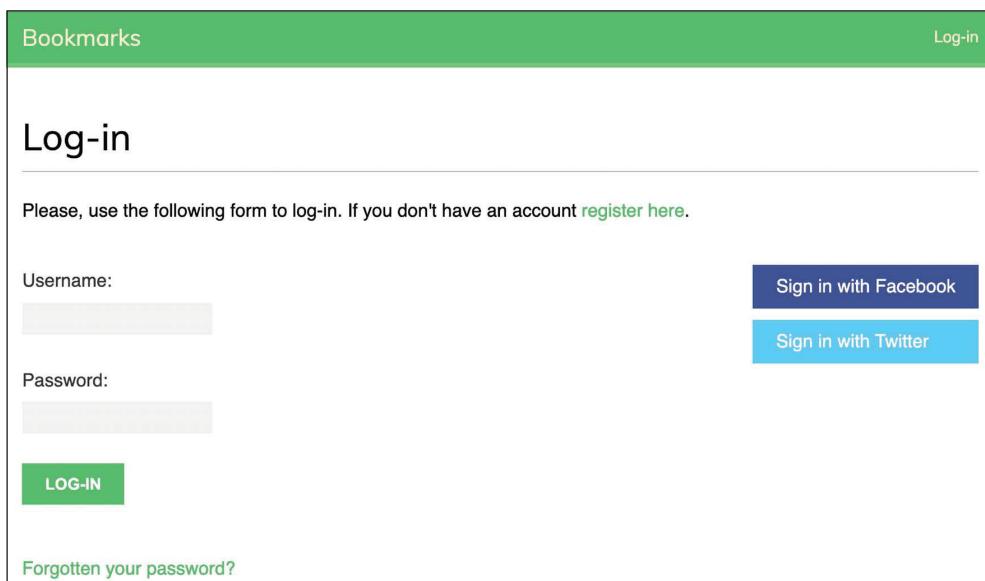


Рис. 5.35. Страница входа в систему, включающая кнопку аутентификации с учетной записью Twitter

Кликните по ссылке **Sign in with Twitter** (Войти с учетной записью Twitter). Вы будете перенаправлены в Twitter, где вам будет предложено авторизовать приложение, как показано ниже:

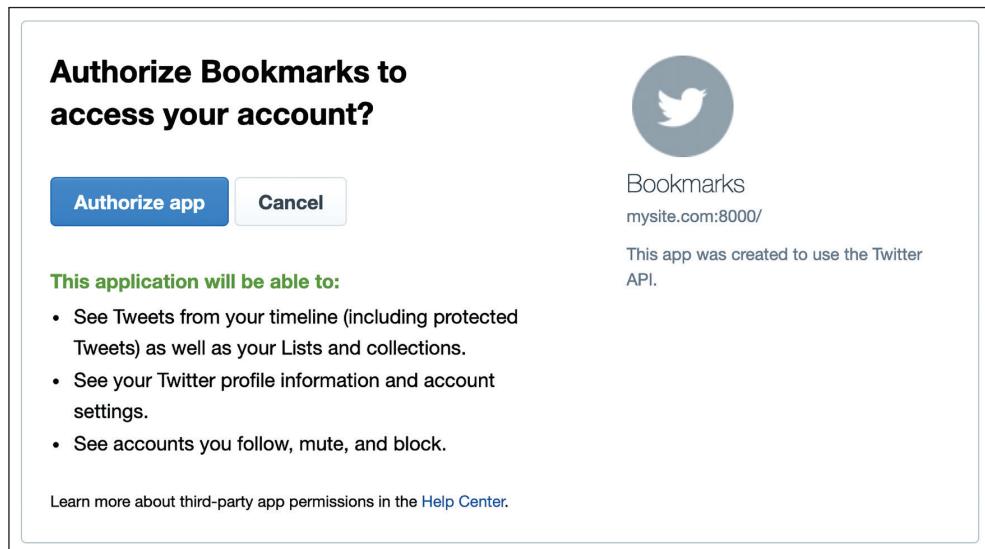


Рис. 5.36. Экран авторизации пользователя через Twitter

Кликните по **Authorize app** (Авторизовать приложение). Вы ненадолго увидите следующую ниже страницу, после чего будете перенаправлены на страницу информационной панели:

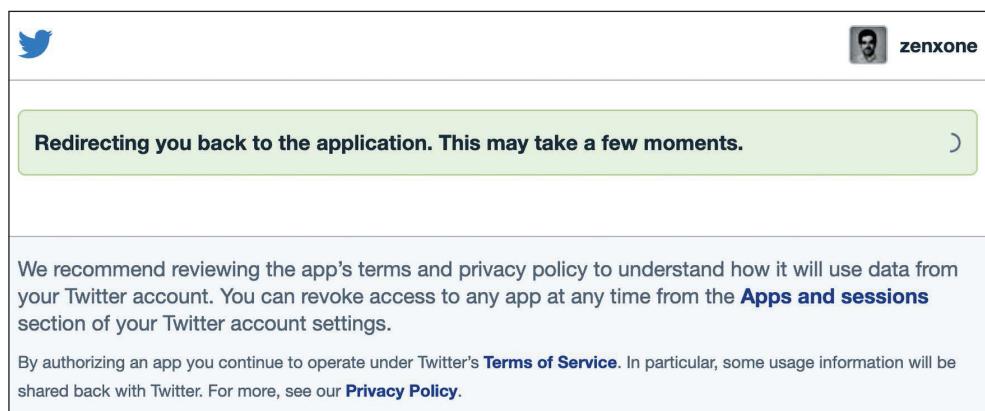


Рис. 5.37. Страница перенаправления при аутентификации пользователя через Twitter

После этого вы будете перенаправлены на страницу информационной панели своего приложения.

Аутентификация с учетной записью Google

Google предлагает социальную аутентификацию с использованием OAuth2. О реализации OAuth2 в Google можно почитать на странице <https://developers.google.com/identity/protocols/OAuth2>.

Для проведения аутентификации с использованием учетной записи Google добавьте следующую ниже строку, выделенную жирным шрифтом, в настроекий параметр AUTHENTICATION_BACKENDS в файле settings.py проекта:

```
AUTHENTICATION_BACKENDS = [  
    'django.contrib.auth.backends.ModelBackend',  
    'account.authentication.EmailAuthBackend',  
    'social_core.backends.facebook.FacebookOAuth2',  
    'social_core.backends.twitter.TwitterOAuth',  
    'social_core.backends.google.GoogleOAuth2',  
]
```

Сначала необходимо создать ключ API в консоли разработчика Google. Пройдите по URL-адресу <https://console.cloud.google.com/projectcreate> в своем браузере. Вы увидите следующий ниже экран:

The screenshot shows the 'New Project' page in the Google Cloud Platform. At the top, there's a blue header bar with the Google Cloud logo and the text 'Google Cloud Platform'. Below it, the main title is 'New Project'. The first field is 'Project name *' with the value 'Bookmarks'. To the right of this field is a question mark icon. The next section is 'Location *' with the value 'No organisation' and a 'BROWSE' button. Below these fields, there's a note: 'Parent organisation or folder'. At the bottom of the form are two buttons: 'CREATE' on the left and 'CANCEL' on the right.

Рис. 5.38. Форма создания проекта Google

В поле **Project name** (Имя проекта) введите Bookmarks (Закладки) и кликните по кнопке **CREATE** (Создать).

Когда новый проект будет готов, проверьте, чтобы в верхней навигационной панели проект был выбран, как показано ниже:



Рис. 5.39. Верхняя навигационная панель консоли Google для разработчиков

После создания проекта выберите в разделе **APIs and services** (API-интерфейсы и сервисы) подпункт **Credentials** (Учетные данные), как показано ниже:



Рис. 5.40. Пункты меню APIs and services

Вы увидите следующий ниже экран:

The screenshot shows the 'Credentials' section of the Google Cloud Platform interface. It includes sections for 'Create credentials to access', 'API keys', and 'OAuth 2.0 Client IDs'. A prominent callout box highlights the 'OAuth client ID' section, which says 'Requests user consent so that your app can access the user's data.' and includes a 'CONFIGURE CONSENT SCREEN' button.

Рис. 5.41. Создание учетных данных API Google

Затем кликните по **CREATE CREDENTIALS** (Создать учетные данные) и выберите **OAuth client ID** (ИД клиента OAuth).

Сначала Google попросит сконфигурировать экран согласия, как показано ниже:



Рис. 5.42. Предупреждение о необходимости сконфигурировать экран согласия OAuth

Мы сконфигурируем страницу, которая будет отображаться пользователям, на которой они будут давать согласие на доступ к вашему сайту с помощью своей учетной записи Google. Кликните по кнопке **CONFIGURE CONSENT SCREEN** (Сконфигурировать экран согласия). Вы будете перенаправлены на следующий ниже экран:



Рис. 5.43. Выбор типа пользователя в настройке экрана согласия Google OAuth

Выберите **External** (Внешний) в поле **User Type** (Тип пользователя) и кликните по кнопке **CREATE** (Создать). Вы увидите следующий ниже экран:

The screenshot shows the 'App information' configuration page. It starts with the heading 'App information' and a descriptive text: 'This shows in the consent screen, and helps end users know who you are and contact you'. Below this is a form field for 'App name *' containing 'Bookmarks'. A note below the field says: 'The name of the app asking for consent'. Another form field for 'User support email *' contains 'myaccount@gmail.com'. A note below this field says: 'For users to contact you with questions about their consent'.

Рис. 5.44. Настойка экрана согласия Google OAuth

В разделе **App name** (Имя приложения) введите **Bookmarks** (Закладки) и выберите свой адрес электронной почты в поле **User support email** (Почта поддержки пользователей).

В разделе **Authorised domains** (Авторизованные домены) введите `mysite.com`, как показано ниже:

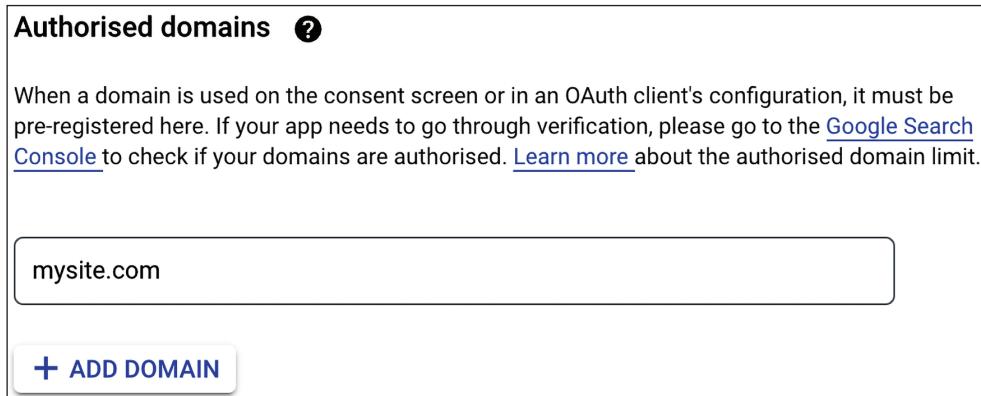


Рис. 5.45. Авторизованные домены Google OAuth

Введите свой адрес электронной почты в разделе **Developer contact information** (Контактная информация разработчика) и кликните по кнопке **SAVE AND CONTINUE** (Сохранить и продолжить).

На шаге 2 **Scopes** (Области применения) ничего не меняйте и кликните по **SAVE AND CONTINUE** (Сохранить и продолжить).

На шаге 3 **Test users** (Тестовые пользователи) добавьте пользователя Google в раздел **Test users** и кликните по кнопке **SAVE AND CONTINUE** (Сохранить и продолжить), как показано ниже:

The screenshot shows the 'Test users' configuration page for a Google OAuth application. At the top, there are four tabs: 'OAuth consent screen' (with a checkmark), 'Scopes' (with a checkmark), 'Test users' (highlighted with a blue circle and the number '3'), and 'Summary' (highlighted with a grey circle and the number '4'). Below the tabs, the heading 'Test users' is displayed. A note states: 'While publishing status is set to 'Testing,' only test users are able to access the app. Allowed user cap prior to app verification is 100, and is counted over the entire lifetime of the app.' A link to 'Learn more' is provided. A button labeled '+ ADD USERS' is visible. Below this, there is a 'Filter' input field with the placeholder 'Enter property name or value' and a question mark icon. A section for 'User information' contains the email 'myaccount@gmail.com' and a trash can icon for deletion. At the bottom, there are two buttons: 'SAVE AND CONTINUE' and 'CANCEL'.

Рис. 5.46. Тестовые пользователи Google OAuth

Вы увидите сводную информацию о конфигурации экрана согласия. Кликните по кнопке **Back to dashboard** (Вернуться к информационной панели).

В меню на левой боковой панели кликните по **Credentials** (Учетные данные) и снова кликните по **Create credentials** (Создать учетные данные), а затем на **OAuth client ID** (ИД клиента OAuth).

В качестве дальнейшего шага введите следующую ниже информацию:

- **Application type** (Тип приложения): выберите **Web application** (Веб-приложение);
- **Name** (Имя): введите **Bookmarks** (Закладки);
- **Авторизованные источники JavaScript**: добавьте `https://mysite.com:8000/`;
- **Authorised redirect URIs** (Авторизованные URI-идентификаторы перевынаправления): добавьте `https://mysite.com:8000/social-auth/complete/google-oauth2/`.

Форма должна выглядеть следующим образом:

[←](#) Create OAuth client ID

A client ID is used to identify a single app to Google's OAuth servers. If your app runs on multiple platforms, each will need its own client ID. See [Setting up OAuth 2.0](#) for more information. [Learn more](#) about OAuth client types.

Application type * ▼

Name *

The name of your OAuth 2.0 client. This name is only used to identify the client in the console and will not be shown to end users.

Authorised JavaScript origins [?](#)

For use with requests from a browser

URIs *

[+ ADD URI](#)

Authorised redirect URIs [?](#)

For use with requests from a web server

URIs *

[+ ADD URI](#)

[CREATE](#) [CANCEL](#)

Рис. 5.47. Форма создания ИД клиента Google OAuth

Кликните по кнопке **CREATE** (Создать). Вы получите ключи **Your Client ID** (ИД клиента) и **Your Client Secret** (Секрет клиента):



Рис. 5.48. ИД клиента и секрет клиента Google OAuth

Добавьте оба ключа в файл `settings.py`, как показано ниже:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = 'XXX' # ИД клиента Google
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = 'XXX' # Секрет клиента Google
```

Отредактируйте шаблон `registration/login.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом, в элемент ``¹:

```
<ul>
    <li class="facebook">
        <a href="{% url "social:begin" "facebook" %}">
            Sign in with Facebook
        </a>
    </li>
    <li class="twitter">
```

¹ Войти с учетной записью Facebook.... – Прим. перев.

```

<a href="{% url "social:begin" "twitter" %}">
    Sign in with Twitter
</a>
</li>
<li class="google">
    <a href="{% url "social:begin" "google-oauth2" %}">
        Sign in with Google
    </a>
</li>
</ul>

```

Примените предоставляемую пакетом Django Extensions команду управления `runserver_plus`, чтобы запустить сервер разработки, как показано ниже:

```
python manage.py runserver_plus --cert-file cert.crt
```

Пройдите по URL-адресу `https://mysite.com:8000/account/login/` в своем браузере. Теперь страница входа должна выглядеть следующим образом:

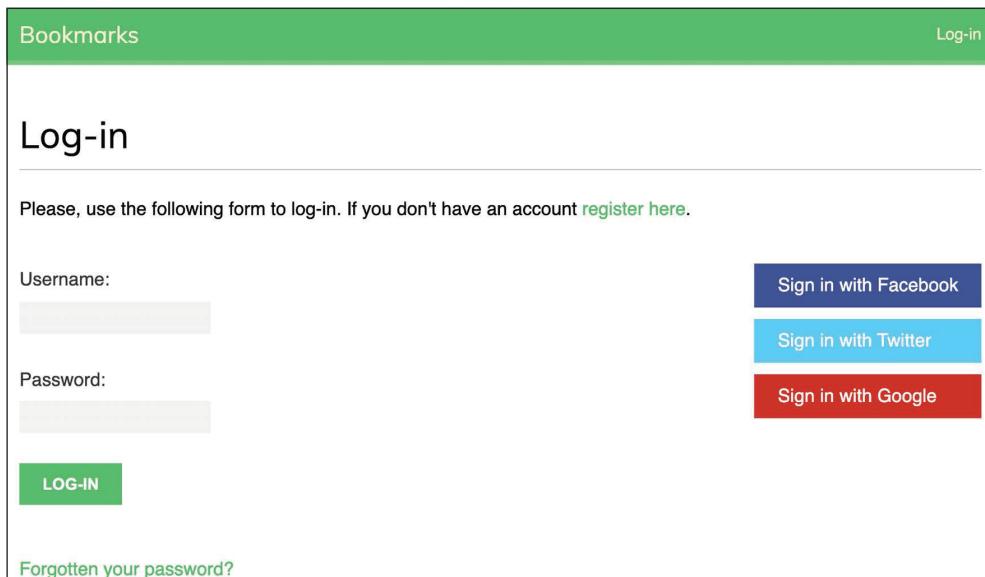


Рис. 5.49. Страница входа, включающая кнопки аутентификации с использованием учетных записей Facebook, Twitter и Google

Кликните по кнопке **Sign in with Google** (Войти с учетной записью Google). Вы увидите следующий ниже экран:

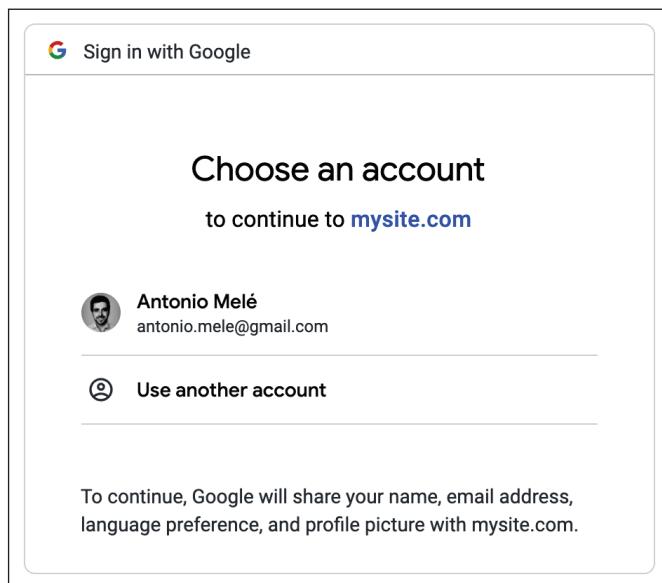


Рис. 5.50. Экран авторизации приложения через Google

Кликните по своей учетной записи Google, чтобы авторизовать приложение. Вы войдете на сайт и будете перенаправлены на страницу информационной панели своего сайта.

Теперь вы добавили социальную аутентификацию в свой проект с помощью нескольких самых популярных социальных платформ. Используя механизм Python Social Auth, можно легко реализовывать социальную аутентификацию с применением других онлайновых сервисов.

Создание профиля пользователей, регистрирующихся посредством социальной аутентификации

Когда пользователь аутентифицируется посредством социальной аутентификации, то если нет существующего пользователя, связанного с этим социальным профилем, создается новый объект `User`. В механизме Python Social Auth используется конвейер, состоящий из набора функций, которые в процессе аутентификации исполняются в определенном порядке. В обязанности этих функций входит получение любых данных о пользователе, создание социального профиля в базе данных и его привязка к существующему пользователю либо создание нового пользователя.

В настоящее время при создании новых пользователей с помощью социальной аутентификации объект `Profile` не создается. Мы добавим в конвейер

новый шаг, чтобы автоматически создавать объект Profile в базе данных при формировании нового пользователя.

Добавьте настроечный параметр SOCIAL_AUTH_PIPELINE в файл settings.py проекта, как показано ниже:

```
SOCIAL_AUTH_PIPELINE = [
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.user.get_username',
    'social_core.pipeline.user.create_user',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
]
```

Это конвейер аутентификации, используемый механизмом Python Social Auth по умолчанию, если не указан иной. Он состоит из нескольких функций, которые выполняют разную работу при аутентификации пользователя. Более подробная информация о стандартном конвейере аутентификации находится на странице <https://python-social-auth.readthedocs.io/en/latest/pipeline.html>.

Давайте разработаем функцию, которая создает объект Profile в базе данных при каждом формировании нового пользователя. Затем мы добавим эту функцию в конвейер социальной аутентификации.

Отредактируйте файл account/authentication.py, добавив следующий ниже исходный код:

```
from account.models import Profile

def create_profile(backend, user, *args, **kwargs):
    """
    Создать профиль пользователя для социальной аутентификации
    """
    Profile.objects.get_or_create(user=user)
```

Функция create_profile принимает два необходимых аргумента:

- `backend`: используемый для аутентификации пользователей бэкенд социальной аутентификации. Напомним, что вы добавили бэкенды социальной аутентификации в настроечный параметр AUTHENTICATION_BACKENDS проекта;
- `user`: экземпляр класса User нового либо существующего пользователя, прошедшего аутентификацию.

Другие аргументы, которые передаются функциям конвейера, можно посмотреть на странице <https://python-social-auth.readthedocs.io/en/latest/pipeline.html#extending-the-pipeline>.

В функции `create_profile` проверяется наличие объекта `user` и используется метод `get_or_create()`, чтобы отыскивать объект `Profile` для данного пользователя, создавая его при необходимости.

Теперь нужно добавить новую функцию в конвейер аутентификации. Добавьте следующую ниже строку, выделенную жирным шрифтом, в параметр `SOCIAL_AUTH_PIPELINE` в файле `settings.py`:

```
SOCIAL_AUTH_PIPELINE = [
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.user.get_username',
    'social_core.pipeline.user.create_user',
    'account.authentication.create_profile',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
]
```

Мы добавили функцию `create_profile` после `social_core.pipeline.create_user`. На этом этапе доступен экземпляр класса `User`. Пользователь может быть существующим либо новым, созданным на этом этапе конвейера. Функция `create_profile` использует экземпляр класса `User`, чтобы отыскивать соответствующий объект `Profile` и создавать новый, если это необходимо.

Обратитесь к списку пользователей на сайте администрирования по адресу <https://mysite.com:8000/admin/auth/user/>. Удалите всех пользователей, созданных с помощью социальной аутентификации.

Затем пройдите по URL-адресу <https://mysite.com:8000/account/login/> и выполните социальную аутентификацию для пользователя, которого вы удалили. Будет создан новый пользователь, а теперь будет создан и объект `Profile`. Обратитесь к адресу <https://mysite.com:8000/admin/account/profile/>, чтобы убедиться, что профиль нового пользователя был создан.

Мы успешно добавили функциональность автоматического создания профиля пользователя для социальной аутентификации.

Механизм Python Social Auth также предлагает конвейерный механизм для потока разъединения. Более подробная информация находится на странице <https://python-social-auth.readthedocs.io/en/latest/pipeline.html#disconnection-pipeline>.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter05>.
- Механизм Python Social Auth: <https://github.com/python-social-auth>.
- Бэкенды аутентификации пакета Python Social Auth: <https://python-social-auth.readthedocs.io/en/latest/backends/index.html#supported-backends>.
- Настройка разрешенных хостов в Django: <https://docs.djangoproject.com/en/4.1/ref/settings/#allowed-hosts>.
- Документация по расширениям Django: <https://django-extensions.readthedocs.io/en/latest/>.
- Портал Facebook для разработчиков: <https://developers.facebook.com/apps/>.
- Приложения для Twitter: <https://developer.twitter.com/en/apps/create>.
- Реализация стандарта OAuth2 в Google: <https://developers.google.com/identity/protocols/OAuth2>.
- Учетные данные API-интерфейсов Google: <https://console.developers.google.com/apis/credentials>.
- Конвейер механизма Python Social Auth: <https://python-social-auth.readthedocs.io/en/latest/pipeline.html>.
- Расширение конвейера механизма Python Social Auth: <https://python-social-auth.readthedocs.io/en/latest/pipeline.html#extending-the-pipeline>.
- Конвейер механизма Python Social Auth для разъединения: <https://python-social-auth.readthedocs.io/en/latest/pipeline.html#disconnection-pipeline>.

Резюме

В этой главе вы добавили социальную аутентификацию на свой сайт, чтобы пользователи имели возможность применять свои существующие учетные записи Facebook, Twitter или Google с целью входа на вашу платформу. Вы использовали механизм Python Social Auth и реализовали социальную аутентификацию с помощью стандартного промышленного протокола авторизации OAuth 2.0. Вы также научились раздавать сервер разработки по HTTPS с помощью пакета расширений Django Extensions. Наконец, вы адаптировали конвейер аутентификации под конкретно-прикладную задачу автоматического создания профилей для новых пользователей.

В следующей главе вы создадите систему управления визуальными прикладками. Вы разработаете модели со взаимосвязями многие-ко-многим и адаптируете поведение форм под конкретно-прикладную задачу. Вы научитесь генерировать миниатюры изображений и создавать AJAX-функции с помощью JavaScript и Django.

6

Распространение контента на веб-сайте

В предыдущей главе вы использовали механизм Django Social Auth, чтобы добавлять социальную аутентификацию на свой сайт при помощи учетных записей Facebook, Google и Twitter. Вы научились раздавать сервер разработки по HTTPS на локальной машине с помощью пакета расширений Django Extensions. Вы адаптировали конвейер социальной аутентификации под конкретно-прикладную задачу автоматического создания профиля для новых пользователей.

В этой главе вы научитесь создавать букмарклет JavaScript, чтобы делиться контентом с других сайтов на вашем сайте, и реализуете в своем проекте функциональные возможности AJAX, используя JavaScript и Django.

В данной главе будут рассмотрены следующие темы:

- создание взаимосвязей многие-ко-многим;
- адаптация поведения форм под конкретно-прикладную задачу;
- использование JavaScript вместе с Django;
- формирование букмарклета JavaScript;
- генерирование миниатюр изображений с помощью easy-thumbnails;
- реализация асинхронных HTTP-запросов с помощью JavaScript и Django;
- разработка бесконечной постраничной прокрутки.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter06>.

Все используемые в данной главе пакеты Python включены в файл requirements.txt в исходном коде к этой главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Создание веб-сайта для управления визуальными закладками

Теперь мы научимся предоставлять пользователям возможность формировать закладки на изображения, которые они находят на других сайтах, и делиться ими на вашем сайте. В целях разработки такой функциональности понадобятся следующие элементы:

- 1) модель данных для хранения изображений и связанной с ними информации;
- 2) форма и представление для работы с закачиванием изображений на сайт;
- 3) исходный код бокмаклета на языке JavaScript, который может исполняться на любом сайте. Этот исходный код будет отыскивать изображения на странице и давать пользователям возможность выбирать изображение, которое они хотят пометить закладкой.

Сначала создайте новое приложение в каталоге проекта `bookmarks`, выполнив следующую ниже команду в командной оболочке:

```
django-admin startapp images
```

Добавьте новое приложение в настроечный параметр `INSTALLED_APPS` в файле `settings.py` проекта, как показано ниже:

```
INSTALLED_APPS = [  
    # ...  
    'images.apps.ImagesConfig',  
]
```

Мы активировали приложение `images` в проекте.

Разработка модели изображения

Отредактируйте файл `models.py` приложения `images`, добавив следующий ниже исходный 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(max_length=2000)
image = models.ImageField(upload_to='images/%Y/%m/%d/')
description = models.TextField(blank=True)
created = models.DateField(auto_now_add=True)

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

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

Это модель, которая будет использоваться для хранения изображений на платформе. Давайте посмотрим на поля данной модели:

- **user**: здесь указывается объект `User`, который сделал закладку на это изображение. Это поле является внешним ключом, поскольку оно определяет взаимосвязь один-ко-многим: пользователь может отправлять несколько изображений, но каждое изображение отправляется одним пользователем. Мы использовали `CASCADE` для параметра `on_delete`, чтобы связанные изображения удалялись при удалении пользователя;
- **title**: заголовок изображения;
- **slug**: короткое обозначение, содержащее только буквы, цифры, подчеркивания или дефисы, которое будет использоваться для создания красивых дружественных для поисковой оптимизации URL-адресов;
- **url**: изначальный URL-адрес этого изображения. Мы используем `max_length`, чтобы определить максимальную длину, равную 2000 символов;
- **image**: файл изображения;
- **description**: optionalное описание изображения;
- **created**: дата и время, когда объект был создан в базе данных. Мы добавили `auto_now_add`, чтобы устанавливать текущее время/дату автоматически при создании объекта.

В `Meta`-классе модели мы определили индекс базы данных в убывающем порядке по полю `created`. Мы также добавили атрибут `ordering`, сообщая Django, что по умолчанию он должен сортировать результаты по созданному полю. Мы указываем убывающий порядок, используя дефис перед именем поля, например `-created`, с тем чтобы новые изображения отображались первыми.



Индексы базы данных повышают производительность запросов. Рассмотрите возможность создания индексов для полей, которые часто запрашиваются методами `filter()`, `exclude()` или `order_by()`. Поля `ForeignKey` либо поля с параметром `unique=True` подразумевают создание индекса. Подробнее об индексах баз данных можно узнать по адресу <https://docs.djangoproject.com/en/4.1/ref/models/options/#django.db.models.Options.indexes>.

Мы переопределим метод `save()` модели `Image`, чтобы автоматически генерировать поля `slug` на основе значения поля `title`. Импортируйте функцию `slugify()` и добавьте метод `save()` в модель `Image`, как показано ниже. Новые строки выделены жирным шрифтом:

```
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().save(*args, **kwargs)
```

Если при сохранении объекта `Image` поле `slug` является пустым, то `slug` генерируется автоматически из поля `title` изображения с помощью функции `slugify()`. Затем объект сохраняется. Благодаря автоматическому генерированию слага из заголовка пользователям не придется указывать слаг, когда они делятся изображениями на сайте.

Создание взаимосвязей многие-ко-многим

Далее мы добавим в модель `Image` еще одно поле, чтобы хранить данные о пользователях, которым понравилось изображение. В данном случае понадобится взаимосвязь многие-ко-многим, поскольку пользователю может нравиться несколько изображений, и каждое изображение может нравиться нескольким пользователям.

Добавьте в модель `Image` следующее ниже поле:

```
users_like = models.ManyToManyField(settings.AUTH_USER_MODEL,
                                    related_name='images_liked',
                                    blank=True)
```

При определении поля `ManyToManyField` Django создает промежуточную таблицу соединения, используя первичные ключи обеих моделей. На рис. 6.1 показана таблица базы данных, которая будет создана для этой взаимосвязи:

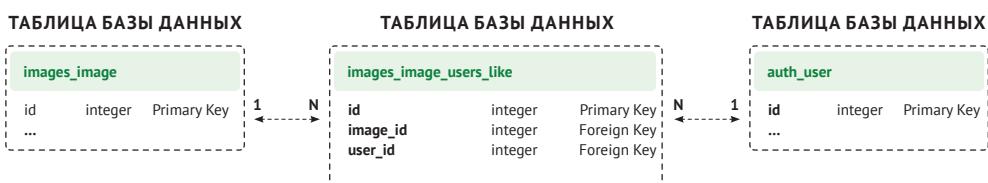


Рис. 6.1. Промежуточная таблица базы данных для взаимосвязи многие-ко-многим

Таблица `images_image_users_like` создается Django как промежуточная таблица, которая имеет ссылки на таблицу `images_image` (модель `Image`) и таблицу `auth_user` (модель `User`). Поле `ManyToManyField` может быть определено в любой из двух связанных моделей.

Как и в случае с полями `ForeignKey`, атрибут `related_name` поля `ManyToManyField` позволяет называть связь из связанного объекта назад к данному объекту. Поля `ManyToManyField` предоставляют менеджер взаимосвязей `многие-ко-многим`, который позволяет получать связанные объекты, например `image.users_like.all()`, либо получать их из объекта пользователя, например `user.images_liked.all()`.

Подробнее о взаимосвязях `многие-ко-многим` можно узнать на странице https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/.

Откройте командную оболочку и выполните следующую ниже команду, чтобы создать первоначальную миграцию:

```
python manage.py makemigrations images
```

Результат должен быть похож на следующий ниже:

```
Migrations for 'images':
  images/migrations/0001_initial.py
    - Create model Image
    - Create index images_imag_created_d57897_idx on field(s) -created of model
```

Теперь выполните следующую ниже команду, чтобы применить миграцию:

```
python manage.py migrate images
```

Вы получите результат, содержащий такую строку:

```
Applying images.0001_initial... OK
```

Теперь модель `Image` синхронизирована с базой данных.

Регистрация модели изображения на сайте администрирования

Отредактируйте файл `admin.py` приложения `images`, чтобы зарегистрировать модель `Image` на сайте администрирования, как показано ниже:

```
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']
```

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver_plus --cert-file cert.crt
```

Пройдите по URL-адресу <https://127.0.0.1:8000/admin/> в своем браузере, и вы увидите модель Image на сайте администрирования, как показано ниже:



Рис. 6.2. Блок *Images* на индексной странице сайта администрирования

Вы завершили создание модели для хранения изображений. И теперь вы научитесь реализовывать форму, чтобы получать изображения по их URL-адресам и сохранять их с помощью модели Image.

Отправка контента с других сайтов

Мы предоставим пользователям возможность отмечать закладкой изображения с внешних сайтов и делиться ими на нашем сайте. Пользователи будут указывать URL-адрес изображения, название и optionalное описание. Мы создадим форму и представление скачивания изображения, а также новый объект Image в базе данных.

Начнем с создания формы для передачи новых изображений на обработку.

Внутри каталога приложения `images` создайте новый файл `forms.py` и добавьте в него следующий ниже исходный 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,
        }
```

Мы определили форму `ModelForm` из модели `Image`, включая только поля `title`, `url` и `description`. Пользователи не будут вводить URL-адрес изображения прямо в форму. Вместо этого мы предоставим им инструмент JavaScript, который позволит им выбирать изображение с внешнего сайта, а форма будет получать URL-адрес изображения в качестве параметра. Мы переопределили стандартный виджет поля `url`, чтобы использовать виджет `HiddenInput`. Этот виджет прорисовывается как HTML-элемент `input` с атрибутом `type="hidden"`. Мы используем данный виджет, потому что не хотим, чтобы это поле было видимым для пользователя.

Очистка полей формы

Для того чтобы убедиться, что предоставленный URL-адрес изображения является валидным, мы проверим, что имя файла заканчивается расширением `.jpg`, `.jpeg` либо `.png`, чтобы разрешить делиться только файлами JPEG и PNG. В целях реализации валидации полей в предыдущей главе мы использовали условное обозначение `clean_<fieldname>()`. Этот метод исполняется для каждого поля при его наличии, когда `is_valid()` вызывается на экземпляре формы. В методе `clean` можно изменять значение поля или вызывать ошибки валидации поля.

В файле `forms.py` приложения `images` добавьте следующий ниже метод в класс `ImageCreateForm`:

```
def clean_url(self):
    url = self.cleaned_data['url']
    valid_extensions = ['jpg', 'jpeg', 'png']
    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
```

Здесь мы определили метод `clean_url()`, чтобы очищать поле `url`. Исходный код работает следующим образом:

- 1) значение поля `url` извлекается путем обращения к словарю `cleaned_data` экземпляра формы;
- 2) URL-адрес разбивается на части, чтобы проверить наличие валидного расширения у файла. Если расширение невалидно, то выдается ошибка `ValidationError`, и экземпляр формы не валидируется.

В дополнение к валидации данного URL-адреса необходимо также скачать файл изображения и сохранить его. Например, мы могли бы использовать представление, которое работает с формой, чтобы скачивать файл изображения. Вместо этого давайте применим более общий подход, переопределив метод `save()` модельной формы, чтобы выполнять эту работу при сохранении формы.

Установка библиотеки `requests`

Когда пользователь помечает изображение закладкой, необходимо скачивать файл изображения по его URL-адресу. Для этого мы будем использовать библиотеку Python под названием `requests`, являющуюся самой популярной HTTP-библиотекой на Python. В ней абстрагируется сложность работы с HTTP-запросами и предоставляется очень простой интерфейс для использования HTTP-сервисов. Документация по библиотеке `requests` находится на странице <https://requests.readthedocs.io/en/master/>.

Следующей ниже командой откройте командную оболочку и установите библиотеку `requests`:

```
pip install requests==2.28.1
```

Теперь мы переопределим метод `save()` класса `ImageCreateForm` и воспользуемся библиотекой `requests`, чтобы получать изображение по его URL-адресу.

Переопределение метода `save()` класса `ModelForm`

Как вы уже знаете, `ModelForm` предоставляет метод `save()`, чтобы сохранять текущий экземпляр модели в базе данных и возвращать объект. Этот метод получает булев параметр `commit`, который позволяет указывать, нужно ли сохранять объект в базе данных. Если `commit` равен `False`, то метод `save()` будет возвращать экземпляр модели, но не будет сохранять его в базе данных. Мы переопределим метод `save()` формы, чтобы получать файл изображения по переданному URL-адресу и сохранить его в файловой системе.

Добавьте инструкции импорта в верхнюю часть файла `forms.py`:

```
from django.core.files.base import ContentFile
from django.utils.text import slugify
import requests
```

Затем добавьте метод `save()` в форму `ImageCreateForm`:

```
def save(self, force_insert=False,
         force_update=False,
         commit=True):
    image = super().save(commit=False)
    image_url = self.cleaned_data['url']
    name = slugify(image.title)
    extension = image_url.rsplit('.', 1)[1].lower()
    image_name = f'{name}.{extension}'
    # скачать изображение с данного URL-адреса
```

```
response = requests.get(image_url)
image.image.save(image_name,
                  ContentFile(response.content),
                  save=False)
if commit:
    image.save()
return image
```

Мы переопределили метод `save()`, сохранив параметры, требуемые классом `ModelForm`. Приведенный выше исходный код объясняется следующим образом.

1. Новый экземпляр изображения создается путем вызова метода `save()` формы с `commit=False`.
2. URL-адрес изображения извлекается из словаря `clean_data` формы.
3. Имя изображения генерируется путем комбинирования названия изображения с изначальным расширением файла изображения.
4. Библиотека Python `requests` используется для скачивания изображения путем отправки HTTP-запроса методом `GET` с использованием URL-адреса изображения. Ответ сохраняется в объекте `response`.
5. Вызывается метод `save()` поля `image`, передавая ему объект `ContentFile`, экземпляр которого заполнен содержимым скачанного файла. Таким путем файл сохраняется в каталог `media` проекта. Параметр `save=False` передается для того, чтобы избежать сохранения объекта в базе данных.
6. Для того чтобы оставить то же поведение, что и в изначальном методе `save()` модельной формы, форма сохраняется в базе данных только в том случае, если параметр `commit` равен `True`.

Теперь необходимо разработать представление для создания экземпляра формы и работы по ее передаче на обработку.

Отредактируйте файл `views.py` приложения `images`, добавив следующий ниже исходный код. Новый исходный код выделен жирным шрифтом:

```
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 в форме валидны
            cd = form.cleaned_data
            new_image = form.save(commit=False)
            # назначить текущего пользователя элементу
```

```

new_image.user = request.user
new_image.save()
messages.success(request,
    'Image added successfully')
# перенаправить к представлению детальной
# информации о только что созданном элементе
return redirect(new_image.get_absolute_url())
else:
    # скомпоновать форму с данными,
    # предоставленными бокмаклетом методом GET
    form = ImageCreateForm(data=request.GET)
return render(request,
    'images/image/create.html',
    {'section': 'images',
     'form': form})

```

Здесь мы создали представление хранения изображений на сайте. В представление `image_create` был добавлен декоратор `login_required`, чтобы предотвращать доступ неавтентифицированных пользователей. Вот как это представление работает.

1. Для создания экземпляра формы необходимо предоставить начальные данные через HTTP-запрос методом `GET`. Эти данные будут состоять из атрибутов `url` и `title` изображения с внешнего веб-сайта. Оба параметра будут заданы в запросе `GET` бокмаклетом JavaScript, который мы создадим позже. Пока же можно допустить, что эти данные будут иметься в запросе.
2. После того как форма передана на обработку с помощью HTTP-запроса методом `POST`, она валидируется методом `form.is_valid()`. Если данные в форме валидны, то создается новый экземпляр `Image` путем сохранения формы методом `form.save(commit=False)`. Новый экземпляр в базе данных не сохраняется, если `commit=False`.
3. В новый экземпляр изображения добавляется связь с текущим пользователем, который выполняет запрос: `new_image.user = request.user`. Так мы будем знать, кто закачивал каждое изображение.
4. Объект `Image` сохраняется в базе данных.
5. Наконец, с помощью встроенного в Django фреймворка сообщений создается сообщение об успехе, и пользователь перенаправляется на канонический URL-адрес нового изображения. Мы еще не реализовали метод `get_absolute_url()` модели `Image`; мы сделаем это позже.

Создайте новый файл `urls.py` внутри приложения `images` и добавьте в него следующий ниже исходный код:

```

from django.urls import path
from . import views

app_name = 'images'

```

```
urlpatterns = [
    path('create/', views.image_create, name='create'),
]
```

Отредактируйте главный файл `urls.py` проекта `bookmarks`, вставив шаблоны для приложения `images`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
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 создать шаблон для прорисовки формы. Внутри каталога приложения `images` создайте следующую ниже каталожную структуру:

```
templates/
    images/
        image/
            create.html
```

Отредактируйте новый шаблон `create.html`, добавив следующий ниже исходный код:

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

{% block title %}Bookmark an image{% endblock %}

{% block content %}
    <h1>Bookmark an image</h1>
    
    <form method="post">
        {{ form.as_p }}
        {% csrf_token %}
        <input type="submit" value="Bookmark it!">
    </form>
{% endblock %}
```

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver_plus --cert-file cert.crt
```

Пройдите по URL-адресу `https://127.0.0.1:8000/images/create/?title=...&url=...` в своем браузере, включив GET-параметры `title` и `url` и указав в последнем параметре URL-адрес существующего JPEG-изображения. Например, можно использовать следующий URL-адрес: `https://127.0.0.1:8000/images/create/?title=%20Django%20and%20Duke&url=https://upload.wikimedia.org/wikipedia/commons/8/85/Django_Reinhardt_and_Duke_Ellington_%28Gottlieb%29.jpg`.

Вы увидите форму с предварительным просмотром изображения, как показано ниже:



Рис. 6.3. Страница новой закладки на изображение

Добавьте описание и кликните по кнопке **BOOKMARK IT!** (Пометить закладкой). Новый объект `Image` будет сохранен в базе данных. Однако будет получена ошибка, указывающая на то, что модель `Image` не имеет метода `get_absolute_url()`, как показано ниже:

AttributeError

```
AttributeError: 'Image' object has no attribute 'get_absolute_url'
```

Рис. 6.4. Ошибка, показывающая, что объект `Image` не имеет атрибута `get_absolute_url`

Пока не беспокойтесь об этой ошибке; чуть позже мы реализуем метод `get_absolute_url` в модели `Image`.

Пройдите по URL-адресу `https://127.0.0.1:8000/admin/images/image/` в своем браузере и проверьте, чтобы новый объект `image` был сохранен, как показано ниже:

Action:	TITLE	SLUG	IMAGE	CREATED
<input type="checkbox"/>	Django and Duke	django-and-duke	images/2022/01/10/django-and-duke.jpg	Jan. 10, 2022

1 image

Рис. 6.5. Страница списка изображений на сайте администрирования, показывающая созданный объект `Image`

Разработка бокмаклета с помощью JavaScript

Бокмаклет – это закладка, хранящаяся в веб-браузере и содержащая исходный код JavaScript с целью расширения функциональности браузера. При нажатии на закладку в браузерной строке *Bookmarks* (Закладки) или панели *Favorites* (Избранное) на отображаемом в браузере веб-сайте исполняется исходный код JavaScript. Такой подход очень полезен для создания инструментов, которые взаимодействуют с другими веб-сайтами.

В некоторых онлайновых сервисах, таких как Pinterest, используется свой собственный бокмаклет, чтобы предоставлять пользователям возможность делиться контентом с других сайтов на своей платформе. Бокмаклет Pinterest, именуемый *браузерной кнопкой*, доступен на странице <https://about.pinterest.com/en/browser-button>. Бокмаклет Pinterest предоставляется как расширение для Google Chrome, дополнение для Microsoft Edge или обычный бокмаклет JavaScript для Safari и других браузеров, который можно перетаскивать в панель закладок браузера. Закладка позволяет пользователям сохранять изображения или веб-сайты в своей учетной записи Pinterest (рис. 6.6).

Давайте создадим бокмаклет для вашего сайта аналогичным образом. Для этого мы будем использовать JavaScript.

Вот как ваши пользователи будут добавлять бокмаклет в свой браузер и его использовать:

- 1) пользователь перетаскивает ссылку с вашего сайта на панель закладок своего браузера. Ссылка содержит исходный код JavaScript в атрибуте `href`. Этот исходный код будет сохранен в закладке;
- 2) пользователь переходит на любой веб-сайт и кликает по закладке в панели закладок или панели избранного. Исходный код JavaScript закладки будет выполнен.

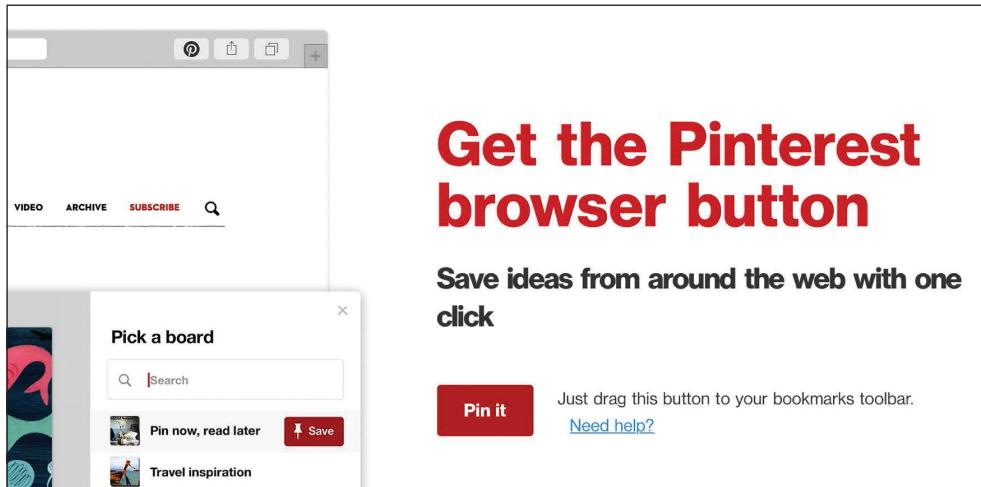


Рис. 6.6. Букмаклет *Pin it* в Pinterest

Поскольку исходный код JavaScript будет храниться в закладке, мы не сможем обновлять его после того, как пользователь добавит его в свою панель закладок. Это важный недостаток, который можно устранить, реализовав запускающий скрипт. Пользователи будут сохранять запускающий скрипт как закладку, а запускающий скрипт будет загружать фактический букмаклет JavaScript с URL-адреса. Благодаря этому появляется возможность обновлять исходный код букмаклета в любое время. Именно такой подход мы будем использовать для разработки букмаклета. Давайте начнем!

Создайте новый шаблон в папке `images/templates/` и назовите его `bookmarklet_launcher.js`. Это будет запускающий скрипт. Добавьте в новый файл следующий ниже исходный код JavaScript:

```
(function(){
  if(!window.bookmarklet) {
    bookmarklet_js = document.body.appendChild(document.
      createElement('script'));
    bookmarklet_js.src = '//127.0.0.1:8000/static/js/bookmarklet.js?r=' +Math.
      floor(Math.random()*9999999999999999);
    window.bookmarklet = true;
  }
  else {
    bookmarkletLaunch();
  }
})();
```

Приведенный выше скрипт проверяет, не был ли букмаклет уже загружен, проверяя значение переменной окна `bookmarklet` с помощью булева выражения `if(!window.bookmarklet)`:

- если `window.bookmarklet` не определена или не имеет истинного значения (считается `true` в булевом контексте), то файл JavaScript загружается путем добавления элемента `<script>` в тело HTML-документа, загруженного в браузер. Атрибут `src` используется для загрузки URL-адреса скрипта `bookmarklet.js` со случайным 16-значным целочисленным параметром, генерируемым посредством `Math.random()*9999999999999999`. Используя случайное число, мы предотвращаем загрузку файла из кеша браузера. Если JavaScript букмаклета был загружен ранее, то другое значение данного параметра заставит браузер снова загрузить скрипт с исходного URL-адреса. При таком подходе мы обеспечиваем, чтобы букмаклет всегда запускал самый актуальный исходный код JavaScript;
- если `window.bookmarklet` определена и имеет истинное значение, то исполняется функция `bookmarkletLaunch()`. Мы определим функцию `bookmarkletLaunch()` как глобальную в скрипте `bookmarklet.js`.

Проверяя переменную окна `bookmarklet`, мы предотвращаем загрузку исходного кода букмаклета более одного раза, если пользователь неоднократно кликает на букмаклете.

Вы создали исходный код запуска букмаклета. Фактический исходный код букмаклета будет находиться в статическом файле `bookmarklet.js`. Применение запускающего исходного кода позволяет обновлять исходный код букмаклета в любое время, не требуя от пользователей изменения закладки, которую они ранее добавили в браузер.

Давайте добавим программу запуска букмаклета на страницы информационной панели, чтобы пользователи имели возможность добавлять его в панель закладок своего браузера.

Отредактируйте шаблон `account/dashboard.html` приложения `account`, приведя его к следующему виду. Новые строки выделены жирным шрифтом:

```
{% 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" %}">
        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 %}
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк; в Django многострочные теги не поддерживаются.

Теперь на информационной панели отображается общее число изображений, помеченных пользователем закладкой. Мы добавили шаблонный тег `{% with %}`, чтобы создавать переменную с общим числом помеченных закладкой изображений текущего пользователя. Мы вставили ссылку с атрибутом `href`, содержащую скрипт запуска бокмаклкета. Этот исходный код JavaScript загружается из шаблона `bookmarklet_launcher.js`.

Пройдите по URL-адресу `https://127.0.0.1:8000/account/` в своем браузере. Вы должны увидеть следующую ниже страницу:

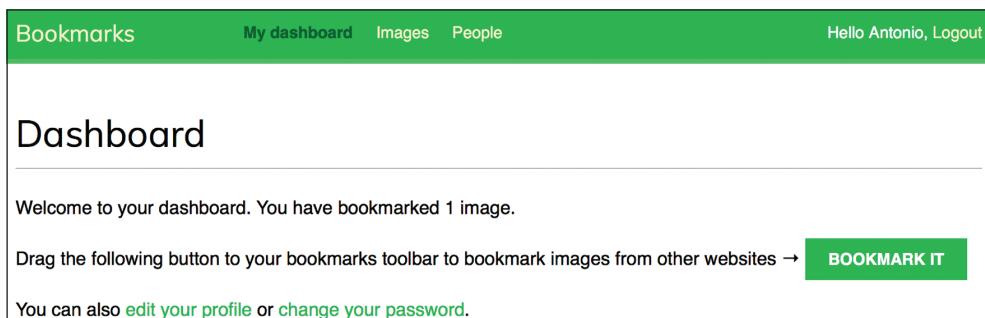


Рис. 6.7. Страница информационной панели, в том числе общее число помеченных закладкой изображений и кнопка для бокмаклкета

Теперь внутри каталога приложения `images` создайте следующие ниже каталоги и файлы:

```
static/
  js/
    bookmarklet.js
```

Каталог `static/css/` находится в каталоге приложения `images` в исходном коде, который прилагается к этой главе. Скопируйте каталог `css/` в каталог `static/` вашего исходного кода. Содержимое каталога находится на странице <https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter06/bookmarks/images/static>.

Файл `css/bookmarklet.css` содержит стили для бокмаклкета JavaScript. Теперь каталог `static/` должен содержать следующую ниже файловую структуру:

```
css/  
  bookmarklet.css  
js/  
  bookmarklet.js
```

Отредактируйте статический файл `bookmarklet.js`, добавив следующий ниже исходный код JavaScript:

```
const siteUrl = '//127.0.0.1:8000/';  
const styleUrl = siteUrl + 'static/css/bookmarklet.css';  
const minWidth = 250;  
const minHeight = 250;
```

Вы объявили четыре константы, которые будут использоваться букмаклелом. Эти константы таковы:

- `siteUrl` и `styleUrl`: базовый URL-адрес веб-сайта и базовый URL-адрес статических файлов;
- `minWidth` и `minHeight`: минимальная ширина и высота изображений, которые букмаклел будет забирать с сайта, в пикселях. Букмаклел будет выявлять изображения, ширина которых составляет не менее 250px каждая.

Отредактируйте статический файл `bookmarklet.js`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
const siteUrl = '//127.0.0.1:8000/';  
const styleUrl = siteUrl + 'static/css/bookmarklet.css';  
const minWidth = 250;  
const minHeight = 250;  
  
// загрузить CSS  
var head = document.getElementsByTagName('head')[0];  
var link = document.createElement('link');  
link.rel = 'stylesheet';  
link.type = 'text/css';  
link.href = styleUrl + '?r=' + Math.floor(Math.random()*9999999999999999);  
head.appendChild(link);
```

Этот раздел загружает таблицу стилей CSS для букмаклела. Мы используем JavaScript для манипулирования объектной моделью документа **DOM (Document Object Model)**. DOM представляет собой HTML-документ в памяти и создается браузером при загрузке веб-страницы. DOM строится как дерево объектов, которые составляют структуру и содержимое HTML-документа.

Приведенный выше исходный код генерирует объект, эквивалентный следующему ниже исходному коду JavaScript, и добавляет его в элемент `<head>` HTML-страницы:

```
<link rel="stylesheet" type="text/css" href= "//127.0.0.1:8000/static/css/bookmarklet.css?r=1234567890123456">
```

Давайте рассмотрим, как это делается.

- С помощью `document.getElementsByTagName()` извлекается элемент `<head>` сайта. Эта функция извлекает все HTML-элементы страницы с переданным тегом. Используя `[0]`, мы получаем доступ к первому найденному элементу. Мы обращаемся к первому элементу, потому что все HTML-документы должны иметь один элемент `<head>`.
- С помощью `document.createElement('link')` создается элемент `<link>`.
- Устанавливаются атрибуты `rel` и `type` элемента `<link>`. Это эквивалентно HTML `<link rel="stylesheet" type="text/css">`.
- В атрибуте `href` элемента `<link>` задается URL-адрес таблицы стилей `bookmarklet.css`. В качестве параметра URL-адреса используется 16-значное случайное число, чтобы не дать браузеру загружать файл из кеша.
- С помощью `head.appendChild(link)` новый элемент `<link>` добавляется в элемент `<head>` HTML-страницы.

Теперь мы создадим HTML-элемент, чтобы отображать на веб-сайте конейнер, в котором букмарклет будет исполняться. HTML-контейнер будет использоваться для отображения всех найденных на сайте изображений и предоставит пользователям возможность выбирать изображение, которым они хотят поделиться. В нем будут использоваться стили CSS, определенные в таблице стилей `bookmarklet.css`.

Отредактируйте статический файл `bookmarklet.js`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
const siteUrl = '//127.0.0.1:8000/';
const styleUrl = siteUrl + 'static/css/bookmarklet.css';
const minWidth = 250;
const minHeight = 250;

// загрузить CSS
var head = document.getElementsByTagName('head')[0];
var link = document.createElement('link');
link.rel = 'stylesheet';
link.type = 'text/css';
link.href = styleUrl + '?r=' + Math.floor(Math.random()*9999999999999999);
head.appendChild(link);

// загрузить HTML
var body = document.getElementsByTagName('body')[0];
boxHtml = '
<div id="bookmarklet">
  <a href="#" id="close">&times;</a>
  <h1>Select an image to bookmark:</h1>
  <div class="images"></div>
'
```

```
</div>';
body.innerHTML += boxHTML;
```

С помощью этого исходного кода извлекается элемент `<body>` DOM-модели, и в него добавляется новый исходный код HTML путем видоизменения его свойства `innerHTML`. В тело страницы добавляется новый элемент `<div>`. Контеинер `<div>` состоит из следующих элементов:

- ссылка для закрытия контеинера, определенная с помощью `×`;
- название, определенное с помощью `<h1>Select an image to bookmark:</h1>`;
- элемент `<div>` для списка изображений, найденных на сайте, определенный с помощью `<div class="images"></div>`. Этот контеинер изначально пуст и будет заполнен изображениями, найденными на сайте.

Контеинер HTML, включая ранее загруженные стили CSS, будет выглядеть так, как показано на рис. 6.8.

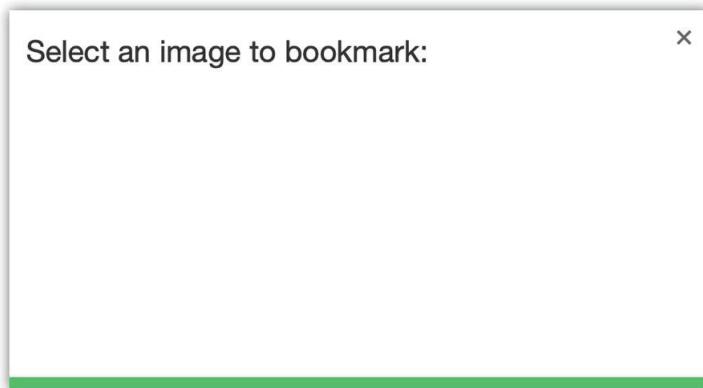


Рис. 6.8. Контеинер для выбора изображения

Теперь давайте реализуем функцию запуска бокмаклета. Отредактируйте статический файл `bookmarklet.js`, добавив следующий исходный код в нижнюю его часть:

```
function bookmarkletLaunch() {
  bookmarklet = document.getElementById('bookmarklet');
  var imagesFound = bookmarklet.querySelector('.images');

  // очистить найденные изображения
  imagesFound.innerHTML = '';
  // показать бокмаклет
  bookmarklet.style.display = 'block';
```

```
// событие закрытия
bookmarklet.querySelector('#close')
    .addEventListener('click', function(){
        bookmarklet.style.display = 'none'
    });
}

// запустить бокмаклет
bookmarkletLaunch();
```

Это функция `bookmarkletLaunch()`. Перед определением этой функции загружается CSS бокмаклера и в DOM страницы добавляется HTML-контейнер. Функция `bookmarkletLaunch()` работает следующим образом.

1. Извлекается главный контейнер бокмаклера путем получения элемента DOM с ИД бокмаклера с помощью `document.getElementById()`.
2. Элемент `bookmarklet` используется для извлечения дочернего элемента с классом `images`. Метод `querySelector()` позволяет извлекать элементы DOM с помощью селекторов CSS. Селекторы дают возможность отыскивать элементы DOM, к которым применяется набор правил CSS. Список селекторов CSS находится на странице https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors, а дополнительную информацию о том, как находить элементы DOM с помощью селекторов, можно почитать на странице https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Location_DOM_elements_using_selectors.
3. Контейнер `images` очищается путем установки его атрибута `innerHTML` равным пустой строке, а бокмаклер отображается на странице путем установки значения свойства CSS `display` равным `block`.
4. Селектор `#close` используется для отыскания элемента DOM с ИД `close`. Событие `click` прикрепляется к этому элементу с помощью метода `addEventListener()`. Когда пользователи кликают на элементе, главный контейнер бокмаклера скрывается путем установки значения его свойства `display` равным `none`.

Функция `bookmarkletLaunch()` исполняется после ее определения.

После загрузки стилей CSS и HTML-контейнера бокмаклера необходимо найти элементы изображения в DOM текущей веб-страницы. Изображения с минимально необходимым размером должны добавляться в HTML-контейнер бокмаклера. Отредактируйте статический файл `bookmarklet.js`, добавив следующий ниже исходный код, выделенный жирным шрифтом, в конец функции `bookmarklet()`:

```
function bookmarkletLaunch() {
    bookmarklet = document.getElementById('bookmarklet');
    var imagesFound = bookmarklet.querySelector('.images');

    // очистить найденные изображения
    imagesFound.innerHTML = '';
```

```
// показать букмаклет
bookmarklet.style.display = 'block';

// событие закрытия
bookmarklet.querySelector('#close')
    .addEventListener('click', function(){
        bookmarklet.style.display = 'none'
    });

// найти изображения в DOM с минимальными размерами
images = document.querySelectorAll('img[src$=".jpg"], img[src$=".jpeg"], img[src$=".png"]');
images.forEach(image => {
    if(image.naturalWidth >= minWidth
        && image.naturalHeight >= minHeight)
    {
        var imageFound = document.createElement('img');
        imageFound.src = image.src;
        imagesFound.append(imageFound);
    }
})
}

// запустить букмаклет
bookmarkletLaunch();
```

В приведенном выше исходном коде селекторы `img[src$=".jpg"]`, `img[src$=".jpeg"]` и `img[src$=".png"]` используются для отыскания всех элементов DOM ``, атрибут `src` которых заканчивается соответственно на `.jpg`, `.jpeg` либо `.png`. Использование этих селекторов с `document.querySelectorAll()` позволяет отыскивать все изображения в формате JPEG и PNG, отображаемые на веб-сайте. Прокручивание результатов в цикле выполняется методом `forEach()`. Маленькие изображения отфильтровываются, потому что они не считаются релевантными. В качестве результатов рассматриваются только те изображения, размер которых превышает размер, указанный в переменных `minWidth` и `minHeight`. Для каждого найденного изображения создается новый элемент ``, при этом атрибут `src`, содержащий источник URL-адреса, копируется из изначального изображения и добавляется в контейнер `imagesFound`.

Из соображений безопасности ваш браузер не разрешит запускать букмаклет по HTTP на сайте, раздаваемом по HTTPS. Именно по этой причине для работы сервера разработки мы продолжаем использовать RunServerPlus с применением автоматически генерированного сертификата TLS/SSL. Напомним, что вы научились обеспечивать работу сервера разработки по HTTPS в главе 5 «Реализация социальной аутентификации».

В производственной среде потребуется валидный сертификат TLS/SSL. Владея доменным именем, можно обратиться в доверяемый центр сертификации (CA) с просьбой выпустить для него сертификат TLS/SSL, чтобы

браузеры могли верифицировать его подлинность. Если вы хотите получить доверяемый сертификат для реального домена, то можете воспользоваться сервисом *Let's Encrypt*. *Let's Encrypt*¹ – это некоммерческий центр сертификации, который бесплатно упрощает получение и обновление доверяемых сертификатов TLS/SSL. Более подробная информация по нему находится на странице <https://letsencrypt.org>.

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver_plus --cert-file cert.crt
```

Пройдите по URL-адресу <https://127.0.0.1:8000/account/> в своем браузере. Войдите на сайт под существующим пользователем, затем кликните по кнопке **BOOKMARK IT** (Пометить закладкой) и перетащите ее в панель закладок браузера, как показано ниже:



Рис. 6.9. Добавление кнопки **BOOKMARK IT** на панель закладок

Откройте в браузере выбранный вами сайт и кликните по кнопке **Bookmark it** в панели закладок. Вы увидите, что на сайте появится новое белое наложение, отображающее все найденные изображения JPEG и PNG размером более 250×250 пикселов. На рис. 6.10 показан букмарклет, работающий на сайте <https://amazon.com/>.

Если HTML-контейнер не появляется, то следует проверить журнал консоли оболочки RunServer. Если вы увидите ошибку MIME-типа, то, скорее всего, ваши файлы соотнесения MIME неверны либо нуждаются в обновлении. Правильное соотнесение файлов JavaScript и CSS применяется путем добавления следующей ниже строки в файл `settings.py`:

```
if DEBUG:
    import mimetypes
    mimetypes.add_type('application/javascript', '.js', True)
    mimetypes.add_type('text/css', '.css', True)
```

¹ Переводится как «Давайте зашифруем». – Прим. перев.



Рис. 6.10. Букмарклет, загруженный на amazon.com

HTML-контейнер содержит изображения, которые можно помечать закладкой. Теперь давайте реализуем функциональность, позволяющую пользователям кликать на нужном изображении, чтобы помечать его закладкой.

Откройте статический файл `js/bookmarklet.js` и добавьте следующий ниже исходный код в нижнюю часть функции `bookmarklet()`:

```
function bookmarkletLaunch() {
    bookmarklet = document.getElementById('bookmarklet');
    var imagesFound = bookmarklet.querySelector('.images');

    // очистить найденные изображения
    imagesFound.innerHTML = '';

    // показать букмарклет
    bookmarklet.style.display = 'block';

    // событие закрытия
    bookmarklet.querySelector('#close')
        .addEventListener('click', function(){
            bookmarklet.style.display = 'none'
        });

    // найти изображения в DOM с минимальными размерами
    images = document.querySelectorAll('img[src$=".jpg"], img[src$=".jpeg"], img[src$=".png"]');
    images.forEach(image => {
```

```
if(image.naturalWidth >= minWidth
    && image.naturalHeight >= minHeight)
{
    var imageFound = document.createElement('img');
    imageFound.src = image.src;
    imagesFound.append(imageFound);
}
})

// событие выбора изображения
imagesFound.querySelectorAll('img').forEach(image => {
    image.addEventListener('click', function(event){
        imageSelected = event.target;
        bookmarklet.style.display = 'none';
        window.open(siteUrl + 'images/create/?url='
            + encodeURIComponent(imageSelected.src)
            + '&title='
            + encodeURIComponent(document.title),
            '_blank');
    })
})
}

// запустить бокмаклет
bookmarkletLaunch();
```

Приведенный выше исходный код работает следующим образом.

1. К каждому элементу изображения в контейнере `imagesFound` прикрепляется событие `click()`.
2. Когда пользователь кликает по любому изображению, элемент изображения, на котором было сделано нажатие, сохраняется в переменной `imageSelected`.
3. Затем бокмаклет скрывается путем установки свойства `display` равным `none`.
4. Открывается новое окно браузера с URL-адресом закладки нового изображения на сайте. Содержимое элемента `<title>` веб-сайта передается в URL-адрес в параметре `title` HTTP-метода GET, а URL-адрес выбранного изображения – в параметре `url`.

Пройдите по новому URL-адресу в своем браузере, например <https://commons.wikimedia.org/>, как показано на рис. 6.11.



Кликните по бокмаклету **Bookmark it**, чтобы отобразить наложение, которое позволит выбрать изображения. Вы увидите его, как показано ниже:



Рис. 6.11. Веб-сайт Wikimedia Commons

Изображение на рис. 6.11–6.14: Стая журавлей (*Grus grus*) в долине Хула, Северный Израиль, работа Томере (Лицензия: Creative Commons Attribution-Share Alike 4.0 International: <https://creativecommons.org/licenses/by-sa/4.0/deed.en>).



Рис. 6.12. Букмарклет, загруженный на внешний веб-сайт

Если вы кликнете на изображении, то будете перенаправлены на страницу создания закладки на изображение, передав название сайта и URL-адрес вы-

бранного изображения в качестве GET-параметров. Страница будет выглядеть следующим образом:



Рис. 6.13. Форма для нанесения закладки на изображение

Поздравляем! Это был ваш первый букмарклет на JavaScript, и он полностью интегрирован в ваш проект Django. Далее мы создадим представление детальной информации об изображениях и реализуем для них канонический URL-адрес.

Создание представления детальной информации об изображениях

Теперь давайте создадим простое представление детальной информации, чтобы отображать изображения, которые были помечены закладками на сайте. Откройте файл `views.py` приложения `images` и добавьте в него следующий ниже исходный код:

```
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 простое представление вывода изображения на страницу. Отредактируйте файл `urls.py` приложения `images`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [
    path('create/', views.image_create, name='create'),
    path('detail///', bold(
        views.image_detail, name='detail'),
]
```

Отредактируйте файл `models.py` приложения `images`, добавив метод `get_absolute_url()` в модель `Image`, как показано ниже:

```
from django.urls import reverse

class Image(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('images:detail', args=[self.id,
                                              self.slug])
```

Напомним, что общепринятым способом предоставления канонических URL-адресов объектам является определение метода `get_absolute_url()` в модели.

Наконец, внутри каталога шаблонов `/templates/images/image/` создайте шаблон для приложения `images` и назовите его `detail.html`. Добавьте в него следующий ниже исходный код:

```
{% 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>
</div>
```

```
  {{ image.description|linebreaks }}
```

```
</div>
```

```
<div class="image-likes">
```

```
{% for user in image.users_like.all %}
```

```
  <div>
```

```
    {% if user.profile.photo %}
```

```
      
```

```
    {% endif %}
```

```
    <p>{{ user.first_name }}</p>
```

```
  </div>
```

```
{% empty %}
```

```
  Nobody likes this image yet.
```

```
{% endfor %}
```

```
</div>
```

```
{% endwith %}
```

```
{% endblock %}
```

Это шаблон, который будет отображать представление детальной информации о помеченном закладкой изображении. Мы использовали тег `{% with %}`, чтобы создавать переменную `total_likes` с помощью результата набора запросов `QuerySet`, который подсчитывает все «лайки» пользователей. Благодаря этому мы избегаем двойного вычисления одного и того же набора запросов (сначала – чтобы вывести на страницу общее количество лайков, а затем – чтобы использовать шаблонный фильтр `pluralize`). Мы также вставили описание изображения и добавили цикл `{% for %}`, чтобы прокручивать `image.users_like.all` в цикле для отображения всех пользователей, которым нравится это изображение.



Всякий раз, когда в шаблоне требуется повторить запрос, следует использовать шаблонный тег `{% with %}`. Это делается во избежание дополнительных запросов к базе данных.

Теперь откройте в браузере внешний URL-адрес и примените букмаклет, чтобы пометить закладкой новое изображение. После отправки изображения вы будете перенаправлены на страницу детальной информации об изображении. Данная страница будет содержать сообщение об успехе, как показано ниже:

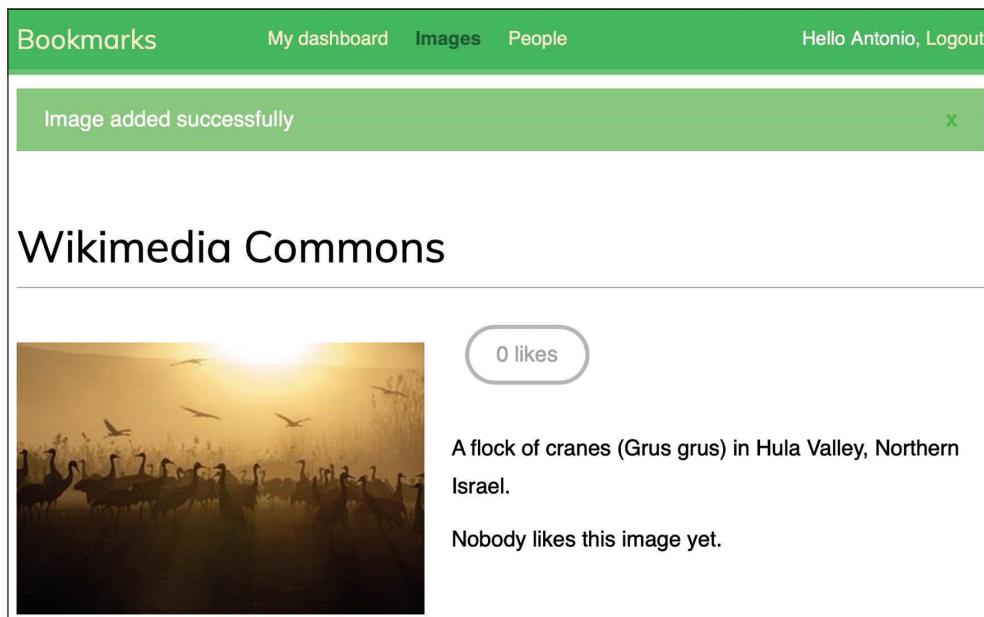


Рис. 6.14. Страница детальной информации об изображении для помеченного закладкой изображения

Отлично! Вы завершили работу с функциональностью букмарклета. Далее вы научитесь создавать миниатюры изображений.

Создание миниатюр изображений с помощью easy-thumbnails

На странице детальной информации отображается исходное изображение, но размеры разных изображений могут разительно отличаться. Размер файла некоторых изображений бывает очень большим, и их загрузка может занимать слишком много времени. Лучший способ единообразного отображения оптимизированных изображений состоит в генерировании миниатюр. Миниатюра – это маленькое изображение, представляющее более крупное изображение. Миниатюры быстрее загружаются в браузер и являются отличным способом унифицировать изображения разных размеров. Мы будем использовать приложение Django под названием `easy-thumbnails`, чтобы генерировать миниатюры изображений, помечаемых пользователями закладками.

Следующей ниже командой откройте терминал и установите `easy-thumbnails`:

```
pip install easy-thumbnails==2.8.1
```

Отредактируйте файл `settings.py` проекта `bookmarks`, добавив `easy_thumbnails` в настроечный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [  
    # ...  
    'easy_thumbnails',  
]
```

Затем выполните следующую ниже команду, чтобы синхронизировать приложение с базой данных:

```
python manage.py migrate
```

Вы увидите результат, включающий такие строки:

```
Applying easy_thumbnails.0001_initial... OK  
Applying easy_thumbnails.0002_thumbnaildimensions... OK
```

Приложение `easy-thumbnails` предлагает различные способы формирования миниатюр изображений. Оно предоставляет шаблонный тег `{% thumbnail %}`, чтобы генерировать миниатюры в шаблонах и конкретно-прикладное поле `ImageField`, если вы хотите формировать миниатюры в своих моделях. Давайте воспользуемся шаблонным тегом.

Откройте шаблон `images/image/detail.html` и взгляните на следующую ниже строку:

```

```

Следующие ниже строки должны заменить предыдущую:

```
{% load thumbnail %}  
<a href="{{ image.image.url }}>  
      
</a>
```

Мы определили миниатюру с фиксированной шириной 300 пикселов и гибкой высотой, чтобы поддерживать соотношение сторон, используя значение 0. Когда пользователь загружает эту страницу в первый раз, будет сформировано изображение-миниатюра. Миниатюра хранится в том же каталоге, что и исходный файл. Местоположение определяется настроечным параметром

MEDIA_ROOT и атрибутом upload_to поля image модели Image. Сгенерированная миниатюра будет раздаваться в последующих запросах.

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver_plus --cert-file cert.crt
```

Зайдите на страницу детальной информации о существующем изображении. Будет сгенерирована миниатюра, которая отобразится на сайте. Кликните правой кнопкой мыши по изображению и откройте его в новой вкладке браузера, как показано ниже:



Рис. 6.15. Открыть изображение в новой вкладке браузера

Проверьте URL-адрес сгенерированного изображения в своем браузере. Он должен выглядеть следующим образом:



127.0.0.1:8000/media/images/2022/01/10/django-and-duke.jpg.300x0_q85.jpg ↗

Рис. 6.16. URL-адрес сгенерированного изображения

За исходным именем файла следует дополнительная информация о настройках, использованных для создания миниатюры. В случае данного изображения JPEG вы увидите имя файла наподобие `filename.jpg.300x0_q85.jpg`, где `300x0` – это параметры размера, используемые для генерирования миниатюры, а `85` – значение качества JPEG, используемого библиотекой по умолчанию для генерирования миниатюры.

С помощью параметра `quality` можно использовать другое значение качества. Для установки самого высокого качества JPEG можно использовать значение `100`, как показано ниже: `{% thumbnail image.image 300x0 quality=100 %}`. Более высокое качество подразумевает более крупный размер файла.

Приложение `easy-thumbnails` предлагает несколько вариантов адаптации миниатюр под конкретно-прикладную задачу, включая алгоритмы обрезки и различные эффекты. Если с генерированием миниатюр возникнут проблемы, то в файл `settings.py` можно добавить настроечный параметр `THUMBNAIL_DEBUG = True`, чтобы получать отладочную информацию. Полную документацию по `easy-thumbnails` можно почитать на странице <https://easy-thumbnails.readthedocs.io/>.

Добавление асинхронных действий с помощью JavaScript

На страницу детальной информации об изображении мы добавим кнопку `like` (нравится), чтобы пользователи имели возможность по ней кликать, помечая изображение своим лайком. Когда пользователи будут кликать по кнопке `like`, мы будем отправлять HTTP-запрос на веб-сервер с помощью JavaScript. Такой подход позволит выполнять действие `like` без перезагрузки всей страницы. В целях обеспечения этой функциональности мы реализуем представление, которое позволяет пользователям ставить/убирать лайк (`like/unlike`) изображениям.

Интерфейс Fetch API на JavaScript – это встроенный способ выполнения асинхронных HTTP-запросов к веб-серверам из веб-браузеров. Используя интерфейс Fetch API, можно отправлять данные на веб-сервер и получать их с веб-сервера без необходимости обновления всей страницы. Fetch API был запущен как современный преемник встроенного в браузеры объекта XMLHttpRequest (XHR), используемого для выполнения HTTP-запросов без перезагрузки страницы. Набор приемов веб-разработки для асинхронной отправки данных на веб-сервер и получения данных с веб-сервера без перезагрузки страницы также называется AJAX, что расшифровывается как Asynchronous

JavaScript and XML¹. Название AJAX вводит в заблуждение, поскольку запросы AJAX могут обмениваться данными не только в формате XML, но и в таких форматах, как JSON, HTML и обычный текст. В интернете можно встретить упоминания об API Fetch и AJAX во взаимозаменяемой форме.

Информация об интерфейсе Fetch API находится на странице https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch.

Мы начнем с реализации представления действияй *like* (нравится) и *unlike* (не нравится), а затем добавим исходный код JavaScript в соответствующий шаблон, чтобы выполнять асинхронные HTTP-запросы.

Отредактируйте файл `views.py` приложения `images`, добавив следующий ниже исходный код:

```
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 Image.DoesNotExist:
            pass
    return JsonResponse({'status': 'error'})
```

В новом представлении использованы два декоратора. Декоратор `login_required` не дает пользователям, не вошедшим в систему, обращаться к этому представлению. Декоратор `require_POST` возвращает объект `HttpResponseNotAllowed` (код состояния, равный 405), в случае если HTTP-запрос выполнен не методом POST. При таком подходе этому представлению разрешаются запросы только методом POST.

Django также предоставляет декоратор `require_GET`, чтобы разрешать запросы только методом GET, и декоратор `require_http_methods`, на вход которого можно передавать список разрешенных методов.

Указанное представление ожидает следующие ниже POST-параметры:

- `image_id`: ИД объекта изображения, над которым пользователь выполняет действие;

¹ Асинхронный JavaScript и XML. – Прим. перев.

- `action`: действие, которое пользователь хочет выполнить. Оно должно быть строковым литералом со значением `like` либо `unlike`.

Мы использовали менеджер, предоставляемый веб-фреймворком Django для поля `users_like` со взаимосвязью многие-ко-многим модели `Image`, для того чтобы добавлять либо удалять объекты из связи методами `add()` или `remove()`. Если метод `add()` вызывается, передавая объект, который уже присутствует в наборе связанных объектов, то этот объект не будет дублироваться. Если метод `remove()` вызывается с объектом, которого нет в наборе связанных объектов, то ничего не произойдет. Еще одним полезным методом менеджера взаимосвязей многие-ко-многим является `clear()`, который удаляет все объекты из набора связанных объектов.

Для того что сгенерировать ответ представления, мы использовали класс `JsonResponse` веб-фреймворка Django, который возвращает HTTP-ответ с типом контента `application/json`, конвертируя переданный объект в JSON.

Отредактируйте файл `urls.py` приложения `images`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [
    path('create/', views.image_create, name='create'),
    path('detail///', views.image_detail, name='detail'),
    path('like/', views.image_like, name='like'),
]
```

Загрузка JavaScript в DOM

Далее нужно добавить исходный код JavaScript в шаблон детальной информации об изображении. Для того чтобы применять JavaScript в своих шаблонах, сначала в шаблон `base.html` проекта добавляется базовая обертка.

Отредактируйте шаблон `base.html` приложения `account`, вставив следующий ниже исходный код, выделенный жирным шрифтом, перед закрывающим HTML-тегом `</body>`:

```
<!DOCTYPE html>
<html>
<head>
...
</head>
<body>
...
<script>
    document.addEventListener('DOMContentLoaded', (event) => {
        // DOM загружена
        {% block domready %}
        {% endblock %}
    })

```

```
})
</script>
</body>
</html>
```

Мы добавили тег `<script>`, чтобы вставить исходный код JavaScript. Метод `document.addEventListener()` используется для определения функции, которая будет вызываться при наступлении заданного события. Мы передаем имя события `DOMContentLoaded`, которое срабатывает, когда первоначальный HTML-документ полностью загружен и иерархия Объектной модели документа **DOM (Document Object Model)** полностью построена. Используя это событие, мы обеспечиваем, чтобы DOM была построена полностью, перед тем как взаимодействовать с HTML-элементами и манипулировать DOM-моделью. Исходный код внутри функции будет исполнен только после того, как DOM будет готова.

Внутри обработчика готовности документа мы включили блок шаблонов Django под названием `domready`. Любой шаблон, расширяющий шаблон `base.html`, может использовать этот блок для вставки того или иного исходного кода JavaScript, который будет исполняться, когда DOM будет готова.

Не путайте исходный код JavaScript и шаблонные теги Django. Язык шаблонов Django используется на стороне сервера для генерирования HTML-документа, а JavaScript исполняется в браузере на стороне клиента. В некоторых случаях удобно генерировать исходный код JavaScript динамически с помощью Django, чтобы иметь возможность использовать результаты наборов запросов `QuerySet` или вычислений на стороне сервера для определения значений переменных JavaScript.

Примеры этой главы содержат исходный код JavaScript в шаблонах Django. Предпочтительным методом добавления исходного кода JavaScript в свои шаблоны является загрузка `.js`-файлов, раздаваемых сервером как статические файлы, в особенности если используются большие скрипты.

Защита от подделки межсайтовых HTTP-запросов на JavaScript

Вы узнали о подделке межсайтовых запросов **CSRF (cross-site request forgery)** в главе 2 «Усовершенствование блога за счет продвинутых функциональностей». В случае активной защиты от CSRF Django ищет токен CSRF во всех запросах, отправляемых методом `POST`. При передаче формы на обработку можно использовать шаблонный тег `{% csrf_token %}`, чтобы отправлять токен вместе с формой. Выполняемые на JavaScript HTTP-запросы должны также передавать токен CSRF в каждом запросе, отправляемом методом `POST`.

Django позволяет задавать конкретно-прикладной заголовок `X-CSRFToken` в HTTP-запросах со значением токена CSRF.

Для того чтобы включить токен в исходящие от JavaScript HTTP-запросы, необходимо получить токен CSRF из cookie-файла `csrftoken`, который зада-

ется веб-фреймворком Django, в случае если защита CSRF активна. Для работы с cookie-файлами мы будем использовать библиотеку JavaScript Cookie. JavaScript Cookie – это легковесный API на JavaScript для работы с cookie-файлами. Подробнее о ней можно узнать на странице <https://github.com/js-cookie/js-cookie>.

Отредактируйте шаблон `base.html` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом, в нижнюю часть элемента `<body>`, как показано ниже:

```
<!DOCTYPE html>
<html>
<head>
...
</head>
<body>
...
<script src="//cdn.jsdelivr.net/npm/js-cookie@3.0.1/dist/js.cookie.min.js"></script>
<script>
const csrftoken = Cookies.get('csrftoken');
document.addEventListener('DOMContentLoaded', (event) => {
    // DOM загружена
    {% block domready %}
    {% endblock %}
})
</script>
</body>
</html>
```

Мы реализовали следующую функциональность:

- 1) плагин JS Cookie загружается из публичной сети доставки контента CDN¹;
- 2) значение cookie-файла `csrftoken` извлекается методом `Cookies.get()` и сохраняется в константе JavaScript `csrftoken`.

Все запросы JavaScript на доставку ресурса, в которых используются небезопасные HTTP-методы, такие как POST или PUT, необходимо вставлять токен CSRF. Позже мы будем вставлять константу `csrftoken` в конкретно-прикладной HTTP-заголовок X-CSRFToken при отправке HTTP-запросов методом POST.

Более подробная информация о защите от CSRF в Django и в AJAX находится по адресу <https://docs.djangoproject.com/en/4.1/ref/csrf/#ajax>.

Далее мы реализуем исходный код HTML и JavaScript, для того чтобы пользователи имели возможность ставить лайк/дизлайк изображениям.

¹ Англ. Content Delivery Network – Прим. перев.

Выполнение HTTP-запросов с помощью JavaScript

Отредактируйте шаблон `images/image/detail.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

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

{% block title %}{{ image.title }}{% endblock %}

{% block content %}
<h1>{{ image.title }}</h1>
{% load thumbnail %}
<a href="{{ image.image.url }}">

</a>
{% with total_likes=image.users_like.count users_like=image.users_like.all %}
<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 %}unlike{% endif %}" class="like button">
{% if request.user not in users_like %}
    Like
{% else %}
    Unlike
{% endif %}
</a>
</div>
{{ image.description|linebreaks }}
</div>
<div class="image-likes">
{% for user in users_like %}
<div>
    {% if user.profile.photo %}
        
    {% endif %}
        <p>{{ user.first_name }}</p>
</div>
    {% empty %}
        Nobody likes this image yet.
    {% endfor %}
</div>
{% endwith %}
{% endblock %}
```

В приведенном выше исходном коде в шаблонный тег `{% with %}` была добавлена еще одна переменная, чтобы хранить результаты запроса `image.users_like.all` и избегать многократного исполнения запроса к базе данных. Указанная переменная используется для проверки наличия текущего пользователя в списке с помощью `{% if request.user in users_like %}`, а затем с помощью `{% if request.user not in users_like %}`. Эта же переменная используется для прокручивания пользователей, которые поставили этому изображению лайк, в цикле с помощью `{% for user in users_like %}`.

На указанную страницу мы добавили общее число пользователей, которые поставили изображению лайк, и добавили ссылку, по которой пользователь может поставить лайк/дизлайк изображению. Связанный набор объектов, `users_like`, используется для проверки наличия `request.user` в связанном наборе объектов, для отображения текста *Like* (Понравилось) или *Unlike* (Не понравилось) на основе текущих взаимосвязей между пользователем и этим изображением. Следующие ниже атрибуты были добавлены в ссылочный HTML-элемент `<a>`:

- `data-id`: ИД выводимого на странице изображения;
- `data-action`: действие, выполняемое, когда пользователь кликает по ссылке. Им может быть `like` либо `unlike`.



Любой атрибут любого HTML-элемента, имя которого начинается с `data`, является атрибутом данных. Атрибуты данных используются для хранения конкретно-прикладных данных вашего приложения.

Теперь мы отправим значения атрибутов `data-id` и `data-action` в HTTP-запросе в представление `image_like`. Когда пользователь кликает по ссылке `like/unlike`, в браузере необходимо выполнить следующие ниже действия:

- 1) отправить HTTP-запрос методом POST в представление `image_like`, передав ему `id` изображения и параметры `action`;
- 2) если HTTP-запрос был успешным, то обновить атрибут `data-action` HTML-элемента `<a>` противоположным действием (`like/unlike`) и соответствующим образом видоизменить его текст, который отображается на странице;
- 3) обновить общее число лайков, отображаемых на странице.

Добавьте следующий ниже блок `domready` в нижней части шаблона `images/image/detail.html`:

```
{% block domready %}  
const url = '{% url "images:like" %}';  
var options = {  
    method: 'POST',  
    headers: {'X-CSRFToken': csrfToken},  
    mode: 'same-origin'  
}
```

```
document.querySelector('a.like')
    .addEventListener('click', function(e){
        e.preventDefault();
        var likeButton = this;
    });
{%- endblock %}
```

Приведенный выше исходный код работает следующим образом.

1. Шаблонный тег `{% url %}` используется для формирования URL-адреса `images:like`. Генерированный URL-адрес сохраняется в константе JavaScript `url`.
2. Создается объект `options` с опциями, которые будут переданы в HTTP-запрос с помощью Fetch API. Они таковы:
 - `method`: используемый HTTP-метод. В данном случае это `POST`;
 - `headers`: дополнительные включаемые в запрос HTTP-заголовки. Мы включаем заголовок `X-CSRFToken` с константой `csrfToken`, значение которой мы определили в шаблоне `base.html`;
 - `mode`: режим HTTP-запроса. Мы используем `same-origin`, чтобы указывать, что запрос делается к тому же источнику. Дополнительная информация о режимах находится на странице [https://developer.mozilla.org/en-US/docs/Web/API/Request mode](https://developer.mozilla.org/en-US/docs/Web/API/Request	mode).
3. Селектор `a.like` используется для отыскания всех элементов `<a>` HTML-документа с классом `like` посредством `document.querySelector()`.
4. Прослушиватель событий определен для события клика по элементам, выбираемым селектором. Эта функция исполняется всякий раз, когда пользователь кликает по ссылке `like/unlike`.
5. Внутри функции-обработчика используется `e.preventDefault()`, чтобы избежать стандартного поведения элемента `<a>`. Такой подход предотвратит стандартное поведение ссылочного элемента, остановит распространение события и предотвратит переход по ссылке на URL-адрес.
6. Переменная `likeButton` используется для хранения ссылки на элемент `this`, для которого было инициировано событие.

Теперь нужно отправить HTTP-запрос с помощью Fetch API. Отредактируйте блок `domready` шаблона `images/image/detail.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
{% block domready %}
const url = '{% url "images:like" %}';
var options = {
    method: 'POST',
    headers: {'X-CSRFToken': csrfToken},
    mode: 'same-origin'
}

document.querySelector('a.like')
    .addEventListener('click', function(e){
```

```
e.preventDefault();
var likeButton = this;

// добавить тело запроса
var formData = new FormData();
formData.append('id', likeButton.dataset.id);
formData.append('action', likeButton.dataset.action);
options['body'] = formData;

// отправить HTTP-запрос
fetch(url, options)
.then(response => response.json())
.then(data => {
  if (data['status'] === 'ok')
  {
  }
})
);
{%
  endblock %}
```

Новый исходный код работает следующим образом.

- Объект `FormData` создается для формирования набора пар ключ/значение, представляющих поля формы и их значения. Указанный объект хранится в переменной `formData`.
- Ожидаемые встроенным в Django представлением `image_like` параметры `id` и `action` добавляются в объект `formData`. Значения этих параметров извлекаются из нажатого элемента `likeButton`. Доступ к атрибутам `data-id` и `data-action` осуществляется посредством `dataset.id` и `dataset.action`.
- В объект `options` добавляется новый ключ `body`, который будет использоваться для HTTP-запроса. Значением этого ключа является объект `formData`.
- Fetch API используется путем вызова функции `fetch()`. Определенная ранее переменная `url` передается в качестве запрашиваемого URL-адреса, а объект `options` передается в качестве опций запроса.
- Функция `fetch()` возвращает обещание, которое конвертируется с помощью объекта `Response`, являющегося представлением HTTP-ответа. Метод `.then()` используется для формирования обработчика обещания. С целью извлечения содержимого тела JSON используется `response.json()`. Подробнее об объекте `Response` можно узнать на странице <https://developer.mozilla.org/en-US/docs/Web/API/Response>.
- Метод `.then()` используется снова, чтобы определить обработчика для извлеченных из JSON данных. В этом обработчике атрибут `status` полученных данных проверяется на наличие у него значения `ok`.

Мы добавили функциональность по отправке HTTP-запроса и обработке ответа. После успешного запроса нужно изменить кнопку и связанное с ней

действие на противоположное: с *like* на *unlike* либо с *unlike* на *like*. Благодаря этому пользователи смогут отменять свое действие.

Отредактируйте блок `domready` шаблона `images/image/detail.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
{% block domready %}

const url = '{% url "images:like" %}';
var options = {
    method: 'POST',
    headers: {'X-CSRFToken': csrfToken},
    mode: 'same-origin'
}

document.querySelector('a.like')
    .addEventListener('click', function(e){
        e.preventDefault();
        var likeButton = this;

        // добавить тело запроса
        var formData = new FormData();
        formData.append('id', likeButton.dataset.id);
        formData.append('action', likeButton.dataset.action);
        options['body'] = formData;

        // отправить HTTP-запрос
        fetch(url, options)
            .then(response => response.json())
            .then(data => {
                if (data['status'] === 'ok') {
                    var previousAction = likeButton.dataset.action;

                    // переключить текст кнопки и атрибут data-action
                    var action = previousAction === 'like' ? 'unlike' : 'like';
                    likeButton.dataset.action = action;
                    likeButton.innerHTML = action;

                    // обновить количество лайков
                    var likeCount = document.querySelector('span.count .total');
                    var totalLikes = parseInt(likeCount.innerHTML);
                    likeCount.innerHTML = previousAction === 'like' ? totalLikes + 1 : totalLikes - 1;
                }
            })
        });
    });

{% endblock %}
```

Приведенный выше исходный код работает следующим образом.

- Предыдущее действие кнопки извлекается из атрибута `data-action` ссылки и сохраняется в переменной `previousAction`.
- Переключается атрибут `data-action` ссылки и текст ссылки, что позволяет пользователям отменять свое действие.
- Общее количество лайков извлекается из DOM с помощью селектора `span.count.total`, а значение конвертируется в целое число посредством `parseInt()`. Общее количество лайков увеличивается или уменьшается в зависимости от выполненного действия (`like` либо `unlike`).

Откройте в своем браузере страницу детальной информации об изображении, которое вы закачали на сайт. Вы должны увидеть следующее ниже первоначальное количество лайков и кнопку **LIKE** (Нравится), как показано ниже:



Рис. 6.17. Количество лайков и кнопка **LIKE**
в шаблоне детальной информации об изображении

Кликните по кнопке **LIKE** (Нравится). Вы заметите, что общее количество лайков увеличилось на единицу, а текст кнопки изменился на **UNLIKE** (Не нравится), как показано ниже:



Рис. 6.18. Количество лайков
и кнопка после нажатия кнопки **LIKE**

Если кликнуть по кнопке **UNLIKE**, то действие будет выполнено, а затем текст кнопки снова изменится на **LIKE**, и общее количество соответствующим образом изменится.

При программировании на JavaScript, в особенности при выполнении AJAX-запросов, рекомендуется использовать инструмент отладки JavaScript и HTTP-запросов. Большинство современных браузеров содержат инструменты разработчика для отладки JavaScript. Для этого обычно нужно кликнуть правой кнопкой мыши в любом месте веб-сайта, чтобы открыть контекстное меню, и нажать **Inspect** (Проинспектировать) или **Inspect Element** (Проинспектировать элемент), чтобы получить доступ к браузерным инструментам веб-разработчика.

В следующем далее разделе вы научитесь использовать асинхронные HTTP-запросы с помощью JavaScript и Django, чтобы реализовывать бесконечную постраничную прокрутку.

Добавление бесконечной постраничной прокрутки в список изображений

Далее нужно вывести список всех изображений, помеченных закладкой на веб-сайте. Мы будем использовать запросы JavaScript, чтобы разработать функциональность бесконечной прокрутки. Бесконечная прокрутка достигается за счет автоматической загрузки следующих результатов, когда пользователь прокручивает страницу до конца.

Давайте реализуем представление списка изображений, которое будет обрабатывать как стандартные запросы браузера, так и запросы, исходящие из JavaScript. Когда пользователь будет загружать страницу списка изображений в первый раз, мы будем отображать первую страницу изображений. Когда он будет прокручивать страницу до конца, мы будем получать следующую страницу элементов с помощью JavaScript и добавлять ее в конец главной страницы.

Одно и то же представление будет обрабатывать как стандартную, так и AJAX-ориентированную бесконечную постраничную прокрутку. Отредактируйте файл `views.py` приложения `images`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.http import HttpResponseRedirect
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')
    images_only = request.GET.get('images_only')
    try:
        images = paginator.page(page)
    except PageNotAnInteger:
        # Если страница не является целым числом,
        # то доставить первую страницу
        images = paginator.page(1)
    except EmptyPage:
        if images_only:
            # Если AJAX-запрос и страница вне диапазона,
            # то вернуть пустую страницу
            return HttpResponseRedirect('')
```

```

# Если страница вне диапазона,
# то вернуть последнюю страницу результатов
images = paginator.page(paginator.num_pages)

if images_only:
    return render(request,
                  'images/image/list_images.html',
                  {'section': 'images',
                   'images': images})

return render(request,
              'images/image/list.html',
              {'section': 'images',
               'images': images})

```

В этом представлении создается набор запросов `QuerySet`, чтобы извлекать все изображения из базы данных. Затем формируется объект `Paginator`, чтобы разбивать результаты на страницы, беря по восемь изображений на страницу. Извлекается HTTP GET-параметр `page`, чтобы получить запрошенный номер страницы. Извлекается HTTP GET-параметр `images_only`, чтобы узнать, должна ли прорисовываться вся страница целиком или же только новые изображения. Мы будем прорисовывать всю страницу целиком, когда она запрашивается браузером. Однако мы будем прорисовывать HTML только с новыми изображениями в случае запросов Fetch API, поскольку мы будем добавлять их в существующую HTML-страницу.

Иключение `EmptyPage` будет вызываться в случае, если запрошенная страница находится вне допустимого диапазона. Если это так и нужно прорисовывать только изображения, то будет возвращаться пустой `HttpResponse`. Такой подход позволит останавливать AJAX-ориентированное постраничное разбиение на стороне клиента при достижении последней страницы. Результаты прорисовываются с использованием двух разных шаблонов:

- в случае HTTP-запросов на JavaScript, которые будут содержать параметр `images_only`, будет прорисовываться шаблон `list_images.html`. Этот шаблон будет содержать изображения только запрошенной страницы;
- в случае браузерных запросов будет прорисовываться шаблон `list.html`. Этот шаблон будет расширять шаблон `base.html`, чтобы отображать всю страницу целиком, и будет вставлять шаблон `list_images.html`, который будет вставлять список изображений.

Отредактируйте файл `urls.py` приложения `images`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```

urlpatterns = [
    path('create/', views.image_create, name='create'),
    path('detail/<int:id>/<slug:slug>',
         views.image_detail, name='detail'),
    path('like/', views.image_like, name='like'),
    path('', views.image_list, name='list'),
]

```

Наконец, необходимо создать упомянутые здесь шаблоны. Внутри каталога `images/image/template` создайте новый шаблон и назовите его `list_images.html`. Добавьте в него следующий ниже исходный код:

```
{% load thumbnail %}  
{% for image in images %}  
    <div class="image">  
        <a href="{{ image.get_absolute_url }}>  
            {% thumbnail image.image 300x300 crop="smart" as im %}  
            <a href="{{ image.get_absolute_url }}>  
                  
            <a href="{{ image.get_absolute_url }}" class="title">  
                {{ image.title }}  
            </a>  
        </div>  
    </div>  
{% endfor %}
```

Приведенный выше шаблон отображает список изображений. Он будет использоваться для того, чтобы возвращать результаты AJAX-запросов. В показанном выше исходном коде изображения в списке прокручиваются в цикле, и по каждому изображению генерируется квадратная миниатюра. Размер миниатюр нормализуется до 300×300 пикселов. Также используется опция умной обрезки `smart`. Данная опция указывает на то, что изображение должно постепенно обрезаться до требуемого размера путем удаления срезов с краев с наименьшей энтропией.

В том же каталоге создайте еще один шаблон и назовите его `images/image/list.html`. Добавьте в него следующий ниже исходный код:

```
{% extends "base.html" %}  
  
{% block title %}Images bookmarked{% endblock %}  
  
{% block content %}  
    <h1>Images bookmarked</h1>  
    <div id="image-list">  
        {% include "images/image/list_images.html" %}  
    </div>  
{% endblock %}
```

Шаблон списка расширяет шаблон `base.html`. Во избежание дублирования исходного кода вставляется шаблон `images/image/list_images.html`, служащий для вывода изображений на страницу. Шаблон `images/image/list.html` будет

содержать исходный код JavaScript для загрузки дополнительных страниц при прокрутке страницы до самого ее низа.

Отредактируйте шаблон `images/image/list.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

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

{% block title %}Images bookmarked{% endblock %}

{% block content %}
<h1>Images bookmarked</h1>
<div id="image-list">
    {% include "images/image/list_images.html" %}
</div>
{% endblock %}

{% block domready %}
var page = 1;
var emptyPage = false;
var blockRequest = false;

window.addEventListener('scroll', function(e) {
    var margin = document.body.clientHeight - window.innerHeight - 200;
    if(window.pageYOffset > margin && !emptyPage && !blockRequest) {
        blockRequest = true;
        page += 1;

        fetch('?images_only=1&page=' + page)
        .then(response => response.text())
        .then(html => {
            if (html === '') {
                emptyPage = true;
            }
            else {
                var imageList = document.getElementById('image-list');
                imageList.insertAdjacentHTML('beforeEnd', html);
                blockRequest = false;
            }
        })
    }
});

// Запустить события прокрутки
const scrollEvent = new Event('scroll');
window.dispatchEvent(scrollEvent);
{% endblock %}
```

Приведенный выше исходный код обеспечивает функциональность бесконечной прокрутки. Исходный код JavaScript вставляется в блок `domready`, определенный в шаблоне `base.html`. Исходный код работает следующим образом.

1. Определяются переменные:
 - `page`: сохраняет текущий номер страницы;
 - `empty_page`: позволяет узнать, не находится ли пользователь на последней странице и не извлекает ли он пустую страницу. Как только будет получена пустая страница, перестанут отправляться дополнительные HTTP-запросы, потому что будет считаться, что результатов больше нет;
 - `block_request`: запрещает отправку дополнительных запросов во время выполнения HTTP-запроса.
2. Применяется `window.addEventListener()`, чтобы захватывать событие прокрутки и определять для него функцию-обработчик.
3. Вычисляется переменная `margin`, чтобы получать разницу между общей высотой документа и внутренней высотой окна, потому что эта высота, на которую пользователь может прокручивать оставшийся контент. Из результата вычитается значение `200`, чтобы следующая страница загружалась, когда пользователь будет находиться ближе, чем на `200` пикселов к нижней части страницы.
4. Перед отправкой HTTP-запроса делается проверка, что:
 - смещение `window.pageYOffset` превышает рассчитанный край `margin`;
 - пользователь не попал на последнюю страницу результатов (`emptyPage` должно быть `false`);
 - нет другого текущего HTTP-запроса (`blockRequest` должен иметь значение `false`).
5. Если предыдущие условия соблюdenы, то `blockRequest` устанавливается равным `true`, чтобы событие прокрутки не запускало дополнительные HTTP-запросы, и счетчик страниц увеличивается на `1`, чтобы получить следующую страницу.
6. Используется `fetch()`, чтобы отправлять HTTP-запрос методом `GET`, устанавливая параметры URL-адреса `image_only=1` для получения HTML только для изображений вместо всей HTML-страницы, и `page` для запрошенного номера страницы.
7. Содержимое тела извлекается из HTTP-ответа методом `response.text()`, и возвращаемый HTML обрабатывается в согласованном порядке:
 - **если в ответе содержимого нет:** был достигнут конец результатов, и подлежащих загрузке страниц больше нет. Значение `emptyPage` устанавливается равным `true`, чтобы предотвратить дополнительные HTTP-запросы;
 - **если ответ содержит данные:** в HTML-элемент с ИД `image-list` добавляются данные. Содержимое страницы расширяется по вертикали, добавляя результаты, когда пользователь приближается к нижней части страницы. Блокировка дополнительных HTTP-запросов снимается, устанавливая значение `blockRequest` равным `false`.

8. Под прослушивателем событий имитируется первоначальное событие прокрутки после загрузки страницы. Событие возникает, создавая новый объект Event, а затем оно запускается методом `window.dispatchEvent()`. Благодаря этому обеспечивается, чтобы событие срабатывало в случае, если первоначальный контент укладывается в окно и не имеет прокрутки.

Пройдите по URL-адресу <https://127.0.0.1:8000/images/> в своем браузере. Вы увидите список изображений, которые уже пометили закладкой. Он должен выглядеть примерно так:

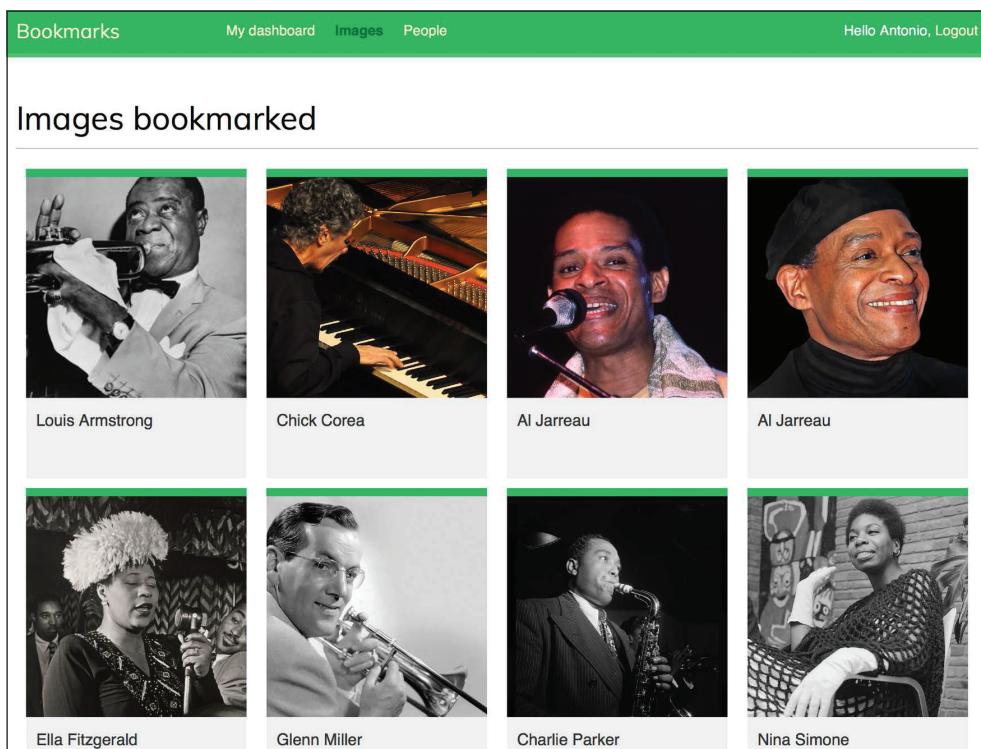


Рис. 6.19. Страница бесконечной постраничной прокрутки со списком изображений

Авторство изображений на рис. 6.19:

- Чик Кориа, автор: ataelw (лицензия: Creative Commons Attribution 2.0 Типовая форма: <https://creativecommons.org/licenses/by/2.0/>);
- Эл Джарро – Дюссельдорф, 1981, автор: Эдди Лауманс, он же RX-Guru (лицензия: Creative Commons Attribution 3.0 Непортируированная форма: <https://creativecommons.org/licenses/by/3.0/>);
- Эл Джарро, авторы: Kingkongphoto и www.celebrity-photos.com (лицензия: Creative Commons Attribution-ShareAlike 2.0 Типовая форма: <https://creativecommons.org/licenses/by-sa/2.0/>).

Прокрутите страницу вниз, чтобы загрузить дополнительные страницы. Проверьте, чтобы с помощью букмарклета было помечено закладкой более восьми изображений, потому что именно это число изображений отображается на каждой странице.

Вдобавок можно использовать браузерные инструменты разработчика, чтобы отслеживать AJAX-запросы.

Для этого обычно нужно кликнуть правой кнопкой мыши в любом месте веб-сайта, чтобы открыть контекстное меню, и нажать **Inspect** (Проинспектировать) или **Inspect Element** (Проинспектировать элемент) для получения доступа к браузерным инструментам веб-разработчика. Найдите панель сетевых запросов. Перезагрузите страницу и прокрутите страницу вниз, чтобы загрузить новые страницы. Вы увидите запрос первой страницы и AJAX-запросы дополнительных страниц, как на рис. 6.20:

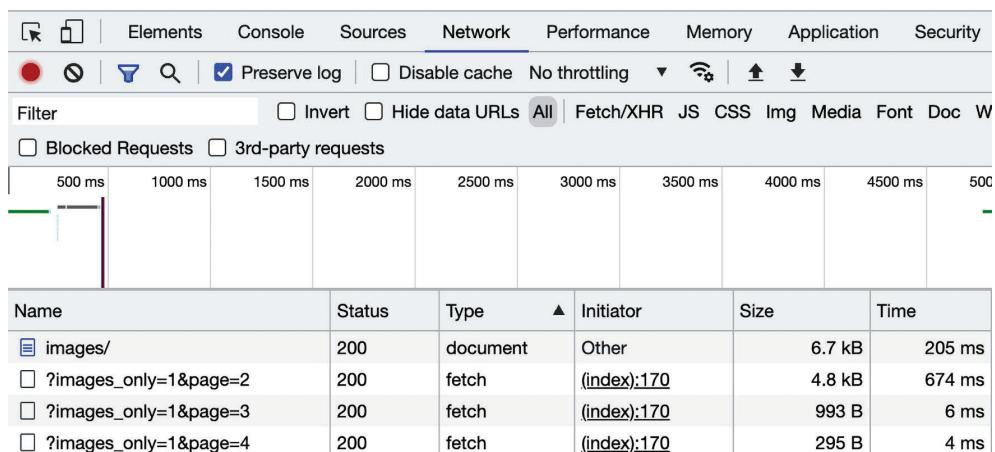


Рис. 6.20. HTTP-запросы, регистрируемые в браузерных инструментах разработчика

В командной оболочке, в которой вы запускаете Django, вы также увидите запросы, представленные в следующем виде:

```
[08/Aug/2022 08:14:20] "GET /images/ HTTP/1.1" 200
[08/Aug/2022 08:14:25] "GET /images/?images_only=1&page=2 HTTP/1.1" 200
[08/Aug/2022 08:14:26] "GET /images/?images_only=1&page=3 HTTP/1.1" 200
[08/Aug/2022 08:14:26] "GET /images/?images_only=1&page=4 HTTP/1.1" 200
```

Наконец, отредактируйте шаблон `base.html` приложения `account`, добавив URL-адрес элемента `images`, выделенного жирным шрифтом:

```
<ul class="menu">
...
<li {% if section == "images" %}class="selected"{% endif %}>
```

```
<a href="{% url "images:list" %}">Images</a>
</li>
...
</ul>
```

Теперь к списку изображений можно обращаться из главного меню.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter06>.
- Индексы баз данных: <https://docs.djangoproject.com/en/4.1/ref/models/options/#django.db.models.Options.indexes>.
- Взаимосвязи многие-ко-многим: https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/.
- Библиотека Python requests: <https://requests.readthedocs.io/en/master/>.
- Браузерная кнопка Pinterest: <https://about.pinterest.com/en/browser-button>.
- Статический контент для приложения account: <https://github.com/Packt-Publishing/Django-4-by-Example/tree/main/Chapter06/bookmarks/images/static>.
- Селекторы CSS: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors.
- Отыскание элементов DOM с помощью селекторов CSS: https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Location_DOM_elements_using_selectors.
- Бесплатный автоматизированный центр сертификации Let's Encrypt: <https://letsencrypt.org>.
- Приложение Django easy-thumbnails: <https://easy-thumbnails.readthedocs.io/>.
- Использование интерфейса JavaScript Fetch API: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch.
- Библиотека JavaScript Cookie: <https://github.com/js-cookie/js-cookie>.
- Защита от CSRF в Django и в AJAX: <https://docs.djangoproject.com/en/4.1/ref/csrf/#ajax>.
- Режим запроса в интерфейсе JavaScript Fetch API: <https://developer.mozilla.org/en-US/docs/Web/API/Request/mode>.
- Ответ в интерфейсе JavaScript Fetch API: <https://developer.mozilla.org/en-US/docs/Web/API/Response>.

Резюме

В этой главе вы создали модели со взаимосвязями многие-ко-многим и научились адаптировать поведение форм под конкретно-прикладную задачу. Вы создали бокмаклет JavaScript, чтобы делиться изображениями из других веб-сайтов на своем сайте. Здесь также было рассмотрено создание миниатюр изображений с помощью приложения `easy-thumbnails`. Наконец, вы реализовали представления на базе AJAX посредством интерфейса JavaScript Fetch API и добавили бесконечную постраничную прокрутку в представление списка изображений.

В следующей главе вы научитесь разрабатывать систему подписки и поток активности. Вы поработаете с обобщенными отношениями, сигналами и денормализацией. Вы также научитесь использовать быстрое хранилище данных Redis вместе с Django, чтобы вести подсчет просмотров изображений и генерировать рейтинг изображений.

7

Отслеживание действий пользователя

В предыдущей главе вы разработали бокмаклет JavaScript, чтобы делиться контентом с других веб-сайтов на своей платформе. Вы также внедрили в свой проект асинхронные действия с помощью JavaScript и создали бесконечную прокрутку.

В этой главе вы научитесь разрабатывать систему подписки и создавать поток активности пользователей. Вы также узнаете, как работают сигналы Django, и интегрируете в свой проект хранилище данных Redis с быстрым вводом-выводом с целью хранения определенного количества просмотров элементов.

В данной главе будут рассмотрены следующие темы:

- разработка системы подписки;
- создание взаимосвязей многие-ко-многим с промежуточной моделью;
- создание приложения потока активности;
- добавление обобщенных отношений в модели;
- оптимизация наборов запросов для связанных объектов;
- использование сигналов для денормализации количественных данных;
- использование меню отладочных инструментов Django Debug Toolbar для получения соответствующей отладочной информации;
- ведение подсчета просмотров изображений с помощью хранилища Redis;
- создание рейтинга самых просматриваемых изображений с помощью хранилища Redis.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter07>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Разработка системы подписки

Давайте разработаем для вашего проекта систему подписки. Под ней подразумевается, что ваши пользователи смогут подписываться друг на друга и отслеживать то, чем другие пользователи делятся на платформе. Взаимосвязь между пользователями классифицируется как связь многие-ко-многим: пользователь может следить за несколькими пользователями, а за ним, в свою очередь, могут следить нескольких пользователей.

Формирование взаимосвязей многие-ко-многим с промежуточной моделью

В предыдущих главах вы создавали взаимосвязи многие-ко-многим, добавляя `ManyToManyField` в одну из связанных моделей и позволяя Django создавать таблицу базы данных для этой взаимосвязи. Такой подход применим для большинства случаев, но иногда для подобной взаимосвязи возникает потребность в создании промежуточной модели. Создание промежуточной модели необходимо, когда требуется хранить дополнительную информацию о взаимосвязи, например дату создания взаимосвязи или поле, описывающее природу взаимосвязи.

Давайте создадим промежуточную модель с целью формирования взаимосвязей между пользователями. Есть две причины использования промежуточной модели:

- используется встроенная в Django модель `User` и нужно избежать ее изменения;
- нужно хранить время создания взаимосвязи.

Отредактируйте файл `models.py` приложения `account`, добавив следующий ниже исходный код:

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

```
def __str__(self):
    return f'{self.user_from} follows {self.user_to}'
```

В приведенном выше исходном коде показана модель `Contact`, которая будет использоваться для взаимосвязей пользователей. Она содержит следующие поля:

- `user_from`: внешний ключ (`ForeignKey`) для пользователя, который создает взаимосвязь;
- `user_to`: внешний ключ (`ForeignKey`) для пользователя, на которого есть подписка;
- `created`: поле `DateTimeField` с параметром `auto_now_add=True` для хранения времени создания взаимосвязи.

Для полей `ForeignKey` индекс базы данных создается автоматически. В `Meta`-классе модели такой индекс определен в убывающем порядке по полю `created`. Также был добавлен атрибут `ordering`, чтобы сообщать Django, что по умолчанию он должен сортировать результаты по полю `created`. Используя дефис перед именем поля, указывается убывающий порядок. Например, `-created`.

Используя ORM, взаимосвязь между пользователем, `user1`, подписанным на другого пользователя, `user2`, можно создать, например, так:

```
user1 = User.objects.get(id=1)
user2 = User.objects.get(id=2)
Contact.objects.create(user_from=user1, user_to=user2)
```

Родственные менеджеры, `rel_from_set` и `rel_to_set`, будут возвращать набор запросов для модели `Contact`. Для того чтобы обратиться к конечной стороне взаимосвязи из модели `User`, желательно, чтобы `User` содержал поле `ManyToManyField`, как показано ниже:

```
following = models.ManyToManyField('self',
                                   through=Contact,
                                   related_name='followers',
                                   symmetrical=False)
```

Исходный код в приведенном выше примере сообщает Django, что для взаимосвязи нужно использовать конкретно-прикладную промежуточную модель, что достигается путем добавления параметра `through=Contact` в объект `ManyToManyField`. Это взаимосвязь многие-ко-многим из модели `User` в саму себя; в поле `ManyToManyField` делается ссылка '`self`', чтобы создать взаимосвязь с той же моделью.



Если во взаимосвязи многие-ко-многим требуются дополнительные поля, то следует создать конкретно-прикладную модель с внешним ключом (`ForeignKey`) для каждой стороны взаимосвязи. В одну из связанных моделей следует добавить поле `ManyToManyField` и сообщить Django, что нужно использовать вашу промежуточную модель, включив ее в параметр `through`.

Если бы модель User была частью вашего приложения, то вы могли бы добавить упомянутое выше поле в модель. Однако изменить класс User напрямую невозможно, потому что он принадлежит приложению `django.contrib.auth`. Давайте воспользуемся немного другим подходом и будем добавлять это поле в модель User динамически.

Отредактируйте файл `models.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
from django.contrib.auth import get_user_model

# ...

# Добавить следующее поле в User динамически
user_model = get_user_model()
user_model.add_to_class('following',
    models.ManyToManyField('self',
        through=Contact,
        related_name='followers',
        symmetrical=False))
```

Здесь модель User извлекается встроенной в Django типовой функцией `get_user_model()`. Метод `add_to_class()` моделей Django применяется для того, чтобы динамически подправлять модель User.

Имейте в виду, что использование метода `add_to_class()` не является рекомендуемым способом добавления полей в модели. Тем не менее в данном случае его можно использовать, чтобы избежать создания конкретно-прикладной модели User, сохраняя все преимущества встроенной в Django модели User.

Также упрощается способ извлечения связанных объектов с использованием ORM-преобразователя посредством методов `user.followers.all()` и `user.following.all()`. Применяется промежуточная модель `Contact` и избегаются сложные запросы, которые потребовали бы дополнительных соединений в базе данных, как это было бы в случае, если бы вы определили взаимосвязи в своей конкретно-прикладной модели `Profile`. Таблица для этой взаимосвязи многие-ко-многим будет создана с использованием модели `Contact`. Таким образом, из динамически добавляемого поля `ManyToManyField` не будет следовать, что модель User будет вызывать какие-либо изменения в базе данных.

Имейте в виду, что в большинстве случаев предпочтительнее добавлять поля в созданную ранее модель `Profile`, а не вносить динамические правки в модель User. В идеале существующую модель User изменять не следует. Django позволяет применять конкретно-прикладные модели пользователя. Если вы хотите использовать такую модель пользователя, то ознакомьтесь с документацией по адресу <https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#specifying-a-custom-user-model>.

Обратите внимание, что взаимосвязь содержит параметр `symmetrical=False`. При определении поля `ManyToManyField` в модели, создавая взаимосвязь с самой моделью, Django навязывает взаимосвязи симметричность. В данном же случае устанавливается параметр `symmetrical=False`, чтобы определить не-

симметричную взаимосвязь (если я на вас подписываюсь, то это не означает, что вы автоматически подписываетесь на меня).



При использовании промежуточной модели для взаимосвязей многие-ко-многим некоторые методы родственного менеджера отключаются, такие как `add()`, `create()` или `remove()`. Вместо этого нужно создавать или удалять экземпляры промежуточной модели.

Выполните следующую ниже команду, чтобы создать первоначальные миграции для приложения `account`:

```
python manage.py makemigrations account
```

Вы получите результат, подобный следующему ниже:

```
Migrations for 'account':
  account/migrations/0002_auto_20220124_1106.py
    - Create model Contact
    - Create index account_con_created_8bdae6_idx on field(s) -created of model
      contact
```

Теперь выполните следующую ниже команду, чтобы синхронизировать приложение с базой данных:

```
python manage.py migrate account
```

Вы должны увидеть результат, который содержит такую строку:

```
Applying account.0002_auto_20220124_1106... OK
```

Теперь модель `Contact` синхронизирована с базой данных, и вы можете создавать взаимосвязи между пользователями. Однако ваш сайт пока не предлагает возможности просмотра пользователей или просмотра профиля того либо иного пользователя. Давайте создадим представления списка и детальной информации для модели `User`.

Создание представлений списка и детальной информации для профилей пользователей

Откройте файл `views.py` приложения `account` и добавьте в него следующий ниже исходный код, выделенный жирным шрифтом:

```
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 представления списка и детальной информации для объектов `User`. Представление `user_list` получает всех активных пользователей. Модель `User` содержит флаг `is_active`, который маркирует, считается учетная запись пользователя активной или нет. Запрос фильтруется по параметру `is_active=True`, чтобы возвращать только активных пользователей. Это представление возвращает все результаты, но его можно улучшить, добавив постраничную разбивку так же, как это делалось для представления `image_list`.

В представлении `user_detail` используется функция сокращенного доступа `get_object_or_404()`, чтобы извлекать активного пользователя с переданным пользовательским именем (`username`). Данное представление возвращает HTTP-ответ 404, если активный пользователь с переданным пользовательским именем не найден.

Отредактируйте файл `urls.py` приложения `account`, добавив шаблон URL-адреса для каждого представления, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
urlpatterns = [
    # ...
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    path('edit/', views.edit, name='edit'),
    path('users/', views.user_list, name='user_list'),
    path('users/<username>', views.user_detail, name='user_detail'),
]
```

Шаблон URL-адреса `user_detail` будет использоваться для того, чтобы генерировать канонический URL-адрес для пользователей. Метод `get_absolute_`

`url()` уже в модели определен и будет возвращать канонический URL-адрес для каждого объекта. Еще одним способом указания URL-адреса для модели является добавление в проект настроичного параметра `ABSOLUTE_URL_OVERRIDES`.

Отредактируйте файл `settings.py` проекта, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.urls import reverse_lazy

# ...

ABSOLUTE_URL_OVERRIDES = {
    'auth.user': lambda u: reverse_lazy('user_detail',
                                        args=[u.username])
}
```

Django добавляет метод `get_absolute_url()` динамически в любые модели, которые появляются в настроичном параметре `ABSOLUTE_URL_OVERRIDES`. Этот метод возвращает соответствующий URL-адрес для заданной модели, которая указана в настроичном параметре. URL-адрес `user_detail` возвращается для заданного пользователя. Теперь метод `get_absolute_url()` можно использовать для экземпляра модели `User`, чтобы получать соответствующий URL-адрес.

Следующей ниже командой откройте оболочку Python:

```
python manage.py shell
```

Затем выполните приведенный далее исходный код, чтобы его протестировать:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.latest('id')
>>> str(user.get_absolute_url())
'/account/users/ellington/'
```

Возвращаемый URL-адрес соответствует ожидаемому формату `/account/users/<username>/`.

Теперь необходимо создать шаблоны только что разработанных представлений. Добавьте следующий ниже каталог и файлы в каталог `templates/account/` приложения `account`:

```
/user/
    detail.html
    list.html
```

Отредактируйте шаблон `account/user/list.html`, добавив такой исходный код:

```
{% 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 }}">  
                      
                </a>  
                <div class="info">  
                    <a href="{{ user.get_absolute_url }}" class="title">  
                        {{ user.get_full_name }}  
                    </a>  
                </div>  
            </div>  
        {% endfor %}  
    </div>  
{% endblock %}
```

Приведенный выше шаблон позволяет перечислять всех активных пользователей на сайте. Заданные пользователи прокручиваются в цикле, и шаблонный тег `{% thumbnail %}` из `easy-thumbnails` используется для генерирования миниатюр изображений, относящихся к данному профилю.

Обратите внимание, что у пользователей должно быть изображение профиля. Для того чтобы использовать типовое изображение для тех пользователей, у которых нет изображения профиля, можно добавить инструкцию `if/else`, чтобы проверять наличие у пользователя фотографии профиля, например `{% if user.profile.photo %} {# миниатюра фотографии #} {% else %} {# типовое изображение #} {% endif %}`.

Откройте шаблон `base.html` проекта и вставьте URL-адрес `user_list` в атрибут `href` следующего ниже пункта меню. Новый исходный код выделен жирным шрифтом:

```
<ul class="menu">  
    ...  
    <li {% if section == "people" %}class="selected"{% endif %}>  
        <a href="{% url "user_list" %}">People</a>  
    </li>  
</ul>
```

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/account/users/` в своем браузере. Вы должны увидеть список пользователей, подобный следующему ниже:

The screenshot shows a web interface with a green header bar. On the left of the header are links for 'Bookmarks', 'My dashboard', 'Images', and 'People'. On the right, it says 'Hello Tesla, Logout'. Below the header, the word 'People' is centered above a horizontal line. Underneath this line are three circular profile pictures arranged horizontally. From left to right, they are labeled 'Tesla', 'Einstein', and 'Turing' below their respective images.

Рис. 7.1. Страница списка пользователей с миниатюрами изображений профилей

Напомним, что если у вас возникли трудности с генерированием миниатюр, то в свой файл `settings.py` можно добавить настроечный параметр `THUMBNAIL_DEBUG = True`, чтобы получать отладочную информацию в оболочке.

Отредактируйте шаблон `account/user/detail.html` приложения `account`, добавив следующий ниже исходный 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">  
          
    </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 %}unfollow{% endif %}" 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_images.html" with images=user.images_created.all %}  
</div>  
{% endwith %}  
{% endblock %}
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк; Django не поддерживает многострочные теги.

В шаблоне детальной информации отображается профиль пользователя, а шаблонный тег `{% thumbnail %}` используется для вывода изображения профиля на странице. При этом отображается общее число подписчиков и ссылка, чтобы подписаться либо отписаться от пользователя. Эта ссылка будет использоваться для подписки/отписки от конкретного пользователя. Атрибуты `data-id` и `data-action` HTML-элемента `<a>` содержат ИД пользователя и первоначальное действие, которое необходимо выполнять при нажатии на ссылочный элемент, – `follow` (подписаться) либо `unfollow` (отписаться). Первоначальное действие (`follow` либо `unfollow`) зависит от того, является запрашивающей страницу пользователь уже подписчиком пользователя или нет. Изображения, отмеченные пользователем закладкой, отображаются путем вставки шаблона `images/image/list_images.html`.

Снова откройте браузер и кликните на пользователе, который отметил несколько изображений закладкой. Страница пользователя будет выглядеть следующим образом:



Рис. 7.2. Страница детальной информации о пользователе



Изображение Чика Кориа, автор: ataelw (лицензия: Creative Commons Attribution 2.0 Типовая форма: <https://creativecommons.org/licenses/by/2.0/>).

Добавление действий пользователя по подписке/отписке с помощью JavaScript

Давайте добавим функциональность подписки на пользователей и отписки от пользователей. Мы создадим новое представление подписки/отписки и реализуем асинхронный HTTP-запрос с помощью JavaScript для действия по подписке/отписке.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST
from .models import Contact

# ...

@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': 'error'})
    return JsonResponse({'status': 'error'})
```

Представление `user_follow` очень похоже на представление `image_like`, которое вы создали в главе 6 «Распространение контента на веб-сайте». Поскольку вы применяете конкретно-прикладную промежуточную модель для пользовательской взаимосвязи многие-ко-многим, стандартные методы

add() и remove() автоматического менеджера полей ManyToManyField недоступны. Вместо этого для создания или удаления пользовательских взаимосвязей применяется промежуточная модель Contact.

Отредактируйте файл urls.py приложения account, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    path('edit/', views.edit, name='edit'),
    path('users/', views.user_list, name='user_list'),
    path('users/follow/', views.user_follow, name='user_follow'),
    path('users/<username>/', views.user_detail, name='user_detail'),
]
```

Проверьте, чтобы приведенный выше шаблон был помещен перед шаблоном URL-адреса user_detail. В противном случае любые запросы к /users/follow/ будут совпадать с регулярным выражением шаблона user_detail, и вместо этого будет выполняться указанное представление. Напомним, что в каждом HTTP-запросе веб-фреймворк Django проверяет запрошенный URL-адрес на соответствие каждому шаблону в порядке появления и останавливается при первом совпадении.

Отредактируйте шаблон user/detail.html приложения account, добавив следующий ниже исходный код:

```
{% block domready %}
var const = '{% url "user_follow" %}';
var options = {
  method: 'POST',
  headers: {'X-CSRFToken': csrfToken},
  mode: 'same-origin'
}

document.querySelector('a.follow')
  .addEventListener('click', function(e){
  e.preventDefault();
  var followButton = this;

  // добавить тело запроса
  var formData = new FormData();
  formData.append('id', followButton.dataset.id);
  formData.append('action', followButton.dataset.action);
  options['body'] = formData;

  // отправить HTTP-запрос
  fetch(url, options)
})
```

```
.then(response => response.json())
.then(data => {
  if (data['status'] === 'ok') {
    var previousAction = followButton.dataset.action;

    // переключить текст кнопки и data-action
    var action = previousAction === 'follow' ? 'unfollow' : 'follow';
    followButton.dataset.action = action;
    followButton.innerHTML = action;

    // обновить количество подписчиков
    var followerCount = document.querySelector('span.count .total');
    var totalFollowers = parseInt(followerCount.innerHTML);
    followerCount.innerHTML = previousAction === 'follow' ? totalFollowers + 1 :
    totalFollowers - 1;
  }
})
});
{%
  endblock %}
```

Приведенный выше шаблонный блок содержит исходный код JavaScript, который выполняет асинхронный HTTP-запрос на подписку или отписку от того или иного пользователя, а также переключает ссылку подписки/отписки. Интерфейс Fetch API используется для выполнения запроса AJAX и установки как атрибута `data-action`, так и текста HTML-элемента `<a>` на основе его предыдущего значения. После завершения действия также обновляется отображаемое на странице общее число подписчиков.

Откройте страницу детальной информации о существующем пользователе и кликните по ссылке **FOLLOW** (Подписаться), чтобы протестировать только что созданную функциональность. Вы увидите, что количество подписчиков увеличилось:



Рис. 7.3. Количество подписчиков и кнопка подписаться/отписаться

Теперь система подписки завершена, и пользователи могут подписываться друг на друга. Далее мы создадим поток активности, создав соответствующий контент для каждого пользователя, основанный на людях, на которых они подписаны.

Разработка типового приложения для потока активности

Многие социальные веб-сайты показывают своим пользователям поток активности, предоставляя им возможность отслеживать то, что другие пользователи делают на платформе. Поток активности – это список последних действий, выполненных пользователем или группой пользователей. Например, новостная лента Facebook является потоком активности. Примеры действий могут быть такими: *пользователь X пометил изображение Y закладкой* или *теперь пользователь X подписан на пользователя Y*.

Сейчас вы разработаете приложение для потока активности, которое будет давать пользователю возможность видеть недавние взаимодействия пользователей, на которых он подписан. Для этого понадобится модель, которая будет хранить действия пользователей на сайте, и простой способ добавления действий в новостную ленту.

Следующей ниже командой создайте новое приложение с именем `actions` внутри проекта:

```
python manage.py startapp actions
```

Добавьте новое приложение в настроечный параметр `INSTALLED_APPS` в файле `settings.py` проекта, чтобы активировать приложение в проекте. Новая строка выделена жирным шрифтом:

```
INSTALLED_APPS = [  
    # ...  
    'actions.apps.ActionsConfig',  
]
```

Отредактируйте файл `models.py` приложения `actions`, добавив следующий ниже исходный код:

```
from django.db import models  
  
class Action(models.Model):  
    user = models.ForeignKey('auth.User',  
                           related_name='actions',  
                           on_delete=models.CASCADE)  
    verb = models.CharField(max_length=255)  
    created = models.DateTimeField(auto_now_add=True)
```

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

В приведенном выше исходном коде показана модель `Action`, которая будет использоваться для хранения действий пользователя. Поля этой модели таковы:

- `user`: пользователь, выполнивший действие; это внешний ключ (`ForeignKey`) для встроенной в Django модели `User`;
- `verb`: глагол, описывающий действие, которое выполнил пользователь;
- `created`: дата и время создания этого действия. Параметр `auto_now_add=True` используется для того, чтобы автоматически устанавливать текущую дату и время при первом сохранении объекта в базе данных.

В `Meta`-классе модели был определен индекс базы данных в убывающем порядке по полю `created`. Кроме того, был добавлен атрибут `ordering`, чтобы сообщать Django, что по умолчанию результаты следует сортировать по полю `created` в убывающем порядке.

В этой базовой модели можно хранить только такие действия, как: *пользователь X что-то сделал*. Необходимо иметь дополнительное поле `ForeignKey`, чтобы хранить действия, связанные с целевым объектом `target`, например *пользователь X пометил изображение Y закладкой* или *теперь пользователь X подписан на пользователя Y*. Как вы уже знаете, обычный внешний ключ (`ForeignKey`) может указывать только на одну модель. Вместо этого понадобится способ, которым целевой объект действия будет экземпляром существующей модели. В этом поможет встроенный в Django фреймворк типов контента `contenttypes`.

Применение фреймворка `contenttypes`

Django содержит фреймворк `contenttypes`, расположенный в приложении `django.contrib.contenttypes`. Указанное приложение может отслеживать все установленные в проекте модели и предоставляет типовой интерфейс взаимодействия с этими моделями.

Приложение `django.contrib.contenttypes` включается в настроечный параметр `INSTALLED_APPS` по умолчанию при создании нового проекта с помощью команды `startproject`. Он используется другими пакетами `contrib`, такими как фреймворк аутентификации и приложение администрирования.

Приложение `contenttypes` содержит модель `ContentType`. Экземпляры этой модели представляют фактические модели вашего приложения, а новые экземпляры `ContentType` создаются автоматически при установке в проект новых моделей. Модель `ContentType` имеет следующие поля:

- `app_label`: указывает имя приложения, которому модель принадлежит. Оно берется автоматически из атрибута `app_label` модельных Meta-опций. Например, ваша модель `Image` принадлежит приложению `images`;
- `model`: имя модельного класса;
- `name`: указывает удобочитаемое имя модели. Оно берется автоматически из атрибута `verbose_name` модельных Meta-опций.

Давайте посмотрим, как взаимодействовать с объектами `ContentType`. Следующей ниже командой откройте оболочку:

```
python manage.py shell
```

Объект `ContentType`, соответствующий конкретной модели, можно получать путем выполнения запроса с атрибутами `app_label` и `model`, как показано ниже:

```
>>> from django.contrib.contenttypes.models import ContentType  
>>> image_type = ContentType.objects.get(app_label='images', model='image')  
>>> image_type  
<ContentType: images | image>
```

Из объекта `ContentType` можно получать модельный класс, вызвав его метод `model_class()`:

```
>>> image_type.model_class()  
<class 'images.models.Image'>
```

Объект `ContentType` также принято получать для определенного модельного класса, как показано ниже:

```
>>> from images.models import Image  
>>> ContentType.objects.get_for_model(Image)  
<ContentType: images | image>
```

Это всего лишь несколько примеров использования объектов приложения `contenttypes`, и Django предлагает много других способов работы с ними. Официальная документация по фреймворку `contenttypes` находится на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/contenttypes/>.

Добавление обобщенных отношений в модели

В обобщенных отношениях объекты `ContentType` играют роль указателей на модель, используемую для взаимосвязи. При этом для того чтобы установить обобщенное отношение, в модели понадобятся три поля:

- поле `ForeignKey` для `ContentType`: оно будет сообщать о модели, используемой для взаимосвязи;

- поле для хранения первичного ключа связанного объекта: обычно это PositiveIntegerField, чтобы сочетаться со встроенными в Django автоматическими полями первичных ключей;
- поле для определения обобщенного отношения и управления им с использованием двух предыдущих полей: для этой цели фреймворк contenttypes предлагает поле GenericForeignKey.

Отредактируйте файл models.py приложения actions, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
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',
                            on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    created = models.DateTimeField(auto_now_add=True)
    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)
    target = GenericForeignKey('target_ct', 'target_id')

    class Meta:
        indexes = [
            models.Index(fields=['-created']),
            models.Index(fields=['target_ct', 'target_id']),
        ]
        ordering = ['-created']
```

В модель Action были добавлены следующие поля:

- target_ct: поле ForeignKey, указывающее на модель ContentType;
- target_id: PositiveIntegerField для хранения первичного ключа связанного объекта;
- target: поле GenericForeignKey для связанного объекта на основе комбинации двух предыдущих полей.

Кроме того, был добавлен многопольный индекс, включающий поля target_ct и target_id.

Django не создает поля GenericForeignKey в базе данных. Единственными полями, которые соотносятся с полями базы данных, являются target_ct

и `target_id`. Оба поля имеют атрибуты `empty=True` и `null=True`, поэтому при сохранении объектов `Action` целевой объект `target` не требуется.



Применение обобщенных отношений вместо внешних ключей будет придавать приложениям большую гибкость.

Выполните следующую ниже команду, чтобы создать первоначальные миграции приложения:

```
python manage.py makemigrations actions
```

Вы должны увидеть следующий ниже результат:

```
Migrations for 'actions':
  actions/migrations/0001_initial.py
    - Create model Action
    - Create index actions_act_created_64f10d_idx on field(s) -created of model
      action
      - Create index actions_act_target_f20513_idx on field(s) target_ct,
        target_id of model action
```

Затем выполните приведенную ниже команду, чтобы синхронизировать приложение с базой данных:

```
python manage.py migrate
```

Результат команды должен указывать на то, что новые миграции были применены, как показано далее:

```
Applying actions.0001_initial... OK
```

Давайте добавим модель `Action` на сайт администрирования. Отредактируйте файл `admin.py` приложения `actions`, добавив следующий ниже исходный код:

```
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']
```

Вы только что зарегистрировали модель Action на сайте администрирования.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/actions/action/add/` в своем браузере. Вы должны увидеть страницу создания нового объекта Action, как показано далее:

Рис. 7.4. Страница добавления действия на сайте администрирования

Как вы заметили, на приведенном выше снимке экрана показаны только поля `target_ct` и `target_id`, которые соотносятся с фактическими полями базы данных. Поле `GenericForeignKey` в форме не отображается. Поле `target_ct` позволяет выбирать любую зарегистрированную модель своего проекта Django. Выбор типов контента можно ужимать путем выбора из ограниченного набора моделей, используя атрибут `limit_choices_to` в поле `target_ct`; атрибут `limit_choices_to` позволяет ограничивать содержимое полей `ForeignKey` определенным набором значений.

Внутри каталога приложения `action` создайте новый файл и назовите его `utils.py`. Сейчас необходимо определить функцию быстрого доступа, которая позволит создавать новые объекты `Action` простым способом. Отредактируйте новый файл `utils.py`, добавив следующий ниже исходный код:

```
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 создавать действия, которые опционально включают целевой объект `target`. Эту функцию можно применять в любом месте своего исходного кода в качестве функции сокращенного доступа, чтобы добавлять новые действия в поток активности.

Игнорирование повторных действий в потоке активности

Иногда могут возникать ситуации, когда пользователи будут кликать по кнопке **Like** или **Unlike** несколько раз либо неоднократно выполнять одно и то же действие за короткий промежуток времени. Они легко будут приводить к сохранению и отображению повторяющихся действий. Во избежание этого давайте усовершенствуем функцию `create_action()`, чтобы игнорировать очевидные повторяющиеся действия.

Отредактируйте файл `utils.py` приложения `actions`, как показано ниже:

```
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):
    # проверить, не было ли каких-либо аналогичных
    # действий, совершенных 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:
        # никаких существующих действий не найдено
        action = Action(user=user, verb=verb, target=target)
        action.save()
        return True
    return False
```

Функция `create_action()` была изменена, чтобы избегать сохранения повторяющихся действий и возвращать булево значение, сообщающее о том, не было ли действие сохранено. Вот как игнорируются повторы:

- 1) сначала берется текущее время. Это делается с помощью встроенного в Django метода `timezone.now()`. Указанный метод делает то же самое,

что и `datetime.datetime.now()`, но возвращает объект с учетом часового пояса. Django предоставляет настроочный параметр `USE_TZ`, которым активируется либо деактивируется поддержка часового пояса. Стандартный файл `settings.py`, созданный с помощью команды `startproject`, содержит `USE_TZ=True`;

- 2) переменная `last_minute` используется для хранения даты/времени давностью одна минута назад и для получения любых идентичных действий, выполненных пользователем с тех пор;
- 3) если за последнюю минуту не было идентичного действия, то создается объект `Action`. При этом возвращается `True`, если объект `Action` был создан, либо `False` в противном случае.

Добавление действий пользователя в поток активности

Теперь самое время добавить несколько действий в представления, чтобы сформировать поток активности для ваших пользователей. Действие будет сохраняться для каждого следующего ниже взаимодействия:

- пользователь отмечает изображение закладкой;
- пользователю нравится изображение;
- пользователь создает учетную запись;
- пользователь только что подписался на другого пользователя.

Отредактируйте файл `views.py` приложения `images`, добавив следующую ниже инструкцию импорта:

```
from actions.utils import create_action
```

В представлении `image_create` сразу после сохранения изображения добавьте `create_action()`, как показано ниже. Новая строка выделена жирным шрифтом:

```
@login_required
def image_create(request):
    if request.method == 'POST':
        # форма отправлена
        form = ImageCreateForm(data=request.POST)
        if form.is_valid():
            # данные в форме валидны
            cd = form.cleaned_data
            new_image = form.save(commit=False)
            # назначить текущего пользователя элементу
            new_image.user = request.user
            new_image.save()
            create_action(request.user, 'bookmarked image', new_image)
            messages.success(request, 'Image added successfully')
```

```
# перенаправить к представлению детальной
# информации о только что созданном элементе
return redirect(new_image.get_absolute_url())

else:
    # скомпоновать форму с данными,
    # предоставленными бокмакрелтом методом GET
    form = ImageCreateForm(data=request.GET)
return render(request,
              'images/image/create.html',
              {'section': 'images',
               'form': form})
```

В представлении `image_like` сразу после добавления пользователя во взаимосвязь `users_like` добавьте `create_action()`, как показано ниже. Новая строка выделена жирным шрифтом:

```
@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)
                create_action(request.user, 'likes', image)
            else:
                image.users_like.remove(request.user)
            return JsonResponse({'status':'ok'})
        except Image.DoesNotExist:
            pass
    return JsonResponse({'status':'error'})
```

Теперь отредактируйте файл `views.py` приложения `account`, добавив следующую ниже инструкцию импорта:

```
from actions.utils import create_action
```

В представлении `register` сразу после создания объекта `Profile` добавьте `create_action()`, как показано ниже. Новая строка выделена жирным шрифтом:

```
def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
```

```

# Создать новый объект пользователя,
# но пока не сохранять его
new_user = user_form.save(commit=False)
# Установить выбранный пароль
new_user.set_password(
    user_form.cleaned_data['password'])
# Сохранить объект User
new_user.save()
# Создать профиль пользователя
Profile.objects.create(user=new_user)
create_action(new_user, 'has created an account')
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})

```

В представление `user_follow` добавьте `create_action()`, как показано ниже. Новая строка выделена жирным шрифтом:

```

@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)
                create_action(request.user, 'is following', user)
            else:
                Contact.objects.filter(user_from=request.user,
                                      user_to=user).delete()
        return JsonResponse({'status':'ok'})
    except User.DoesNotExist:
        return JsonResponse({'status':'error'})
    return JsonResponse({'status':'error'})

```

Как видно из приведенного выше исходного кода, благодаря модели `Action` и вспомогательной функции сохранять новые действия в потоке активности довольно легко.

Отображение потока активности

Наконец, требуется способ отображения потока активности по каждому пользователю. В связи с этим необходимо добавить поток активности на информационную панель пользователя. Отредактируйте файл `views.py` приложения `account`, импортировав модель `Action` и видоизменив представление информационной панели, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
from actions.models import Action

# ...

@login_required
def dashboard(request):
    # По умолчанию показать все действия
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
    if following_ids:
        # Если пользователь подписан на других,
        # то извлечь только их действия
        actions = actions.filter(user_id__in=following_ids)
    actions = actions[:10]
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard',
                   'actions': actions})
```

В приведенном выше представлении из базы данных извлекаются все действия, за исключением тех, которые выполняются текущим пользователем. По умолчанию извлекаются последние действия, выполненные всеми пользователями на платформе. Если пользователь подписан на других пользователей, то запрос ограничивается, чтобы получать только те действия, которые выполняются пользователями, на которых он подписан. Наконец, результат ограничивается первыми 10 возвращаемыми действиями. Метод `order_by()` в наборе запросов `QuerySet` не используется, потому что вы опираетесь на заранее заданный порядок сортировки, указанный в `Meta-опциях` модели `Action`. Недавние действия будут первыми, поскольку в модели `Action` было задано `ordering = ['-created']`.

Оптимизация наборов запросов, предусматривающих связанные объекты

Всякий раз, когда извлекается объект `Action`, обычно обращаются к связанному с ним объекту `User` и к связанному с пользователем объекту `Profile`.

Встроенный в Django ORM-преобразователь предлагает простой способ одновременного извлечения связанных объектов, что позволяет избегать дополнительных запросов к базе данных.

Применение метода `select_related()`

Django предлагает QuerySet-метод под названием `select_related()`, который позволяет извлекать связанные объекты для взаимосвязей один-ко-многим. Это транслируется в один более сложный набор запросов, но зато позволяет избегать дополнительных запросов при доступе к связанным объектам. Метод `select_related` предназначен для полей `ForeignKey` и `OneToOne`. Он работает, выполняя SQL-инструкцию `JOIN` и включая поля связанного объекта в инструкцию `SELECT`.

Для того чтобы воспользоваться преимуществами метода `select_related()`, отредактируйте следующую ниже строку приведенного выше исходного кода в файле `views.py` приложения `account`, добавив `select_related`, включая поля, которые будут использоваться, например, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
@login_required
def dashboard(request):
    # По умолчанию показать все действия
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)

    if following_ids:
        # Если пользователь подписан на других,
        # то извлечь только их действия
        actions = actions.filter(user_id__in=following_ids)
    actions = actions.select_related('user', 'user_profile')[:10]
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard',
                   'actions': actions})
```

Аргумент `user_profile` используется для того, чтобы выполнять операцию соединения на таблице `Profile` в одном SQL-запросе. Если вызвать `select_related()` без передачи каких-либо аргументов, то он будет извлекать объекты из всех взаимосвязей с внешними ключами `ForeignKey`. Следует всегда ограничивать метод `select_related()` взаимосвязями, которые будут доступны позже.



Осторожное применение метода `select_related()` может значительно сократить время исполнения запросов.

Применение метода `prefetch_related()`

Метод `select_related()` поможет повышать производительность при извлечении связанных объектов во взаимосвязях один-ко-многим. Однако он не работает для взаимосвязей многие-ко-многим или многие-к-одному (поля `ManyToMany` или обратного внешнего ключа `ForeignKey`). Django предлагает другой `QuerySet`-метод под названием `prefetch_related()`, который в дополнение к взаимосвязям, поддерживаемым методом `select_related()`, успешно работает для взаимосвязей многие-ко-многим и многие-к-одному. Метод `prefetch_related()` выполняет отдельный поиск по каждой взаимосвязи и соединяет результаты с помощью Python. Этот метод также поддерживает упреждающую выборку полей `GenericRelation` и `GenericForeignKey`.

Отредактируйте файл `views.py` приложения `account`, чтобы завершить свой запрос, добавив в него функцию `prefetch_related()` для целевого поля `GenericForeignKey`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
@login_required
def dashboard(request):
    # Display all actions by default
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
    if following_ids:
        # Если пользователь подписан на других,
        # то извлечь только их действия
        actions = actions.filter(user_id__in=following_ids)
    actions = actions.select_related('user', 'user_profile')[:10]\
        .prefetch_related('target')[:10]
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard',
                   'actions': actions})
```

Теперь этот запрос оптимизирован под получение действий пользователя, включая связанные объекты.

Создание шаблонов действий

Теперь давайте создадим шаблон для отображения того или иного объекта `Action`. Создайте новый каталог внутри каталога приложения `actions` и назовите его `templates`. Добавьте в него следующую ниже файловую структуру:

```
actions/
  action/
    detail.html
```

Отредактируйте файл шаблона `action/action/detail.html`, добавив следующие ниже строки:

```
{% 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>
        {% 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>
                {% 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 %}
```

Это шаблон, который будет использоваться для отображения объекта `Action`. Вначале используется шаблонный тег `{% with %}`, чтобы извлекать выполняющего действие пользователя и связанный с ним объект `Profile`. Затем, если объект `Action` имеет связанный объект `target`, отображается изображение объекта `target`. Наконец, отображается ссылка на выполняющего действие пользователя, глагол и объект `target`, если он есть.

Отредактируйте шаблон `account/dashboard.html` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом, в нижнюю часть блока `content`:

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

{% block title %}Dashboard{% endblock %}

{% block content %}

...

<h2>What's happening</h2>
<div id="action-list">
    {% for action in actions %}
        {% include "actions/action/detail.html" %}
    {% endfor %}
</div>
{% endblock %}
```

Пройдите по URL-адресу `http://127.0.0.1:8000/account/` в своем браузере. Войдите в систему как существующий пользователь и выполните несколько действий, чтобы они были сохранены в базе данных. Затем войдите в систему, используя другого пользователя, подпишитесь на предыдущего пользователя и посмотрите на генерированный поток действий на странице информационной панели.

Это должно выглядеть следующим образом:

What's happening



3 minutes ago

Einstein likes Alternating electric current generator



5 minutes ago

Einstein bookmarked image Turing Machine



2 days, 2 hours ago

Tesla likes Chick Corea

Рис. 7.5. Поток активности текущего пользователя



Авторство изображений на рис. 7.5:

- Асинхронный электродвигатель Теслы, автор: Стас (лицензия: Creative Commons Attribution Share-Alike 3.0). Непортированная форма: <https://creativecommons.org/licenses/by-sa/3.0/>);
- Модель машины Тьюринга Davey 2012, автор: Рокки Акоста (лицензия: Creative Commons Attribution 3.0 Непортированная форма: <https://creativecommons.org/licenses/by/3.0/>);
- Чик Кориа, автор: ataelw (лицензия: Creative Commons Attribution 2.0 Типовая форма: <https://creativecommons.org/licenses/by/2.0/>).

Вы только что создали полный поток активности своих пользователей и можете легко добавлять в него новые действия пользователей. В поток активности также можно добавлять функциональность бесконечной прокрутки, реализовав того же AJAX-ориентированного постраничного разбивщика, которого вы использовали для представления `image_list`. Далее вы научитесь использовать сигналы Django для денормализации количеств действий.

Использование сигналов для денормализации количественных данных

В некоторых случаях вы, возможно, захотите проводить денормализацию своих данных. Денормализация делает данные избыточными, но при этом она оптимизирует производительность операций чтения. Например, можно скопировать связанные данные в объект, чтобы избежать дорогостоящих запросов к базе данных, предусматривающих операцию чтения при извлечении связанных данных. В работе с денормализацией нужно быть осмотрительным и начинать ее применять только тогда, когда она действительно нужна. Самая большая проблема, с которой вы столкнетесь при денормализации, заключается в том, что денормализованные данные сложно обновлять.

Давайте взглянем на пример того, как улучшать свои запросы путем денормализации количественных данных. Вы будете выполнять денормализацию данных из своей модели `Image` и использовать сигналы Django, чтобы поддерживать данные в обновленном состоянии.

Работа с сигналами

Django идет в комплекте с диспетчером сигналов, который позволяет функциям-получателям получать уведомления о том, когда происходят те или иные действия. Сигналы очень полезны, когда требуется, чтобы исходный код делал что-то всякий раз, когда происходит что-то еще. Сигналы позволяют отцеплять логику: можно улавливать определенное действие, независимо от приложения или исходного кода, вызвавшего это действие, и реализовывать логику, которая выполняется всякий раз, когда это действие происходит. Например, можно разработать функцию-получатель сигналов, которая будет выполняться всякий раз при сохранении объекта `User`. Также можно создавать свои собственные сигналы, чтобы другие могли получать уведомления, когда происходит событие.

Django предоставляет моделям ряд сигналов, расположенных в `django.db.models.signals`. Вот несколько таких сигналов:

- `pre_save` и `post_save` отправляются до или после вызова метода `save()` модели;
- `pre_delete` и `post_delete` отправляются до или после вызова метода `delete()` модели или объекта `QuerySet`;
- `m2m_changed` отправляется при изменении поля `ManyToManyField` в модели.

Это всего лишь часть встроенных в Django сигналов. Список всех встроенных сигналов находится на странице <https://docs.djangoproject.com/en/4.1/ref/signals/>.

Допустим, вы хотите извлекать изображения по популярности. Для того чтобы получать изображения, упорядоченные по числу пользователей, которым они нравятся, можно использовать встроенные в Django функции агрегирования. Напомним, что вы использовали функции агрегирования Django в главе 3 «Расширение приложения для ведения блога». В следующем ниже примере исходного кода изображения будут извлекаться в соответствии с числом их лайков:

```
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')
```

Однако с точки зрения производительности упорядочивание изображений путем подсчета их общих количеств лайков (`likes`) будет затратнее, чем упорядочивание их по полю, в котором эти общие количества хранятся. В модель `Image` можно добавить поле, чтобы денормализовать общее количество лайков с целью значительного повышения производительности запросов, в которых это поле используется. Вопрос в том, как поддерживать это поле в обновленном состоянии.

Отредактируйте файл `models.py` приложения `images`, добавив следующее ниже поле `total_likes` в модель `Image`. Новый исходный код выделен жирным шрифтом:

```
class Image(models.Model):
    # ...
    total_likes = models.PositiveIntegerField(default=0)

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

Поле `total_likes` позволит хранить общее количество пользователей, которым понравилось каждое изображение. Денормализация количеств широка применяется тогда, когда нужно по ним фильтровать или упорядочить набор запросов `QuerySet`. Индекс базы данных по полю `total_likes` был добавлен в убывающем порядке, так как планируется, что изображения будут извлекаться упорядоченными в убывающем порядке по их общему числу лайков.



Перед денормализацией полей следует учитывать, что существует несколько способов улучшения производительности. Прежде чем начинать денормализацию данных, рассмотрите возможность применения индексов базы данных, оптимизации запросов и кеширования.

Выполните следующую ниже команду, чтобы создать миграции для добавления нового поля в таблицу базы данных:

```
python manage.py makemigrations images
```

Вы должны увидеть такой результат:

```
Migrations for 'images':
  images/migrations/0002_auto_20220124_1757.py
    - Add field total_likes to image
    - Create index images_imag_total_l_0bcd7e_idx on field(s) -total_likes of
      model image
```

Затем выполните ниже следующую команду, чтобы применить миграцию:

```
python manage.py migrate images
```

Результат должен содержать такую строку:

```
Applying images.0002_auto_20220124_1757... OK
```

Теперь нужно прикрепить функцию-получатель к сигналу `m2m_changed`.

Внутри каталога приложения `images` создайте новый файл и назовите его `signals.py`. Добавьте в него следующий ниже исходный 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()
```

Сперва, используя декоратор `receiver()`, в качестве функции-получателя регистрируется функция `users_like_changed`. Она привязывается к сигналу `m2m_changed`. Затем эта функция соединяется с `Image.users_like.through`, чтобы функция вызывалась только в том случае, если сигнал `m2m_changed` был запущен этим отправителем. Есть и альтернативный метод регистрации функции-получателя; он состоит в использовании метода `connect()` объекта `Signal`.



Сигналы Django бывают синхронными и блокирующими. Не путайте сигналы с асинхронными заданиями. Однако когда ваш исходный код получает уведомление с помощью сигнала, то можно комбинировать и то, и другое для запуска асинхронных заданий. Вы научитесь создавать асинхронные задания с помощью очереди заданий Celery в главе 8 «Разработка интернет-магазина».

Необходимо соединить функцию-получатель с сигналом, чтобы она вызывалась всякий раз, когда сигнал отправляется. Рекомендуемый метод регистрации своих собственных сигналов состоит в импортировании их в метод `ready()` класса конфигурации вашего приложения. Django предоставляет реестр приложений, который позволяет конфигурировать и контролировать ваши приложения.

Конфигурационные классы приложений

Django позволяет задавать приложениям конфигурационные классы. При создании приложения с помощью команды `startapp` Django добавляет файл `apps.py` в каталог приложения, включая базовую конфигурацию приложения, которая наследует от класса `AppConfig` .

Конфигурационный класс приложения позволяет хранить метаданные и конфигурацию приложения, а также обеспечивает интроспекцию приложения. Дополнительная информация о конфигурациях приложений находится на странице <https://docs.djangoproject.com/en/4.1/ref/applications/>.

Для того чтобы зарегистрировать свои функции-получатели (`receiver`) сигналов при использовании декоратора `receiver()`, нужно просто импортировать модуль `signal` своего приложения в метод `ready()` конфигурационного класса приложения. Этот метод вызывается сразу после того, как реестр приложений будет полностью заполнен. В указанный метод также должны быть включены любые другие инициализации приложения.

Отредактируйте файл `apps.py` приложения `images`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.apps import AppConfig

class ImagesConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'images'

    def ready(self):
        # импортировать обработчики сигналов
        import images.signals
```

Сигналы для этого приложения импортируются в методе `ready()`, для того чтобы они импортировались при загрузке приложения `images`.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Откройте браузер, чтобы просмотреть страницу детальной информации об изображении, и кликните по кнопке **Like** (Нравится).

Перейдите на сайт администрирования, откройте URL-адрес редактирования изображения, например <http://127.0.0.1:8000/admin/images/image/1/change/>, и посмотрите на атрибут `total_likes`. Вы должны увидеть, что атрибут `total_likes` обновляется общим числом пользователей, которым нравится изображение, как показано ниже:

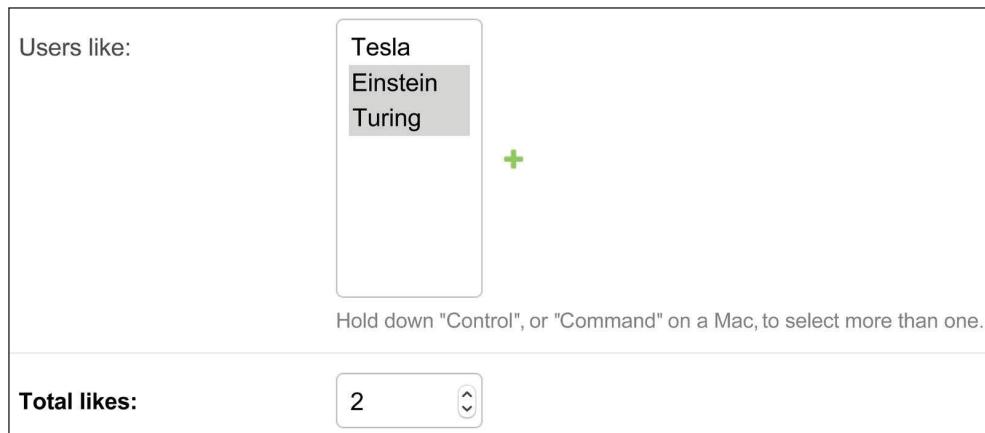


Рис. 7.6. Страница редактирования изображения на сайте администрации, включая денормализацию общего числа лайков

Теперь можно использовать атрибут `total_likes`, чтобы упорядочивать изображения по популярности или отображать значение в любом месте, избегая сложных запросов для его вычисления.

Рассмотрим следующий ниже запрос, чтобы получать изображения, упорядоченные в убывающем порядке по количеству лайков:

```
from django.db.models import Count

images_by_popularity = Image.objects.annotate(
    likes=Count('users_like')).order_by('-likes')
```

Теперь приведенный выше запрос можно записать следующим образом:

```
images_by_popularity = Image.objects.order_by('-total_likes')
```

Такой подход приводит к менее дорогостоящему SQL-запросу благодаря денормализации общего числа лайков к изображениям. Кроме того, вы научились использовать сигналы Django.



Сигналы следует использовать с осторожностью, поскольку они затрудняют понимание процедуры управления. Во многих случаях можно обойтись без использования сигналов, если известны получатели, которые должны уведомляться.

Теперь необходимо установить начальные значения количеств для остальных объектов `Image`, чтобы они соответствовали текущему состоянию базы данных.

Следующей ниже командой откройте оболочку:

```
python manage.py shell
```

Выполните такой исходный код в оболочке:

```
>>> from images.models import Image
>>> for image in Image.objects.all():
...     image.total_likes = image.users_like.count()
...     image.save()
```

Вы вручную обновили количество лайков для существующих изображений в базе данных. С этого момента функция-получатель сигналов `users_like_changed` будет обрабатывать обновление поля `total_likes` всякий раз, когда изменяются объекты, соединенные взаимосвязью многие-ко-многим.

Далее вы научитесь использовать меню отладочных инструментов Django Debug Toolbar, чтобы получать соответствующую отладочную информацию о запросах, включая время исполнения, выполненные запросы SQL, прорисованные шаблоны, зарегистрированные сигналы и многое другое.

Использование меню отладочных инструментов Django

К этому моменту вы уже знакомы с отладочной страницей Django. В предыдущих главах вы несколько раз видели отличительную желто-серую отладочную страницу. Например, в главе 2 «Совершенствование блога за счет продвинутых функциональностей» в разделе «Обработка ошибок постраничной разбивки» на отладочной странице показывалась информация, связанная с необработанными исключениями при реализации постраничной разбивки объектов.

Отладочная страница Django предоставляет полезную информацию об отладке. Вместе с тем есть приложение Django, которое содержит более детальную отладочную информацию и бывает очень удобным при разработке.

Меню отладочных инструментов Django Debug Toolbar – это внешнее приложение Django, которое позволяет просматривать соответствующую отладочную информацию о текущем цикле запроса/ответа. Информация разделена на несколько панелей, которые отображают различную информацию, включая данные запроса/ответа, используемые версии пакетов Python, время исполнения, настроочные параметры, заголовки, SQL-запросы, применяемые шаблоны, кеш, сигналы и журнальные данные.

Документация по меню отладочных инструментов Django Debug Toolbar находится на странице <https://django-debug-toolbar.readthedocs.io/>.

Установка меню отладочных инструментов Django

Установите django-debug-toolbar через pip, используя следующую ниже команду:

```
pip install django-debug-toolbar==3.6.0
```

Отредактируйте файл settings.py проекта, добавив debug_toolbar в настроечный параметр INSTALLED_APPS, как показано ниже. Новая строка выделена жирным шрифтом:

```
INSTALLED_APPS = [  
    # ...  
    'debug_toolbar',  
]
```

В том же файле добавьте следующую ниже строку, выделенную жирным шрифтом, в настроечный параметр MIDDLEWARE:

```
MIDDLEWARE = [  
    'debug_toolbar.middleware.DebugToolbarMiddleware',  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

Меню отладочных инструментов Django Debug Toolbar в основном реализовано как промежуточный программный компонент. Порядок следования

таких компонентов в настроечном параметре MIDDLEWARE имеет значение. Компонент DebugToolbarMiddleware должен располагаться перед любым другим промежуточным компонентом, за исключением промежуточного компонента, кодирующего содержимое ответа, например GZipMiddleware, который, если он присутствует, должен стоять первым.

Добавьте следующие ниже строки в конец файла settings.py:

```
INTERNAL_IPS = [  
    '127.0.0.1',  
]
```

Меню отладочных инструментов Django будет отображаться только в том случае, если ваш IP-адрес соответствует записи в настроечном параметре INTERNAL_IPS. Для того чтобы предотвратить отображение отладочной информации в производственной среде, ПО меню отладочных инструментов Django Debug Toolbar проверяет, что в настроечном параметре DEBUG установлено значение True.

Отредактируйте главный файл urls.py проекта, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом, в список urlpatterns:

```
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')),  
    path('__debug__/', include('debug_toolbar.urls')),  
]
```

Теперь меню отладочных инструментов Django Debug Toolbar установлено в вашем проекте. Давайте попробуем!

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/images/> в своем браузере. Теперь вы должны увидеть сворачиваемое боковое меню с правой стороны. Оно должна выглядеть следующим образом:

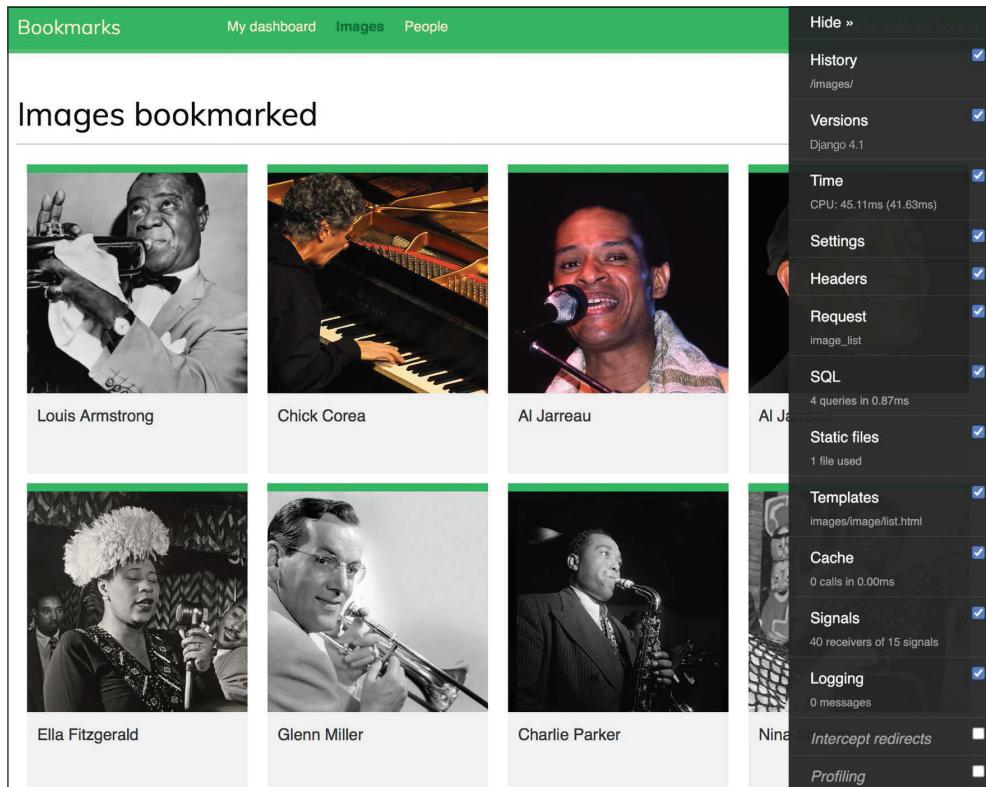


Рис. 7.7. Боковое меню отладочных инструментов Django

✍

Авторство изображений на рис. 7.7:

- Чик Кориа, автор: ataelw (лицензия: Creative Commons Attribution 2.0 Типовая форма: <https://creativecommons.org/licenses/by/2.0/>);
- Эл Джарро – Дюссельдорф, 1981, автор: Эдди Лауманс, он же RX-Guru (лицензия: Creative Commons Attribution 3.0 Unported: <https://creativecommons.org/licenses/by/3.0/>);
- Эл Джарро, авторы: Kingkongphoto и www.celebrity-photos.com (лицензия: Creative Commons Attribution-ShareAlike 2.0 Типовая форма: <https://creativecommons.org/licenses/by-sa/2.0/>).

Если боковое меню отладочных инструментов не появляется, то проверьте журнал консоли оболочки RunServer. Если вы видите ошибку MIME-типа, то, скорее всего, ваши файлы соотнесения MIME неверны либо нуждаются в обновлении.

Правильное соотнесение для файлов JavaScript и CSS достигается путем добавления следующих ниже строк в файл `settings.py`:

```
if DEBUG:
    import mimetypes
    mimetypes.add_type('application/javascript', '.js', True)
    mimetypes.add_type('text/css', '.css', True)
```

Панели меню отладочных инструментов Django

Меню отладочных инструментов Django Debug Toolbar содержит несколько панелей, которые упорядочивают отладочную информацию цикла запрос/ответ. Боковое вертикальное меню содержит ссылки на каждую панель, и в любой панели можно использовать флажок, чтобы ее активировать либо деактивировать. Изменение будет применено к следующему запросу. Это бывает удобно, когда та или иная панель не интересует, а вычисление добавляет к запросу слишком много накладных расходов.

Кликните по **Time** (Время) в боковом меню. Вы увидите следующую ниже панель:



Рис. 7.8. Панель **Time** – меню отладочных инструментов Django

Панель **Time** вставляет таймер для разных фаз цикла запроса/ответа. Она также показывает CPU, прошедшее время и количество переключений контекста. Если вы используете Windows, то панель **Time** вы не увидите. В Windows на инструментальной панели доступно и отображается только общее время.

Кликните по **SQL** в боковом меню. Вы увидите следующую ниже панель:



Рис. 7.9. Панель **SQL** – меню отладочных инструментов Django

Здесь можно наблюдать различные выполненные SQL-запросы. Эта информация помогает определять ненужные запросы, повторяющиеся запросы, которые можно реиспользовать, или длительные запросы, которые можно оптимизировать. Основываясь на своих находках, вы можете улучшать наборы запросов QuerySets в своих представлениях, при необходимости создавать новые индексы по полям модели или кешировать информацию, когда это необходимо. В этой главе вы научились оптимизировать запросы, которые предусматривают взаимосвязи, используя методы `select_related()` и `prefetch_related()`. Вы научитесь кешировать данные в главе 14 «Прорисовка и кеширование контента».

Кликните по **Templates** (Шаблоны) в боковом меню. Вы увидите следующую ниже панель:

The screenshot shows the 'Templates' panel of the Django Debug Toolbar. At the top, it says 'Templates (3 rendered)'. Below that, there are sections for 'Template paths' (None), 'Templates' (listing 'images/image/list.html' with its path and a 'Toggle context' link), 'base.html' (with its path and a 'Toggle context' link), and 'images/image/list_images.html' (with its path and a 'Toggle context' link). At the bottom, there is a section for 'Context processors' listing several Django built-in context processors: 'django.template.context_processors.csrf', 'django.template.context_processors.debug', 'django.template.context_processors.request', 'django.contrib.auth.context_processors.auth', and 'django.contrib.messages.context_processors.messages', each with a 'Toggle context' link.

Рис. 7.10. Панель **Templates** – меню отладочных инструментов *Django*

На этой панели показаны различные шаблоны, используемые при прорисовке контента, пути к шаблонам и используемый контекст. Также можно увидеть различные используемые процессоры контекста. Вы узнаете о процессорах контекста в главе 8 «Разработка интернет-магазина».

Кликните по **Signals** (Сигналы) в боковом меню. Вы увидите следующую панель:

Signal	Receivers
class_prepared	
connection_created	
got_request_exception	
m2m_changed	users_like_changed
post_delete	
post_init	ImageField.update_dimension_fields, ImageField.update_dimension_fields
post_migrate	create_permissions, create_contenttypes
post_save	signal_committed_filefields
pre_delete	
pre_init	
pre_migrate	inject_rename_contenttypes_operations
pre_save	find_uncommitted_filefields
request_finished	close_caches, close_old_connections, reset_urlconf
request_started	reset_queries, close_old_connections
setting_changed	reset_cache, clear_cache_handlers, update_installed_apps, update_connections_time_zone, clear_routers_cache, reset_template_engines, clear_serializers_cache, language_changed, localize_settings_changed, file_storage_changed, complex_setting_changed, root_urlconf_changed, static_storage_changed, static_finders_changed, auth_password_validators_changed, user_model_swapped, update_toolbar_config, reset_hashers, update_level_tags, FileSystemStorage._clear_cached_properties, FileSystemStorage._clear_cached_properties, FileSystemStorage._clear_cached_properties, StaticFileStorage._clear_cached_properties, FileSystemStorage._clear_cached_properties, ThumbnailFileStorage._clear_cached_properties

Рис. 7.11. Панель **Signals** – меню отладочных инструментов Django

На этой панели можно увидеть все сигналы, которые зарегистрированы в вашем проекте, и функции-получатели, прикрепленные к каждому сигналу. Например, тут можно найти созданную ранее функцию-получатель `users_like_changed`, прикрепленную к сигналу `m2m_changed`. Другие сигналы и получатели являются частью различных приложений Django.

Мы рассмотрели некоторые панели, поставляемые вместе с меню отладочных инструментов Django Debug Toolbar. Помимо встроенных панелей, на странице <https://django-debug-toolbar.readthedocs.io/en/latest/panels.html#first-party-panels> находятся дополнительные сторонние панели, которые можно загружать и использовать.

Команды меню отладочных инструментов Django

Помимо панелей отладки запросов/ответов, меню отладочных инструментов Django Debug Toolbar предоставляет команду управления по отладке SQL-запросов, исполняемых с помощью Django ORM. Команда управления `debugsqlshell` является репликой команды оболочки Django, но показывает инструкции SQL-запросов.

Следующей ниже командой откройте оболочку:

```
python manage.py debugsqlshell
```

Исполните приведенный далее исходный код:

```
>>> from images.models import Image  
>>> Image.objects.get(id=1)
```

Вы увидите такой результат:

```
SELECT "images_image"."id",  
       "images_image"."user_id",  
       "images_image"."title",  
       "images_image"."slug",  
       "images_image"."url",  
       "images_image"."image",  
       "images_image"."description",  
       "images_image"."created",  
       "images_image"."total_likes"  
  FROM "images_image"  
 WHERE "images_image"."id" = 1  
 LIMIT 21 [0.44ms]  
<Image: Django and Duke>
```

Эту команду можно использовать для тестирования ORM-запросов, перед тем как добавлять их в представления. Результирующую инструкцию SQL и время ее исполнения можно проверить для каждого ORM-вызыва.

В следующем далее разделе вы научитесь подсчитывать просмотры изображений с помощью резидентной базы данных Redis, которая обеспечивает доступ к данным с низкой задержкой и высокой пропускной способностью.

Подсчет просмотров изображений с помощью хранилища Redis

Redis (Remote Dictionary Server) – это продвинутая база данных в формате ключ-значение, позволяющая хранить различные типы данных. Она также имеет чрезвычайно быстрые операции ввода-вывода. Redis все хранит в памяти, но состояние данных можно сохранять, периодически сбрасывая набор данных на диск или добавляя каждую команду в журнал. Redis очень универсальна по сравнению с другими хранилищами в формате ключ-значение: она предоставляет набор мощных команд и поддерживает различные структуры

данных, такие как строки, хеши, списки, множества, упорядоченные множества и даже битовые карты или структуры данных Hyper-LogLog¹.

Хотя SQL лучше всего подходит для определяемого схемой постоянного хранения данных, Redis предлагает многочисленные преимущества при работе с быстро меняющимися данными, энергозависимым хранилищем или когда требуется быстрое кеширование. Давайте посмотрим, как можно использовать Redis для формирования новых функциональностей в проекте.

Дополнительная информация о базе данных Redis находится на ее домашней странице по адресу <https://redis.io/>.

База данных Redis предоставляет образ Docker, который упрощает развертывание Redis-сервера со стандартной конфигурацией.

Установка платформы Docker

Docker – это популярная платформа контейнеризации с открытым исходным кодом. Она обеспечивает разработчикам возможность упаковки приложений в контейнеры, упрощая процесс разработки, запуска, управления и распространения приложений.

Сначала скачайте и установите Docker для вашей ОС. Инструкции по скачиванию и установке Docker в Linux, macOS и Windows находятся на странице <https://docs.docker.com/get-docker/>.

Установка хранилища Redis

После установки платформы Docker на своем компьютере с Linux, macOS или Windows можно легко получить образ Redis платформы Docker. Запустите следующую ниже команду из оболочки:

```
docker pull redis
```

Она скачает образ Redis платформы Docker на локальный компьютер. Информация об официальном образе Redis платформы Docker находится на странице https://hub.docker.com/_/redis. Другие альтернативные способы установки базы данных Redis расположены на странице <https://redis.io/download/>.

Исполните следующую ниже команду в оболочке, чтобы запустить контейнер Docker для Redis:

¹ Битовые карты хранилища Redis – это расширение строкового типа данных, позволяющее обращаться со строковым литералом как с битовым вектором. Hyper-LogLog – это структура данных, которая используется для подсчета количества уникальных элементов в множестве с использованием малого, постоянного объема памяти. – Прим. перев.

```
docker run -it --rm --name redis -p 6379:6379 redis
```

Эта команда запускает Redis в контейнере Docker. Опция `-it` сообщает Docker, что можно переходить прямо внутрь контейнера для интерактивного ввода. Параметр `--rm` сообщает Docker, что при выходе из контейнера нужно очищать контейнер и удалять файловую систему автоматически. Опция `--name` используется для назначения контейнеру имени. Опция `-p` применяется для публикации порта 6379, на котором работает Redis, на тот же порт интерфейса хоста. В Redis по умолчанию используется порт 6379.

Вы должны увидеть результат, который заканчивается следующими ниже строками:

```
# Server initialized  
* Ready to accept connections
```

Оставьте сервер Redis работать на порту 6379 и откройте еще одну оболочку. Следующей ниже командой запустите клиента Redis:

```
docker exec -it redis sh
```

Вы увидите строку с символом решетки:

```
#
```

Следующей ниже командой запустите клиента Redis:

```
# redis-cli
```

Вы увидите приглашение оболочки клиента Redis. Например:

```
127.0.0.1:6379>
```

Клиент хранилища Redis позволяет исполнять команды Redis непосредственно из оболочки. Давайте попробуем несколько команд. Введите команду `SET` в оболочке Redis, чтобы сохранить значение в ключе:

```
127.0.0.1:6379> SET name "Peter"  
OK
```

Приведенная выше команда создает ключ `name` со строковым значением `"Peter"` в хранилище Redis. Результат `OK` указывает на то, что ключ был успешно сохранен.

Затем извлеките значение с помощью команды `GET`, как показано ниже:

```
127.0.0.1:6379> GET name  
"Peter"
```

Далее можно проверить существование ключа, используя команду EXISTS. Эта команда возвращает 1, если данный ключ существует, и 0 в противном случае:

```
127.0.0.1:6379> EXISTS name
(integer) 1
```

С помощью команды EXPIRE можно установить срок истечения ключа, причем она позволяет устанавливать время жизни ключа в секундах. Еще одним вариантом является использование команды EXPIREAT, которая на входе ожидает отметку времени в формате Unix. Срок истечения ключа общепринято использовать, когда Redis применяется в качестве кеша или для хранения волатильных данных:

```
127.0.0.1:6379> GET name
"Peter"
127.0.0.1:6379> EXPIRE name 2
(integer) 1
```

Подождите более двух секунд и попробуйте снова получить тот же ключ:

```
127.0.0.1:6379> GET name
(nil)
```

Ответ (nil), то есть нулевой ответ, означает, что ключ не найден. Кроме того, любой ключ можно удалить с помощью команды DEL, как показано ниже:

```
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)
```

Это всего лишь базовые команды для операций на ключах. Список всех команд Redis находится на странице <https://redis.io/commands/>, список всех типов данных Redis – на странице <https://redis.io/docs/manual/data-types/>.

Использование хранилища Redis вместе с Python

Для использования Redis вместе с Python понадобятся привязки. Следующей ниже командой установите redis-рү посредством pip:

```
pip install redis==4.3.4
```

Документация по `redis-py` находится на странице <https://redis-py.readthedocs.io/>.

Пакет `redis-py` взаимодействует с Redis, предоставляя Python'овский интерфейс, соответствующий синтаксису команд Redis. Следующей ниже командой откройте оболочку Python:

```
python manage.py shell
```

Выполните такой исходный код:

```
>>> import redis  
>>> r = redis.Redis(host='localhost', port=6379, db=0)
```

Приведенный выше исходный код создает соединение с базой данных Redis. В Redis базы данных идентифицируются по целочисленному индексу, а не по имени базы данных. По умолчанию клиент подключается к базе данных 0. Число доступных баз данных, на которое Redis рассчитан изначально, равно 16, но это значение можно изменить в конфигурационном файле `redis.conf`.

Далее с помощью оболочки Python установите значение ключа:

```
>>> r.set('foo', 'bar')  
True
```

Команда возвращает `True`, указывая на то, что ключ успешно создан. Теперь с помощью команды `get()` можно получить значение ключа повторно:

```
>>> r.get('foo')  
b'bar'
```

Как вы заметили из приведенного выше примера, методы Redis подчиняются синтаксису команд Redis.

Давайте интегрируем Redis в проект. Отредактируйте файл `settings.py` проекта `bookmarks`, добавив в него следующие ниже настроочные параметры:

```
REDIS_HOST = 'localhost'  
REDIS_PORT = 6379  
REDIS_DB = 0
```

Это настроочные параметры Redis-сервера и базы данных, которую вы будете использовать для своего проекта.

Хранение просмотров изображений в хранилище Redis

Давайте найдем способ сохранить общее число просмотров изображения. Если реализовать это с помощью Django ORM, то такая реализация будет предусматривать SQL-запрос UPDATE всякий раз, когда изображение отображается на странице.

Если вместо этого использовать Redis, то просто нужно увеличивать хранящийся в памяти счетчик, что будет приводить к гораздо большей производительности и меньшим накладным расходам.

Отредактируйте файл `views.py` приложения `images`, добавив следующий ниже исходный код после существующих инструкций `import`:

```
import redis
from django.conf import settings

# соединить с redis
r = redis.Redis(host=settings.REDIS_HOST,
                 port=settings.REDIS_PORT,
                 db=settings.REDIS_DB)
```

В приведенном выше исходном коде устанавливается соединение с базой данных Redis, чтобы использовать ее в своих представлениях. Отредактируйте файл `views.py` приложения `images`, видоизменив представление `image_detail`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # увеличить общее число просмотров изображения на 1
    total_views = r.incr(f'image:{image.id}:views')
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

В этом представлении используется команда `incr`, которая увеличивает значение данного ключа на 1. Если ключ не существует, то команда `incr` его создает. Метод `incr()` возвращает окончательное значение ключа после выполнения операции. Его значение сохраняется в переменной `total_views`, которая затем передается в контекст шаблона. Ключ Redis создается, используя формат `object-type:id:field` (например, `image:33:id`).



По традиции при именовании ключей Redis знак двоеточия используется в качестве разделителя для создания ключей в именном пространстве (или пространстве имен). Благодаря этому имена ключей становятся весьма подробными, а родственные ключи имеют в своих именах общие части одной и той же схемы.

Отредактируйте шаблон `images/image/detail.html` приложения `images`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
...
<div class="image-info">
    <div>
        <span class="count">
            <span class="total">{{ total_likes }}</span>
            like{{ total_likes|pluralize }}
        </span>
        <span class="count">
            {{ total_views }} view{{ total_views|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>
...
```

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Откройте страницу детальной информации об изображении в своем браузере и перезагрузите ее несколько раз. Вы увидите, что всякий раз, когда представление обрабатывается, общее количество отображаемых просмотров увеличивается на 1. Взгляните на следующий ниже пример:



Рис. 7.12. Страница детальной информации об изображении, включающая количество лайков и просмотров

Отлично! Вы успешно интегрировали Redis в проект с целью подсчета просмотров изображений. В следующем разделе вы научитесь формировать рейтинг самых просматриваемых изображений с помощью Redis.

Хранение рейтинга в хранилище Redis

Теперь с помощью хранилища Redis мы создадим что-то посложнее. Мы будем использовать Redis для хранения рейтинга самых просматриваемых изображений на платформе. Для этого мы будем использовать сортированные множества Redis. Сортированное множество – это неповторяющаяся коллекция строковых значений, в которой каждый элемент ассоциирован с баллом. Элементы сортируются по их баллу.

Отредактируйте файл `views.py` приложения `images`, добавив следующий ниже исходный код, выделенный жирным шрифтом, в представление `image_detail`:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # увеличить общее число просмотров изображения на 1
    total_views = r.incr(f'image:{image.id}:views')
    # увеличить рейтинг изображения на 1
    r.zincrby('image_ranking', 1, image.id)
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

Команда `zincrby()` используется для сохранения просмотров изображений в сортированном множестве с ключом `image:ranking`. В нем будут храниться `id` изображения и соответствующий балл, равный 1, который будет добавлен к общему баллу этого элемента сортированного множества. Такой подход позволит отслеживать все просмотры изображений в глобальном масштабе и иметь сортированное множество, упорядоченное по общему числу просмотров.

Теперь создайте новое представление, чтобы отображать рейтинг наиболее просматриваемых изображений. Добавьте следующий ниже исходный код в файл `views.py` приложения `images`:

```
@login_required
def image_ranking(request):
    # получить словарь рейтинга изображений
    image_ranking = r.zrange('image_ranking', 0, -1,
                             desc=True)[:10]
    image_ranking_ids = [int(id) for id in image_ranking]
    # получить наиболее просматриваемые изображения
    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})
```

Представление `image_ranking` работает следующим образом.

- Для получения элементов сортированного множества используется команда `zrange()`. Эта команда ожидает конкретно-прикладной диапазон в соответствии с самым низким и самым высоким баллами. Используя 0 в качестве наименьшего значения и -1 в качестве наибольшего, базе данных Redis сообщается, что нужно вернуть все элементы сортированного множества. Для извлечения элементов, упорядоченных по убыванию балла, также указывается параметр `desc=True`. Наконец, результаты нарезаются, используя `[:10]`, чтобы получить первые 10 элементов с наивысшим баллом.
- Создается список возвращаемых ИД изображений и сохраняется в переменной `image_ranking_ids` в виде списка целых чисел. По этим идентификаторам извлекаются объекты `Image`, и с помощью функции `list()` запрос принудительно исполняется. Важно вызвать принудительное исполнение набора запросов `QuerySet`, потому что для него будет использоваться списковый метод `sort()` (на этом этапе вместо набора запросов `QuerySet` нужен список объектов).

3. Объекты `Image` сортируются по индексу их появления в рейтинге изображений. Теперь в шаблоне можно использовать список `most_viewed`, чтобы отобразить 10 самых просматриваемых изображений.

Внутри каталога `images/image/template` приложения `images` создайте новый шаблон `rating.html` и добавьте в него следующий ниже исходный 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 %}
```

Этот шаблон довольно простой. Содержащиеся в списке `most_viewed` объекты `Image` прокручиваются в цикле, и их имена отображаются, включая ссылку на страницу детальной информации об изображении.

Наконец, необходимо создать шаблон URL-адреса для нового представления. Отредактируйте файл `urls.py` приложения `images`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [
    path('create/', views.image_create, name='create'),
    path('detail/<int:id>/<slug:slug>',
         views.image_detail, name='detail'),
    path('like/', views.image_like, name='like'),
    path('', views.image_list, name='list'),
    path('ranking/', views.image_ranking, name='ranking'),
]
```

Запустите сервер разработки, обратитесь к своему сайту из своего веб-браузера и загрузите страницу детальной информации об изображении несколько раз с разными изображениями. Затем обратитесь к URL-адресу `http://127.0.0.1:8000/images/ranking/` из своего браузера. Вы должны увидеть рейтинг изображений, как показано ниже:

Images ranking

1. Chick Corea
2. Louis Armstrong
3. Al Jarreau
4. Django Reinhardt
5. Django and Duke

Рис. 7.13. Страница рейтинга, сформированная на основе данных, извлеченных из Redis

Отлично! Вы только что создали рейтинг с помощью Redis.

Следующие шаги с Redis

Redis не является заменой базы данных SQL, но предлагает быстроерезидентное хранилище, которое больше подходит для определенных задач. Добавьте его в свой стек и используйте тогда, когда действительно чувствуете, что это необходимо. Ниже приведено несколько сценариев, в которых Redis бывает полезен.

- **Подсчет количеств:** как вы уже убедились, с помощью Redis очень легко управлять количествами. Команды `incg()` и `incby()` можно использовать для подсчета элементов.
- **Хранение последних элементов:** с помощью команд `lpush()` и `rpush()` можно добавлять элементы в начало/конец списка. Удаление и возврат первого/последнего элемента осуществляются посредством команд `lpop()`/`rpop()`. Длину списка можно обрезать с использованием `ltrim()`, чтобы поддерживать заданную длину.
- **Очереди:** в дополнение к командам `push` и `pop` Redis предлагает блокирование команды очереди.
- **Кеширование:** применение команд `expire()` и `expireat()` позволяет использовать Redis в качестве кеша. Кроме того, существуют сторонние для Django механизмы кеширования Redis.
- **Публикация/подписка:** Redis предоставляет команды для подписки/отписки и отправки сообщений в каналы.
- **Рейтинги и списки лидеров:** сортированные множества Redis сбаллами позволяют очень легко создавать списки лидеров.
- **Отслеживание в режиме реального времени:** благодаря быстрому вводу-выводу Redis идеально подходит для реально-временных сценариев.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter07>.
- Конкретно-прикладные модели пользователей: <https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#specifying-a-custom-user-model>.
- Фреймворк `contenttypes`: <https://docs.djangoproject.com/en/4.1/ref/contrib/contenttypes/>.
- Встроенные в Django сигналы: <https://docs.djangoproject.com/en/4.1/ref/signals/>.
- Конфигурационные классы приложений: <https://docs.djangoproject.com/en/4.1/ref/applications/>.
- Документация по меню отладочных инструментов Django Debug Toolbar: <https://django-debug-toolbar.readthedocs.io/>.
- Сторонние панели для меню отладочных инструментов Django Debug Toolbar: <https://django-debug-toolbar.readthedocs.io/en/latest/panels.html#third-party-panels>.
- Резидентное хранилище данных Redis: <https://redis.io/>.
- Инструкции по скачиванию и установке платформы Docker: <https://docs.docker.com/get-docker/>.
- Официальный образ Redis платформы Docker: https://hub.docker.com/_/redis.
- Опции скачивания Redis: <https://redis.io/download/>.
- Команды Redis: <https://redis.io/commands/>.
- Типы данных Redis: <https://redis.io/docs/manual/data-types/>.
- Документация по пакету `redis-py`: <https://redis-py.readthedocs.io/>.

Резюме

В этой главе вы разработали систему подписки, используя взаимосвязи многие-ко-многим с промежуточной моделью. Вы также создали поток активности, применяя обобщенные отношения, и оптимизировали наборы запросов с целью извлечения связанных объектов. Затем вы ознакомились с сигналами Django и создали функцию-получатель сигналов с целью де-нормализации количеств связанных объектов. Рассмотрели классы конфигурации приложений, которые использовались для загрузки обработчиков сигналов. Вы добавили меню отладочных инструментов Django Debug Toolbar в проект. Вы также научились устанавливать в свой проект и конфигурировать резидентное хранилище Redis. Наконец, вы применили Redis в проекте, чтобы хранить просмотры элементов, и с помощью нее сформировали рейтинг изображений.

В следующей главе вы научитесь разрабатывать интернет-магазин. Вы создадите каталог товаров и сформируете корзину покупок, используя сеансы. Вы научитесь создавать конкретно-прикладные процессоры контекста, а также будете управлять заказами клиентов и отправлять асинхронные уведомления с помощью Celery и RabbitMQ.

8

Разработка интернет-магазина

В предыдущей главе вы создали систему подписки и разработали поток активности пользователей. Вы также узнали, как работают сигналы Django, и интегрировали Redis в свой проект, чтобы вести подсчет просмотров изображений.

В этой главе вы начнете новый проект Django, состоящий из полнофункционального интернет-магазина. Данная и следующие две главы покажут, как разрабатывать ключевые функциональности платформы электронной коммерции. Ваш интернет-магазин предоставит клиентам возможность просматривать товары, добавлять их в корзину, применять коды скидок, проходить процесс оформления платежа, оплачивать кредитной картой и получать счета-фактуры. Вы также имплементируете рекомендательный механизм, чтобы рекомендовать товары своим клиентам, и будете использовать интернационализацию, чтобы предлагать свой сайт на нескольких языках.

В этой главе научитесь:

- создавать каталог товаров;
- создавать корзину покупок, используя сеансы Django;
- создавать конкретно-прикладные процессоры контекста;
- управлять заказами клиентов;
- конфигурировать в своем проекте очередь заданий Celery с помощью брокера сообщений RabbitMQ;
- отправлять асинхронные уведомления клиентам с помощью Celery;
- проводить мониторинг Celery посредством инструмента Flower.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter08>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Создание проекта интернет-магазина

Давайте начнем с нового проекта Django по разработке интернет-магазина. Ваши пользователи смогут просматривать каталог товаров и добавлять товары в корзину. Наконец, они смогут оформлять и размещать заказ. В этой главе будут рассмотрены следующие ниже функциональности интернет-магазина:

- создание моделей каталога товаров, добавление их на административный сайт и разработка базовых представлений для отображения каталога;
- разработка системы корзины покупок с использованием сеансов Django, предоставляющей пользователям возможность сохранять выбранные товары во время просмотра сайта;
- создание формы и функциональности размещения заказов на сайте;
- отправка пользователям асинхронного электронного письма о подтверждении заказа после его размещения.

Откройте оболочку и примените следующую ниже команду, чтобы создать новую виртуальную среду для этого проекта в каталоге `env`:

```
python -m venv env/myshop
```

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

```
source env/myshop/bin/activate
```

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

```
.\env\myshop\Scripts\activate
```

Приглашение оболочки отобразит вашу активную виртуальную среду, как показано ниже:

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

Следующей ниже командой установите Django в вашей виртуальной среде:

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

Запустите новый проект под названием `myshop` с приложением под названием `shop`, открыв оболочку и выполнив такую команду:

```
django-admin startproject myshop
```

Создана начальная структура проекта. Примените следующие ниже команды, чтобы попасть в каталог проекта и создать новое приложение с именем `shop`:

```
cd myshop/
django-admin startapp shop
```

Отредактируйте файл `settings.py`, добавив в список `INSTALLED_APPS` следующую ниже строку, выделенную жирным шрифтом:

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

Теперь приложение в этом проекте является активным. Давайте определим модели для каталога товаров.

Создание моделей каталога товаров

Каталог магазина будет состоять из товаров, разделенных на разные категории. У каждого товара будет имя, optionalное описание, optionalное изображение, цена и наличие.

Отредактируйте файл `models.py` только что созданного вами приложения `shop`, добавив следующий ниже исходный код:

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200,
                           unique=True)

    class Meta:
        ordering = ['name']
        indexes = [
            models.Index(fields=['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)
slug = models.SlugField(max_length=200)
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']
    indexes = [
        models.Index(fields=['id', 'slug']),
        models.Index(fields=['name']),
        models.Index(fields=['-created']),
    ]

    def __str__(self):
        return self.name

```

Это модели `Category` и `Product`. Модель `Category` состоит из поля `name` и уникального поля `slug` (уникальность подразумевает создание индекса). В `Meta`-классе модели `Category` определен индекс по полю `name`.

Поля модели `Product` таковы:

- `category`: внешний ключ (`ForeignKey`) к модели `Category`. Это взаимосвязь один-ко-многим: товар принадлежит одной категории, а категория содержит несколько товаров;
- `name`: название товара;
- `slug`: слаг этого товара для создания красивых URL-адресов;
- `image`: optionalное изображение товара;
- `description`: optionalное описание товара;
- `price`: в этом поле используется тип Python `decimal.Decimal`, чтобы хранить десятичное число фиксированной точности. Максимальное количество цифр (включая десятичные разряды) устанавливается с помощью атрибута `max_digits`, а десятичных разрядов – с помощью атрибута `decimal_places`;
- `available`: булево значение, указывающее на наличие или отсутствие товара. Оно будет использоваться для активирования/деактивирования товара в каталоге;

- `created`: в этом поле хранится информация о дате/времени создания объекта;
- `updated`: в этом поле хранится информация о дате/времени обновления объекта в последний раз.

Во избежание проблем с округлением в поле `price` вместо типа `FloatField` используется тип `DecimalField`.



Для хранения значений денежных сумм всегда следует использовать `DecimalField`. Внутри `FloatField` применяется Python'овский тип `float`, тогда как внутри `DecimalField` – Python'овский тип `Decimal`. Используя тип `Decimal`, вы избежите проблем с округлением чисел с плавающей запятой.

В Meta-классе модели `Product` был определен многопольный индекс по полям `id` и `slug`. Оба поля индексируются вместе с целью повышения производительности запросов, в которых эти два поля используются.

Планируется, что товары будут запрашиваться как по идентификатору, так и по слагу. Добавлен индекс по полю `name` и индекс по полю `created`. Перед именем поля использован дефис, чтобы определить индекс в убывающем порядке.

На рис. 8.1 показаны две созданные модели данных:

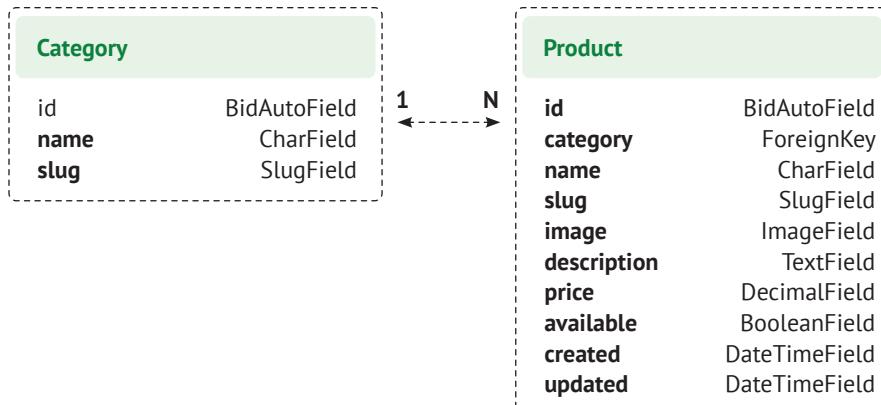


Рис. 8.1. Модели каталога товаров

На рис. 8.1 показаны различные поля моделей данных и взаимосвязи один-ко-многим между моделями `Category` и `Product`.

Результатом этих моделей будут следующие ниже таблицы базы данных, которые показаны на рис. 8.2.



Рис. 8.2. Таблицы базы данных для моделей каталога товаров

Взаимосвязь один-ко-многим между обеими таблицами определяется полем **category_id** в таблице **shop_product**, которая используется для хранения ИД связанной категории по каждому объекту **Product**.

Давайте создадим первоначальные миграции базы данных для приложения **shop**. Поскольку в своих моделях вы собираетесь работать с изображениями, необходимо установить библиотеку **Pillow**. Напомним, что в главе 4 «Разработка социального веб-сайта» вы научились устанавливать библиотеку **Pillow**, для того чтобы с ее помощью управлять изображениями. Откройте оболочку и следующей ниже командой установите **Pillow**:

```
pip install Pillow==9.2.0
```

Теперь выполните следующую команду, чтобы создать начальные миграции для проекта:

```
python manage.py makemigrations
```

Вы увидите такой результат:

```
Migrations for 'shop':
  shop/migrations/0001_initial.py
    - Create model Category
    - Create model Product
    - Create index shop_catego_name_289c7e_idx on field(s) name of model
      category
    - Create index shop_produc_id_f21274_idx on field(s) id, slug of model
      product
    - Create index shop_produc_name_a2070e_idx on field(s) name of model
      product
    - Create index shop_produc_created_ef211c_idx on field(s) -created of model
      product
```

Выполните следующую ниже команду, чтобы синхронизировать базу данных:

```
python manage.py migrate
```

Вы увидите результат, который содержит такую строку:

```
Applying shop.0001_initial... OK
```

Теперь база данных синхронизирована с вашими моделями.

Регистрация моделей каталога на сайте администрации

Давайте добавим ваши модели на сайт администрирования, чтобы легко управлять категориями и товарами. Отредактируйте файл `admin.py` приложения `shop`, добавив следующий ниже исходный 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',)}
```

Напомним, что атрибут `prepopulated_fields` используется для того, чтобы указывать поля, значение которых устанавливается автоматически с использованием значения других полей. Как вы уже убедились, это удобно для генерирования слагов.

Атрибут `list_editable` в классе `ProductAdmin` используется для того, чтобы задать поля, которые можно редактировать, находясь на странице отображения списка на сайте администрирования. Такой подход позволит редактировать несколько строк одновременно. Любое поле в `list_editable` также должно быть указано в атрибуте `list_display`, поскольку редактировать можно только отображаемые поля.

Теперь следующей ниже командой создайте сайт суперпользователя:

```
python manage.py createsuperuser
```

Введите желаемое пользовательское имя, адрес электронной почты и пароль. Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/shop/product/add/> в своем браузере и войдите под именем пользователя, которого вы только что создали. Добавьте новую категорию и товар, используя интерфейс администрирования. Форма для добавления товара должна выглядеть следующим образом:

Add product

Category: Tea

Name: Green tea

Slug: green-tea

Image: Choose File no file selected

Description: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Price: 30.00

Available

Save and add another Save and continue editing SAVE

Рис. 8.3. Форма для создания товара

Кликните по кнопке **Save** (Сохранить). Страница списка изменения товара на странице администрирования будет выглядеть следующим образом:

The screenshot shows the Django admin 'Products' list view. At the top, there's a header with 'Django administration', a welcome message for 'ADMIN', and links to 'VIEW SITE / CHANGE PASSWORD / LOG OUT'. Below the header, the URL 'Home · Shop > Products' is visible. The main area displays a table with one row for 'Green tea'. The table columns are: NAME, SLUG, PRICE, AVAILABLE, CREATED, and UPDATED. The 'AVAILABLE' column has a dropdown menu set to 'True'. The 'CREATED' and 'UPDATED' columns show specific dates: Jan. 30, 2022, 8:46 p.m. and Jan. 31, 2022, 3:36 p.m. respectively. To the right of the table is a 'FILTER' sidebar with sections for 'By available' (with 'All' selected), 'By created' (with 'Any date' selected), and 'By updated' (with 'Any date' selected). A 'Save' button is located at the bottom right of the table area.

Рис. 8.4. Страница списка изменения товара

Формирование представлений каталога

Для того чтобы отобразить каталог товаров на странице, необходимо создать представление перечисления списка всех товаров или фильтрации товаров по заданной категории. Отредактируйте файл `views.py` приложения `shop`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
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})
```

В приведенном выше исходном коде набор запросов QuerySet фильтруется с параметром `available=True`, чтобы получать только те товары, которые имеются в наличии. Опциональный параметр `category_slug` используется для дополнительной фильтрации товаров по заданной категории.

Еще требуется представление извлечения и отображения одного товара. Добавьте следующее ниже представление в файл `views.py`:

```
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})
```

На вход в представление `product_detail` должны передаваться параметры `id` и `slug`, чтобы извлекать экземпляр класса `Product`. Указанный экземпляр можно получить только по ИД, так как это уникальный атрибут. Однако в URL-адрес еще включается слаг, чтобы формировать дружественные для поисковой оптимизации URL-адреса товаров.

После разработки представлений списка товаров и детальной информации о товаре для них необходимо определить шаблоны URL-адресов. Создайте новый файл в каталоге приложения `shop` и назовите его `urls.py`. Добавьте в него следующий ниже исходный код:

```
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'),
]
```

Это шаблоны URL-адресов для каталога товаров. Для представления `product_list` здесь определено два разных шаблона URL-адреса: шаблон с именем `product_list`, который вызывает представление `product_list` без каких-либо параметров, и шаблон с именем `product_list_by_category`, который передает представлению параметр `category_slug`, чтобы фильтровать товары в соответствии с заданной категорией. Для представления `product_detail` был до-

бавлен шаблон, который передает в него параметры `id` и `slug`, чтобы извлекать конкретный товар.

Отредактируйте файл `urls.py` проекта `myshop`, придав ему следующий вид:

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

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

В главные шаблоны URL-адресов проекта вставляются URL-адреса приложения `shop` в конкретно-прикладном именном пространстве под тем же именем `shop`.

Далее отредактируйте файл `models.py` приложения `shop`, импортировав функцию `reverse()` и добавив метод `get_absolute_url()` в модели `Category` и `Product`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
from django.db import models
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, `get_absolute_url()` – это общепринятый способ получения URL-адреса заданного объекта.

Здесь используются шаблоны URL-адресов, которые были только что определены в файле `urls.py`.

Создание шаблонов каталога

Теперь нужно создать шаблоны для представлений списка товаров и детальной информации о товаре. Внутри каталога приложения `shop` создайте следующую ниже структуру каталогов и файлов:

```
templates/
  shop/
    base.html
  product/
    list.html
    detail.html
```

Необходимо определить базовый шаблон, а затем расширить его в шаблонах списка товаров и детальной информации о товаре. Отредактируйте шаблон `shop/base.html`, добавив следующий ниже исходный 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>
```

Это базовый шаблон, который будет использоваться для магазина. Для того чтобы включить используемые в шаблонах стили CSS и изображения, необходимо скопировать сопровождающие эту главу статические файлы, которые находятся в каталоге `static` приложения `shop`. Скопируйте их в то же место в своем проекте. Содержимое каталога находится на странице <https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter08/myshop/shop/static>.

Отредактируйте шаблон `shop/product/list.html`, добавив следующий ниже исходный код:

```
{% 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 %}
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк.

Это шаблон списка товаров. Он расширяет шаблон `shop/base.html` и использует контекстную переменную `categories`, чтобы отображать все категории на боковой панели, а также `products` для отображения товаров на текущей странице. Один и тот же шаблон используется как для списка всех имеющихся в наличии товаров, так и для списка товаров, отфильтрованных по категории. Поскольку поле `image` модели `Product` может быть пустым, необходимо предоставить типовое изображение для тех товаров, у которых нет изображения.

Это изображение находится в каталоге статических файлов с относительным путем `img/no_image.png`.

Поскольку для хранения изображений товаров используется тип `ImageField`, для раздачи скачанных файлов изображений нужен сервер разработки.

Отредактируйте файл `settings.py` проекта `myshop`, добавив следующие ниже настроочные параметры:

```
MEDIA_URL = 'media/'  
MEDIA_ROOT = BASE_DIR / 'media'
```

Параметр `MEDIA_URL` – это базовый URL-адрес, по которому раздаются закачанные пользователями медиафайлы. Параметр `MEDIA_ROOT` – это локальный путь, по которому эти файлы находятся и который следует формировать динамически, добавляя значение переменной `BASE_DIR` в качестве префикса.

Для того чтобы Django раздавал закачанные медиафайлы с помощью сервера разработки, отредактируйте файл `mainurls.py` проекта `myshop`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
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('', include('shop.urls', namespace='shop')),  
]  
  
if settings.DEBUG:  
    urlpatterns += static(settings.MEDIA_URL,  
                          document_root=settings.MEDIA_ROOT)
```

Напомним, что во время разработки статические файлы раздаются только таким образом. В производственной среде никогда не следует раздавать статические файлы с помощью Django; сервер разработки Django не способен эффективно раздавать статические файлы. В главе 17 «Выход в прямой эфир» будет объяснено, как раздавать статические файлы в производственной среде.

С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Добавьте пару товаров в свой магазин с помощью сайта администрирования и пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере. Вы увидите страницу списка товаров, которая будет выглядеть примерно так:

My shop

Your cart is empty.

Categories

All

Tea

Products

Green tea
\$30.00

Red tea
\$45.50

Tea powder
\$21.20

Рис. 8.5. Страница списка товаров

Изображения в этой главе:

- Зеленый чай: фото Цзя Е на Unsplash;
- Красный чай: фото Манки Кима на Unsplash;
- Чайный порошок: фото Фуонг Нгуена на Unsplash.

Если создать товар с помощью сайта администрирования и не закачать его изображение, то вместо него по умолчанию будет отображаться изображение `no_image.png`:

NO IMAGE
AVAILABLE

Green tea
\$30.00

Red tea
\$45.50

Tea powder
\$21.20

Рис. 8.6. Список товаров, отображающий типовое изображение, используемое для товаров, у которых нет изображения

Отредактируйте шаблон `shop/product/detail.html`, добавив следующий ниже исходный код:

```
{% 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 %}

```

В приведенном выше исходном коде метод `get_absolute_url()` вызывается с объектом связанной категории, чтобы отобразить имеющиеся товары, принадлежащие к той же категории.

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере и кликните по любому товару, чтобы увидеть страницу детальной информации о товаре. Она будет выглядеть следующим образом:

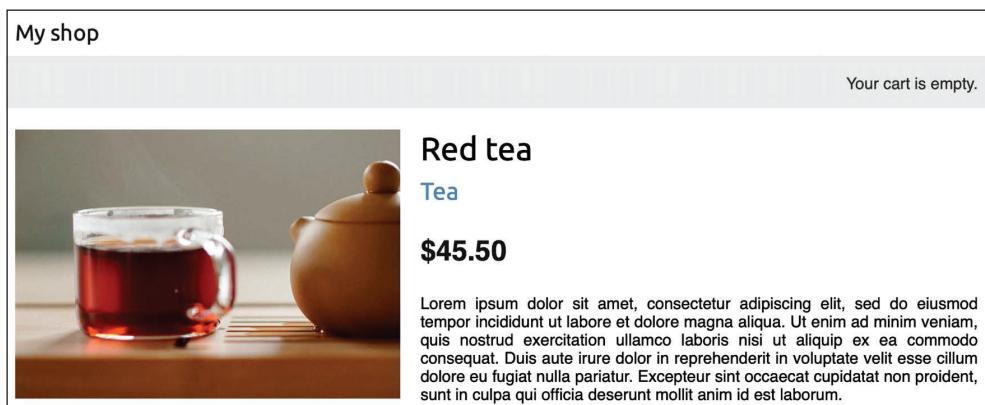


Рис. 8.7. Страница детальной информации о товаре

Вы создали базовый каталог товаров. Далее вы реализуете корзину покупок, которая позволит пользователям добавлять в нее любые товары во время просмотра интернет-магазина.

Разработка корзины покупок

После разработки каталога товаров следующим шагом будет создание корзины покупок, чтобы пользователи могли выбирать товары, которые они хотят приобрести. Корзина покупок позволяет пользователям выбирать товары и устанавливать сумму, на которую они желают сделать заказ, а затем временно сохранять эту информацию, пока они просматривают сайт, до тех пор, пока они в конечном итоге не разместят заказ. Состояние корзины покупок должно храниться в сеансе, чтобы товарные позиции корзины оставались в ней во время посещения магазина пользователем.

Для хранения состояния корзины будет использоваться встроенный в Django фреймворк сеансов. Корзина будет оставаться в сеансе до тех пор, пока сеанс не завершится либо пользователь не оформит заказ. Также нужно будет создать дополнительные модели Django для корзины и ее товарных позиций.

Использование сеансов Django

Django предоставляет фреймворк сеансов, который поддерживает анонимные и пользовательские сеансы. Фреймворк сеансов позволяет хранить произвольные данные каждого посетителя. Сеансовые данные хранятся на стороне сервера, а cookie-файлы содержат ИД сеанса, если только вы не используете cookie-ориентированный сеансовый механизм. Сеансовый промежуточный компонент управляет отправкой и получением cookie-файлов. Стандартный сеансовый механизм хранит сеансовые данные в базе данных, но существует возможность выбирать и другие сеансовые механизмы.

В целях использования сеансов необходимо, чтобы настроочный параметр MIDDLEWARE проекта содержал 'django.contrib.sessions.middleware.SessionMiddleware'. Данный промежуточный компонент управляет сеансами. Он добавляется в настрочный параметр MIDDLEWARE по умолчанию при создании нового проекта с помощью команды `startproject`.

Сеансовый промежуточный компонент делает текущий сеанс доступным в объекте `request`. К текущему сеансу можно обращаться, используя `request.session`, трактуя его как словарь Python для хранения и извлечения сеансовых данных. Словарь `session` по умолчанию принимает любой объект Python, который можно сериализовать в JSON. Значение переменной в сеансе устанавливается следующим образом:

```
request.session['foo'] = 'bar'
```

Извлечение сеансового ключа выполняется так:

```
request.session.get('foo')
```

Удаление ключа, который ранее был сохранен в сеансе, выполняется следующим образом:

```
del request.session['foo']
```



При входе пользователей на сайт их анонимный сеанс теряется, и создается новый сеанс для аутентифицированных пользователей. Если в анонимном сеансе хранятся элементы, которые необходимо сохранить после входа пользователя в систему, то придется копировать данные старого сеанса в новый сеанс. Это делается путем извлечения сеансовых данных перед входом пользователя в систему с помощью функции `login()`, встроенной в Django системы аутентификации, и сохранения их в сеансе после этого.

Настроочные параметры сеанса

С целью конфигурирования сеансов для своего проекта можно использовать несколько настроочных параметров. Наиболее важным является `SESSION_ENGINE`. Этот параметр позволяет указывать место, где будут храниться сеансы. По умолчанию Django хранит сеансы в базе данных, используя модель `Session` приложения `django.contrib.sessions`.

Django предлагает следующие варианты хранения сеансовых данных:

- **сеансы на основе базы данных:** сеансовые данные хранятся в базе данных. Этот сеансовый механизм применяется по умолчанию;
- **сеансы на основе файлов:** сеансовые данные хранятся в файловой системе;
- **сеансы на основе кеша:** сеансовые данные хранятся в кеш-памяти. Кеширующие бэкенды указываются в настроочном параметре `CACHES`. Хранение сеансовых данных в системе кеширования обеспечивает наилучшую производительность;
- **кешированные сеансы на основе базы данных:** сеансовые данные хранятся в кеше со сквозной записью¹ и в базе данных. Операции чтения используют базу данных только в том случае, если данные еще не находятся в кеше;
- **сеансы на основе cookie-файлов:** сеансовые данные хранятся в cookie-файлах, которые отправляются в браузер.

¹ Сквозная запись (write-through) – это метод хранения, при котором данные одновременно записываются в кеш и в соответствующую ячейку основной памяти. Кешированные данные позволяют быстро извлекать их по требованию, тогда как те же данные в основной памяти гарантируют, что ничего не будет потеряно в случае сбоя. – Прим. перев.



В целях повышения производительности следует использовать сеансовый механизм на основе кеша. Django поддерживает кеш-бэкенд Memcached прямо «из коробки», при этом еще имеются сторонние механизмы кеширования для Redis и других систем кеширования.

Сеансы можно адаптировать под конкретно-прикладную задачу с использованием конкретных настроек параметров. Вот несколько важных настроек параметров, связанных с сеансом:

- `SESSION_COOKIE_AGE`: продолжительность сеансовых cookie-файлов в секундах. По умолчанию равна 1 209 600 (две недели);
- `SESSION_COOKIE_DOMAIN`: домен, используемый для сеансовых cookie-файлов. Установите его значение равным `mydomain.com`, чтобы активировать междоменные cookie-файлы, или задайте `None` для стандартного доменного cookie-файла;
- `SESSION_COOKIE_HTTPONLY`: булево значение, указывающее на использование/неиспользование флага `HttpOnly` для сеансового cookie-файла. Если для этого параметра установлено значение `True`, то клиентский JavaScript не сможет получать доступ к сеансовому cookie-файлу. По умолчанию равно `True` с целью повышения защиты от перехвата пользовательского сеанса;
- `SESSION_COOKIE_SECURE`: булево значение, указывающее, что cookie-файл следует отправлять только в том случае, если соединение работает по протоколу HTTPS. По умолчанию равно `False`;
- `SESSION_EXPIRE_AT_BROWSER_CLOSE`: булево значение, указывающее, что сеанс должен истечь при закрытии браузера. По умолчанию равно `False`;
- `SESSION_SAVE_EVERY_REQUEST`: булево значение, которое, если оно равно `True`, будет сохранять сеанс в базе данных при каждом запросе. Срок истечения сеанса также обновляется при каждом его сохранении. По умолчанию равно `False`.

Список всех сеансовых настроек параметров и их заданные по умолчанию значения находятся по адресу <https://docs.djangoproject.com/en/4.1/ref/settings/#sessions>.

Срок истечения сеанса

Используя настройочный параметр `SESSION_EXPIRE_AT_BROWSER_CLOSE`, можно применять сеансы браузерной продолжительности либо постоянные (персистентные) сеансы. По умолчанию значение этого параметра установлено равным `False`, что принудительно устанавливает продолжительность сеанса равной значению, хранящемуся в настройочном параметре `SESSION_COOKIE_AGE`. Если значение параметра `SESSION_EXPIRE_AT_BROWSER_CLOSE` установлено равным `True`, то сеанс будет истекать, когда пользователь закроет браузер, и параметр `SESSION_COOKIE_AGE` не будет иметь никакого эффекта.

Для перезаписи продолжительности текущего сеанса можно воспользоваться методом `set_expiry()` объекта `request.session`.

Хранение корзин покупок в сеансах

Теперь нужно создать простую структуру, которую можно сериализовывать в JSON, чтобы хранить товарные позиции корзины в сеансе.

Корзина должна содержать следующие данные по каждому содержащемуся в ней товару:

- ИД экземпляра класса `Product`;
- выбранное количество товара;
- цена за единицу товара.

Поскольку цены на товары могут варьироваться, давайте воспользуемся подходом, при котором цена товара будет сохраняться вместе с самим товаром при его добавлении в корзину. Поступая таким образом, в момент, когда пользователи добавляют товар в свою корзину, будет использоваться текущая цена товара, независимо от того, изменится цена товара впоследствии или нет. Это означает, что цена, которую товар имеет, когда клиент добавляет его в корзину, остается для этого клиента в сеансе до завершения оформления заказа либо завершения сеанса.

Далее необходимо создать функциональность создания корзин покупок и связывания их с сеансами. Она должна работать следующим образом:

- в случае когда нужна корзина, проверить, что конкретно-прикладной сеансовый ключ установлен. Если в сеансе корзина не установлена, то создается новая корзина и сохраняется в сеансовом ключе корзины;
- в случае поочередных запросов выполнить ту же проверку и извлечь товарные позиции корзины из сеансового ключа корзины. При этом товарные позиции корзины извлекаются из сеанса, а связанные с ними объекты `Product` – из базы данных.

Отредактируйте файл `settings.py` проекта, добавив следующий ниже настроочный параметр:

```
CART_SESSION_ID = 'cart'
```

Это ключ, который будет использоваться для хранения корзины в пользовательском сеансе. Поскольку сеансы Django управляются по каждому посетителю, для всех сеансов можно использовать один и тот же сеансовый ключ корзины.

Давайте создадим приложение для управления корзинами покупок. Откройте терминал и создайте новое приложение, выполнив следующую ниже команду из каталога проекта:

```
python manage.py startapp cart
```

Затем отредактируйте файл `settings.py` проекта, добавив новое приложение в настроечный параметр `INSTALLED_APPS`, используя следующую ниже строку, выделенную жирным шрифтом:

```
INSTALLED_APPS = [  
    # ...  
    'shop.apps.ShopConfig',  
'cart.apps.CartConfig',  
]
```

Внутри каталога приложения `cart` создайте новый файл и назовите его `cart.py`. Добавьте в него следующий ниже исходный код:

```
from decimal import Decimal  
from django.conf import settings  
from shop.models import Product  
  
class Cart:  
    def __init__(self, request):  
        """  
        Инициализировать корзину.  
        """  
        self.session = request.session  
        cart = self.session.get(settings.CART_SESSION_ID)  
        if not cart:  
            # сохранить пустую корзину в сеансе  
            cart = self.session[settings.CART_SESSION_ID] = {}  
        self.cart = cart
```

Это класс `Cart`, который позволит управлять корзиной покупок. Переменная `cart` должна быть инициализирована объектом `request`. Текущий сеанс сохраняется посредством инструкции `self.session = request.session`, чтобы сделать его доступным для других методов класса `Cart`.

Сначала с помощью метода `self.session.get(settings.CART_SESSION_ID)` делается попытка получить корзину из текущего сеанса. Если в сеансе корзины нет, то путем задания пустого словаря в сеансе создается пустая корзина.

Далее надо сформировать словарь `cart` с идентификаторами товаров в качестве ключей и словарем, который будет содержать количество и цену, по каждому ключу товара. Поступая таким образом, можно гарантировать, что товар не будет добавляться в корзину более одного раза. Благодаря этому также упрощается извлечение товаров из корзины.

Давайте создадим метод добавления товаров в корзину или обновления их количества. Добавьте следующие ниже методы `add()` и `save()` в класс `Cart`:

```
class Cart:  
    # ...  
    def add(self, product, quantity=1, override_quantity=False):
```

```
"""
Добавить товар в корзину либо обновить его количество.
"""

product_id = str(product.id)
if product_id not in self.cart:
    self.cart[product_id] = {'quantity': 0,
                           'price': str(product.price)}
if override_quantity:
    self.cart[product_id]['quantity'] = quantity
else:
    self.cart[product_id]['quantity'] += quantity
self.save()

def save(self):
    # пометить сеанс как "измененный",
    # чтобы обеспечить его сохранение
    self.session.modified = True
```

Метод `add()` принимает на входе следующие ниже параметры:

- `product`: экземпляр `product` для его добавления в корзину либо его обновления;
- `quantity`: опциональное целое число с количеством товара. По умолчанию равен 1;
- `override_quantity`: это булево значение, указывающее, нужно ли заменить количество переданным количеством (`True`) либо прибавить новое количество к существующему количеству (`False`).

ИД товара используется в качестве ключа в словаре содержимого корзины. ИД товара конвертируется в строковое значение, потому что Django использует JSON для сериализации сеансовых данных, а JSON допускает только строковые имена ключей. ИД товара является ключом, а сохраняемое значение – словарем с числами количества и цены товара. Цена товара конвертируется из десятичного числа фиксированной точности в строковое значение для его сериализации. Наконец, вызывается метод `save()`, чтобы сохранить корзину в сеансе.

Метод `save()` помечает сеанс как измененный, используя `session.modified = True`. Это сообщает Django о том, что сеанс изменился и его необходимо сохранить.

Также потребуется метод удаления товаров из корзины. Добавьте следующий ниже метод в класс `Cart`:

```
class Cart:
    ...
    def remove(self, product):
        """
```

```
Удалить товар из корзины.  
"""  
product_id = str(product.id)  
if product_id in self.cart:  
    del self.cart[product_id]  
    self.save()
```

Метод `remove()` удаляет данный товар из словаря `cart` и вызывает метод `save()`, чтобы обновить корзину в сеансе.

Кроме того, нужно будет прокручивать в цикле содержащиеся в корзине товарные позиции и обращаться к соответствующим экземплярам класса `Product`. Для этого в своем классе можно определить метод `__iter__()`. Добавьте следующий ниже метод в класс `Cart`:

```
class Cart:  
    # ...  
    def __iter__(self):  
        """  
        Прокрутить товарные позиции корзины в цикле и  
        получить товары из базы данных.  
        """  
        product_ids = self.cart.keys()  
        # получить объекты product и добавить 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 методе `__iter__()` извлекаются присутствующие в корзине экземпляры класса `Product`, чтобы включить их в товарные позиции корзины. Текущая корзина копируется в переменную `cart`, и в нее добавляются экземпляры класса `Product`. Наконец, товары корзины прокручиваются в цикле, конвертируя цену каждого товара обратно в десятичное число фиксированной точности и добавляя в каждый товар атрибут `total_price`. Метод `__iter__()` позволит легко прокручивать товарные позиции корзины в представлениях и шаблонах.

Кроме того, понадобится способ возвращать общее число товаров в корзине. Когда функция `len()` исполняется с объектом в качестве аргумента, Python вызывает свой метод `__len__()` для получения его длины. Далее будет определен конкретно-прикладной метод `__len__()`, чтобы возвращать общее число товаров, хранящихся в корзине.

Добавьте следующий ниже метод `__len__()` в класс `Cart`:

```
class Cart:
    # ...
    def __len__(self):
        """
        Подсчитать все товарные позиции в корзине.
        """
        return sum(item['quantity'] for item in self.cart.values())
```

В результате будет возвращаться сумма количеств всех товаров в корзине. Добавьте следующий ниже метод расчета общей стоимости товаров в корзине:

```
class Cart:
    # ...
    def get_total_price(self):
        return sum(Decimal(item['price']) * item['quantity']
                  for item in self.cart.values())
```

Наконец, добавьте метод очистки сеанса корзины:

```
class Cart:
    # ...
    def clear(self):
        # удалить корзину из сеанса
        del self.session[settings.CART_SESSION_ID]
        self.save()
```

Теперь класс `Cart` готов к управлению корзинами покупок.

Создание представлений корзины покупок

Теперь, когда у нас имеется класс `Cart` для управления корзиной, необходимо создать представления, чтобы добавлять, обновлять или удалять товары из корзины. Нужно создать следующие ниже представления:

- представление добавления или обновления товаров в корзине, которое может обрабатывать текущие и новые количества;
- представление удаления товаров из корзины;
- представление отображения товарных позиций корзины и итоговых величин.

Добавление товаров в корзину

Для того чтобы добавить товары в корзину, нужна форма, позволяющая пользователю выбирать количество. Внутри каталога приложения `cart` создайте файл `forms.py`, добавив следующий ниже исходный код:

```
from django import forms

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)
    override = forms.BooleanField(required=False,
                                  initial=False,
                                  widget=forms.HiddenInput)
```

Эта форма будет использоваться для добавления товаров в корзину. Ваш класс `CartAddProductForm` содержит следующие два поля:

- `quantity`: позволяет пользователю выбирать количество от 1 до 20. Для конвертирования входных данных в целое число используется поле `TypedChoiceField` вместе с `coerce=int`;
- `override`: позволяет указывать, должно ли количество быть прибавлено к любому существующему количеству в корзине для этого товара (`False`) или же существующее количество должно быть переопределено данным количеством (`True`). Для этого поля используется виджет `HiddenInput`, так как это поле не будет показываться пользователю.

Давайте создадим представление добавления товаров в корзину. Отредактируйте файл `views.py` приложения `cart`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
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'],
        override_quantity=cd['override'])
return redirect('cart:cart_detail')
```

Это представление добавления товаров в корзину или обновления количества существующих товаров. В нем используется декоратор `require_POST`, чтобы разрешать запросы только методом `POST`. Указанное представление получает ИД товара в качестве параметра. Затем извлекается экземпляр класса `Product` с заданным ИД и выполняется валидация формы посредством `CartAddProductForm`. Если форма валидна, то товар в корзине либо добавляется, либо обновляется. Представление перенаправляет на URL-адрес `cart_detail`, который будет отображать содержимое корзины. Вы создадите представление `cart_detail` чуть позже.

Также потребуется представление удаления товаров из корзины. Добавьте следующий ниже исходный код в файл `views.py` приложения `cart`:

```
@require_POST
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')
```

Представление `cart_remove` получает ИД товара в качестве параметра. В нем используется декоратор `require_POST`, чтобы разрешать запросы только методом `POST`. Экземпляр товара извлекается с заданным ИД, и товар удаляется из корзины. Затем пользователь перенаправляется на URL-адрес `cart_detail`.

Наконец, потребуется представление отображения корзины и ее товаров. Добавьте следующее ниже представление в файл `views.py` приложения `cart`:

```
def cart_detail(request):
    cart = Cart(request)
    return render(request, 'cart/detail.html', {'cart': cart})
```

Представление `cart_detail` получает текущую корзину, чтобы ее отобразить.

Вы создали представления добавления товаров в корзину, обновления количества, удаления товаров из корзины и отображения содержимого корзины. Давайте добавим шаблоны URL-адресов этих представлений. Внутри каталога приложения `cart` создайте новый файл и назовите его `urls.py`. Добавьте в него следующие ниже URL-адреса:

```
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'),
]
```

Отредактируйте главный файл `urls.py` проекта `myshop`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом, чтобы включить URL-адреса корзины:

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

Проверьте, чтобы этот шаблон URL-адреса был вставлен перед шаблоном `shop.urls`, так как он более строгий, чем последний.

Разработка шаблона отображения корзины

Представления `cart_add` и `cart_remove` не будут прорисовывать никаких шаблонов, но вам нужно создать шаблон для представления `cart_detail`, чтобы отображались товарные позиции корзины и итоговые величины.

Внутри каталога приложения `cart` создайте следующую ниже файловую структуру:

```
templates/
    cart/
        detail.html
```

Отредактируйте шаблон `cart/detail.html`, добавив приведенный ниже исходный код:

```
{% 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>
                    <form action="{% url "cart:cart_remove" product.id %}"
method="post">
                        <input type="submit" value="Remove">
                        {% csrf_token %}
                    </form>
                </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 %}
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк.

Это шаблон, который используется для отображения содержимого корзины. Он содержит таблицу с товарами, хранящимися в текущей корзине. Пользователи имеют возможность изменять количество выбранных товаров, используя форму, которая отправляется методом POST в представление `cart_add`. Кроме того, пользователи могут удалять товары из корзины с помощью кнопки **Delete** (Удалить), предоставляемой для каждого из них. Наконец, используется HTML-форма с атрибутом `action`, которая указывает на URL-адрес `cart_remove`, включая ИД товара.

Добавление товаров в корзину

Теперь необходимо добавить кнопку **Add to cart** (Добавить в корзину) на страницу детальной информации о товаре. Отредактируйте файл `views.py` приложения `shop`, добавив `CartAddProductForm` в представление `product_detail`, как показано ниже:

```
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})
```

Отредактируйте шаблон `shop/product/detail.html` приложения `shop`, добавив следующую ниже форму к цене товара, как показано ниже. Новые строки выделены жирным шрифтом:

```
...
<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 }}
...
```

С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Теперь откройте адрес `http://127.0.0.1:8000/` в своем браузере и пройдите на страницу детальной информации о товаре. Она будет содержать форму для выбора количества перед добавлением товара в корзину. Страница будет выглядеть так:

My shop

Your cart is empty.

Red tea

Tea

\$45.50

Quantity: Add to cart

Рис. 8.8. Страница детальной информации о товаре, включающая форму для добавления товара в корзину

Выберите количество и кликните по кнопке **Add to cart** (Добавить в корзину). Форма передается на обработку в представление `cart_add` методом POST. Представление добавляет товар в корзину в сеансе, включая его текущую цену и выбранное количество. Затем оно перенаправляет пользователя на страницу детальной информации о корзине, которая будет выглядеть, как на рис. 8.9.

Image	Product	Quantity	Remove	Unit price	Price
	Red tea	2	Remove	\$45.50	\$91.00
Total					\$91.00

[Continue shopping](#) [Checkout](#)

Рис. 8.9. Страница детальной информации о корзине

Обновление количества товаров в корзине

Когда пользователи увидят корзину, они, возможно, захотят изменить количество товаров перед размещением заказа. В целях обеспечения этой функциональности необходимо разрешить пользователям изменять количество на странице детальной информации о корзине.

Отредактируйте файл `views.py` приложения `cart`, добавив следующие ниже строки, выделенные жирным шрифтом, в представление `cart_detail`:

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(initial={
            'quantity': item['quantity'],
            'override': True})
    return render(request, 'cart/detail.html', {'cart': cart})
```

По каждому товару в корзине создается экземпляр `CartAddProductForm`, чтобы разрешить изменение количества товара. Форма инициализируется текущим количеством товара, и поле `override` получает значение `True`, чтобы при передаче формы на обработку в представление `cart_add` текущее количество заменялось новым.

Теперь отредактируйте шаблон `cart/detail.html` приложения `cart` и найдите следующую ниже строку:

```
<td>{{ item.quantity }}</td>
```

Замените указанную строку таким исходным кодом:

```
<td>
    <form action="{% url "cart:cart_add" product.id %}" method="post">
        {{ item.update_quantity_form.quantity }}
        {{ item.update_quantity_form.override }}
        <input type="submit" value="Update">
        {% csrf_token %}
    </form>
</td>
```

С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/cart/` в своем браузере.

Вы увидите форму для редактирования количества по каждой товарной позиции корзины, как показано ниже:

Your shopping cart					
Image	Product	Quantity	Remove	Unit price	Price
	Red tea	2 <input type="button" value=""/>	<input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$45.50 \$91.00
Total					\$91.00
					<input type="button" value="Continue shopping"/> <input type="button" value="Checkout"/>

Рис. 8.10. Страница детальной информации о корзине, включающая форму для обновления количества товаров

Измените количество товара и кликните по кнопке **Update** (Обновить), чтобы протестировать новую функциональность. Вы также можете удалить товар из корзины, кликнув по кнопке **Remove** (Удалить).

Создание процессора контекста для текущей корзины

Возможно, вы заметили, что сообщение **Your cart is empty** (Ваша корзина пуста) отображается в шапке сайта, даже если в корзине есть товары. Вместо этого вы должны отображать общее количество товаров в корзине и общую стоимость. Поскольку эта информация должна отображаться на всех страницах, необходимо создать процессор контекста, чтобы включать текущую корзину в контекст запроса, независимо от представления, которым обрабатывается запрос.

Процессоры контекста

Процессор контекста – это функция Python, которая принимает объект `request` в качестве аргумента и возвращает словарь, который добавляется в контекст запроса. Процессоры контекста оказываются как нельзя кстати, когда нужно сделать что-то глобально доступным для всех шаблонов.

По умолчанию при создании нового проекта с помощью команды `start-project` проект содержит следующие ниже процессоры контекста шаблона в параметре `context_processors` внутри настроичного параметра `TEMPLATES`:

- `django.template.context_processors.debug`: устанавливает булевые переменные `debug` и `sql_queries` в контексте, представляющие список SQL-запросов, исполняемых в запросе;
- `django.template.context_processors.request`: устанавливает переменную `request` в контексте;

- `django.contrib.auth.context_processors.auth`: устанавливает переменную `user` в запросе;
- `django.contrib.messages.context_processors.messages`: устанавливает переменную `messages` в контексте, содержащую все сообщения, генерированные с использованием фреймворка сообщений.

Django также активирует `django.template.context_processors.csrf`, чтобы избегать атак с подделкой межсайтовых запросов (**CSRF**). Этого процессора контекста нет в настроечном параметре, но он всегда активирован, и его невозможно деактивировать из соображений безопасности.

Список всех встроенных процессоров контекста находится по адресу <https://docs.djangoproject.com/en/4.1/ref/templates/api/#built-in-template-context-processors>.

Установка корзины в контекст запроса

Давайте создадим процессор контекста, чтобы установить текущую корзину в контекст запроса. С помощью него можно получать доступ к корзине в любом шаблоне.

Внутри каталога приложения `cart` создайте новый файл и назовите его `context_processors.py`. Процессоры контекста могут располагаться в вашем исходном коде где угодно, но размещение их здесь будет поддерживать ваш код в хорошо организованном состоянии. Добавьте в файл следующий ниже исходный код:

```
from .cart import Cart

def cart(request):
    return {'cart': Cart(request)}
```

Здесь в процессоре контекста создается экземпляр класса `Cart` с объектом `request` в качестве параметра и обеспечивается его доступность для шаблонов в виде переменной `cart`.

Отредактируйте файл `settings.py` проекта, добавив `cart.context_processors.cart` в опцию `context_processors` внутри настроечного параметра `TEMPLATES`, как показано ниже. Новая строка выделена жирным шрифтом:

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

```
'django.contrib.messages.context_processors.messages',
'cart.context_processors.cart',
],
},
],
]
```

Процессор контекста переменной `cart` будет исполняться всякий раз, когда шаблон прорисовывается с использованием встроенного в Django контекстного класса `RequestContext`. Переменная `cart` будет устанавливаться в контекст ваших шаблонов. Подробнее о классе `RequestContext` можно почитать по адресу <https://docs.djangoproject.com/en/4.1/ref/templates/api/#django.template.RequestContext>.



Процессоры контекста исполняются во всех запросах, в которых используется контекстный класс `RequestContext`. В случае если ваша функциональность не требуется во всех шаблонах, в особенности если она предусматривает запросы к базе данных, возможно, вместо процессора контекста вы захотите создать конкретно-прикладной шаблонный тег.

Затем откройте шаблон `shop/base.html` приложения `shop` и найдите следующие ниже строки:

```
<div class="cart">
    Your cart is empty.
</div>
```

Замените эти строки таким исходным кодом:

```
<div class="cart">
    {% with total_items=cart|length %}
        {% if total_items > 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>
```

С помощью следующей команды перезапустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере и добавьте несколько товаров в корзину.

Теперь в шапке сайта можно увидеть общее количество товаров в корзине и их общую стоимость, как показано ниже:

The screenshot shows a web application interface for a shopping cart. At the top, there's a header with the text "My shop". Below it, a message "Your cart: 3 items, \$121.00" is circled in red. The main content area is titled "Your shopping cart". It contains a table with the following data:

Image	Product	Quantity	Remove	Unit price	Price
	Green tea	1 <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$30.00	\$30.00
	Red tea	2 <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$45.50	\$91.00
Total				\$121.00	

At the bottom right of the page are two buttons: "Continue shopping" and "Checkout".

Рис. 8.11. Шапка сайта, отображающая текущие товары в корзине

Вы завершили работу над корзиной. Далее вы создадите функциональность по регистрации заказов клиентов.

Регистрация заказов клиентов

После оформления заказа необходимо сохранить заказ в базе данных. Заказы будут содержать информацию о клиентах и товарах, которые они покупают.

Следующей ниже командой создайте новое приложение для управления заказами клиентов:

```
python manage.py startapp orders
```

Отредактируйте файл `settings.py` проекта, добавив новое приложение в параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [
    # ...
    'shop.apps.ShopConfig',
    'cart.apps.CartConfig',
    'orders.apps.OrdersConfig',
]
```

Вы активировали приложение `orders`.

Создание моделей заказа

Одна модель понадобится для хранения детальной информации о заказе и вторая – для хранения приобретенных товаров, включая их цену и количество. Отредактируйте файл `models.py` приложения `orders`, добавив следующий ниже исходный код:

```
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']
        indexes = [
            models.Index(fields=['-created']),
        ]

    def __str__(self):
        return f'Order {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 str(self.id)

    def get_cost(self):
        return self.price * self.quantity
```

Модель `Order` содержит несколько полей для хранения информации о клиенте и булево поле `paid`, которое по умолчанию имеет значение `False`. Позже это поле будет использоваться для того, чтобы различать оплаченные и неоплаченные заказы. Здесь также определен метод `get_total_cost()`, который получает общую стоимость товаров, приобретенных в этом заказе.

Модель `OrderItem` позволяет хранить товар, количество и цену, уплаченную за каждый товар. Здесь определен метод `get_cost()`, который возвращает стоимость товара путем умножения цены товара на количество.

Выполните следующую ниже команду, чтобы создать начальные миграции для приложения `orders`:

```
python manage.py makemigrations
```

Вы увидите примерно такой результат:

```
Migrations for 'orders':
  orders/migrations/0001_initial.py
    - Create model Order
    - Create model OrderItem
    - Create index orders_order_created_743fca_idx on field(s) -created of model
      order
```

Выполните следующую ниже команду, чтобы применить новую миграцию:

```
python manage.py migrate
```

Вы увидите следующий ниже результат:

```
Applying orders.0001_initial... OK
```

Теперь модели заказов синхронизированы с базой данных.

Включение моделей заказа на сайт администрирования

Добавьте добавим модели заказа на сайт администрирования. Отредактируйте файл `admin.py` приложения `orders`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
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]
```

Класс `ModelInline` используется с моделью `OrderItem`, чтобы включать ее *внутристрочно*¹ в класс `OrderAdmin`. Атрибут `inlines` позволяет вставлять модель в ту же страницу редактирования, что и связанная с ней модель.

С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/orders/order/add/` в своем браузере. Вы увидите такую страницу:

¹ Англ. inline; син. локально. – Прим. перев.

Add order

First name:

Last name:

Email:

Address:

Postal code:

City:

Paid

ORDER ITEMS			
PRODUCT	PRICE	QUANTITY	DELETE?
<input type="text"/> 	<input type="text"/> ₽	<input type="text"/> 1 ₽	
<input type="text"/> 	<input type="text"/> ₽	<input type="text"/> 1 ₽	
<input type="text"/> 	<input type="text"/> ₽	<input type="text"/> 1 ₽	

[+ Add another Order item](#)

Рис. 8.12. Форма для добавления заказа, включающая OrderItemInline

Создание заказов клиентов

Созданные ранее модели заказа будут использоваться для вывода товарных позиций корзины в постоянное хранилище, когда пользователь наконец разместит заказ. Новый заказ будет создан после следующих ниже шагов:

- 1) предоставить пользователю форму заказа, чтобы тот заполнил ее своими данными;
- 2) создать новый экземпляр `Order` с введенными данными и создать связанный экземпляр `OrderItem` для каждого товара в корзине;
- 3) очистить все содержимое корзины и перенаправить пользователя на страницу успеха.

Сперва потребуется форма для ввода детальной информации о заказе. Внутри каталога приложения `orders` создайте новый файл и назовите его `forms.py`. Добавьте в него следующий ниже исходный 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']
```

Это форма, которая будет использоваться для создания новых объектов Order. Теперь потребуется представление обработки формы и создания нового заказа. Отредактируйте файл `views.py` приложения `orders`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
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'])
            # очистить 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 представлении `order_create` текущая корзина извлекается из сеанса посредством инструкции `cart = Cart(request)`.

В зависимости от метода запроса выполняется следующая работа:

- запрос методом GET:** создает экземпляр формы `OrderCreateForm` и прорисовывает шаблон `orders/order/create.html`;
- запрос методом POST:** выполняет валидацию отправленных в запросе данных. Если данные валидны, то в базе данных создается новый заказ, используя инструкцию `order = form.save()`. Товарные позиции корзины

прокручиваются в цикле, и для каждой из них создается `OrderItem`. Наконец, содержимое корзины очищается, и шаблон `orders/order/created.html` прорисовывается.

Внутри каталога приложения `orders` создайте новый файл и назовите его `urls.py`. Добавьте в него следующий ниже исходный код:

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

app_name = 'orders'

urlpatterns = [
    path('create/', views.order_create, name='order_create'),
]
```

Это шаблон URL-адреса представления `order_create`.

Отредактируйте файл `urls.py` проекта `myshop` и вставьте следующий ниже шаблон. Не забудьте поместить его перед шаблоном `shop.urls`, как показано далее. Новая строка выделена жирным шрифтом:

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

Откройте шаблон `cart/detail.html` приложения `cart` и найдите вот эту строку:

```
<a href="#" class="button">Checkout</a>
```

Добавьте URL-адрес `order_create` в HTML-атрибут `href`, как показано ниже:

```
<a href="{% url "orders:order_create" %}" class="button">
    Checkout
</a>
```

Теперь пользователи могут переходить со страницы детальной информации о корзине в форму для заказа.

Остается определить шаблоны создания заказов. Внутри каталога приложения `orders` создайте следующую ниже файловую структуру:

```
templates/
    orders/
```

```
order/
  create.html
  created.html
```

Отредактируйте шаблон `orders/order/create.html`, добавив следующий ниже исходный 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 method="post" class="order-form">
        {{ form.as_p }}
        <p><input type="submit" value="Place order"></p>
        {% csrf_token %}
    </form>
{% endblock %}
```

Этот шаблон отображает товарные позиции корзины, включая итоговые величины и форму для размещения заказа.

Отредактируйте шаблон `orders/order/created.html`, добавив следующий ниже исходный код:

```
{% 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 %}
```

Это шаблон, который прорисовывается при успешном создании заказа.

Запустите сервер веб-разработки, чтобы загрузить новые файлы. Откройте адрес <http://127.0.0.1:8000/> в своем браузере, добавьте пару товаров в корзину и пройдите на страницу оформления заказа. Вы увидите следующую ниже форму:

The screenshot shows a web application interface for a shopping cart. At the top, it says "My shop" and "Your cart: 4 items, \$166.50". Below this, the title "Checkout" is displayed. On the left, there is a form for entering personal details:

- First name: Antonio
- Last name: Melé
- Email: antonio.mele@zenxit.com
- Address: 1 Bank Street
- Postal code: E14 4AD
- City: London

On the right, there is a summary section titled "Your order" which lists the items purchased:

Item	Quantity	Unit Price	Total Price
Red tea	3x	\$136.50	\$136.50
Green tea	1x	\$30.00	\$30.00
Total:			\$166.50

A blue button at the bottom left of the form area is labeled "Place order".

Рис. 8.13. Страница создания заказа, включающая форму для оформления заказа и детальную информацию

Заполните форму валидными данными и кликните по кнопке **Place order** (Разместить заказ). Заказ будет создан, и вы увидите страницу успеха, как показано ниже:



Рис. 8.14. Шаблон созданного заказа, отображающий номер заказа

Заказ зарегистрирован, корзина очищена.

Возможно, вы заметили, что сообщение **Your cart is empty** (Ваша корзина пуста) отображается в шапке после завершения заказа. Это вызвано тем, что корзина была очищена. Данное сообщение можно легко убрать в представлениях, в контексте шаблона которых есть объект `order`.

Отредактируйте шаблон `shop/base.html` приложения `shop`, заменив следующую ниже строку, выделенную жирным шрифтом:

```
...
<div class="cart">
    {% with total_items=cart|length %}
        {% if total_items > 0 %}
            Your cart:
            <a href="{% url "cart:cart_detail" %}">
                {{ total_items }} item{{ total_items|pluralize }},
                ${{ cart.get_total_price }}
            </a>
        {% elif not order %}
            Your cart is empty.
        {% endif %}
    {% endwith %}
</div>
...
```

При создании заказа сообщение **Your cart is empty** больше отображаться не будет.

Теперь откройте сайт администрирования по адресу `http://127.0.0.1:8000/admin/orders/order/`. Вы увидите, что заказ был успешно создан, например:

Select order to change										
Action:		ID	FIRST NAME	LAST NAME	EMAIL	ADDRESS	POSTAL CODE	CITY	PAID	CREATED
<input type="checkbox"/>	1	Antonio	Melé		antonio.mele@zenxit.com	1 Bank Street	E14 4AD	London		Jan. 31, 2022, 5:46 p.m.
1 order										

Рис. 8.15. Раздел списка изменений заказов на сайте администрирования, включающий созданный заказ

Вы реализовали систему заказов. Далее вы научитесь создавать асинхронные задания, чтобы отправлять пользователям электронные письма о подтверждении размещенного ими заказа.

Асинхронные задания

При получении HTTP-запроса необходимо как можно быстрее вернуть ответ пользователю. Напомним, что в главе 7 «Отслеживание действий пользователя» вы использовали меню отладочных инструментов Django Debug Toolbar, чтобы проверять время различных фаз цикла запроса/ответа и время исполнения выполненных SQL-запросов. Каждое исполняемое в ходе цикла запроса/ответа задание прибавляет к общему времени ответа. Длительные задания могут серьезно замедлять ответ сервера. Как возвращать быстрый ответ пользователю, при этом выполняя времязатратные задания? Это можно делать с помощью асинхронного исполнения заданий.

Работа с асинхронными заданиями

Исполняя определенные задания в фоновом режиме, можно снимать нагрузку с цикла запрос/ответ. Например, платформа для видеообмена позволяет пользователям закачивать видео на платформу, но требует много времени для транскодирования закачанных видео. Когда пользователь закачивает видео, сайт может вернуть ответ, информирующий о том, что транскодирование скоро начнется, и начать транскодирование видео асинхронно. Еще один пример – отправка электронных писем пользователям. Если ваш сайт отправляет уведомления по электронной почте из представления, то соединение по простому протоколу передачи почты (**SMTP**) может завершиться ошибкой либо замедлить ответ. Отправляя электронное письмо асинхронно, избегается блокировка исполнения исходного кода.

Асинхронное исполнение доказало свою актуальность особенно в отношении процессов, интенсивных по объему обработки данных, используемых ресурсов и времени, или процессов, подверженных сбоям, для которых может потребоваться политика повторных попыток соединения.

Работники, очереди сообщений и брокеры сообщений

В то время как ваш веб-сервер обрабатывает запросы и возвращает ответы, вам нужен второй, основанный на заданиях сервер под названием **работник**¹, чтобы обрабатывать асинхронные задания. Один или несколько работников могут работать и выполнять задания в фоновом режиме. Эти работники могут обращаться к базе данных, обрабатывать файлы, отправлять электронные письма и т. д. Работники могут даже ставить будущие задания в очередь. При этом главный веб-сервер будет оставаться свободным для обработки HTTP-запросов.

Для того чтобы сообщать работникам о подлежащих исполнению заданиях, нужно отправлять **сообщения**. Связь осуществляется через брокеров путем добавления сообщения в **очередь сообщений**, которая в сущности представляет собой структуру данных, организованную по принципу «первым вошел – первым вышел» (**FIFO**). Когда брокер становится доступным, он берет первое сообщение из очереди и запускает исполнение соответствующего задания. По завершении брокер берет следующее сообщение из очереди и отправляет соответствующее задание на исполнение. Брокеры простояивают, когда очередь сообщений пуста. При использовании нескольких брокеров каждый брокер принимает первое доступное сообщение в том порядке, в котором они становятся доступными. Очередь обеспечивает, чтобы каждый брокер получал только по одному заданию за раз и чтобы ни одно задание не обрабатывалось более чем одним работником.

На рис. 8.16 показан процесс работы очереди сообщений:



Рис. 8.16. Асинхронное исполнение с использованием очереди сообщений и работников

Производитель отправляет сообщение в очередь, а работники обрабатывают сообщения в порядке очереди; первое добавленное в очередь сообщение является первым, которое должно быть обработано работником(ами).

Для того чтобы управлять очередью сообщений, нужен **брокер сообщений**. Брокер сообщений используется для трансляции сообщений в формаль-

¹ Англ. worker – Прим. перев.

ный протокол обмена сообщениями и управления очередями для нескольких получателей. Он обеспечивает надежное хранение и гарантированную доставку сообщений. Брокер сообщений позволяет создавать очереди, маршрутизировать сообщения, распределять сообщения между работниками и т. д.

Использование Django с Celery и RabbitMQ

Celery – это очередь заданий, которая может обрабатывать огромное количество сообщений. Она будет использоваться для формирования асинхронных заданий как функций Python в приложениях Django. Мы будем запускать работников Celery, которые будут прослушивать брокера сообщений, чтобы получать новые сообщения для обработки асинхронных заданий.

Используя очередь заданий Celery, можно не только легко создавать асинхронные задания и обеспечивать их наискорейшее исполнение работниками, но и планировать их запуск в конкретное время. Документация Celery находится на странице <https://docs.celeryq.dev/en/stable/index.html>.

Очередь заданий Celery общается посредством сообщений и требует, чтобы брокер сообщений был посредником между клиентами и работниками. С Celery могут работать несколько вариантов брокера сообщений, включая хранилища данных в формате ключ/значение, такие как Redis, или фактический брокер сообщений, такой как RabbitMQ.

Брокер сообщений RabbitMQ является наиболее широко используемым из всех. Он поддерживает несколько протоколов обмена сообщениями, таких как продвинутый протокол очередности сообщений AMQP¹, и является рекомендуемым обработчиком сообщений для очереди заданий Celery. Брокер сообщений RabbitMQ легковесен, легко развертывается и приспособлен под конфигурирование с целью обеспечения масштабируемости и высокой доступности.

На рис. 8.17 показано, как Django, Celery и RabbitMQ будут использоваться для исполнения асинхронных заданий:



Рис. 8.17. Архитектура асинхронных заданий с участием Django, RabbitMQ и Celery

¹ Англ. Advanced Message Queuing Protocol. – Прим. перев.

Установка очереди заданий Celery

Давайте установим очередь заданий Celery и интегрируем ее в проект. Установите Celery посредством pip, используя следующую ниже команду:

```
pip install celery==5.2.7
```

Введение в очередь заданий Celery находится на странице <https://docs.celeryq.dev/en/stable/getting-started/introduction.html>.

Установка брокера сообщений RabbitMQ

Сообщество RabbitMQ предоставляет образ Docker, который упрощает развертывание сервера RabbitMQ со стандартной конфигурацией. Напомним, что вы научились устанавливать Docker в главе 7 «Отслеживание действий пользователя».

После установки Docker на свой компьютер можно легко получить образ брокера сообщений RabbitMQ платформы Docker, выполнив следующую ниже команду из командной оболочки:

```
docker pull rabbitmq
```

Она скачает образ RabbitMQ платформы Docker на ваш локальный компьютер. Информация об официальном образе RabbitMQ платформы Docker находится на странице https://hub.docker.com/_/rabbitmq.

Если вы хотите установить RabbitMQ исходно на свой компьютер вместо использования Docker, то подробные руководства по установке для различных операционных систем находятся на странице <https://www.rabbitmq.com/download.html>.

Исполните следующую ниже команду в оболочке, чтобы запустить сервер RabbitMQ с Docker:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672  
rabbitmq:management
```

Эта команда сообщает RabbitMQ о том, что нужно работать на порту 5672, и затем запускается его пользовательский веб-интерфейс управления на порту 15672.

Вы увидите результат, который содержит следующие ниже строки:

```
Starting broker...  
...  
completed with 4 plugins.  
Server startup complete; 4 plugins started.
```

Сервер брокера сообщений RabbitMQ работает на порту 5672 и готов принимать сообщения.

Доступ к встроенному в RabbitMQ пользовательскому интерфейсу управления

Пройдите по URL-адресу <http://127.0.0.1:15672/> в своем браузере. Вы увидите экран входа во встроенный в RabbitMQ пользовательский интерфейс управления. Он будет выглядеть так:



Рис. 8.18. Экран входа во встроенный в RabbitMQ пользовательский интерфейс управления

Ведите `guest` в качестве пользовательского имени и пароль и кликните по **Login** (Войти). Вы увидите следующий ниже экран:

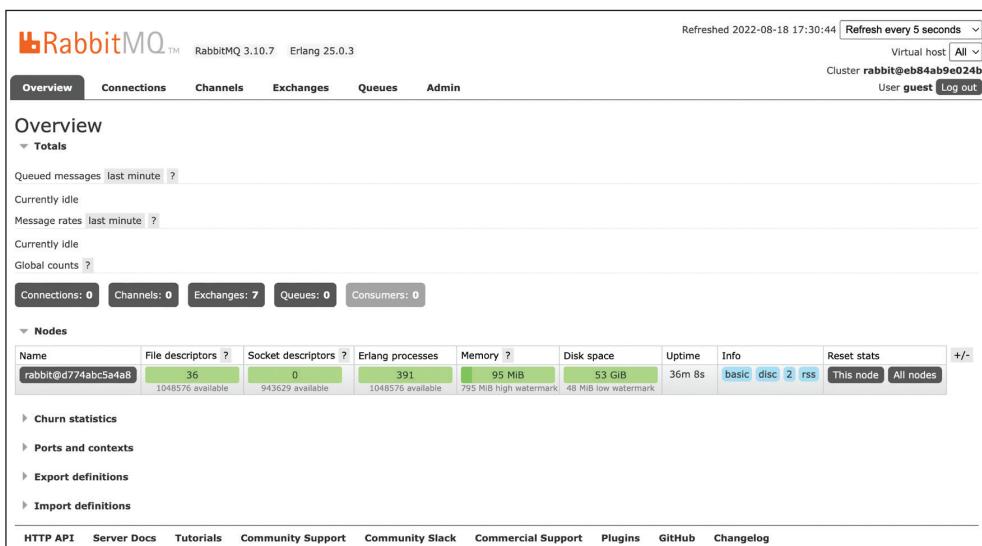


Рис. 8.19. Информационная панель пользовательского интерфейса управления

Это пользователь-администратор, используемый в RabbitMQ по умолчанию. На данном экране можно отслеживать текущую активность брокера сообщений RabbitMQ. Вы видите, что имеется один работающий узел без зарегистрированных соединений или очередей.

Если RabbitMQ работает в рабочей среде, то необходимо создать нового пользователя-администратора и удалить используемого по умолчанию гостевого пользователя. Это можно сделать в разделе **Admin** (Администратор) пользовательского интерфейса управления.

Теперь давайте добавим очередь заданий Celery в проект. Затем ее запустим и проверим соединение с брокером сообщений RabbitMQ.

Добавление очереди заданий Celery в проект

Экземпляру Celery необходимо предоставить конфигурацию. Рядом с файлом `settings.py` проекта `myshop` создайте новый файл и назовите его `celery.py`. Этот файл будет содержать конфигурацию Celery для вашего проекта. Добавьте в него следующий ниже исходный код:

```
import os
from celery import Celery

# задать стандартный модуль настроек Django
# для программы 'celery'.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myshop.settings')
app = Celery('myshop')
app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()
```

В приведенном выше исходном коде делается следующее:

- задается переменная `DJANGO_SETTINGS_MODULE` для встроенной в Celery программы командной строки;
- посредством инструкции `app = Celery('myshop')` создается экземпляр приложения;
- используя метод `config_from_object()`, загружается любая конкретно-прикладная конфигурация из настроек проекта. Атрибут `namespace` задает префикс, который будет в вашем файле `settings.py` у настроек, связанных с Celery. Задав именное пространство `CELERY`, все настройки Celery должны включать в свое имя префикс `CELERY_` (например, `CELERY_BROKER_URL`);
- наконец, сообщается, чтобы очередь заданий Celery автоматически обнаруживала асинхронные задания в ваших приложениях. Celery будет искать файл `tasks.py` в каждом каталоге приложений, добавленных в `INSTALLED_APPS`, чтобы загружать определенные в нем асинхронные задания.

Далее необходимо импортировать модуль `celery` в файл `__init__.py` проекта, чтобы он загружался при запуске Django.

Отредактируйте файл `myshop/__init__.py`, добавив в него следующий ниже исходный код:

```
# импортировать celery
from .celery import app as celery_app

__all__ = ['celery_app']
```

Вы добавили очередь заданий Celery в проект Django и теперь можете начать ее использовать.

Запуск работника Celery

Работник Celery – это процесс, или рабочий узел, который занимается служебными функциями, такими как отправка/получение сообщений очереди, регистрация заданий, уничтожение зависших заданий, отслеживание состояния и т. д. Экземпляр работника может потреблять любое число очередей сообщений.

Откройте еще одну оболочку и запустите работника Celery из каталога проекта, используя следующую ниже команду:

```
celery -A myshop worker -l info
```

Теперь работник Celery запущен и готов к обработке заданий. Давайте проверим наличие соединения между Celery и RabbitMQ.

Пройдите по URL-адресу <http://127.0.0.1:15672/> в своем браузере, чтобы получить доступ к встроенному в RabbitMQ пользовательскому интерфейсу управления. Вы увидите график в разделе **Queued messages** (Сообщения в очереди) и еще один в разделе **Message rates** (Частота сообщений), как на рис. 8.20.

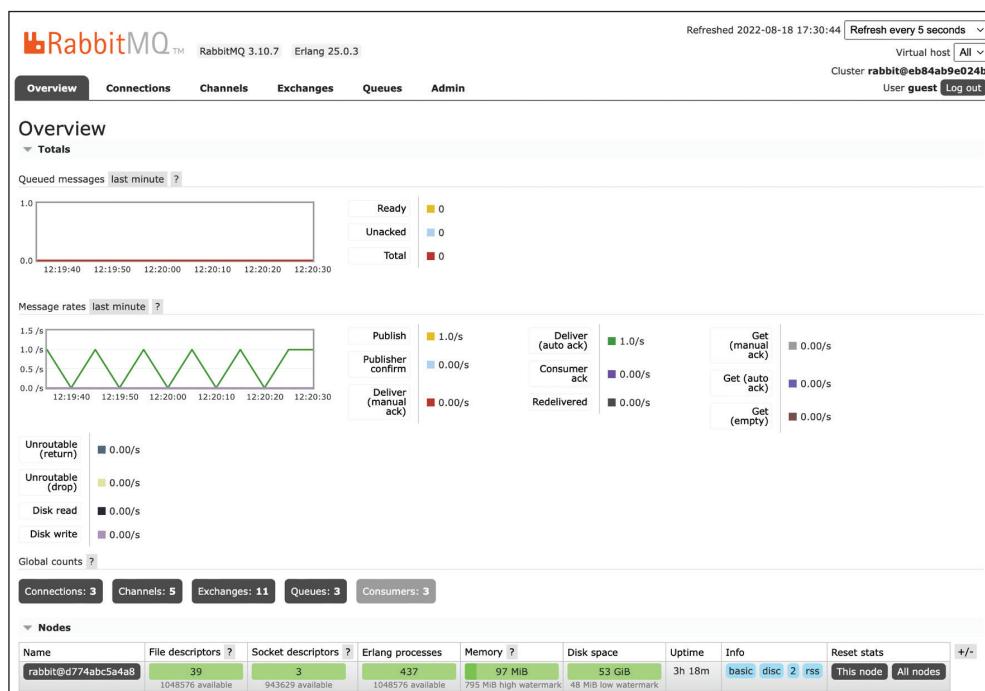


Рис. 8.20. Информационная панель управления RabbitMQ, отображающая соединения и очередь

Очевидно, что сообщений в очереди нет, так как мы еще не отправляли сообщения в очередь сообщений. График в разделе **Message rates** (Частота сообщений) должен обновляться каждые пять секунд; частоту обновления можно увидеть в правом верхнем углу экрана. На этот раз и **Connections** (Соединения), и **Queues** (Очереди) должны отображать число больше нуля.

Теперь можно начать программировать асинхронные задания.



Параметр `CELERY_ALWAYS_EAGER` позволяет исполнять задания локально в синхронном режиме, не отправляя их в очередь. Это удобно при выполнении модульных тестов или исполнении приложения в локальной среде без запуска программы Celery.

Добавление асинхронных заданий в приложение

Давайте будем отправлять пользователю электронное письмо о подтверждении всякий раз, когда заказ размещается в интернет-магазине. Мы реализуем отправку электронной почты в функции Python и зарегистрируем ее в Celery как задание. Затем мы добавим ее в представление `order_create` для асинхронного исполнения заданий.

При исполнении представления `order_create` Celery будет отправлять сообщение в очередь сообщений, управляемую брокером сообщений RabbitMQ, а затем брокер будет исполнять асинхронное задание, которое мы определили в функции Python.

По традиции простое обнаружение заданий Celery заключается в формировании асинхронных заданий для приложения в модуле `tasks` в каталоге приложения.

Создайте новый файл внутри приложения `orders` и назовите его `tasks.py`. Это место, где Celery будет искать асинхронные задания. Добавьте в него следующий ниже исходный код:

```
from celery import shared_task
from django.core.mail import send_mail
from .models import Order

@shared_task
def order_created(order_id):
    """
    Задание по отправке уведомления по электронной почте
    при успешном создании заказа.
    """

    order = Order.objects.get(id=order_id)
    subject = f'Order nr. {order.id}'
    message = f'Dear {order.first_name},\n\n' \
              f'You have successfully placed an order.' \
              f'\nYour order ID is {order.id}.'
    mail_sent = send_mail(subject,
```

```
    message,
    'admin@myshop.com',
    [order.email])
return mail_sent
```

Задание `order_created` было определено с помощью декоратора `@shared_task`. Как видите, задание Celery – это просто функция Python, декорированная функцией-декоратором `@shared_task`. Функция задания `order_created` получает параметр `order_id`. При исполнении задания рекомендуется всегда передавать идентификаторы функциям задания и извлекать объекты из базы данных. Тем самым избегается доступ к устаревшей информации, поскольку данные в базе данных могли измениться за то время, пока задание стояло в очереди. Для отправки уведомления по электронной почте разместившему заказ пользователю была использована предоставляемая веб-фреймворком Django функция `send_mail()`.

Вы научились конфигурировать Django под использование SMTP-сервера в главе 2 «Усовершенствование блога за счет продвинутых функциональностей». Если вы не хотите устанавливать настроочные параметры электронной почты, то можете сообщить Django, что нужно писать электронные письма в консоль, добавив следующий ниже настроочный параметр в файл `settings.py`:

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



Асинхронные задания следует использовать не только для времязатратных процессов, но и для других процессов, выполнение которых не занимает так много времени, но которые подвержены сбоям соединения либо требуют политики повторных попыток соединения.

Теперь необходимо добавить задание в представление `order_create`. Отредактируйте файл `views.py` приложения `orders`, импортировав задание и вызвав асинхронное задание `order_created` после очистки корзины, как показано ниже:

```
from .tasks import order_created
# ...

def order_create(request):
    ...
    if request.method == 'POST':
        ...
        if form.is_valid():
            ...
            cart.clear()
            # запустить асинхронное задание
```

```
order_created.delay(order.id)
# ...
```

Метод `delay()` задания вызывается для его асинхронного исполнения. Задание будет добавлено в очередь сообщений и выполнено работником Celery как можно скорее.

Проверьте, чтобы RabbitMQ был запущен. Затем остановите процесс работника Celery и следующей ниже командой снова его запустите:

```
celery -A myshop worker -l info
```

Теперь работник Celery зарегистрировал задание. В еще одной оболочке следующей ниже командой запустите сервер разработки из каталога проекта:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/` в браузере, добавьте товары в корзину и завершите заказ. В оболочке, в которой вы запустили работника очереди заданий Celery, вы увидите примерно такой результат:

```
[2022-02-03 20:25:19,569: INFO/MainProcess] Task orders.tasks.order_
created[a94dc22e-372b-4339-bff7-52bc83161c5c] received
...
[2022-02-03 20:25:19,605: INFO/ForkPoolWorker-8] Task orders.tasks.
order_created[a94dc22e-372b-4339-bff7-52bc83161c5c] succeeded in
0.015824042027816176s: 1
```

Задание `order_created` было исполнено, и уведомление о заказе отправлено по электронной почте. Если вы используете почтовый бэкенд `console.EmailBackend`, то электронная почта не отправляется, но вы будете видеть электронную почту в консоли.

Отслеживание Celery с помощью инструмента Flower

Помимо встроенного в RabbitMQ пользовательского интерфейса управления, можно использовать другие инструменты мониторинга асинхронных заданий, исполняемых с помощью Celery. Flower представляет собой удобный веб-инструмент для отслеживания работы Celery.

Следующей ниже командой установите Flower:

```
pip install flower==1.1.0
```

После установки мониторингового инструмента Flower можно запустить следующей ниже командой в новой оболочке из каталога проекта:

```
celery -A myshop flower
```

Пройдите по URL-адресу <http://localhost:5555/dashboard> в своем браузере. Вы увидите активных работников Celery и статистику асинхронных заданий. Экран должен выглядеть следующим образом:

The screenshot shows the Flower dashboard interface. At the top, there are tabs for 'Flower' (selected), 'Dashboard', 'Tasks', and 'Broker'. On the right, there are links for 'Docs' and 'Code'. Below the tabs, there are five status counters: Active: 0, Processed: 0, Failed: 0, Succeeded: 0, and Retried: 0. A search bar labeled 'Search:' is present. The main area displays a table of workers. The columns are: Worker Name, Status, Active, Processed, Failed, Succeeded, Retried, and Load Average. One row is shown, corresponding to the worker 'celery@Antonios-MBP.home', which is listed as 'Online'. The 'Active' column shows 0, 'Processed' shows 0, 'Failed' shows 0, 'Succeeded' shows 0, 'Retried' shows 0, and 'Load Average' shows 2.91, 3.11, 3.88. A note at the bottom says 'Showing 1 to 1 of 1 entries'.

Рис. 8.21. Информационная панель Flower

Вы увидите активного работника, имя которого начинается с **celery@** и статус которого будет **Online**.

Кликните по имени работника, а затем перейдите на вкладку **Queues** (Очереди). Вы увидите следующий ниже экран:

The screenshot shows the 'Queues' tab for the worker 'celery@Antonios-MBP.home'. At the top, it says 'Worker: celery@Antonios-MBP.home'. There is a 'Refresh' button. Below the worker name, there are tabs for 'Pool', 'Broker', 'Queues' (which is selected), 'Tasks', 'Limits', 'Config', 'System', and 'Other'. The 'Queues' section has a heading 'Active queues being consumed from'. It shows a table with one row for the queue 'celery'. The columns are: Name, Exclusive, Durable, Routing key, No ACK, Alias, Queue arguments, Binding arguments, and Auto delete. The 'Name' column shows 'celery', 'Exclusive' shows 'False', 'Durable' shows 'True', 'Routing key' shows 'celery', 'No ACK' shows 'False', 'Alias' shows 'None', 'Queue arguments' shows 'None', 'Binding arguments' shows 'None', and 'Auto delete' shows 'False'. To the right of the table is a red button labeled 'Cancel Consumer'.

Рис. 8.22. Flower – Очереди заданий работника Celery

Здесь можно увидеть активную очередь с именем **celery**. Это активный потребитель очереди, соединенный с брокером сообщений.

Откройте вкладку **Tasks** (Задания). Вы увидите следующий ниже экран:

The screenshot shows the Flower dashboard for a Celery worker named **celery@Antonios-MBP.home**. The worker status is listed as **Online**. Below the status, there is a table titled **Processed** showing the number of completed tasks. One task is listed: **orders.tasks.order_created** with a value of **5**.

Processed	number of completed tasks
orders.tasks.order_created	5

Рис. 8.23. Flower – Задания работника Celery

Здесь вы увидите обработанные задания и количество их исполнений. Вы должны увидеть задание `order_created` и общее время его исполнения. Это число может варьироваться в зависимости от того, сколько заказов было размещено.

Пройдите по URL-адресу `http://localhost:8000/` в своем браузере. Добавьте несколько товаров в корзину, а затем завершите процесс оформления заказа.

Пройдите по URL-адресу `http://localhost:5555/dashboard` в браузере. Мониторинговый инструмент Flower зарегистрировал задание как обработанное. Теперь вы должны увидеть 1 в столбце **Processed** (Обработанное) и 1 в столбце **Succeeded** (Успешное):

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@Antonios-MBP.home	Online	0	1	0	1	0	5.51, 4.14, 4.2

Рис. 8.24. Flower – Работники Celery

В разделе **Tasks** (Задания) вы увидите дополнительные сведения о каждом задании, зарегистрированном в Celery:

Flower										Docs	Code	
Dashboard										Tasks	Broker	
Show 10 entries										Search:		
Name	UUID	State	args	kwargs	Result	Received	Started	Runtime	Worker			
<code>orders.tasks.order_created</code>	<code>abb1048f-9a2a-4b59-8b39-8c6b804bebdc</code>	SUCCESS	(10,)	{}	1	2022-02-06 15:54:55.803	2022-02-06 15:54:55.803	0.086	<code>celery@Antonios-MBPhome</code>			

Рис. 8.25. Flower – Задания Celery

Документация по мониторинговому инструменту Flower находится на странице <https://flower.readthedocs.io/>.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter08>.
- Статические файлы проекта: <https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter08/myshop/shop/static>.
- Настройки сеанса Django: <https://docs.djangoproject.com/en/4.1/ref/settings/#sessions>.
- Встроенные в Django процессоры контекста: <https://docs.djangoproject.com/en/4.1/ref/templates/api/#built-in-template-context-processors>.
- Информация о RequestContext: <https://docs.djangoproject.com/en/4.1/ref/templates/api/#django.template.RequestContext>.
- Документация Celery: <https://docs.celeryq.dev/en/stable/index.html>.
- Введение в очередь заданий Celery: <https://docs.celeryq.dev/en/stable/getting-started/introduction.html>.
- Официальный образ брокера сообщений RabbitMQ платформы Docker: https://hub.docker.com/_/rabbitmq.
- Инструкции по установке брокера сообщений RabbitMQ: <https://www.rabbitmq.com/download.html>.
- Документация по мониторинговому инструменту Flower: <https://flower.readthedocs.io/>.

Резюме

В этой главе вы создали базовое приложение электронной коммерции. Вы разработали каталог товаров и сформировали корзину на основе сессий. Вы имплементировали конкретно-прикладной процессор контекста, чтобы сделать корзину доступной для всех шаблонов, и создали форму для размещения заказов. Вы также научились реализовывать асинхронные задания с помощью очереди заданий Celery и брокера сообщений RabbitMQ.

В следующей главе вы научитесь интегрировать платежный шлюз в свой магазин, добавлять конкретно-прикладные действия на сайт администрирования, экспортить данные в формате CSV и динамически создавать PDF-файлы.

9

Управление платежами и заказами

В предыдущей главе вы создали базовый интернет-магазин с каталогом товаров и корзиной покупок. Вы научились использовать сеансы Django и создали конкретно-прикладной процессор контекста. Вы также научились запускать асинхронные задания с помощью очереди заданий Celery и брокера сообщений RabbitMQ.

В этой главе вы научитесь интегрировать платежный шлюз в свой сайт, чтобы пользователи имели возможность оплачивать кредитной картой. Вы также расширите сайт администрирования различными функциональными возможностями.

В этой главе вы будете:

- интегрировать платежный шлюз Stripe в свой проект;
- обрабатывать платежи по кредитным картам с помощью Stripe;
- оперировать платежными уведомлениями;
- экспортить заказы в файлы CSV;
- создавать конкретно-прикладные представления сайта администрирования;
- динамически генерировать счета в формате PDF.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter09>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Интеграция платежного шлюза

Платежный шлюз – это технология, используемая продавцами для обработки платежей от интернет-клиентов. Используя платежный шлюз, можно управ-

лять заказами клиентов и делегировать обработку платежей надежной и безопасной третьей стороне. Используя доверяемый платежный шлюз, не придется беспокоиться о технической сложности, безопасности и нормативных сложностях обработки кредитных карт в своей собственной системе.

Существует несколько поставщиков платежных шлюзов на выбор. Мы займемся интеграцией платежного шлюза очень популярного обработчика платежей Stripe, который среди прочих используется такими онлайновыми сервисами, как Shopify, Uber, Twitch и GitHub.

Платформа Stripe предоставляет интерфейс прикладного программирования (**API¹**), который позволяет обрабатывать онлайновые платежи с использованием нескольких методов платежа, таких как кредитная карта, Google Pay и Apple Pay. Подробнее о Stripe можно узнать на странице <https://www.stripe.com/>.

Платформа Stripe предоставляет различные цифровые продукты, связанные с обработкой платежей. Она может управлять разовыми платежами, регулярными платежами за подписные услуги, многосторонними платежами для платформ и торговых площадок и многим другим.

Обработчик платежей Stripe предлагает различные методы интеграции, от размещенных в Stripe платежных форм до полностью адаптируемых под конкретно-прикладную процедуру оформления платежа. Мы встроим платежный инструмент Stripe Checkout (Оформление платежей через Stripe), который состоит из оптимизированной под конверсию платежной страницы. Пользователи смогут легко оплачивать заказываемые ими товары с помощью кредитной карты либо других методов платежа. Мы будем получать уведомления о платеже от Stripe. С документацией по платежному инструменту Stripe Checkout можно ознакомиться на странице <https://stripe.com/docs/payments/checkout>.

Задействуя платежный инструмент Stripe Checkout для обработки платежей, вы опираетесь на безопасное решение, соответствующее требованиям индустрии платежных карт (**PCI²**). Вы сможете получать платежи из Google Pay, Apple Pay, Afterpay, Alipay, автоплатежей³ SEPA, автоплатежей Bacs, автоплатежей BECS, iDEAL, Sofort, GrabPay, FPX и других методов платежа.

Создание учетной записи Stripe

Для интеграции платежного шлюза на сайт понадобится учетная запись Stripe. Давайте создадим учетную запись, чтобы протестировать API Stripe. Пройдите по URL-адресу <https://dashboard.stripe.com/register> в своем браузере. Вы увидите примерно такую форму:

¹ Англ. Application Programming Interface. – *Прим. перев.*

² Англ. Payment Card Industry. – *Прим. перев.*

³ Англ. direct debit (прямой дебет). Это договоренность с банком, которая позволяет третьей стороне снимать деньги со счета человека в согласованные даты, обычно для оплаты счетов. – *Прим. перев.*

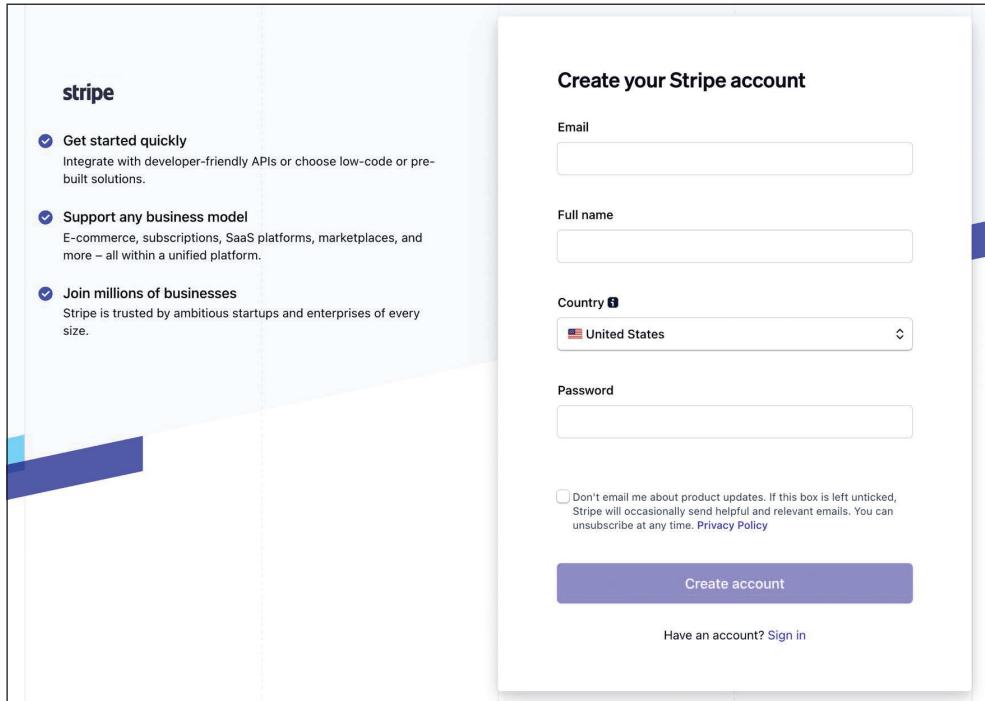


Рис. 9.1. Форма для регистрации в Stripe

Заполните форму своими данными и кликните по **Create account** (Создать учетную запись). Вы получите электронное письмо от Stripe со ссылкой для подтверждения вашего адреса электронной почты. Электронное письмо будет выглядеть так:

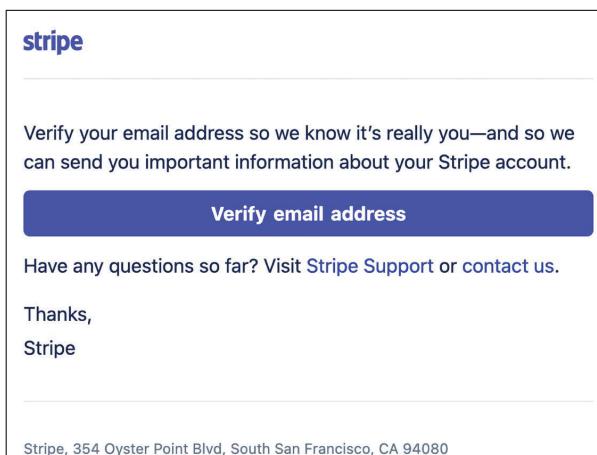


Рис. 9.2. Верификационное электронное письмо для подтверждения вашего адреса электронной почты

Откройте письмо в папке входящих сообщений и кликните по **Verify email address** (Подтвердить адрес электронной почты).

Вы будете перенаправлены на экран информационной панели Stripe, который будет выглядеть следующим образом:

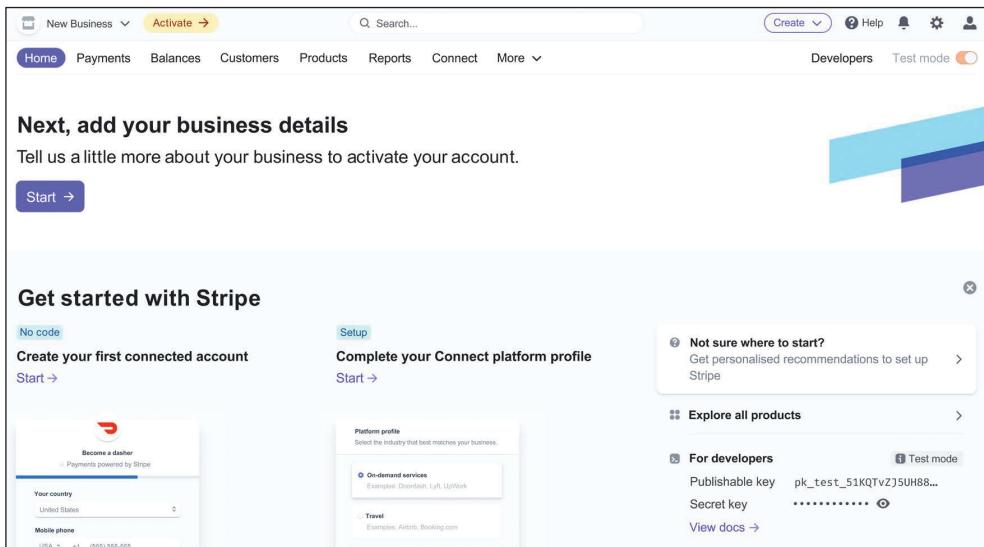


Рис. 9.3. Информационная панель Stripe
после верификации адреса электронной почты

В правом верхнем углу экрана вы увидите, что активирован тестовый режим (**Test mode**). Stripe предоставляет тестовую и производственную среды. Если вы владеете бизнесом или являетесь фрилансером, то можете добавить данные о своем бизнесе, чтобы активировать учетную запись и получить доступ к обработке реальных платежей. Однако для имплементации и тестирования платежей через Stripe это не обязательно, так как мы будем работать в тестовой среде.

Для обработки платежей необходимо добавить имя учетной записи. Пройдите по URL-адресу <https://dashboard.stripe.com/settings/account> в своем браузере. Вы увидите следующий ниже экран:



Рис. 9.4. Настройки учетной записи Stripe

В разделе **Account name** (Имя учетной записи) введите имя по вашему выбору и кликните по **Save** (Сохранить). Вернитесь на информационную панель Stripe. Вы увидите, что в заголовке будет отображаться имя вашей учетной записи:



Рис. 9.5. Заголовок информационной панели Stripe, включающий имя учетной записи

Мы продолжим, установив SDK Stripe для Python и добавив Stripe в проект Django.

Установка библиотеки Stripe

Обработчик платежей Stripe предоставляет библиотеку Python, которая упрощает работу с API. Мы собираемся интегрировать платежный шлюз в проект с помощью библиотеки `stripe`.

Исходный код библиотеки Python `stripe` можно найти на странице <https://github.com/stripe/stripe-python>.

С помощью следующей команды установите библиотеку `stripe` из оболочки:

```
pip install stripe==4.0.2
```

Добавление Stripe в проект

Пройдите по URL-адресу <https://dashboard.stripe.com/test/apikeys> в своем браузере. К этой странице также можно обратиться из информационной панели Stripe, кликнув по **Developers** (Разработчики), а затем кликнув по **API keys** (Ключи API). Вы увидите следующий ниже экран:

The screenshot shows the 'API keys' section of the Stripe dashboard. It has a header with 'API keys' and a link to 'Learn more about API authentication'. Below the header, there's a note: 'Viewing test API keys. Toggle to view live keys.' and a button to 'Viewing test data'. The main section is titled 'Standard keys' with the sub-note: 'These keys will allow you to authenticate API requests. [Learn more](#)'. It lists two keys: 'Publishable key' and 'Secret key'. The 'Publishable key' row contains the token: pk_test_51KQTVzJ5UH88gi9TqRLwQzR0gZopjf7as5Dxwk129qR2hB4KLCh6sgP6bDkeT2oCUq0WJZm0Cfe xgfzkgHDts1Jv00XqYm42PK'. The 'Secret key' row contains a placeholder token and a 'Reveal test key' button. The table has columns: NAME, TOKEN, LAST USED, and CREATED.

NAME	TOKEN	LAST USED	CREATED	...
Publishable key	pk_test_51KQTVzJ5UH88gi9TqRLwQzR0gZopjf7as5Dxwk129qR2hB4KLCh6sgP6bDkeT2oCUq0WJZm0Cfe xgfzkgHDts1Jv00XqYm42PK	-	7 Feb	...
Secret key	[REDACTED] Reveal test key	-	7 Feb	...

Рис. 9.6. Экран тестовых ключей API Stripe

Платформа Stripe предоставляет пару ключей для двух разных сред: тестовой и производственной. Для каждой среды существует **Publishable key** (Публикуемый ключ) и **Secret key** (Секретный ключ). Публикуемые ключи тестового режима имеют префикс `pk_test_`, а публикуемые ключи эфирного режима имеют префикс `pk_live_`. Секретные ключи тестового режима имеют префикс `sk_test_`, а секретные ключи эфирного режима имеют префикс `sk_live_`.

Эта информация понадобится для аутентификации запросов к API Stripe. Свой закрытый ключ нужно всегда держать в секрете, и хранить его в надежном месте. Публикуемый ключ можно использовать в исходном коде на стороне клиента, например в скриптах JavaScript. Подробнее о ключах API Stripe можно узнать на странице <https://stripe.com/docs/keys>.

Добавьте следующие ниже настроочные параметры в файл `settings.py` проекта:

```
# Настроочные параметры Stripe
STRIPE_PUBLISHABLE_KEY = '' # Публикуемый ключ
STRIPE_SECRET_KEY = ''      # Секретный ключ
STRIPE_API_VERSION = '2022-08-01'
```

Замените значения `STRIPE_PUBLISHABLE_KEY` и `STRIPE_SECRET_KEY` предоставленным Stripe тестовым публикуемым ключом и секретным ключом. Вы будете использовать API Stripe версии 2022-08-01. Примечания к релизу

этой версии API можно посмотреть на странице <https://stripe.com/docs/upgrades#2022-08-01>.



В проекте используются ключи тестовой среды. При выходе в прямой эфир и валидации своей учетной записи Stripe вы получите ключи производственной среды. В главе 17 «Выход в прямой эфир» вы научитесь конфигурировать параметры под несколько сред.

Давайте интегрируем платежный шлюз в процесс оформления платежа. Документация Python по Stripe находится на странице <https://stripe.com/docs/api?lang=python>.

Формирование процесса платежа

Процесс оформления платежа будет работать следующим образом:

- 1) добавить товары в корзину;
- 2) оформить заказ;
- 3) ввести данные кредитной карты и оплатить.

Мы собираемся создать новое приложение, которое будет служить для управления платежами. Создайте новое приложение в проекте, используя следующую ниже команду:

```
python manage.py startapp payment
```

Отредактируйте файл `settings.py` проекта, добавив новое приложение в настроочный параметр `INSTALLED_APPS`, как показано ниже. Новая строка выделена жирным шрифтом:

```
INSTALLED_APPS = [  
    # ...  
    'shop.apps.ShopConfig',  
    'cart.apps.CartConfig',  
    'orders.apps.OrdersConfig',  
    'payment.apps.PaymentConfig',  
]
```

Теперь приложение `payment` в проекте является активным.

В настоящее время пользователи могут размещать заказы, но не могут их оплачивать. После того как клиенты будут размещать заказ, необходимо их перенаправлять к процессу платежа.

Отредактируйте файл `views.py` приложения `orders`, вставив следующие ниже инструкции импорта:

```
from django.urls import reverse
from django.shortcuts import render, redirect
```

В этом же файле найдите следующие ниже строки представления `order_create`:

```
# запустить асинхронное задание
order_created.delay(order.id)
return render(request,
              'orders/order/created.html',
              locals())
```

Замените их таким исходным кодом:

```
# запустить асинхронное задание
order_created.delay(order.id)
# задать заказ в сеансе
request.session['order_id'] = order.id
# перенаправить к платежу
return redirect(reverse('payment:process'))
```

Отредактированное представление должно выглядеть следующим образом:

```
from django.urls import reverse
from django.shortcuts import render, redirect
# ...

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'])
            # очистить корзину
            cart.clear()
            # запустить асинхронное задание
            order_created.delay(order.id)
            # задать заказ в сеансе
```

```

request.session['order_id'] = order.id
# перенаправить к платежу
return redirect(reverse('payment:process'))
else:
    form = OrderCreateForm()
return render(request,
              'orders/order/create.html',
              {'cart': cart, 'form': form})

```

При размещении нового заказа вместо прорисовки шаблона `orders/order/created.html` ИД заказа сохраняется в сеансе пользователя, и пользователь перенаправляется на URL-адрес `payment:process`. Чуть позже мы реализуем этот URL-адрес. Напомним, что для того, чтобы задание `order_created` было поставлено в очередь и исполнено, программа Celery должна работать.

Давайте интегрируем платежный шлюз.

Интеграция платежного инструмента *Stripe Checkout*

Платежный инструмент Stripe Checkout состоит из размещенной в Stripe страницы оформления платежа, которая дает пользователю возможность вводить платежные данные, обычно кредитную карту, и взимает платеж. Если платеж проходит успешно, то Stripe перенаправляет клиента на страницу успеха. Если платеж отменяется клиентом, то клиент перенаправляется на страницу отмены.

Мы реализуем три представления:

- `payment_process`: создает сеанс оформления платежа, **Checkout Session**, и перенаправляет клиента к размещеннной на Stripe платежной форме. Сеанс оформления платежа – это программное представление того, что видит клиент, когда его перенаправляют к платежной форме, включая товары, количество, валюту и сумму платежа;
- `payment_completed`: отображает сообщение об успешных платежах. Пользователь перенаправляется к этому представлению, если платеж прошел успешно;
- `payment_canceled`: отображает сообщение об отмененных платежах. Пользователь перенаправляется к этому представлению, если платеж был отменен.

На рис. 9.7 показана процедура оформления платежа.

Полный процесс оформления платежа будет выглядеть следующим образом.

1. После создания заказа пользователь перенаправляется к представлению `payment_process`. Пользователю предварительно отправляется сводная информация о заказе и кнопка для перехода к платежу.
2. Когда пользователь переходит к оплате, создается сеанс оформления платежа Stripe. Сеанс оформления платежа включает в себя список приобретаемых пользователем товаров, URL-адрес перенаправления пользователя в случае успешного платежа и URL-адрес перенаправления пользователя в случае отмены платежа.

3. Представление перенаправляет пользователя к размещённой на Stripe странице оформления платежа. На этой странице есть платежная форма. Клиент вводит данные своей кредитной карты и передает форму на обработку.
4. Stripe обрабатывает платеж и перенаправляет клиента к представлению `payment_completed`. Если клиент не завершает платеж, то вместо этого Stripe перенаправляет клиента к представлению `payment_canceled`.



Рис. 9.7. Процедура оформления платежа

Давайте начнем разрабатывать представления платежа. Отредактируйте файл `views.py` приложения `payment`, добавив в него следующий ниже исходный код:

```
from decimal import Decimal
import stripe
from django.conf import settings
from django.shortcuts import render, redirect, reverse,\n    get_object_or_404
from orders.models import Order\n\n# создать экземпляр Stripe
stripe.api_key = settings.STRIPE_SECRET_KEY
stripe.api_version = settings.STRIPE_API_VERSION\n\ndef payment_process(request):
    order_id = request.session.get('order_id', None)
    order = get_object_or_404(Order, id=order_id)
```

```

if request.method == 'POST':
    success_url = request.build_absolute_uri(
        reverse('payment:completed'))
    cancel_url = request.build_absolute_uri(
        reverse('payment:canceled'))
    # данные сеанса оформления платежа Stripe
    session_data = {
        'mode': 'payment',
        'client_reference_id': order.id,
        'success_url': success_url,
        'cancel_url': cancel_url,
        'line_items': []
    }
    # создать сеанс оформления платежа Stripe
    session = stripe.checkout.Session.create(**session_data)
    # перенаправить к платежной форме Stripe
    return redirect(session.url, code=303)
else:
    return render(request, 'payment/process.html', locals())

```

В приведенном выше исходном коде импортируется модуль `stripe` и с помощью значения настроичного параметра `STRIPE_SECRET_KEY` задается ключ API Stripe. Кроме того, с помощью значения настроичного параметра `STRIPE_API_VERSION` задается используемая версия API.

Представление `payment_process` выполняет следующую работу.

1. Текущий объект `Order` извлекается по сеансовому ключу `order_id`, который ранее был сохранен в сеансе представлением `order_create`.
2. Объект `Order` извлекается из базы данных по данному `order_id`. Если при использовании функции сокращенного доступа `get_object_ or_404()` возникает исключение `Http404` (страница не найдена), то заказ с заданным ИД не найден.
3. Если представление загружается с помощью запроса методом `GET`, то прорисовывается и возвращается шаблон `payment/process.html`. Этот шаблон будет содержать сводную информацию о заказе и кнопку для перехода к платежу, которая будет генерировать запрос методом `POST` к представлению.
4. Если представление загружается с помощью запроса методом `POST`, то сеанс Stripe оформления платежа создается с использованием `Stripe.checkout.Session.create()` со следующими ниже параметрами:
 - `mode`: режим сеанса оформления платежа. Здесь используется значение `payment`, указывающее на разовый платеж. На странице https://stripe.com/docs/api/checkout/sessions/object#checkout_session_object-mode можно увидеть другие принятые для этого параметра значения;
 - `client_reference_id`: уникальная ссылка для этого платежа. Она будет использоваться для согласования сеанса оформления платежа Stripe с заказом. Передавая ИД заказа, платежи Stripe связываются с зака-

- зами в вашей системе, и вы сможете получать уведомления от Stripe о платежах, чтобы помечать заказы как оплаченные;
- `success_url`: URL-адрес, на который Stripe перенаправляет пользователя в случае успешного платежа. Здесь используется `request.build_absolute_uri()`, чтобы формировать абсолютный URI-идентификатор из пути URL-адреса. Документация по этому методу находится по адресу https://docs.djangoproject.com/en/4.1/ref/request-response/#django.http.HttpRequest.build_absolute_uri;
 - `cancel_url`: URL-адрес, на который Stripe перенаправляет пользователя в случае отмены платежа;
 - `line_items`: это пустой список. Далее он будет заполнен приобретаемыми товарными позициями заказа.
5. После создания сеанса оформления платежа возвращается HTTP-перенаправление с кодом состояния, равным 303, чтобы перенаправить пользователя к Stripe. Код состояния 303 рекомендуется для перенаправления веб-приложений на новый URI-идентификатор после выполнения HTTP-запроса методом POST.

Все параметры создания объекта сеанса Stripe можно увидеть на странице <https://stripe.com/docs/api/checkout/sessions/create>.

Давайте заполним список `line_items` товарными позициями заказа, чтобы создать сеанс оформления платежа. Каждая товарная позиция будет содержать название товара, сумму к оплате, используемую валюту и приобретаемое количество.

Добавьте следующий ниже исходный код, выделенный жирным шрифтом, в представление `payment_process`:

```
def payment_process(request):  
    order_id = request.session.get('order_id', None)  
    order = get_object_or_404(Order, id=order_id)  
    if request.method == 'POST':  
        success_url = request.build_absolute_uri(  
            reverse('payment:completed'))  
        cancel_url = request.build_absolute_uri(  
            reverse('payment:canceled'))  
        # данные сеанса Stripe оформления платежа  
        session_data = {  
            'mode': 'payment',  
            'success_url': success_url,  
            'cancel_url': cancel_url,  
            'line_items': []  
        }  
        # добавить товарные позиции заказа  
        # в сеанс оформления платежа Stripe  
        for item in order.items.all():  
            session_data['line_items'].append({  
                'price_data': {  
                    'unit_amount': int(item.price * Decimal('100')),  
                    'currency': 'USD'  
                },  
                'quantity': item.quantity  
            })  
    return HttpResponseRedirect(success_url)
```

```

    'currency': 'usd',
    'product_data': [
        {
            'name': item.product.name,
        },
    ],
    'quantity': item.quantity,
}
# создать сеанс оформления платежа Stripe
session = stripe.checkout.Session.create(**session_data)
# перенаправит к платежной форме Stripe
return redirect(session.url, code=303)
else:
    return render(request, 'payment/process.html', locals())

```

По каждой товарной позиции используется следующая информация:

- `price_data`: информация, связанная с ценой;
- `unit_amount`: сумма в центах, которую необходимо получить при оплате. Это положительное целое число, показывающее, сколько взимать в наименьшей денежной единице без десятичных знаков. Например, 10 долларов будет равно 1000 (то есть 1000 центам). Цена товара, `item.price`, умножается на `Decimal('100')`, чтобы получить значение в центах, а затем конвертируется в целое число;
- `currency`: используемая валюта в трехбуквенном формате ISO. Значение `usd` используется для долларов США. Список поддерживаемых валют можно увидеть на странице <https://stripe.com/docs/currencies>;
- `product_data`: информация, связанная с товаром:
- `name`: название товара;
- `quantity`: число приобретаемых единиц товара.

Теперь представление `payment_process` готово. Давайте создадим простые представления для страниц успешного и отмененного платежей.

Добавьте следующий ниже исходный код в файл `views.py` приложения `payment`:

```

def payment_completed(request):
    return render(request, 'payment/completed.html')

def payment_canceled(request):
    return render(request, 'payment/canceled.html')

```

Внутри каталога приложения `payment` создайте новый файл и назовите его `urls.py`. Добавьте в него следующий ниже исходный код:

```

from django.urls import path
from . import views

app_name = 'payment'

```

```
urlpatterns = [
    path('process/', views.payment_process, name='process'),
    path('completed/', views.payment_completed, name='completed'),
    path('canceled/', views.payment_canceled, name='canceled'),
]
```

Это URL-адреса рабочего процесса платежа. Мы включили следующие шаблоны URL-адресов:

- `process`: представление, которое отображает пользователю сводную информацию о заказе, создает сеанс оформления заказа в Stripe и перенаправляет пользователя к размещенной в Stripe платежной форме;
- `completed`: представление, к которому Stripe перенаправляет пользователя в случае успешного прохождения платежа;
- `canceled`: представление, к которому Stripe перенаправляет пользователя в случае отмены платежа.

Отредактируйте основной файл `urls.py` проекта `myshop` и вставьте шаблоны URL-адресов для приложения `payment`, как показано ниже:

```
urlpatterns = [
    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('', include('shop.urls', namespace='shop')),
]
```

Новый путь был помещен перед шаблоном `shop.urls`, чтобы избежать не-преднамеренного совпадения шаблона с шаблоном, определенным в `shop.urls`. Напомним, что Django просматривает каждый шаблон URL-адреса по порядку и останавливается на первом, который совпадает с запрошенным URL-адресом.

Давайте создадим шаблон для каждого представления. Внутри каталога приложения `payment` создайте следующую ниже файловую структуру:

```
templates/
    payment/
        process.html
        completed.html
        canceled.html
```

Отредактируйте шаблон `payment/process.html`, добавив в него следующий ниже исходный код:

```
{% extends "shop/base.html" %}
{% load static %}
```

```

{% block title %}Pay your order{% endblock %}

{% block content %}
<h1>Order summary</h1>
<table class="cart">
  <thead>
    <tr>
      <th>Image</th>
      <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>
        
      </td>
      <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="4">Total</td>
      <td class="num">${{ order.get_total_cost }}</td>
    </tr>
  </tbody>
</table>
<form action="{% url "payment:process" %}" method="post">
  <input type="submit" value="Pay now">
  {% csrf_token %}
</form>
{% endblock %}

```

Это шаблон отображения сводной информации о заказе пользователю и предоставления возможности клиенту перейти к платежу. Он содержит форму и кнопку **Pay now** (Оплатить сейчас), чтобы передать ее методом POST на обработку. После передачи формы представление `payment_process` создает сеанс оформления платежа Stripe и перенаправляет пользователя к размещенному в Stripe платежной форме.

Отредактируйте шаблон `payment/completed.html`, добавив в него следующий ниже исходный код:

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

{% block title %}Payment successful{% endblock %}

{% block content %}
<h1>Your payment was successful</h1>
<p>Your payment has been processed successfully.</p>
{% endblock %}
```

Это шаблон страницы, на которую пользователь перенаправляется при успешном прохождении платежа.

Отредактируйте шаблон `payment/canceled.html`, добавив в него следующий ниже исходный код:

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

{% block title %}Payment canceled{% endblock %}

{% block content %}
<h1>Your payment has not been processed</h1>
<p>There was a problem processing your payment.</p>
{% endblock %}
```

Это шаблон страницы, на которую пользователь перенаправляется в случае отмены платежа.

Мы реализовали необходимые представления обработки платежей, включая их шаблоны URL-адресов и шаблоны прорисовки. Теперь самое время испытать процесс оформления платежа.

Тестирование процесса оформления заказа

Выполните следующую ниже команду в оболочке, чтобы запустить сервер RabbitMQ в Docker:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672
rabbitmq:management
```

Она запустит RabbitMQ на порту 5672 и веб-интерфейс управления на порту 15672.

Откройте еще одну оболочку и следующей ниже командой запустите разработчика Celery из каталога проекта:

```
celery -A myshop worker -l info
```

Откройте еще одну оболочку и следующей ниже командой запустите сервер разработки из каталога проекта:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/> в своем браузере, добавьте товары в корзину и заполните форму оформления заказа. Кликните по кнопке **Place order** (Разместить заказ). Заказ будет сохранен в базе данных, ИД заказа будет сохранен в текущем сеансе, и вы будете перенаправлены на страницу процесса платежа.

Страница процесса платежа будет выглядеть следующим образом:

The screenshot shows a payment process page for an order. At the top, it says "My shop". Below that, "Order summary" is displayed. The table shows two items: Green tea and Red tea. The total is \$121.00. A "Pay now" button is at the bottom.

Image	Product	Price	Quantity	Total
	Green tea	\$30.00	1	\$30.00
	Red tea	\$45.50	2	\$91.00
Total				\$121.00

Pay now

Рис. 9.8. Страница процесса платежа, включающая сводную информацию о заказе



На этой странице находится сводная информация о заказе и кнопка **Pay now** (Оплатить сейчас). Кликните по кнопке **Pay now**. Представление `payment_process` создаст сеанс оформления платежа Stripe, и вы будете перенаправлены к размещенной в Stripe платежной форме. Вы увидите следующую ниже страницу:

The screenshot shows a mobile-style payment interface for a tea shop. At the top left is a back arrow and the text 'Tea shop TEST MODE'. At the top right is a 'Google Pay' logo. Below this, the text 'Pay Tea shop' and 'US\$121.00' is displayed. To the left, there's a breakdown of items: 'Green tea' (Qty 1, US\$30.00) and 'Red tea' (Qty 2, US\$45.50 each). On the right, there's a section for 'Card information' with fields for 'Email' (placeholder 'Email'), 'Card number' (placeholder '1234 1234 1234 1234') with logos for Visa, Mastercard, American Express, and Discover, 'MM / YY' (placeholder 'MM / YY'), 'CVC' (placeholder 'CVC'), 'Name on card' (placeholder 'Name on card'), 'Country or region' (dropdown menu set to 'United States'), and 'ZIP' (placeholder 'ZIP'). A checkbox labeled 'Save my info for secure 1-click checkout' is checked, with the subtext 'Pay faster on Tea shop and thousands of sites.' Below the form is a large blue 'Pay' button.

Рис. 9.9. Платежная форма для оформления платежа Stripe

Использование тестовых кредитных карт

Платформа Stripe предоставляет различные тестовые кредитные карты от разных эмитентов карт и стран, что позволяет имитировать платежи с целью проверки всех возможных сценариев (успешный платеж, отклоненный платеж и т. д.). В следующей ниже таблице показано несколько карт, которые можно протестировать на предмет разных сценариев:

Результат	Тестовая кредитная карта	CVC	Дата истечения срока действия
Успешный платеж	4242 4242 4242 4242	Любые 3 цифры	Любая дата в будущем
Неуспешный платеж	4000 0000 0000 0002	Любые 3 цифры	Любая дата в будущем
Требуется аутентификация 3D Secure	4000 0025 0000 3155	Любые 3 цифры	Любая дата в будущем

Полный список предназначенных для тестирования кредитных карт находится на странице <https://stripe.com/docs/testing>.

Мы будем использовать тестовую карту 4242 4242 4242 4242, то есть карту Visa, которая возвращает успешную покупку. Мы будем использовать CVC 123 и любую будущую дату истечения срока действия, например 12/29. Введите данные кредитной карты в платежную форму, как показано ниже:

← Tea shop TEST MODE

Pay Tea shop

US\$121.00

Green tea	US\$30.00
Qty 1	
Red tea	US\$91.00
Qty 2	US\$45.50 each

G Pay

Or pay with card

Email
antonio.mele@zenxit.com

Card information
4242 4242 4242 4242 VISA
12 / 29 123

Name on card
Antonio Melé

Country or region
United States ▾
10001

Save my info for secure 1-click checkout
Pay faster on Tea shop and thousands of sites.

Pay

Рис. 9.10. Платежная форма с данными допустимой тестовой кредитной карты

Кликните по кнопке **Pay** (Оплатить). Текст кнопки изменится на **Processing...** (Идет обработка...), как показано на рис. 9.11:

← Tea shop TEST MODE

Pay Tea shop

US\$121.00

Green tea	US\$30.00
Qty 1	
Red tea	US\$91.00
Qty 2	US\$45.50 each

G Pay

Or pay with card

Email
antonio.mele@zenxit.com

Card information
4242 4242 4242 4242 VISA
12 / 29 123

Name on card
Antonio Melé

Country or region
United States
10001

Save my info for secure 1-click checkout
Pay faster on Tea shop and thousands of sites.

Processing... ⏺

Рис. 9.11. Платежная форма в процессе обработки

Через пару секунд вы увидите, что кнопка станет зеленой, как на рис. 9.12:

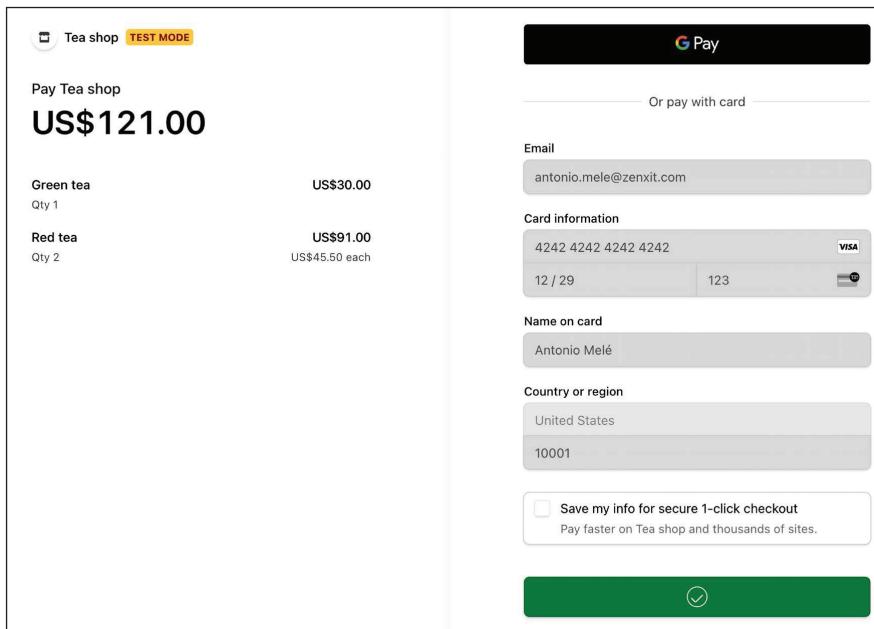


Рис. 9.12. Платежная форма после успешного платежа

Затем Stripe перенаправляет ваш браузер на URL-адрес завершения платежа, который вы указали при создании сеанса оформления платежа. Вы увидите следующую ниже страницу:



Рис. 9.13. Страница успешного платежа

Проверка платежной информации в информационной панели Stripe

Обратитесь к информационной панели Stripe на странице <https://dashboard.stripe.com/test/payments>. В разделе **Payments** (Платежи) вы увидите платеж, как показано на рис. 9.14:



The screenshot shows the Stripe Payments interface. At the top, there's a header with the title 'Payments' and several buttons: 'Filter' (with a count of 1), 'Export', '+ Create payment', and a search bar. Below the header, there's a navigation bar with links: 'All' (underlined), 'Succeeded', 'Refunded', 'Uncaptured', and 'Failed'. A table follows, with columns: 'AMOUNT', 'DESCRIPTION', 'CUSTOMER', and 'DATE'. One row is visible: 'US\$121.00' (Status: Succeeded ✓), 'pi_3Kgp1HJ5UH88gj9T0e3c8Nhy', 'antonio.mele@zenxit.com', '24 Mar, 08:55', and three dots. At the bottom left, it says '1 result'. On the right, there are 'Previous' and 'Next' buttons.

Рис. 9.14. Объект платежа со статусом *Succeeded* на информационной панели Stripe

Платеж имеет статус **Succeeded** (Успешный). Описание платежа содержит ИД платежного намерения, начинающийся с `pi_`. После того как сеанс оформления платежа подтвержден, Stripe создает платежное намерение, связанное с сеансом. Платежное намерение используется для взимания платежа с пользователя. Stripe регистрирует все попытки платежа как платежные намерения. Каждое платежное намерение имеет уникальный ИД и содержит в себе детальную информацию о транзакции, такую как поддерживаемые методы платежа, взимаемая сумма и выбранная валюта. Кликните по транзакции, чтобы перейти к детальной информации о платеже.

Вы увидите экран, показанный на рис. 9.15.

Здесь находится платежная информация и хронология платежа (**Timeline**), включая изменения платежа. В разделе **Checkout summary** (Сводка по оформлению платежа) расположены приобретенные товарные позиции, включая название, количество, цену за единицу и сумму. В разделе **Payment details** (Детальная информация о платеже) находится разбивка уплаченной суммы и комиссии Stripe за обработку платежа.

В этом разделе находится раздел **Payment method** (Метод платежа), содержащий сведения о методе платежа и проверках кредитной карты, выполненных платежным шлюзом Stripe, как на рис. 9.16.

PAYMENT pi_3KgplHJ5UH88gi9T0e3c8Nhy

US\$121.00 USD Succeeded ✓ Refund...

Date 24 Mar, 08:27	Customer Antonio Melé Guest	Payment method 4242	Risk evaluation 55 Normal
-----------------------	--------------------------------	-----------------------------	------------------------------

Timeline + Add note

- Payment succeeded
24 Mar 2022, 08:55
- Payment started
24 Mar 2022, 08:27

Checkout summary

Customer	antonio.mele@zenxit.com		
	Antonio Melé		
	10001 US		
ITEMS	QTY	UNIT PRICE	AMOUNT
Green tea	1	US\$30.00	US\$30.00
Red tea	2	US\$45.50	US\$91.00
	Total		US\$121.00

Payment details

Statement descriptor	Stripe
Amount	US\$121.00
Fee	US\$3.81
Net	US\$117.19
Status	Succeeded
Description	No description

Рис. 9.15. Детальная информация о платеже для транзакции Stripe

Payment method			
ID	pm_1KgqCeJ5UH88gi9TG6fuyETL	Owner	Antonio Melé
Number 4242	Owner email	antonio.mele@zenxit.com
Fingerprint	Ms4UOyABpHZLkN3s	Address	10001, US
Expires	12 / 2029	Origin	United States
Type	Visa credit card	CVC check	Passed
Issuer	Stripe Payments UK Limited	Zip check	Passed

Рис. 9.16. Метод платежа, используемый в транзакции Stripe

В этом разделе вы найдете еще один раздел под названием **Events and logs** (События и журналы), как на рис. 9.17:

Events and logs	
LATEST ACTIVITY	<p>PaymentIntent status: succeeded</p>
ALL ACTIVITY	<p>From Stripe checkout.session.completed View event detail</p> <p>Event data</p> <pre> 1 { 2 "id": "cs_test_a1cCmTfq07pHKqJJ7uW9F4RJMdfHCvYNasrQn0PXrk4xti", 3 "object": "checkout.session", 4 "livemode": false, 5 "payment_intent": "pi_3KgplHJ5UH88gi9T0e3c8Nhy", 6 "status": "complete", 7 "after_expiration": null, 8 "allow_promotion_codes": null, 9 "amount_subtotal": 12100, 10 "amount_total": 12100,</pre> <p>See all 72 lines</p> <ul style="list-style-type: none"> • A Checkout Session was completed 24/03/2022, 08:55:51 • The payment pi_3KgplHJ5UH88gi9T0e3c8Nhy for US\$121.00 has succeeded 24/03/2022, 08:55:50 • ch_3LXk4dJ5UH88gi9T1BepIYFX was charged US\$121.00 24/03/2022, 08:55:50 • A new payment pi_3KgplHJ5UH88gi9T0e3c8Nhy for US\$121.00 was created 24/03/2022, 08:55:49 • 200 OK A request to confirm a Checkout Session completed 24/03/2022, 08:55:31 • 200 OK A request to create a Checkout Session completed 24/03/2022, 08:55:30

Рис. 9.17. События и журналы транзакции Stripe

Этот раздел содержит всю активность, связанную с транзакцией, включая запросы к API платежного шлюза Stripe. Можно кликнуть по любому запросу, чтобы увидеть HTTP-запрос к API Stripe и ответ в формате JSON.

Давайте рассмотрим события активности в хронологическом порядке, снизу вверх.

1. Сначала создается новый сеанс оформления платежа путем отправки запроса методом POST на конечную точку API платежного шлюза Stripe. `/v1/checkout/sessions`. Метод `stripe.checkout.Session.create()` SDK платежного шлюза Stripe, который используется в представлении `payment_process`, формирует и отправляет запрос в API Stripe и обрабатывает ответ, чтобы вернуть сеансовый объект.
2. Пользователь перенаправляется на страницу оформления платежа, на которой он передает платежную форму на обработку. Запрос на подтверждение сеанса оформления платежа отправляется страницей оформления платежа Stripe.
3. Создается новое платежное намерение.
4. Создается взимаемая плата, связанная с платежным намерением.
5. Платежное намерение теперь завершено успешным платежом.
6. Сеанс оформления платежа завершен.

Поздравляем! Вы успешно интегрировали платежный инструмент Stripe Checkout в свой проект. Далее вы научитесь получать уведомления о платежах от Stripe и ссылаться на платежи Stripe в заказах в интернет-магазине.

Применение веб-перехватчиков для получения уведомлений о платежах

Платежный шлюз Stripe может передавать реально-временные события в приложение с помощью веб-перехватчиков. Веб-перехватчик, он же обратный вызов, можно рассматривать как событийно-управляемый API, в отличие от запросно-управляемого API. Вместо частого опрашивания API Stripe, чтобы узнавать о завершении нового платежа, Stripe может отправлять HTTP-запрос на URL-адрес приложения, чтобы уведомлять об успешных платежах в реальном времени. При наступлении события уведомления об этих событиях будут асинхронными, независимо от наших синхронных вызовов API Stripe.

Мы создадим конечную точку веб-перехватчика, чтобы получать события Stripe. Веб-перехватчик будет состоять из представления, которое будет получать полезную нагрузку¹ JSON о событии, для того чтобы ее затем обрабатывать. Мы будем использовать информацию о событии, чтобы помечать заказы как оплаченные после успешного завершения сеанса оформления платежа.

Создание конечной точки веб-перехватчика

Для того чтобы получать события, URL-адреса конечных точек веб-перехватчиков добавляются в учетную запись Stripe. Поскольку мы используем веб-перехватчики и у нас нет размещенного на сервере веб-сайта, общедоступного через публичный URL-адрес, мы будем использовать интерфейс

¹ Англ. payload; синоним «полезные данные». – Прим. перев.

командной строки Stripe (**CLI**), чтобы прослушивать события и пересыпать их в локальную среду.

Пройдите по URL-адресу <https://dashboard.stripe.com/test/webhooks> в своем браузере. Вы увидите следующий ниже экран:

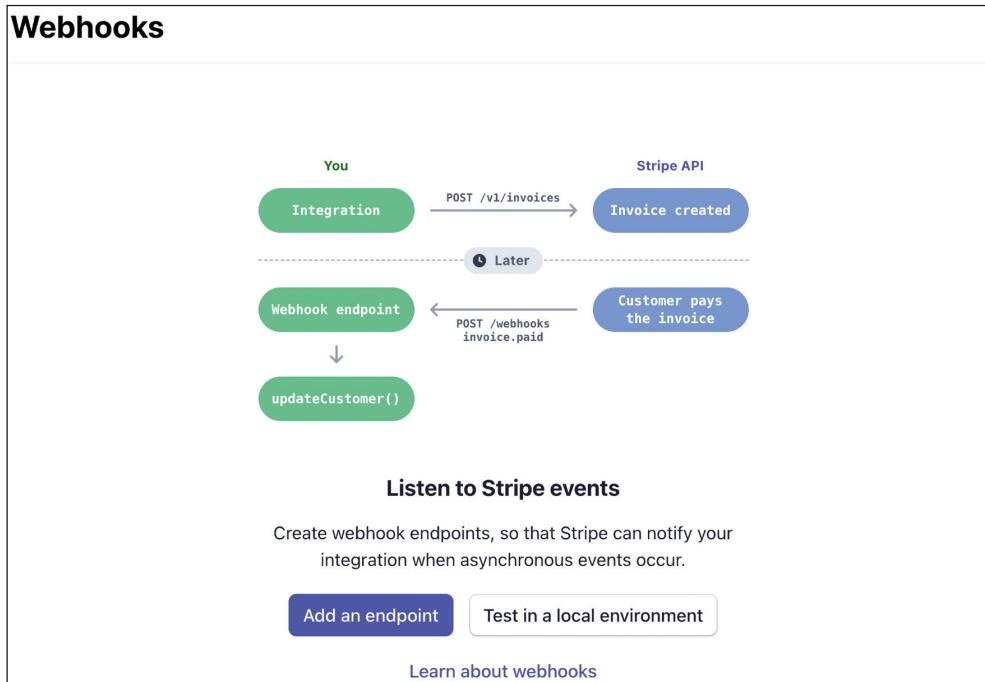


Рис. 9.18. Стандартный экран веб-перехватчиков Stripe

Здесь вы видите схему асинхронного уведомления платежным шлюзом Stripe о вашей интеграции. Вы будете получать уведомления Stripe в реальном времени всякий раз, когда происходит событие. Stripe отправляет различные типы событий, такие как создание сеанса оформления платежа, создание намерения платежа, обновление намерения платежа или завершение сеанса оформления платежа. Список всех отправляемых платежным шлюзом Stripe типов событий находится на странице <https://stripe.com/docs/api/events/types>.

Кликните по **Test in a local environment** (Протестировать в локальной среде). Вы увидите следующий ниже экран:

The screenshot shows the Stripe CLI setup interface. It includes a sidebar with 'Listen to Stripe events' and a main area with three steps:

- 1. Download the CLI and log in with your Stripe account**: Shows the command \$ stripe login and status 'Completed'.
- 2. Forward events to your webhook**: Shows the command \$ stripe listen --forward-to localhost:4242/webhook and status 'Completed'.
- 3. Trigger events with the CLI**: Shows the command \$ stripe trigger payment_intent.succeeded and status 'In progress'.

To the right is a code editor with a Python script:

```

1  # app.py
2  #
3  # Use this sample code to handle webhook events in your application
4  #
5  # 1) Paste this code into a new file (app.py)
6  #
7  # 2) Install dependencies
8  #   pip3 install flask
9  #   pip3 install stripe
10 #
11 # 3) Run the server on http://localhost:4242
12 #   python3 -m flask run --port=4242
13
14 import json
15 import os
16 import stripe
17
18 from flask import Flask, jsonify, request
19
20 # This is your Stripe CLI webhook secret for testing your endpoint
21 endpoint_secret = 'whsec_e46ee9a47be07eec94d46c324d7170'
22
23 app = Flask(__name__)
24
25 @app.route('/webhook', methods=['POST'])
26 def webhook():
27     event = None
28     payload = request.data
29     sig_header = request.headers['STRIPE_SIGNATURE']
30
31     try:
32         event = stripe.Webhook.construct_event(
33             payload, sig_header, endpoint_secret

```

A red circle highlights the line `endpoint_secret = 'whsec_e46ee9a47be07eec94d46c324d7170'`.

Рис. 9.19. Экран установки веб-перехватчика Stripe

На этом экране показаны шаги по прослушиванию событий Stripe из локальной среды. Он также содержит пример конечной точки веб-перехватчика на Python. Скопируйте только значение `endpoint_secret`.

Отредактируйте файл `settings.py` проекта `myshop`, добавив в него следующий ниже настроочный параметр:

```
STRIPE_WEBHOOK_SECRET = ''
```

Замените значение параметра `STRIPE_WEBHOOK_SECRET` значением `endpoint_secret`, которое было предоставлено платежным шлюзом Stripe.

Для того чтобы сформировать конечную точку веб-перехватчика, мы создадим представление, которое будет получать полезную нагрузку JSON с детальной информацией о событии. Мы будем проверять эту информацию о событии, чтобы выяснить, когда сеанс оформления платежа был завершен, и будем помечать соответствующий заказ как оплаченный.

Stripe подписывает события веб-перехватчика, отправляемые на конечные точки, включая заголовок `Stripe-Signature` с подписью в каждом событии. Проверяя подпись Stripe, можно верифицировать, что события были отправлены именно платежным шлюзом Stripe, а не третьей стороной. Если не верифицировать подпись, то злоумышленник сможет намеренно отправлять поддельные события на веб-перехватчики. SDK Stripe предоставляет метод верификации подписей. Мы будем использовать его для создания веб-перехватчика, который верифицирует подпись.

Добавьте новый файл в каталог `payment/application` и назовите его `webhooks.py`. Добавьте следующий ниже исходный код в новый файл `webhooks.py`:

```
import stripe
from django.conf import settings
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from orders.models import Order

@csrf_exempt
def stripe_webhook(request):
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
    event = None

    try:
        event = stripe.Webhook.construct_event(
            payload,
            sig_header,
            settings.STRIPE_WEBHOOK_SECRET)
    except ValueError as e:
        # Недопустимая полезная нагрузка
        return HttpResponse(status=400)
    except stripe.error.SignatureVerificationError as e:
        # Недопустимая подпись
        return HttpResponse(status=400)

    return HttpResponse(status=200)
```

Декоратор `@csrf_exempt` используется для предотвращения выполнения веб-фреймворком Django валидации CSRF, которая делается по умолчанию для всех запросов POST. Для верификации заголовка подписи под событием используется метод `stripe.Webhook.construct_event()` библиотеки Stripe. Если полезная нагрузка события или подпись недопустимы, то возвращается HTTP-ответ 400 Bad Request (Неправильный запрос). В противном случае возвращается HTTP-ответ 200 OK. Это базовая функциональность, необходимая для верификации подписи и конструирования события из полезной нагрузки JSON. Теперь можно реализовать действия конечной точки веб-перехватчика.

Добавьте следующий ниже исходный код, выделенный жирным шрифтом, в представление `stripe_webhook`:

```
@csrf_exempt
def stripe_webhook(request):
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
    event = None

    try:
        event = stripe.Webhook.construct_event(
            payload,
            sig_header,
            settings.STRIPE_WEBHOOK_SECRET)
    except ValueError as e:
        # Недопустимая полезная нагрузка
        return HttpResponse(status=400)
    except stripe.error.SignatureVerificationError as e:
        # Недопустимая подпись
        return HttpResponse(status=400)

    if event.type == 'checkout.session.completed':
        session = event.data.object
        if session.mode == 'payment' and session.payment_status == 'paid':
            try:
                order = Order.objects.get(id=session.client_reference_id)
            except Order.DoesNotExist:
                return HttpResponse(status=404)
            # пометить заказ как оплаченный
            order.paid = True
            order.save()

    return HttpResponse(status=200)
```

В новом исходном коде проверяется, что полученным событием является `checkout.session.completed`. Это событие указывает на успешное завершение сеанса оформления платежа. Если наступает это событие, то извлекается сеансовый объект и делается проверка, не является ли режим (`mode`) сеанса платежным (`payment`), поскольку это ожидаемый режим для разовых платежей. Затем извлекается атрибут `client_reference_id`, который использовался при создании сеанса оформления платежа, и задействуется преобразователем Django ORM, чтобы получить объект `Order` с данным `id`. Если заказ не существует, то вызывается исключение HTTP 404. В противном случае посредством инструкции `order.paid = True` заказ помечается как оплаченный и сохраняется в базе данных.

Отредактируйте файл `urls.py` приложения `payment`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

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

app_name = 'payment'

urlpatterns = [
    path('process/', views.payment_process, name='process'),
    path('completed/', views.payment_completed, name='completed'),
    path('canceled/', views.payment_canceled, name='canceled'),
    path('webhook/', webhooks.stripe_webhook, name='stripe-webhook'),
]
```

Здесь был импортирован модуль веб-перехватчиков и добавлен шаблон URL-адреса для веб-перехватчика Stripe.

Тестирование уведомлений веб-перехватчиков

Для тестирования веб-перехватчиков необходимо установить интерфейс командной строки Stripe (CLI). Это инструмент разработчика, который позволяет тестировать и управлять интеграцией с Stripe прямо из командной оболочки. Инструкция по его установке находится на странице <https://stripe.com/docs/stripe-cli#install>.

Если вы используете macOS или Linux, то можете установить CLI Stripe с помощью утилиты Homebrew, используя следующую ниже команду:

```
brew install stripe/stripe-cl/stripe
```

Если вы используете Windows или работаете с macOS или Linux без Homebrew, то скачайте последнюю версию Stripe CLI для macOS, Linux или Windows с <https://github.com/stripe/stripe-cl/releases/latest> и распакуйте архив. Если вы используете Windows, то запустите распакованный файл .exe.

После установки CLI Stripe выполните следующую ниже команду из оболочки:

```
stripe logi
```

Вы увидите следующий ниже результат:

```
Your pairing code is: xxxx-yyyy-zzzz-oooo
This pairing code verifies your authentication with Stripe.
Press Enter to open the browser or visit https://dashboard.stripe.com/
stripecli/confirm_auth?t=....
```

Нажмите клавишу **Enter** либо пройдите по URL-адресу в браузере. Вы увидите следующий ниже экран:



Рис. 9.20. Экран сопряжения CLI Stripe

Проверьте, чтобы код сопряжения в интерфейсе командной строки Stripe соответствовал коду, указанному на веб-сайте, и кликните по **Allow access** (Разрешить доступ). Вы увидите следующее ниже сообщение:

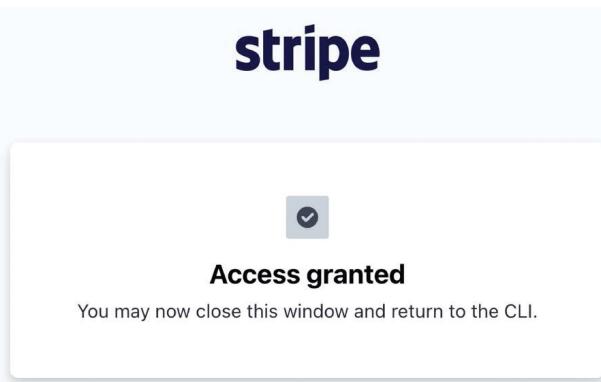


Рис. 9.21. Подтверждение сопряжения CLI Stripe

Теперь выполните следующую ниже команду из своей командной оболочки:

```
stripe listen --forward-to localhost:8000/payment/webhook/
```

Эта команда используется для того, чтобы сообщить Stripe, что надо прослушивать события и пересыпать их на локальный хост. Здесь используется порт 8000, на котором работает сервер разработки Django, и путь /payment/webhook/, который соответствует шаблону URL-адреса веб-перехватчика.

Вы увидите следующий ниже результат:

```
Getting ready... > Ready! You are using Stripe API Version [2022-08-01]. Your webhook signing secret isxxxxxxxxxxxxxxxxxxxx (^C to quit)
```

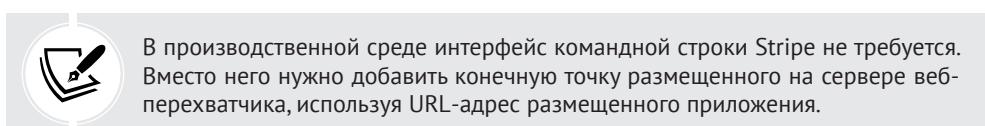
Здесь находится секрет веб-перехватчика. Проверьте, чтобы секрет подпиши веб-перехватчика соответствовал настроенному параметру STRIPE_WEBHOOK_SECRET в файле settings.py проекта.

Пройдите по URL-адресу <https://dashboard.stripe.com/test/webhooks> в своем браузере. Вы увидите следующий ниже экран:

The screenshot shows the Stripe Webhooks dashboard. At the top, there's a section for 'Hosted endpoints' with a button '+ Add endpoint'. Below it, a note says 'Listen to live Stripe events by creating a hosted webhook endpoint when your app is deployed online.' In the 'Local listeners' section, there's a button '+ Add local listener'. A table lists a single endpoint: 'Antonios-MacBook-Pro.local' with port 'localhost:8000/payment/webhook/' under 'DEVICE', '1.11.0' under 'VERSION', and a green button 'Listening' under 'STATUS'.

Рис. 9.22. Страница веб-перехватчиков Stripe

В разделе **Local listeners** (Локальные прослушиватели) вы увидите созданный нами локальный прослушиватель.



Пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере, добавьте несколько товаров в корзину и завершите процесс оформления платежа.

Обратитесь к оболочке, в которой вы используете CLI Stripe:

```
2022-08-17 13:06:13    --> payment_intent.created [evt_...]
2022-08-17 13:06:13    <--  [200] POST http://localhost:8000/payment/webhook/
[evt_...]
```

```
2022-08-17 13:06:13 --> payment_intent.succeeded [evt...]
2022-08-17 13:06:13 <-- [200] POST http://localhost:8000/payment/webhook/
[evt...]

2022-08-17 13:06:13 --> charge.succeeded [evt...]
2022-08-17 13:06:13 <-- [200] POST http://localhost:8000/payment/webhook/
[evt...]

2022-08-17 13:06:14 --> checkout.session.completed [evt...]
2022-08-17 13:06:14 <-- [200] POST http://localhost:8000/payment/webhook/
[evt...]
```

Вы увидите различные события, отправленные Stripe на локальную конечную точку веб-перехватчика. Они таковы (в хронологическом порядке):

- `payment_intent.created`: платежное намерение создано;
- `payment_intent.succeeded`: платежное намерение было успешным;
- `charge.succeeded`: связанный с платежным намерением платеж прошел успешно;
- `checkout.session.completed`: сеанс оформления платежа завершен. Это событие используется для того, чтобы пометить заказ как оплаченный.

Веб-перехватчик `stripe_webhook` возвращает HTTP-ответ 200 OK на все запросы, отправляемые платежным шлюзом Stripe. Однако мы обрабатываем только событие `checkout.session.completed`, чтобы пометить связанный с платежом заказ как оплаченный.

Затем пройдите по URL-адресу `http://127.0.0.1:8000/admin/orders/order/` в своем браузере. Заказ должен быть помечен как оплаченный:



Рис. 9.23. Заказ, помеченный как оплаченный в списке заказов на сайте администрирования

Теперь заказы автоматически помечаются как оплаченные с помощью платежных уведомлений Stripe. Далее вы научитесь ссылаться на платежи Stripe в своих заказах в магазине.

Отсылки к платежам Stripe в заказах

Каждый платеж Stripe имеет уникальный идентификатор. Его можно использовать для того, чтобы связывать каждый заказ с соответствующим платежом Stripe. Мы добавим новое поле в модель `Order` приложения `orders`, чтобы

иметь возможность ссылаться на связанный платеж по его ИД. Такой подход позволит связывать каждый заказ с соответствующей транзакцией Stripe.

Отредактируйте файл `models.py` приложения `orders`, добавив следующее ниже поле в модель `Order`. Новое поле выделено жирным шрифтом:

```
class Order(models.Model):
    # ...
    stripe_id = models.CharField(max_length=250, blank=True)
```

Давайте синхронизируем это поле с базой данных. Примените следующую ниже команду, чтобы создать миграции базы данных для проекта:

```
python manage.py makemigrations
```

Вы увидите такой результат:

```
Migrations for 'orders':
  orders/migrations/0002_order_stripe_id.py
    - Add field stripe_id to order
```

Следующей ниже командой примените миграцию к базе данных:

```
python manage.py migrate
```

Вы увидите результат, который заканчивается вот такой строкой:

```
Applying orders.0002_order_stripe_id... OK
```

Теперь изменения в модели синхронизированы с базой данных, и можно сохранять ИД платежа Stripe по каждому заказу.

Отредактируйте функцию `stripe_webhook` в файле `views.py` приложения `payment`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
# ...
@csrf_exempt
def stripe_webhook(request):
    # ...

    if event.type == 'checkout.session.completed':
        session = event.data.object
        if session.mode == 'payment' and session.payment_status == 'paid':
            try:
                order = Order.objects.get(id=session.client_reference_id)
            except Order.DoesNotExist:
                return HttpResponseRedirect(status=404)
            # пометить заказ как оплаченный
            order.paid = True
```

```
# сохранить ИД платежа Stripe  
order.stripe_id = session.payment_intent  
order.save()  
# запустить асинхронное задание  
payment_completed.delay(order.id)  
  
return HttpResponse(status=200)
```

Благодаря этому изменению при получении уведомления от веб-перехватчика о завершенном сеансе оформления платежа ИД намерения платежа сохраняется в поле `stripe_id` объекта `order`.

Пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере, добавьте товары в корзину и завершите процесс оформления платежа. Затем пройдите по URL-адресу `http://127.0.0.1:8000/admin/orders/order/` в своем браузере и кликните по ИД последнего заказа, чтобы его отредактировать. Поле `stripe_id` должно содержать ИД платежного намерения, как показано на рис. 9.24.



Рис. 9.24. Поле *Stripe id* с ИД платежного намерения

Отлично! Мы успешно ссылаемся на платежи Stripe в заказах. Теперь можно добавлять идентификаторы платежей Stripe в список заказов на сайте администрирования. Мы также можем включать ссылку на каждый ИД платежа, чтобы видеть детальную информацию о платеже на информационной панели Stripe.

Отредактируйте файл `models.py` приложения `orders`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.db import models  
from django.conf import settings  
from shop.models import Product  
  
class Order(models.Model):  
    # ...  
  
    class Meta:  
        # ...  
  
    def __str__(self):  
        return f'Order {self.id}'
```

```

def get_total_cost(self):
    return sum(item.get_cost() for item in self.items.all())

def get_stripe_url(self):
    if not self.stripe_id:
        # никаких ассоциированных платежей
        return ''
    if '_test_' in settings.STRIPE_SECRET_KEY:
        # путь Stripe для тестовых платежей
        path = '/test/'
    else:
        # путь Stripe для настоящих платежей
        path = '/'
    return f'https://dashboard.stripe.com{path}payments/{self.stripe_id}'

```

Здесь в модель Order был добавлен новый метод `get_stripe_url()`. Этот метод используется для возврата URL-адреса информационной панели Stripe для платежа, связанного с заказом. Если ИД платежа не хранится в поле `stripe_id` объекта `Order`, то возвращается пустая строка. В противном случае возвращается URL-адрес платежа в информационной панели Stripe. Далее проверяется наличие подстроки `_test_` в настроекном параметре `STRIPE_SECRET_KEY`, чтобы отличить производственную среду от тестовой. Платежи в производственной среде подчиняются шаблону `https://dashboard.stripe.com/payments/{id}`, тогда как тестовые платежи следуют шаблону `https://dashboard.stripe.com/payments/test/{id}`.

Добавьте ссылку на каждый объект `Order` на странице сайта администрирования с отображением списка.

Отредактируйте файл `admin.py` приложения `orders`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```

from django.utils.safestring import mark_safe

def order_stripe_payment(obj):
    url = obj.get_stripe_url()
    if obj.stripe_id:
        html = f'{obj.stripe\_id}'
        return mark_safe(html)
    return ''
order_stripe_payment.short_description = 'Stripe payment'

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                   'address', 'postal_code', 'city', 'paid',
                   order_stripe_payment, 'created', 'updated']
    # ...

```

Функция `order_stripe_payment()` принимает в качестве аргумента объект `Order` и возвращает HTML-ссылку с URL-адресом платежа Stripe. Django по умолчанию экранирует результат HTML. Мы используем функцию `mark_safe`, чтобы избежать автоматического экранирования.



Во избежание межсайтового скриптинга **XSS (Cross-Site Scripting)** следует избегать использования функции `mark_safe` с входными данными, получаемыми от пользователя. XSS позволяет злоумышленникам внедрять скрипты на стороне клиента в веб-контент, просматриваемый другими пользователями.

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/orders/order/` в своем браузере. Вы увидите новый столбец с названием **STRIPE PAYMENT** (Платеж Stripe) и что ИД платежа Stripe соответствует последнему заказу. Если вы кликните по ИД платежа, то попадете на URL-адрес платежа Stripe, где находятся дополнительные детали платежа.

PAID	STRIPE PAYMENT
	pi_3KgzZVJ5UH88gi9T1l8ofnc6

Рис. 9.25. ИД платежа Stripe
для объекта `order` на сайте администрирования

Теперь вы автоматически сохраняете ИД платежей Stripe в заказах при получении уведомлений о платежах. Вы успешно интегрировали Stripe в свой проект.

Выход в прямой эфир

После того как вы протестировали свою интеграцию, можно подавать заявку на производственную учетную запись Stripe. Когда вы будете готовы перейти в производственную среду, не забудьте заменить свои тестовые учетные данные Stripe в файле `settings.py` на эфирные. Вам также потребуется добавить конечную точку веб-перехватчика по адресу <https://dashboard.stripe.com/webhooks> для размещенного на сервере веб-сайта взамен использования интерфейса командной строки Stripe. Глава 17 «Выход в прямой эфир» научит вас конфигурировать настроочные параметры проекта для нескольких сред.

Экспорт заказов в CSV-файлы

Иногда бывает нужно экспортировать содержащуюся в модели информацию в файл, чтобы иметь возможность импортировать ее в другую систему. Одним из наиболее широко применяемых форматов для экспорта/импорта данных являются разделенные запятыми значения (CSV¹). CSV-файл – это обычный текстовый файл, состоящий из нескольких записей. Обычно одна запись на строку и некий символ-разделитель, обычно запятая, отделяющая поля записи. Мы собираемся адаптировать сайт администрирования под конкретно-прикладной экспорт заказов в CSV-файлы.

Добавление конкретно-прикладных действий на сайт администрирования

Django предлагает широкий спектр возможностей по адаптации сайта администрирования под конкретно-прикладную задачу. Вы собираетесь изменить представление списка объектов, чтобы включить конкретно-прикладное административное действие. Например, можно реализовать конкретно-прикладные административные действия, чтобы позволить сотрудникам применять действия в представлении списка изменений к нескольким элементам одновременно.

Административное действие работает следующим образом: пользователь выбирает объекты на странице списка объектов администрирования с помощью флажков, затем выбирает действие и выполняет его для всех выбранных элементов. На рис. 9.26 показано место, где на сайте администрирования располагаются действия:



Рис. 9.26. Выпадающее меню административных действий

¹ Англ. Comma-Separated Values. – Прим. перев.

Конкретно-прикладное действие создается путем написания обычной функции, которая получает следующие параметры:

- текущий отображаемый ModelAdmin;
- текущий объект запроса как экземпляр HttpRequest;
- набор запросов для выбранных пользователем объектов.

Эта функция будет исполняться, когда действие будет инициироваться с сайта администрирования.

Вы создадите конкретно-прикладное административное действие по скачиванию списка заказов в формате CSV-файла.

Отредактируйте файл `admin.py` приложения `orders`, добавив следующий ниже исходный код перед классом `OrderAdmin`:

```
import csv
import datetime
from django.http import HttpResponseRedirect

def export_to_csv(modeladmin, request, queryset):
    opts = modeladmin.model._meta
    content_disposition = f'attachment; filename={opts.verbose_name}.csv'
    response = HttpResponseRedirect(content_type='text/csv')
    response['Content-Disposition'] = content_disposition
    writer = csv.writer(response)
    fields = [field for field in opts.get_fields() if not \
              field.many_to_many and not field.one_to_many]
    # записать первую строку с информацией заголовка
    writer.writerow([field.verbose_name for field in fields])
    # записать строки данных
    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 этом исходном коде выполняется следующая работа.

1. Создается экземпляр `HttpResponse` с указанием типа содержимого `text/csv`, сообщая браузеру, что ответ должен обрабатываться как CSV-файл. Также добавляется заголовок `Content-Disposition`, указывая на то, что HTTP-ответ содержит вложенный файл.
2. Создается пишущий объект `writer`, который будет писать CSV в объект `response`.

3. Динамически извлекаются поля модели, используя метод `get_fields()` _meta опций модели. Исключаются взаимосвязи многие-ко-многим и один-ко-многим.
4. Пишется строка заголовка, состоящая из имен полей.
5. Заданный набор запросов QuerySet прокручивается в цикле, и пишется строка каждого объекта, возвращаемого набором запросов. При этом обращается внимание на форматирование объектов даты/времени, потому что выходное значение для CSV должно быть строковым.
6. В раскрывающемся списке действий на сайте администрирования адаптируется отображаемое имя действия; это делается за счет установки значения атрибута `short_description` функции.

Вы создали типовое административное действие, которое можно добавлять в любой класс `ModelAdmin`.

Наконец, добавьте новое административное действие `export_to_csv` в класс `OrderAdmin`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                    'address', 'postal_code', 'city', 'paid',
                    order_payment, 'created', 'updated']
    list_filter = ['paid', 'created', 'updated']
    inlines = [OrderItemInline]
    actions = [export_to_csv]
```

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/orders/order/` в своем браузере. Результатирующее административное действие должно выглядеть следующим образом:

Select order to change					
Action: <input type="button" value="Export to CSV"/> <input type="button" value="Go"/> 1 of 25 selected					
	ID	FIRST NAME	LAST NAME	EMAIL	ADDRESS
<input checked="" type="checkbox"/>	5	Antonio	Melé	antonio.mele@zenxit.com	20 W 34th St
<input type="checkbox"/>	4	Antonio	Melé	antonio.mele@zenxit.com	1 Bank Street

Рис. 9.27. Использование конкретно-прикладного административного действия `Export to CSV`

Выберите несколько заказов и выберите действие **Export to CSV** (Экспорт в CSV) в поле выбора, затем кликните по кнопке **Go** (Начать). Ваш браузер скачает сгенерированный CSV-файл с именем `order.csv`. Откройте скачанный файл с помощью текстового редактора. Вы должны увидеть содержимое в следующем ниже формате, включая строку заголовка и строку для каждого выбранного вами объекта `Order`:

```
ID,first name,last name,email,address,postal  
code,city,created,updated,paid,stripe id  
5,antonio,Melé,antonio.mele@zenxit.com,20 W 34th St,10001,New  
York,24/03/2022,24/03/2022,True,pi_3KgzZVJ5UH88gi9T1l8ofnc6  
...
```

Как видите, административные действия создаются довольно просто. Подробнее о создании CSV-файлов с помощью Django можно узнать на странице <https://docs.djangoproject.com/en/4.1/howto/outputting-csv/>.

Далее вы продолжите адаптировать сайт администрирования, создав конкретно-прикладное представление администрирования.

Расширение сайта администрирования за счет конкретно-прикладных представлений

Иногда возникает потребность адаптировать сайт администрирования за рамками того, что возможно, путем конфигурирования класса `ModelAdmin`, создания административных действий и переопределения шаблонов администрирования. Возможно, вы захотите реализовать дополнительные функциональности, недоступные в существующих представлениях или шаблонах администрирования. В этом случае необходимо создать конкретно-прикладное представление администрирования. Конкретно-прикладное представление позволяет создавать любую нужную функциональность; просто нужно обеспечить, чтобы к этому представлению могли получать доступ только штатные пользователи и чтобы внешний вид и поведение администрирования поддерживались постоянными, сделав свой шаблон расширением шаблона администрирования.

Давайте создадим конкретно-прикладное представление для показа информации о заказе. Отредактируйте файл `views.py` приложения `orders`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.urls import reverse  
from django.shortcuts import render, redirect, get_object_or_404  
from django.contrib.admin.views.decorators import staff_member_required
```

```

from .models import OrderItem, Order
from .forms import OrderCreateForm
from .tasks import order_created
from cart.cart import Cart

def order_create(request):
    # ...

@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 значения полей `is_active` и `is_staff` запрашивающего страницу пользователя установлены равными `True`. В этом представлении по заданному ИД извлекается объект `Order` и затем прорисовывается шаблон для отображения заказа.

Далее отредактируйте файл `urls.py` приложения `orders`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```

urlpatterns = [
    path('create/', views.order_create, name='order_create'),
    path('admin/order/<int:order_id>/', views.admin_order_detail,
          name='admin_order_detail'),
]

```

Внутри каталога `templates/` приложения `orders` создайте следующую ниже файловую структуру:

```

admin/
    orders/
        order/
            detail.html

```

Отредактируйте шаблон `detail.html`, добавив в него следующее ниже содержимое:

```

{% extends "admin/base_site.html" %}

{% 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 %}


# Order {{ order.id }}



- Print order



| Created      | {{ order.created }}                                              |
|--------------|------------------------------------------------------------------|
| Customer     | {{ order.first_name }} {{ order.last_name }}                     |
| E-mail       | <a href="mailto:{{ order.email }}">{{ order.email }}</a>         |
| Address      | {{ order.address }},<br>{{ order.postal_code }} {{ order.city }} |
| Total amount | \${{ order.get_total_cost }}                                     |
| Status       | {% if order.paid %}Paid{% else %}Pending payment{% endif %}      |


```

```
<th>Stripe payment</th>
<td>
    {% if order.stripe_id %}
        <a href="{{ order.get_stripe_url }}" target="_blank">
            {{ order.stripe_id }}
        </a>
    {% endif %}
</td>
</tr>
</table>
</div>
<div class="module">
    <h2>Items bought</h2>
    <table style="width:100%">
        <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">${{ item.price }}</td>
                    <td class="num">{{ item.quantity }}</td>
                    <td class="num">${{ 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>
{% endblock %}
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк.

Это шаблон для отображения детальной информации о заказе на сайте администрации. Он расширяет шаблон `admin/base_site.html` сайта администрирования Django, который содержит главную структуру HTML и стили CSS. Для вставки собственного контента используются блоки, определенные в родительском шаблоне. В результате на страницу выводится информация о заказе и приобретенных товарах.

Если вы хотите расширить шаблон администрирования, то для этого необходимо знать его структуру и выявить существующие блоки. Все шаблоны администрирования находятся на странице <https://github.com/django/django/tree/4.0/django/contrib/admin/templates/admin>.

При необходимости шаблон администрирования также можно переопределить. Для этого скопируйте шаблон в каталог `templates/`, оставив тот же относительный путь и имя файла. Встроенный в Django сайт администрирования будет использовать ваш собственный шаблон вместо стандартного.

Наконец, давайте добавим ссылку на каждый объект `Order` на странице сайта администрирования с отображением списка. Отредактируйте файл `admin.py` приложения `orders`, добавив в него следующий ниже исходный код над классом `OrderAdmin`:

```
from django.urls import reverse

def order_detail(obj):
    url = reverse('orders:admin_order_detail', args=[obj.id])
    return mark_safe(f'<a href="{url}">View</a>')
```

Это функция, которая принимает объект `Order` в качестве аргумента и возвращает HTML-ссылку на URL-адрес `admin_order_detail`. По умолчанию Django экранирует HTML-результат, и во избежание автоматического экранирования необходимо использовать функцию `mark_safe`.

Затем отредактируйте класс `OrderAdmin`, как показано ниже, чтобы отобразить ссылку. Новый исходный код выделен жирным шрифтом:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                    'address', 'postal_code', 'city', 'paid',
                    order_payment, 'created', 'updated',
                    order_detail]
    # ...
```

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/orders/order/` в своем браузере. Каждая строка содержит ссылку **View** (Просмотреть), как показано ниже:

PAID	STRIPE PAYMENT	CREATED	UPDATED	ORDER DETAIL
✓	pi_3KgzZVJ5UH88gi9T1l8ofnc6	March 24, 2022, 10:55 p.m.	March 24, 2022, 7:44 p.m.	View

Рис. 9.28. Ссылка **View**, включенная в каждую строку заказа

Кликните по ссылке **View** любого заказа, чтобы загрузить конкретно-прикладную страницу детальной информации о заказе. Вы должны увидеть примерно такую страницу:

Items bought			
PRODUCT	PRICE	QUANTITY	TOTAL
Green tea	\$30.00	1	\$30.00
Total			\$30.00

Рис. 9.29. Конкретно-прикладная страница детальной информации о заказе на сайте администрирования

Теперь, когда вы создали страницу детальной информации о товаре, вы научитесь динамически генерировать счета-фактуры заказов в формате PDF.

Динамическое генерирование счетов-фактур в формате PDF

Теперь, когда у вас есть полноценная система оформления заказов и платежей, можно приступить к работе над генерированием счетов-фактур в формате PDF по каждому заказу. Для создания PDF-файлов предназначено несколько библиотек Python, причем одной из самых популярных, в которой PDF-файлы генерируются исходным кодом Python, является библиотека ReportLab. Информация о том, как выводить PDF-файлы с помощью ReportLab, находится на странице <https://docs.djangoproject.com/en/4.1/howto/outputting-pdf/>.

В большинстве случаев в PDF-файлы придется добавлять конкретно-прикладные стили и форматирование. При этом удобнее прорисовывать HTML-шаблон и конвертировать его в PDF-файл, не используя Python в слое представления. Вы будете следовать этому подходу и использовать модуль

генерирования файлов PDF с помощью Django. Вы будете использовать библиотеку Python WeasyPrint, которая может генерировать PDF-файлы из шаблонов HTML.

Установка библиотеки WeasyPrint

Сначала установите зависимости библиотеки WeasyPrint для вашей операционной системы, перечисленные по адресу https://doc.courtbouillon.org/weasyprint/stable/first_steps.html. Затем, используя следующую ниже команду, установите WeasyPrint посредством pip:

```
pip install WeasyPrint==56.1
```

Создание шаблона PDF

В качестве входных данных для WeasyPrint потребуется HTML-документ. Вы создадите HTML-шаблон, прорисуете его с помощью Django и передадите в WeasyPrint, чтобы сгенерировать PDF-файл.

Внутри каталога `templates/orders/order/` приложения `orders` создайте новый файл шаблона и назовите его `pdf.html`. Добавьте в него следующий ниже исходный код:

```
<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>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Laptop</td>
                <td>1200</td>
                <td>1</td>
            </tr>
            <tr>
                <td>Monitor</td>
                <td>300</td>
                <td>1</td>
            </tr>
            <tr>
                <td>Keyboard</td>
                <td>100</td>
                <td>1</td>
            </tr>
            <tr>
                <td>Mouse</td>
                <td>50</td>
                <td>1</td>
            </tr>
        </tbody>
    </table>
</body>

```

```

<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 счета-фактуры в формате PDF. В данном шаблоне отображается вся детальная информация о заказе и содержащий товары HTML-элемент `<table>`. Вы также вставите сообщение о том, был ли заказ оплачен или нет.

Прорисовка PDF-файлов

Вы создадите представление генерирования счетов-фактур в формате PDF для существующих заказов, пользуясь сайтом администрирования. Откройте в редакторе файл `views.py` внутри каталога приложения `orders` и добавьте в него следующий ниже исходный код:

```

from django.conf import settings
from django.http import HttpResponseRedirect
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'] = f'filename=order_{order.id}.pdf'
weasyprint.HTML(string=html).write_pdf(response,
    stylesheets=[weasyprint.CSS(
        settings.STATIC_ROOT / 'css/pdf.css')])
return response
```

Это представление генерирования счета-фактуры в формате PDF для заказа. Декоратор `staff_member_required` используется в целях обеспечения того, чтобы доступ к этому представлению могли получать только штатные пользователи.

По заданному ИД извлекается объект `Order` и, используя предоставленную веб-фреймворком Django функцию `render_to_string()`, прорисовывается шаблон `orders/order/pdf.html`. Прорисованный HTML сохраняется в переменной `html`.

Затем генерируется новый объект `HttpResponse` с указанием типа содержимого `application/pdf` и с включением заголовка `Content-Disposition`, чтобы задать имя файла. Библиотека WeasyPrint используется для генерирования PDF-файла из прорисованного исходного кода HTML и записи файла в объект `HttpResponse`.

Для добавления стилей CSS в сгенерированный PDF-файл применяется статический файл `css/pdf.css`, который загружается из локального пути, используя настроечный параметр `STATIC_ROOT`. Наконец, возвращается сгенерированный ответ.

Если у вас отсутствуют стили CSS, то не забудьте скопировать статические файлы, расположенные в каталоге `static/` приложения `shop`, в то же место вашего проекта.

Содержимое каталога находится по адресу <https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter09/myshop/shop/static>.

Поскольку необходимо использовать настроечный параметр `STATIC_ROOT`, следует добавить его в свой проект. Это путь в проекте, где хранятся статические файлы. Отредактируйте файл `settings.py` проекта `myshop`, добавив следующий ниже настроечный параметр:

```
STATIC_ROOT = BASE_DIR / 'static'
```

Затем выполните такую команду:

```
python manage.py collectstatic
```

Вы должны увидеть результат, который заканчивается следующим образом:

```
131 static files copied to 'code/myshop/static'.
```

Команда `collectstatic` копирует все статические файлы из ваших приложений в каталог, указанный в настроичном параметре `STATIC_ROOT`. Такой

подход позволяет каждому приложению предоставлять свои собственные статические файлы, используя содержащий их каталог static/. Кроме того, в настроичном параметре STATICFILES_DIRS можно указывать дополнительные источники статических файлов. При исполнении команды collectstatic все каталоги, указанные в списке параметра STATICFILES_DIRS, также будут копироваться в каталог STATIC_ROOT. При очередном исполнении команды collectstatic будет предложено переопределить существующие статические файлы.

Откройте в редакторе файл urls.py внутри каталога приложения orders и добавьте в него следующий ниже шаблон URL, выделенный жирным шрифтом:

```
urlpatterns = [
    # ...
    path('admin/order/<int:order_id>/pdf/',
        views.admin_order_pdf,
        name='admin_order_pdf'),
]
```

Теперь можно отредактировать страницу сайта администрирования с отображением списка для модели Order, чтобы добавить ссылку на PDF-файл по каждому результату. Отредактируйте файл admin.py внутри приложения orders, добавив следующий ниже исходный код над классом OrderAdmin:

```
def order_pdf(obj):
    url = reverse('orders:admin_order_pdf', args=[obj.id])
    return mark_safe(f'PDF')
order_pdf.short_description = 'Invoice'
```

Если для вызываемого объекта указать атрибут short_description, то Django будет использовать его для имени столбца.

Добавьте order_pdf в атрибут list_display класса OrderAdmin, как показано ниже:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                    'address', 'postal_code', 'city', 'paid',
                    order_payment, 'created', 'updated',
                    order_detail, order_pdf]
```

Проверьте, чтобы сервер разработки был запущен. Пройдите по URL-адресу <http://127.0.0.1:8000/admin/orders/order/> в своем браузере. Теперь каждая строка должна содержать ссылку на PDF-файл, как показано ниже:

CREATED	UPDATED	ORDER DETAIL	INVOICE
March 24, 2022, 10:55 p.m.	March 24, 2022, 7:44 p.m.	View	PDF

Рис. 9.30. Ссылка на PDF-файл, включенная в каждую строку заказа

Кликните по ссылке **PDF** напротив любого заказа. Вы должны увидеть примерно такой сгенерированный PDF-файл неоплаченного заказа:

My Shop				
Invoice no. 6				
Mar 24, 2022				
Bill to				
Antonio Melé antonio.mele@zenxit.com 20 W 34th St 10001, New York				
Items bought				
Product	Price	Quantity	Cost	
Green tea	\$30.00	1	\$30.00	
Red tea	\$45.50	2	\$91.00	
Total			\$121.00	
PENDING PAYMENT				

Рис. 9.31. Счет-фактура неоплаченного заказа в формате PDF

В случае же оплаченных заказов вы увидите следующий ниже PDF-файл:

My Shop

Invoice no. 6
Mar 24, 2022

Bill to

Antonio Melé
antonio.mele@zenxit.com
20 W 34th St
10001, New York

Items bought

Product	Price	Quantity	Cost
Green tea	\$30.00	1	\$30.00
Red tea	\$45.50	2	\$91.00
Total			\$121.00

PAID

Рис. 9.32. Счет-фактура оплаченного заказа в формате PDF

Отправка PDF-файлов по электронной почте

При успешном прохождении платежа вы будете отправлять клиенту автоматическое электронное письмо со сгенерированным счетом-фактурой в формате PDF. Для выполнения этого действия вы создадите асинхронное задание.

Внутри каталога приложения `payment` создайте новый файл и назовите его `tasks.py`. Добавьте в него следующий ниже исходный код:

```
from io import BytesIO
from celery import shared_task
import weasyprint
from django.template.loader import render_to_string
from django.core.mail import EmailMessage
from django.conf import settings
from orders.models import Order
```

```
@shared_task
def payment_completed(order_id):
    """
    Задание по отправке уведомления по электронной почте
    при успешной оплате заказа.
    """

    order = Order.objects.get(id=order_id)
    # create invoice e-mail
    subject = f'My Shop - Invoice no. {order.id}'
    message = 'Please, find attached the invoice for your recent purchase.'
    email = EmailMessage(subject,
                          message,
                          'admin@myshop.com',
                          [order.email])

    # сгенерировать 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)

    # прикрепить PDF-файл
    email.attach(f'order_{order.id}.pdf',
                 out.getvalue(),
                 'application/pdf')

    # отправить электронное письмо
    email.send()
```

С помощью декоратора `@shared_task` определяется задание `payment_completed`. В этом задании используется предоставляемый веб-фреймворком Django класс `EmailMessage`, служащий для создания объекта `email`. Затем шаблон прорисовывается в переменную `html` и из прорисованного шаблона генерируется PDF-файл, который выводится в экземпляр `aBytesIO`. Последний представляет собой резидентный байтовый буфер. Затем с помощью метода `attach()` генерированный PDF-файл прикрепляется к объекту `EmailMessage` вместе с содержимым выходного буфера. Наконец, письмо отправляется.

Не забудьте настроить параметры простого протокола передачи почты (**SMTP**) в файле `settings.py` проекта, чтобы отправлять электронные письма. Обратитесь к главе 2 «Усовершенствование блога за счет продвинутых функциональностей», чтобы увидеть рабочий пример конфигурации SMTP. Если вы не хотите устанавливать настроечные параметры электронной почты, то можете сообщить Django, что нужно писать электронные письма в консоль, добавив следующий ниже настроечный параметр в файл `settings.py`:

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

Давайте добавим задание `payment_completed` в конечную точку веб-перехватчика, которая обрабатывает события оформления платежа.

Отредактируйте файл `webhooks.py` приложения `payment`, придав ему следующий вид:

```
import stripe
from django.conf import settings
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from orders.models import Order
from .tasks import payment_completed

@csrf_exempt
def stripe_webhook(request):
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
    event = None

    try:
        event = stripe.Webhook.construct_event(
            payload,
            sig_header,
            settings.STRIPE_WEBHOOK_SECRET)
    except ValueError as e:
        # недопустимая полезная нагрузка
        return HttpResponse(status=400)
    except stripe.error.SignatureVerificationError as e:
        # недопустимая подпись
        return HttpResponse(status=400)

    if event.type == 'checkout.session.completed':
        session = event.data.object
        if session.mode == 'payment' and session.payment_status == 'paid':
            try:
                order = Order.objects.get(id=session.client_reference_id)
            except Order.DoesNotExist:
                return HttpResponse(status=404)
            # пометить заказ как оплаченный
            order.paid = True
            # сохранить ИД платежа Stripe
            order.stripe_id = session.payment_intent
            order.save()
            # запустить асинхронное задание
            payment_completed.delay(order.id)

    return HttpResponse(status=200)
```

Задание `payment_completed` ставится в очередь путем вызова его метода `delay()`. Задание будет добавлено в очередь и исполнено асинхронно работником Celery как можно раньше.

Теперь можно завершить новый процесс оформления заказа, чтобы получать счет-фактуру в формате PDF по электронной почте. Если для своей электронной почты вы используете почтовый бэкенд `console.EmailBackend`, то в оболочке, где вы используете Celery, будет следующий ниже результат:

```
MIME-Version: 1.0
Subject: My Shop - Invoice no. 7
From: admin@myshop.com
To: antonio.mele@zenxit.com
Date: Sun, 27 Mar 2022 20:15:24 -0000
Message-ID: <164841212458.94972.103440689995916799@antonios-mbp.home>

-----8908668108717577350==
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Please, find attached the invoice for your recent purchase.

-----8908668108717577350==
Content-Type: application/pdf
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="order_7.pdf"

JVBERi0xLjCkJfcflqQKMSAwIG9iago8PAovVHlwZSA...
```

Этот результат показывает, что сообщение электронной почты содержит вложение. Вы научились прикреплять файлы к электронным письмам и отправлять их программно.

Поздравляем! Вы завершили интеграцию Stripe и добавили ценные функциональности в свой магазин.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter09>.
- Веб-сайт Stripe: <https://www.stripe.com/>.
- Документация по платежному инструменту Stripe Checkout: <https://stripe.com/docs/payments/checkout>.
- Создание учетной записи Stripe: <https://dashboard.stripe.com/register>.
- Настройки учетной записи Stripe: <https://dashboard.stripe.com/settings/account>.
- Библиотека Stripe на Python: <https://github.com/stripe/stripe-python>.

- Тестовые ключи API Stripe: <https://dashboard.stripe.com/test/apikeys>.
- Документация по ключам API Stripe: <https://stripe.com/docs/keys>.
- Примечания к выпуску API Stripe версии 2022-08-01: <https://stripe.com/docs/upgrades#2022-08-01>.
- Режимы сеанса оформления платежа Stripe: https://stripe.com/docs/api/checkout/sessions/object#checkout_session_object-mode.
- Создание абсолютных URI-идентификаторов с помощью Django: https://docs.djangoproject.com/en/4.1/ref/request-response/#django.http.HttpRequest.build_absolute_uri.
- Создание сеансов Stripe: <https://stripe.com/docs/api/checkout/sessions/create>.
- Валюты, поддерживаемые Stripe: <https://stripe.com/docs/currencies>.
- Информационная панель платежей Stripe Payments: <https://dashboard.stripe.com/test/payments>.
- Кредитные карты для тестирования платежей с помощью Stripe: <https://stripe.com/docs/testing>.
- Веб-перехватчики Stripe: <https://dashboard.stripe.com/test/webhooks>.
- Типы событий, отправляемых Stripe: <https://stripe.com/docs/api/events/types>.
- Установка интерфейса командной строки Stripe: <https://stripe.com/docs/stripe-cli#install>.
- Последний релиз Stripe CLI: <https://github.com/stripe/stripe-cli/releases/latest>.
- Создание CSV-файлов с помощью Django: <https://docs.djangoproject.com/en/4.1/howto/output-csv/>.
- Шаблоны администрирования Django: <https://github.com/django/django/tree/4.0/django/contrib/admin/templates/admin>.
- Вывод PDF-файлов с помощью ReportLab: <https://docs.djangoproject.com/en/4.1/howto/output-pdf/>.
- Установка библиотеки WeasyPrint: <https://weasyprint.readthedocs.io/en/latest/install.html>.
- Статические файлы этой главы: <https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter09/myshop/shop/static>.

Резюме

В этой главе вы интегрировали платежный шлюз Stripe в свой проект и создали конечную точку веб-перехватчика, чтобы получать уведомления о платежах. Вы создали конкретно-прикладное административное действие по экспорту заказов в CSV. Вы также адаптировали встроенный в Django сайт администрирования, используя конкретно-прикладные представления и шаблоны. Наконец, вы научились создавать PDF-файлы с помощью библиотеки WeasyPrint и прикреплять их к электронным письмам.

В следующей главе вы научитесь создавать купонную систему с использованием сеансов Django и создадите механизм рекомендации товаров с помощью Redis.

10

Расширение магазина

В предыдущей главе научились интегрировать платежный шлюз в свой магазин. Вы также научились создавать файлы CSV и PDF.

В этой главе вы добавите в свой магазин купонную систему и создадите в нем механизм рекомендации товаров.

В данной главе будут рассмотрены следующие темы:

- создание купонной системы;
- применение купонов к корзине;
- применение купонов к заказам;
- создание купонов для платежного инструмента Stripe Checkout;
- хранение товаров, которые обычно покупаются вместе;
- создание механизма рекомендации товаров с помощью резидентного хранилища Redis.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter10>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Создание купонной системы

Многие интернет-магазины выдают покупателям купоны, которые можно обменивать на скидки во время покупок. Онлайновый купон обычно состоит из кода, который предоставляется пользователям и действует в течение определенного периода времени.

Вы создадите купонную систему для своего магазина. Ваши купоны будут действительны для клиентов в течение определенного периода времени.

Купоны не будут иметь никаких ограничений по числу их погашения, и они будут применяться к общей стоимости корзины.

Для этой функциональности вам потребуется создать модель хранения кода купона, допустимых временных рамок и применяемой скидки.

Следующей ниже командой создайте новое приложение `coupons` внутри проекта `myshop`:

```
python manage.py startapp coupons
```

Отредактируйте файл `settings.py` проекта `myshop`, добавив приложение в настроочный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [  
    # ...  
    'coupons.apps.CouponsConfig',  
]
```

Теперь новое приложение в проекте Django является активным.

Разработка купонной модели

Давайте начнем с создания модели `Coupon`. Отредактируйте файл `models.py` приложения `coupons`, добавив в него следующий ниже исходный 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)],  
        help_text='Percentage value (0 to 100)')  
    active = models.BooleanField()  
  
    def __str__(self):  
        return self.code
```

Это модель, которую вы будете использовать для хранения купонов. Модель `Coupon` содержит следующие поля:

- `code`: код, который пользователи должны ввести, чтобы применить купон к своей покупке;
- `valid_from`: значение даты/времени, указывающее, когда купон становится действительным;
- `valid_to`: значение даты/времени, указывающее, когда купон становится недействительным;
- `discount`: применяемый уровень скидки (это процент, поэтому принимает значения в интервале от 0 до 100). Для этого поля надо использовать валидаторы, чтобы ограничивать минимальное и максимальное допустимые значения;
- `active`: булево значение, указывающее, является ли купон активным/неактивным.

Выполните следующую ниже команду, чтобы сгенерировать первоначальную миграцию приложения `coupons`:

```
python manage.py makemigrations
```

Результат должен содержать следующие ниже строки:

```
Migrations for 'coupons':  
  coupons/migrations/0001_initial.py  
    - Create model Coupon
```

Затем исполните следующую ниже команду, чтобы применить миграции:

```
python manage.py migrate
```

Вы должны увидеть результат, который содержит вот такую строку:

```
Applying coupons.0001_initial... OK
```

Теперь миграции применены к базе данных. Давайте добавим модель `Coupon` на сайт администрирования. Отредактируйте файл `admin.py` приложения `coupons`, добавив в него следующий ниже исходный код:

```
from django.contrib import admin  
from .models import Coupon  
  
@admin.register(Coupon)  
class CouponAdmin(admin.ModelAdmin):  
    list_display = ['code', 'valid_from', 'valid_to',  
                  'discount', 'active']  
    list_filter = ['active', 'valid_from', 'valid_to']  
    search_fields = ['code']
```

Модель Coupon зарегистрирована на сайте администрирования. Проверьте, чтобы ваш локальный сервер работал, выполнив такую команду:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/coupons/coupon/add/` в своем браузере. Вы должны увидеть следующую ниже форму:

The screenshot shows the Django administration interface for adding a new coupon. The top navigation bar includes links for Home, Coupons, Coupons, Add coupon, and user information (WELCOME, ADMIN, VIEW SITE / CHANGE PASSWORD / LOG OUT). The main content area is titled 'Add coupon'. It contains fields for 'Code' (a text input field), 'Valid from' (Date and Time inputs, both set to 'Today | Now'), and 'Valid to' (Date and Time inputs, both set to 'Now | Now'). Below these are notes: 'Note: You are 1 hour ahead of server time.' and 'Note: You are 1 hour ahead of server time.'. A 'Discount' field (a text input field with a percentage icon) is present, along with an unchecked checkbox for 'Active'. At the bottom right are three buttons: 'Save and add another', 'Save and continue editing', and a large blue 'SAVE' button.

Рис. 10.1. Форма для добавления купона на сайте администрирования

Заполните форму, чтобы создать новый купон, действительный на текущую дату, проверьте, чтобы флажок **Active** (Активный) был установлен, и кликните по кнопке **SAVE** (Сохранить). На рис. 10.2 показан пример создания купона:

Add coupon

Code:	SUMMER
Valid from:	Date: 2022-03-28 Today Calendar Time: 00:00:00 Now Clock
<small>Note: You are 2 hours ahead of server time.</small>	
Valid to:	Date: 2029-09-28 Today Calendar Time: 00:00:00 Now Clock
<small>Note: You are 2 hours ahead of server time.</small>	
Discount:	10
<small>Percentage value (0 to 100)</small>	
<input checked="" type="checkbox"/> Active	

Рис. 10.2. Форма для добавления купона с примером данных

После создания купона страница списка изменений купонов на сайте администрирования будет выглядеть примерно так:

Select coupon to change

Select coupon to change				
<input type="text"/>	<input type="button" value="Search"/>	Action: <input type="button" value="-----"/>	<input type="button" value="Go"/>	0 of 1 selected
CODE	VALID FROM	VALID TO	DISCOUNT	ACTIVE
SUMMER	March 28, 2022, midnight	March 28, 2029, midnight	10	<input checked="" type="checkbox"/>
1 coupon				

Рис. 10.3. Страница списка изменений купонов на сайте администрирования

Далее мы реализуем функциональность применения купонов к корзине.

Применение купона к корзине

На данный момент вы можете хранить новые купоны и делать запросы, чтобы извлекать существующие купоны. Теперь нужно, чтобы клиенты могли применять купоны к своим покупкам. Функциональность применения купона будет следующей.

1. Пользователь добавляет товары в корзину.
2. Пользователь вводит код купона в форму, отображаемую на странице детальной информации о корзине.
3. Когда пользователь вводит код купона и передает форму на обработку, отыскивается существующий купон с данным кодом, который в настоящее время действителен. При этом необходимо убедиться, чтобы код купона совпадал с кодом, введенным пользователем, чтобы атрибут `active` был равен `True` и чтобы текущая дата/время находились между значениями `valid_from` и `valid_to`.
4. Если купон найден, то он сохраняется в сеансе пользователя, и затем отображается корзина, содержащая примененную к ней скидку и обновленную общую сумму.
5. Когда пользователь размещает заказ, купон сохраняется в данном заказе.

Внутри каталога приложения `coupons` создайте новый файл и назовите его `forms.py`. Добавьте в него следующий ниже исходный код:

```
from django import forms

class CouponApplyForm(forms.Form):
    code = forms.CharField()
```

Это форма, которую вы будете использовать для ввода пользователем кода купона. Отредактируйте файл `views.py` внутри приложения `coupons`, добавив в него следующий ниже исходный код:

```
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')
```

Представление `coupon_apply` выполняет валидацию купона и сохраняет его в сеансе пользователя. Здесь декоратор `require_POST` применяется для того, чтобы ограничивать представление запросами POST. В данном представлении выполняется следующая работа.

1. Используя отправленные данные, создается экземпляр формы `CouponApplyForm` и проверяется валидность формы.
2. Если форма валидна, то из словаря `clean_data` формы берется введенный пользователем код (`code`) и делается попытка получить объект `Coupon` с заданным кодом. При этом поиск в поле осуществляется с использованием оператора `iexact`, чтобы отыскать нечувствительное к регистру точное совпадение. Купон должен быть активен в данный момент (`active=True`) и действителен на текущую дату/время. С целью получения текущей даты/времени с учетом часового пояса используется функция Django `timezone.now()`, и полученный результат сравнивается с полями `valid_from` и `valid_to`, просматривая поле с применением операторов соответственно `lte` (меньше или равно) и `gte` (больше или равно).
3. ИД купона сохраняется в сеансе пользователя.
4. Пользователь перенаправляется на URL-адрес `cart_detail`, чтобы отобразить корзину с примененным купоном.

Далее потребуется шаблон URL-адреса представления `coupon_apply`. Внутри каталога приложения `coupons` создайте новый файл и назовите его `urls.py`. Добавьте в него следующий ниже исходный код:

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

app_name = 'coupons'

urlpatterns = [
    path('apply/', views.coupon_apply, name='apply'),
]
```

Затем отредактируйте главный файл `urls.py` проекта `myshop`, вставив шаблон URL-адреса `coupons`, в виде следующей ниже строки, выделенной жирным шрифтом:

```
urlpatterns = [
    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('', include('shop.urls', namespace='shop')),
]

```

Не забудьте поместить этот шаблон перед шаблоном `shop.urls`.

Теперь отредактируйте файл `cart.py` приложения `cart`, вставив в него следующую ниже инструкцию импорта:

```
from coupons.models import Coupon
```

Добавьте следующий ниже исходный код, выделенный жирным шрифтом, в конец метода `__init__(self, request)` класса `Cart`, чтобы инициализировать купон из текущего сеанса:

```

class Cart:

    def __init__(self, request):
        """
        Инициализировать корзину.
        """

        self.session = request.session
        cart = self.session.get(settings.CART_SESSION_ID)
        if not cart:
            # сохранить пустую корзину в сеансе
            cart = self.session[settings.CART_SESSION_ID] = {}
        self.cart = cart
        # сохранить текущий примененный купон
        self.coupon_id = self.session.get('coupon_id')

```

В этом исходном коде делается попытка получить сеансовый ключ `cup_id` из текущего сеанса и сохранить его значение в объекте `Cart`. Добавьте следующие ниже методы, выделенные жирным шрифтом, в объект `Cart`:

```

class Cart:
    # ...

    @property
    def coupon(self):
        if self.coupon_id:
            try:
                return Coupon.objects.get(id=self.coupon_id)

```

```
except Coupon.DoesNotExist:  
    pass  
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 таковы:

- `coupon()`: этот метод определяется как свойство. Если корзина содержит атрибут `coupon_id`, то возвращается объект `Coupon` с заданным ИД;
- `get_discount()`: если в корзине есть купон, то извлекается его уровень скидки и возвращается сумма, которая будет вычтена из общей суммы корзины;
- `get_total_price_after_discount()`: возвращается общая сумма корзины после вычета суммы, возвращаемой методом `get_discount()`.

Теперь класс `Cart` готов обрабатывать купон, примененный к текущему сеансу, и применять соответствующую скидку.

Давайте включим купонную систему в представление детальной информации о корзине. Отредактируйте файл `views.py` приложения `cart`, добавив следующую ниже инструкцию импорта в начало файла:

```
from coupons.forms import CouponApplyForm
```

Ниже в этом файле отредактируйте представление `cart_detail`, добавив в него новую форму следующим образом:

```
def cart_detail(request):  
    cart = Cart(request)  
    for item in cart:  
        item['update_quantity_form'] = CartAddProductForm(initial={  
            'quantity': item['quantity'],  
            'override': True})  
    coupon_apply_form = CouponApplyForm()  
    return render(request,  
                  'cart/detail.html',  
                  {'cart': cart,  
                   'coupon_apply_form': coupon_apply_form})
```

Откройте шаблон `cart/detail.html` приложения `cart` и найдите в нем следующие ниже строки:

```
<tr class="total">
    <td>Total</td>
    <td colspan="4"></td>
    <td class="num">${{ cart.get_total_price }}</td>
</tr>
```

Замените их следующим ниже исходным кодом:

```
{% 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>
```

Это исходный код, служащий для отображения опционального купона и его уровня скидки. Если в корзине есть купон, то отображается первая строка, содержащая общую сумму корзины в качестве промежуточного итога. Затем используется вторая строка, чтобы отобразить текущий купон, примененный к корзине. Наконец, путем вызова метода `get_total_price_after_discount()` объекта `cart` отображается общая цена, содержащая любую скидку.

В том же файле вставьте следующий ниже исходный код после HTML-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 будет отображать форму для ввода кода купона и будет применять его к текущей корзине.

Пройдите по URL-адресу <http://127.0.0.1:8000/> в своем браузере и добавьте товар в корзину. Вы увидите, что на странице корзины появилась форма для применения купона:

The screenshot shows a shopping cart interface. At the top, it says 'Your shopping cart'. Below is a table with the following columns: Image, Product, Quantity, Remove, Unit price, and Price. There is one item listed: 'Tea powder' with a quantity of '1' and a unit price of '\$21.20', totaling '\$21.20'. Below the table, there's a section for applying a coupon with a text input field labeled 'Code:' and a blue 'Apply' button. At the bottom right are 'Continue shopping' and 'Checkout' buttons.

Image	Product	Quantity	Remove	Unit price	Price
	Tea powder	1	<input type="button" value="Update"/>	\$21.20	\$21.20

Total \$21.20

Apply a coupon:

Code:

Рис. 10.4. Страница детальной информации о корзине, содержащей форму для применения купона



Изображение чайного порошка: фото Фуонг Нгуена на Unsplash.

В поле **Code** (Код) введите код купона, который вы создали с помощью сайта администрирования:

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Tea powder	1 <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$21.20	\$21.20
Total \$21.20					

Apply a coupon:

Code:

Рис. 10.5. Страница детальной информации о корзине, содержащей код купона в форме

Кликните по кнопке **Apply** (Применить). Купон будет применен, и в корзине будет отображаться купонная скидка, как показано ниже:

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Tea powder	1 <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$21.20	\$21.20
Subtotal \$21.20					
"SUMMER" coupon (10% off) -\$ 2.12					
Total \$19.08					

Apply a coupon:

Code:

Рис. 10.6. Страница детальной информации о корзине, содержащей примененный купон

Давайте добавим купон в следующий шаг процесса покупки. Откройте шаблон `orders/order/create.html` приложения `orders` и найдите следующие ниже строки:

```
<ul>
  {% for item in cart %}
    <li>
      {{ item.quantity }}x {{ item.product.name }}
      <span>${{ item.total_price }}</span>
    </li>
  {% endfor %}
</ul>
```

Замените их таким исходным кодом:

```
<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 class="neg">- ${{ cart.get_discount|floatformat:2 }}</span>
    </li>
  {% endif %}
</ul>
```

Теперь сводная информация о заказе должна содержать примененный купон, если он есть. Далее найдите следующую ниже строку:

```
<p>Total: ${{ cart.get_total_price }}</p>
```

И замените ее такой:

```
<p>Total: ${{ cart.get_total_price_after_discount|floatformat:2 }}</p>
```

Благодаря этому изменению общая цена тоже будет рассчитываться с учетом купонной скидки.

Пройдите по URL-адресу <http://127.0.0.1:8000/orders/create/> в своем браузере. Вы должны увидеть, что сумма заказа включает примененный купон, как показано ниже:



Рис. 10.7. Сводная информация о заказе, содержащая купон, примененный к корзине

Теперь пользователи могут применять купоны к своей корзине. Однако еще необходимо обеспечить функциональность хранения информации о купоне в том порядке, в котором она создается, когда пользователи оформляют корзину.

Применение купонов к заказам

Вы будете хранить купон, который применялся к каждому заказу. Прежде всего необходимо видоизменить модель Order с учетом хранения связанного объекта Coupon, если он есть.

Отредактируйте файл models.py приложения orders, добавив в него следующие ниже инструкции импорта:

```
from decimal import Decimal
from django.core.validators import MinValueValidator, \
    MaxValueValidator
from coupons.models import Coupon
```

Затем добавьте в модель Order такие поля:

```
class Order(models.Model):
    # ...
    coupon = models.ForeignKey(Coupon,
        related_name='orders',
        null=True,
        blank=True,
        on_delete=models.SET_NULL)
    discount = models.IntegerField(default=0,
        validators=[MinValueValidator(0),
        MaxValueValidator(100)])
```

Эти поля позволяют сохранять опциональный купон заказа и процентную скидку, применяемую к купону. Скидка хранится в связанном объекте Cou-

роп, но ее можно включить в модель Order, чтобы ее сохранять, если купон был видоизменен или удален. Параметр `on_delete` задается равным `models.SET_NULL`, чтобы при удалении купона поле `coupon` устанавливалось равным `Null`, но скидка сохранялась.

Теперь необходимо создать миграцию, чтобы включить новые поля модели Order в базу данных. Выполните следующую ниже команду из командной строки:

```
python manage.py makemigrations
```

Вы должны увидеть результат, подобный приведенному ниже:

```
Migrations for 'orders':
  orders/migrations/0003_order_coupon_order_discount.py
    - Add field coupon to order
    - Add field discount to order
```

Следующей ниже командой примените новую миграцию:

```
python manage.py migrate orders
```

Вы должны увидеть показанное ниже подтверждение, указывающее на то, что новая миграция была применена:

```
Applying orders.0003_order_coupon_order_discount... OK
```

Теперь изменения полей модели Order синхронизированы с базой данных. Отредактируйте файл `models.py`, добавив два новых метода, `get_total_cost_before_discount()` и `get_discount()`, в модель Order, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
class Order(models.Model):
    ...
    def get_total_cost_before_discount(self):
        return sum(item.get_cost() for item in self.items.all())

    def get_discount(self):
        total_cost = self.get_total_cost_before_discount()
        if self.discount:
            return total_cost * (self.discount / Decimal(100))
        return Decimal(0)
```

Затем отредактируйте метод `get_total_cost()` модели Order следующим образом. Новый исходный код выделен жирным шрифтом:

```
def get_total_cost(self):
    total_cost = self.get_total_cost_before_discount()
    return total_cost - self.get_discount()
```

Теперь метод `get_total_cost()` модели `Order` будет учитывать примененную скидку, если она есть.

Отредактируйте файл `views.py` приложения `orders`, видоизменив представление `order_create`, чтобы сохранять связанный купон и его скидку при создании нового заказа. Добавьте следующий ниже исходный код, выделенный жирным шрифтом, в представление `order_create`:

```
def order_create(request):
    cart = Cart(request)
    if request.method == 'POST':
        form = OrderCreateForm(request.POST)
        if form.is_valid():
            order = form.save(commit=False)
            if cart.coupon:
                order.coupon = cart.coupon
                order.discount = cart.coupon.discount
            order.save()
            for item in cart:
                OrderItem.objects.create(order=order,
                                         product=item['product'],
                                         price=item['price'],
                                         quantity=item['quantity'])
            # очистить корзину
            cart.clear()
            # запустить асинхронное задание
            order_created.delay(order.id)
            # задать заказ в сеансе
            request.session['order_id'] = order.id
            # перенаправить к платежу
            return redirect(reverse('payment:process'))
    else:
        form = OrderCreateForm()
    return render(request,
                  'orders/order/create.html',
                  {'cart': cart, 'form': form})
```

В новом исходном коде, используя метод `save()` формы `OrderCreateForm`, создается объект `Order`, избегая его сохранения в базе данных посредством `commit=False`. Если в корзине есть купон, то связанный купон и примененная скидка сохраняются. Затем в базе данных сохраняется объект `order`.

Откройте шаблон `payment/process.html` приложения `payment` и найдите следующие ниже строки:

```
<tr class="total">
    <td>Total</td>
    <td colspan="4"></td>
    <td class="num">${{ order.get_total_cost }}</td>
</tr>
```

Замените их следующим ниже исходным кодом. Новые строки выделены жирным шрифтом:

```
{% if order.coupon %}
<tr class="subtotal">
<td>Subtotal</td>
<td colspan="3"></td>
<td class="num">
    ${{ order.get_total_cost_before_discount|floatformat:2 }}<br/>
</td>
</tr>
<tr>
<td>
    "{{ order.coupon.code }}" coupon
    ({{ order.discount }}% off)
</td>
<td colspan="3"></td>
<td class="num neg">
    - ${{ order.get_discount|floatformat:2 }}<br/>
</td>
</tr>
{% endif %}
<tr class="total">
<td>Total</td>
<td colspan="3"></td>
<td class="num">
    ${{ order.get_total_cost|floatformat:2 }}<br/>
</td>
</tr>
```

Мы обновили сводную информацию о заказе перед платежом.

Следующей ниже командой обеспечьте, чтобы сервер разработки был запущен:

```
python manage.py runserver
```

Проверьте, чтобы Docker был запущен, и выполните следующую ниже команду в другой оболочке, чтобы запустить сервер RabbitMQ с Docker:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672
rabbitmq:management
```

Откройте еще одну оболочку и следующей ниже командой запустите работника Celery из каталога проекта:

```
celery -A myshop worker -l info
```

Откройте дополнительную оболочку и исполните следующую ниже команду, чтобы перенаправлять события Stripe на локальный URL-адрес веб-перехватчика:

```
stripe listen --forward-to localhost:8000/payment/webhook/
```

Пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере и создайте заказ, используя созданный вами купон. После проверки товаров в корзине на странице сводной информации о заказе вы увидите, что купон будет применен к заказу:

Image	Product	Price	Quantity	Total
	Tea powder	\$21.20	1	\$21.20
Subtotal				\$21.20
"SUMMER" coupon (10% off)				- \$2.12
Total				\$19.08

Pay now

Рис. 10.8. Страница сводной информации о заказе, содержащей примененный к заказу купон

Если вы нажмете **Pay now** (Оплатить сейчас), то увидите, что Stripe не знает о применяемой скидке, как показано на рис. 10.9:

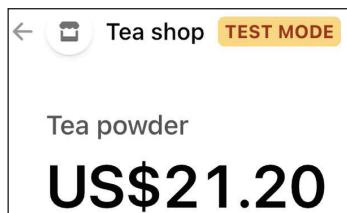


Рис. 10.9. Детальная информация о товарной позиции на странице платежного инструмента Stripe Checkout без скидочного купона

Платформа Stripe показывает полную сумму к оплате без каких-либо вычетов. Это вызвано тем, что скидка не была передана платформе Stripe. Напомним, что в представлении `payment_process` товарные позиции заказа передаются в Stripe как `line_items`, содержащие стоимость и количество каждой товарной позиции заказа.

Создание купонов для платежного инструмента Stripe Checkout

Платформа Stripe позволяет определять скидочные купоны и связывать их с разовыми платежами. Дополнительная информация о создании скидок для платежного инструмента Stripe Checkout находится по адресу <https://stripe.com/docs/payments/checkout/discounts>.

Давайте отредактируем представление `payment_process`, чтобы создать купон для платежного инструмента Stripe Checkout. Откройте файл `views.py` приложения `payment` и добавьте следующий ниже исходный код, выделенный жирным шрифтом, в представление `payment_process`:

```
def payment_process(request):
    order_id = request.session.get('order_id', None)
    order = get_object_or_404(Order, id=order_id)

    if request.method == 'POST':
        success_url = request.build_absolute_uri(
            reverse('payment:completed'))
        cancel_url = request.build_absolute_uri(
            reverse('payment:canceled'))

        # сеансовые данные оформления платежа Stripe
        session_data = {
            'mode': 'payment',
            'client_reference_id': order.id,
            'success_url': success_url,
            'cancel_url': cancel_url,
            'line_items': []
        }

        # добавить товарные позиции заказа в
        # сеанс оформления платежа Stripe
        for item in order.items.all():
            session_data['line_items'].append({
                'price_data': {
                    'unit_amount': int(item.price * Decimal('100')),
                    'currency': 'usd',
                    'product_data': {
                        'name': item.product.name,
                
```

```

        },
    },
    'quantity': item.quantity,
})

# купон Stripe
if order.coupon:
    stripe_coupon = stripe.Coupon.create(
        name=order.coupon.code,
        percent_off=order.discount,
        duration='once')
    session_data['discounts'] = [{{
        'coupon': stripe_coupon.id
}}]

# создать сеанс оформления платежа Stripe
session = stripe.checkout.Session.create(**session_data)

# перенаправить к форме для платежа Stripe
return redirect(session.url, code=303)
else:
    return render(request, 'payment/process.html', locals())

```

В новом исходном коде выполняется проверка на наличие у заказа связанного с ним купона. В этом случае SDK Stripe используется для создания купона Stripe с помощью `stripe.Coupon.create()`. При этом в купоне используются следующие атрибуты:

- `name`: применяется связанный с объектом `order` код (`code`) купона;
- `percent_off`: скидка (`discount`) объекта `order`;
- `duration`: используется значение `once`. Оно указывает Stripe, что это купон для разового платежа.

После создания купона его `id` добавляется в словарь `session_data`, используемый для создания сеанса Stripe Checkout. За счет этого купон связывается с сеансом оформления заказа.

Пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере и совершите покупку, используя созданный вами купон. При перенаправлении на страницу платежного инструмента Stripe Checkout вы увидите примененный купон:

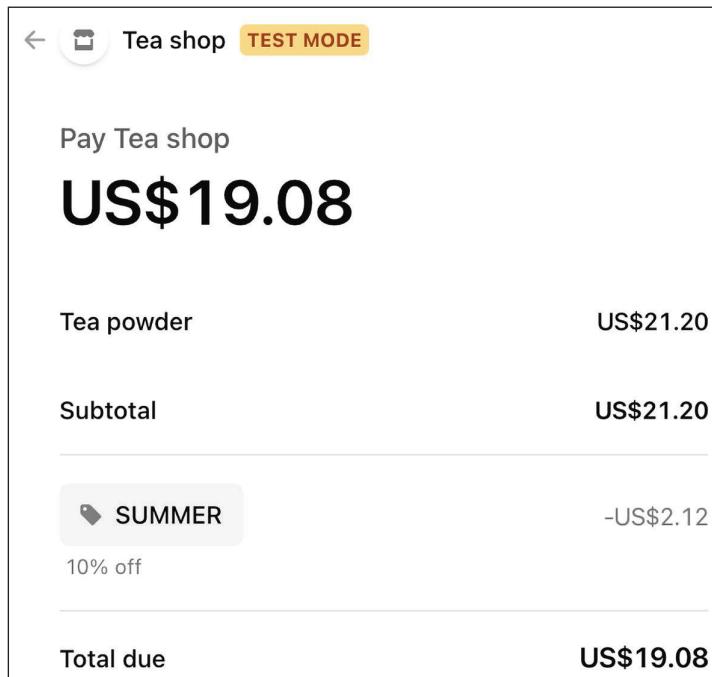


Рис. 10.10. Подробная информация о товаре на странице платежного инструмента Stripe Checkout, содержащая скидочный купон под названием SUMMER

Теперь страница платежного инструмента Stripe Checkout содержит купон заказа, а общая сумма к оплате содержит сумму, вычтенную с использованием купона.

Завершите покупку, а затем пройдите по URL-адресу <http://127.0.0.1:8000/admin/orders/order/> в своем браузере. Кликните по объекту `order`, для которого был использован купон. Форма для редактирования отобразит примененную скидку, как показано на рис. 10.11:

Stripe id:	pi_3KuKIYJ5UH88gj9T0aShmjvC		
Coupon:	SUMMER <input type="button" value="▼"/> <input type="button" value="+"/> <input type="button" value="X"/>		
Discount:	10		
ORDER ITEMS			
PRODUCT	PRICE	QUANTITY	DELETE?
4 1 Q Tea powder	21,20	1	<input type="checkbox"/>

Рис. 10.11. Форма для редактирования заказа, содержащая купон и примененную скидку

Вы организовали успешное хранение купонов к заказам и обработку платежей со скидками. Далее вы добавите купоны в представление детальной информации о заказе на сайте администрирования и в счета-фактуры заказов, прорисованные в формате PDF.

Добавление купонов в заказы на сайте администрирования и в счета-фактуры в формате PDF

Давайте добавим купон на страницу детальной информации о заказе на сайте администрации. Отредактируйте шаблон `admin/orders/order/detail.html` приложения `orders`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
...





```

```

{%- endif %}

<tr class="total">
    <td colspan="3">Total</td>
    <td class="num">
        ${{ order.get_total_cost|floatformat:2 }}
    </td>
</tr>
</tbody>
</table>
...

```

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/orders/order/> в своем браузере и кликните по ссылке **View** (Просмотреть) последнего заказа. Теперь таблица **Items bought** (Приобретенные позиции) будет содержать использованный купон, как показано на рис. 10.12:

Items bought			
PRODUCT	PRICE	QUANTITY	TOTAL
Tea powder	\$21.20	2	\$42.40
Subtotal			\$42.40
"SUMMER" coupon (10% off)			-\$4.24
Total			\$38.16

Рис. 10.12. Страница детальной информации о товаре на сайте администрации, содержащей использованный купон

Теперь давайте изменим шаблон счета-фактуры заказа, чтобы включить используемый для заказа купон. Отредактируйте шаблон `orders/order/detail.pdf` приложения `orders`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```

...
<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 %}

    {% if order.coupon %}
        <tr class="subtotal">
            <td colspan="3">Subtotal</td>
            <td class="num">
                ${{ order.get_total_cost_before_discount|floatformat:2 }}
            </td>
        </tr>
        <tr>
            <td colspan="3">
                "{{ order.coupon.code }}" coupon
                ({{ order.discount }}% off)
            </td>
            <td class="num neg">
                - ${{ order.get_discount|floatformat:2 }}
            </td>
        </tr>
    {% endif %}

    <tr class="total">
        <td colspan="3">Total</td>
        <td class="num">${{ order.get_total_cost|floatformat:2 }}</td>
    </tr>
</tbody>
</table>
...

```

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/orders/order/> в своем браузере и кликните по ссылке **PDF** последнего заказа. Теперь таблица **Items bought** (Приобретенные позиции) будет содержать использованный купон, как показано на рис. 10.13.

Вы успешно добавили купонную систему в свой магазин. Далее вы разработаете механизм рекомендации товаров.

Items bought			
Product	Price	Quantity	Cost
Tea powder	\$21.20	2	\$42.40
Subtotal			\$42.40
"SUMMER" coupon (10% off)			- \$4.24
Total			\$38.16



Рис. 10.13. Счет-фактура заказа в формате PDF, содержащая использованный купон

Разработка рекомендательного механизма

Рекомендательный механизм – это система, которая предсказывает предпочтения или рейтинг, которые пользователь дал бы товарной позиции. Система выбирает релевантные для пользователя товарные позиции, основываясь на его поведении и знаниях о них. В настоящее время рекомендательные системы используются во многих онлайновых сервисах. Они помогают пользователям тем, что подбирают из огромного объема имеющихся данных, которые не имеют к ним никакого отношения, то, что может их заинтересовать. Предоставление хороших рекомендаций повышает вовлеченность пользователей. Сайты электронной коммерции тоже получают выгоду от предложения релевантных рекомендуемых товаров, получаемую путем увеличения среднего дохода в расчете на пользователя.

Вы создадите простой, но мощный рекомендательный механизм, предлагающий товары, которые обычно покупаются вместе. Вы будете предлагать товары на основе исторических продаж, тем самым выявляя товары, которые обычно покупаются вместе. Вы будете предлагать дополнительные товары в двух разных сценариях:

- **страница детальной информации о товаре:** будет отображаться список товаров, которые обычно покупаются вместе с данным товаром. Он будет отображаться как пользователи, которые купили этот товар, также купили X, Y и Z. Для этого сценария нужна структура данных,

которая позволит хранить число покупок каждого товара вместе с отображаемым товаром;

- **страница детальной информации о корзине:** основываясь на товарах, которые пользователи добавляют в корзину, вы будете предлагать товары, которые обычно покупаются вместе с ними. В этом случае балл, который рассчитывается для получения связанных товаров, должен быть агрегирован.

Для хранения товаров, которые обычно покупаются вместе, будет использоваться резидентное хранилище Redis. Напомним, что вы уже использовали Redis в главе 7 «Отслеживание действий пользователя». Если вы еще не установили Redis, то найдете инструкции по его установке в указанной главе.

Рекомендация товаров на основе предыдущих покупок

Мы будем рекомендовать товары пользователям на основе товаров, которые часто покупаются вместе. Для этого мы собираемся хранить ключ по каждому приобретенному на сайте товару в резидентном хранилище Redis. Ключ товара будет содержать сортированное множество Redis с баллами. При совершении каждой новой покупки балл будет увеличиваться на 1 для каждого товара, купленного вместе. Сортированное множество позволит ставить баллы товарам, которые покупаются вместе. В качестве балла товарной позиции мы будем использовать число раз, когда товар покупался вместе с другим товаром.

Не забудьте установить библиотеку `redis`-ру в своей среде следующей ниже командой:

```
pip install redis==4.3.4
```

Отредактируйте файл `settings.py` проекта, добавив в него приведенные далее настроочные параметры:

```
# настроочные параметры Redis
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 1
```

Это настройки, необходимые для установления соединения с сервером Redis. Внутри каталога приложения `shop` создайте новый файл и назовите его `recommender.py`. Добавьте в него следующий ниже исходный код:

```
import redis
from django.conf import settings
from .models import Product
```

```
# соединить с redis
r = redis.Redis(host=settings.REDIS_HOST,
                 port=settings.REDIS_PORT,
                 db=settings.REDIS_DB)

class Recommender:
    def get_product_key(self, id):
        return f'product:{id}:purchased_with'

    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:
                # получить другие товары, купленные
                # вместе с каждым товаром
                if product_id != with_id:
                    # увеличить балл товара,
                    # купленного вместе
                    r.zincrby(self.get_product_key(product_id),
                              1,
                              with_id)
```

Это класс `Recommender`, который предоставит возможность хранить покупки товаров и получать предложения по данному товару или товарам.

Метод `get_product_key()` получает ИД объекта `Product` и формирует ключ сортированного множества Redis, в котором хранятся связанные товары. При этом ключ выглядит как `product:[id]:purchased_with`.

Метод `products_bought()` получает список объектов `Product`, которые были куплены вместе (то есть принадлежат одному заказу).

В этом методе выполняется следующая работа.

1. Берутся товарные идентификаторы для заданных объектов `Product`.
2. Товарные идентификаторы прокручиваются в цикле. По каждому идентификатору товарные идентификаторы снова прокручиваются в цикле, пропуская тот же товар, чтобы получить товары, которые покупаются вместе с каждым товаром.
3. С помощью метода `get_product_id()` по каждому купленному товару берется товарный ключ Redis. Для товара с ИД 33 этот метод возвращает ключ `product:33:purchased_with`. Это ключ сортированного множества, которое содержит идентификаторы товаров, купленных вместе с ним.
4. Балл каждого товарного ИД, содержащегося в сортированном множестве, увеличивается на 1. Балл представляет число раз, когда другой товар был куплен вместе с данным товаром.

Теперь у вас есть способ хранить и назначать баллы товарам, которые были куплены вместе. Далее нужен метод извлечения товаров, которые были куплены вместе для списка заданных товаров. Добавьте следующий ниже метод `submit_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:
        # только 1 товар
        suggestions = r.zrange(
            self.get_product_key(product_ids[0]),
            0, -1, desc=True)[:max_results]
    else:
        # сгенерировать временный ключ
        flat_ids = ''.join([str(id) for id in product_ids])
        tmp_key = f'tmp_{flat_ids}'
        # несколько товаров, объединить баллы всех товаров
        # сохранить полученное сортированное множество
        # во временном ключе
        keys = [self.get_product_key(id) for id in product_ids]
        r.zunionstore(tmp_key, keys)
        # удалить идентификаторы товаров,
        # для которых дается рекомендация
        r.zrem(tmp_key, *product_ids)
        # получить идентификаторы товаров по их количеству,
        # сортировка по убыванию
        suggestions = r.zrange(tmp_key, 0, -1,
                               desc=True)[:max_results]
        # удалить временный ключ
        r.delete(tmp_key)
    suggested_products_ids = [int(id) for id in suggestions]
    # получить предлагаемые товары и
    # отсортировать их по порядку их появления
    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 `offer_products_for()` получает следующие параметры:

- `products`: это список объектов `Product`, для которых нужно получить рекомендации. Он может содержать один или несколько товаров;
- `max_results`: это целое число, представляющее максимальное число возвращаемых рекомендуемых товаров.

В этом методе выполняются следующие действия.

1. Для заданных объектов `Product` берутся идентификаторы товаров.
2. Если указан только один товар, то берутся идентификаторы товаров, которые были куплены вместе с данным товаром, упорядоченные по общему числу раз, когда они покупались вместе. Для этой цели используется команда Redis `ZRANGE`. Число результатов ограничивается числом, указанным в атрибуте `max_results` (по умолчанию 6).
3. Если указано более одного товара, то генерируется временный ключ Redis, сформированный с использованием идентификаторов товаров.

4. Все баллы товарных позиций, содержащихся в сортированном множестве каждого данного товара, объединяются и суммируются. Это делается с помощью команды Redis `ZUNIONSTORE`. Команда `ZUNIONSTORE` выполняет операцию объединения сортированных множеств с заданными ключами и сохраняет агрегированную сумму баллов товарных позиций в новом ключе Redis. Подробнее об этой команде можно прочитать на странице <https://redis.io/commands/zunionstore/>. Агрегированные баллы сохраняются во временном ключе.
5. Поскольку баллы суммируются, можно получить те же товары, по которым берутся рекомендации. Они удаляются из генерированного отсортированного множества с помощью команды `ZREM`.
6. С помощью команды `ZRANGE` из временного ключа извлекаются идентификаторы товаров, упорядоченные по их баллам. Число результатов ограничивается числом, указанным в атрибуте `max_results`. Затем временный ключ удаляется.
7. Наконец, берутся объекты `Product` с заданными идентификаторами и товары упорядочиваются в том же порядке, что и они.

Из практических соображений давайте также добавим метод очистки рекомендаций. Добавьте следующий ниже метод в класс `Recommender`:

```
def clear_purchases(self):  
    for id in Product.objects.values_list('id', flat=True):  
        r.delete(self.get_product_key(id))
```

Теперь давайте испытаем рекомендательный механизм. Проверьте, чтобы в базу данных было включено несколько объектов `Product` и инициализирован контейнер Redis платформы Docker следующей ниже командой:

```
docker run -it --rm --name redis -p 6379:6379 redis
```

Откройте еще одну оболочку и выполните следующую ниже команду, чтобы открыть оболочку Python:

```
python manage.py shell
```

Проверьте, чтобы в вашей базе данных было как минимум четыре разных товара. Извлеките четыре разных товара по их именам:

```
>>> from shop.models import Product  
>>> black_tea = Product.objects.get(name='Black tea')  
>>> red_tea = Product.objects.get(name='Red tea')  
>>> green_tea = Product.objects.get(name='Green tea')  
>>> tea_powder = Product.objects.get(name='Tea powder')
```

Затем добавьте несколько тестовых покупок в рекомендательный механизм:

```
>>> 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])
```

Вы сохранили следующие баллы:

```
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)
```

Это представление данных для товаров, которые были куплены вместе с каждым товаром, включая число раз, когда они покупались вместе.

Давайте извлечем рекомендуемые товары для одного товара:

```
>>> 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>]
```

Хорошо видно, что порядок рекомендуемых товаров зависит от их балла. Давайте получим несколько рекомендуемых товаров с агрегированными баллами:

```
>>> 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>]
```

Вы видите, что порядок предлагаемых товаров совпадает с агрегированными баллами. Например, для `black_tea` и `red_tea` предлагаются следующие товары: `tea_powder` (2+1) и `green_tea` (1+1).

Вы убедились, что рекомендательный алгоритм работает как положено. Теперь давайте отобразим рекомендуемые товары на сайте.

Отредактируйте файл `views.py` приложения `shop`. В представление `product_detail` добавьте функциональность извлечения не более четырех рекомендуемых товаров, как показано ниже:

```
from .recommender import Recommender

def product_detail(request, id, slug):
    product = get_object_or_404(Product,
        id=id,
        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})
```

Отредактируйте шаблон `shop/product/detail.html` приложения `shop`, добавив следующий ниже исходный код после `{{ product.description|linebreaks }}`:

```
{% if recommended_products %}


### People who bought this also bought


{% for p in recommended_products %}


!\[Thumbnail image of {{ p.name }}\]\({% if p.image %}{{ p.image.url }}{% else %}
    {% static \)


{{ p.name }}


{% endfor %}


{% endif %}
```

Запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере. Кликните по любому товару, чтобы просмотреть о нем детальную информацию. Вы должны увидеть, что рекомендуемые товары отображаются под товаром, как показано на рис. 10.14:



Tea powder
Tea
\$21.20

Quantity: Add to cart

People who bought this also bought

NO IMAGE AVAILABLE		
Black tea	Red tea	Green tea

Рис. 10.14. Страница детальной информации о товаре, содержащей рекомендуемые товары



Изображения в этой главе:

- Зеленый чай: фото Цзя Е на Unsplash;
- Красный чай: фото Манки Ким на Unsplash;
- Чайный порошок: фото Phuong Nguyen на Unsplash.

Далее вы также вставите рекомендуемые товары в корзину. Рекомендуемые товары будут основываться на товарах, которые пользователь добавил в корзину.

Отредактируйте `views.py` внутри приложения `cart`, импортировав класс `Recommender` и отредактировав представление `cart_detail`, придая ему следующий вид:

```
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'],
        'override': True})
coupon_apply_form = CouponApplyForm()

r = Recommender()
cart_products = [item['product'] for item in cart]
if(cart_products):
    recommended_products = r.suggest_products_for(
        cart_products,
        max_results=4)
else:
    recommended_products = []
return render(request,
    'cart/detail.html',
    {'cart': cart,
     'coupon_apply_form': coupon_apply_form,
     'recommended_products': recommended_products})
```

Отредактируйте шаблон `cart/detail.html` приложения `cart`, добавив следующий ниже исходный код сразу после `</table>`:

```
{% if recommended_products %}


### People who bought this also bought


{% 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 %}


```

Пройдите по URL-адресу `http://127.0.0.1:8000/en/` в своем браузере и добавьте пару товаров в корзину. Когда вы перейдете по адресу `http://127.0.0.1:8000/en/cart/`, вы должны увидеть агрегированные рекомендуемые товарные позиции в корзине, как показано ниже:

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Green tea	1 <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$30.00	\$30.00
	Tea powder	1 <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$21.20	\$21.20
Total \$51.20					

People who bought this also bought

	
Black tea	Red tea

Apply a coupon:

Coupon:

Рис. 10.15. Страница детальной информации о корзине, содержащей рекомендуемые товары

Поздравляем! Вы создали полный рекомендательный механизм, используя веб-фреймворк Django и резидентное хранилище Redis.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter10>.
- Скидки платежного инструмента Stripe Checkout: <https://stripe.com/docs/payments/checkout/discounts>.
- Команда Redis ZUNIONSTORE: <https://redis.io/commands/zunionstore/>.

Резюме

В этой главе вы создали купонную систему, используя сеансы Django, и интегрировали ее с платежным шлюзом Stripe. Вы также создали рекомендательный механизм с помощью резидентного хранилища Redis, чтобы рекомендовать товары, которые обычно покупаются вместе.

Следующая глава даст вам понимание интернационализации и локализации проектов Django. Вы научитесь переводить исходный код и управлять переводами с помощью приложения Rosetta. Вы внедрите URL-адреса для переводов и разработаете селектор языка. Вы также реализуете переводы моделей с помощью `django-parler` и будете верифицировать локализованные поля форм с помощью `django-localflavor`.

11

Добавление интернационализации в магазин

В предыдущей главе вы добавили купонную систему в магазин и разработали систему рекомендации товаров.

В этой главе вы узнаете, как работают интернационализация и локализация.

В ней будут рассмотрены следующие темы:

- подготовка проекта к интернационализации;
- управление файлами перевода;
- перевод исходного кода Python;
- перевод шаблонов;
- использование приложения Rosetta для управления переводами;
- перевод шаблонов URL-адресов и использование префикса языка в URL-адресах;
- предоставление пользователям возможности переключать язык;
- перевод моделей с помощью модуля `django-parler`;
- использование переводов с ORM-преобразователем;
- адаптация представлений под использование переводов;
- использование локализованных полей форм модуля `django-localflavor`.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter11>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Интернационализация в Django

Django предлагает полную поддержку интернационализации и локализации. Указанная поддержка позволяет переводить приложение на несколько языков и обрабатывает форматирование дат, времен, чисел и часовых поясов в зависимости от локали. Давайте проясним разницу между интернационализацией и локализацией.

Интернационализация (часто сокращенно **i18n**) – это процесс адаптации программного обеспечения под потенциальное использование разных языков и локалей, чтобы избежать его жесткой привязки к определенному языку или локали.

Локализация (сокращенно **l10n**) – это процесс фактического перевода программного обеспечения и его адаптации под конкретную локаль. Сам Django переведен более чем на 50 языков с использованием своего фреймворка интернационализации.

Фреймворт интернационализации позволяет легко помечать подлежащие переводу строковые литералы как в исходном коде Python, так и в шаблонах. В нем используется инструментарий GNU gettext, который служит для генерирования файлов сообщений и управления ими. **Файл сообщения** – это обычный текстовый файл, представляющий язык. Он содержит часть либо все подлежащие переводу строковые литералы, найденные в приложении, и соответствующие им переводы на другой язык. Файлы сообщений имеют расширение `.po`. После завершения перевода файлы сообщений компилируются, чтобы обеспечить быстрый доступ к переведенным строковым литералам. Скомпилированные файлы перевода имеют расширение `.mo`.

Настроочные параметры интернационализации и локализации

Django предоставляет несколько настроочных параметров интернационализации. Наиболее актуальными являются следующие параметры:

- `USE_I18N`: булево значение, указывающее на активацию/деактивацию встроенной в Django системы перевода. По умолчанию равен `True`;
- `USE_L10N`: булево значение, указывающее на активацию/деактивацию локализованного форматирования. Для представления дат и чисел в активном состоянии параметра используются локализованные форматы. По умолчанию равен `False`;
- `USE_TZ`: булево значение, указывающее на учитывание/неучитывание в датах/временах часового пояса. При создании проекта с помощью команды `startproject` данный параметр получает значение `True`;
- `LANGUAGE_CODE`: код языка, применяемый в проекте по умолчанию. Это стандартный формат идентификатора языка, например '`en-us`' для американского английского или '`en-gb`' для британского английского. Для того чтобы он вступил в силу, требуется, чтобы значение настроеч-

ногого параметра USE_I18N было задано равным True. Список допустимых идентификаторов языков находится на странице <http://www.i18nguy.com/unicode/language-identifiers.html>;

- LANGUAGES: кортеж, содержащий имеющиеся для проекта языки. Языки состоят из двухэлементных кортежей: **кода языка и названия языка**. Список имеющихся языков можно посмотреть в django.conf.global_settings. При выборе языков, на которых будет доступен ваш сайт, параметр LANGUAGES следует устанавливать в качестве подмножества этого списка;
- LOCALE_PATHS: список каталогов, в которых Django ищет файлы сообщений, содержащие переводы проекта;
- TIME_ZONE: строковый литерал, представляющий часовой пояс проекта. При создании нового проекта с помощью команды startproject он устанавливается равным 'UTC'. При этом можно устанавливать любой другой часовой пояс, например 'Europe/Madrid'.

Это лишь несколько из имеющихся в Django настроек параметров интернационализации и локализации. Полный их список находится по адресу <https://docs.djangoproject.com/en/4.1/ref/settings/#globalization-i18n-l10n>.

Команды управления интернационализацией

Django содержит следующие команды, служащие для управления переводами:

- makemessages: пробегает по дереву исходного кода, чтобы отыскать все строковые литералы, помеченные для перевода, и создает или обновляет файлы сообщений .po в каталоге locale. По каждому языку создается один файл .po;
- compilemessages: компилирует существующие файлы сообщений .po в файлы .mo, которые используются для извлечения переводов.

Установка инструментария gettext

Для того чтобы иметь возможность создавать, обновлять и компилировать файлы сообщений, понадобится инструментарий gettext. Большинство дистрибутивов Linux уже содержат инструментарий gettext. Если же вы используете macOS, то самый простой способ его установить – применить утилиту Homebrew (расположенную по адресу <https://brew.sh/>) следующей ниже командой:

```
brew install gettext
```

Кроме того, возможно, понадобится связать его принудительно с помощью такой команды:

```
brew link --force gettext
```

Если вы используете Windows, то следуйте инструкциям по адресу <https://docs.djangoproject.com/en/4.1/topics/i18n/translation/#gettext-on-windows>. Предкомпилированный бинарный установщик инструментария gettext для Windows можно скачать со страницы <https://mlocati.github.io/articles/gettext-iconv-windows.html>.

Как добавлять переводы в проект Django

Давайте посмотрим на процесс интернационализации проекта. Он состоит из следующих действий:

- 1) отметить подлежащие переводу строковые литералы в своем коде Python и шаблонах;
- 2) выполнить команду `makemessages`, чтобы создать или обновить файлы сообщений, включающие все переводные строковые литералы из исходного кода;
- 3) перевести содержащиеся в файлах сообщений строковые литералы и скомпилировать их с помощью команды управления `compilemessages`.

Как Django определяет текущий язык

Django поставляется с компонентом фреймворка промежуточных программных компонентов, который определяет текущий язык на основе данных запроса. Это класс `LocaleMiddleware`, который находится в `django.middleware.locale.LocaleMiddleware` и выполняет следующую работу.

1. Если используется `i18n_patterns`, то есть применяются переведенные шаблоны URL-адресов, то в запрошенном URL-адресе отыскивается префикс языка, чтобы определить текущий язык.
2. Если языковой префикс не найден, то в сеансе текущего пользователя отыскивается существующий языковой сеансовый ключ, `LANGUAGE_SESSION_KEY`.
3. Если язык в сеансе не задан, то отыскивается существующий cookie-файл с текущим языком. Конкретно-прикладное имя этого cookie-файла можно указать в настроичном параметре `LANGUAGE_COOKIE_NAME`. По умолчанию cookie-файл имеет имя `django_language`.
4. Если cookie-файл не найден, то отыскивается HTTP-заголовок `Accept-Language` запроса.
5. Если в заголовке `Accept-Language` язык не указан, то Django использует язык, указанный в настроичном параметре `LANGUAGE_CODE`.

По умолчанию Django будет использовать язык, указанный в настроичном параметре `LANGUAGE_CODE`, при условии что не используется промежуточный компонент `LocaleMiddleware`. Описанный здесь процесс применим только при использовании этого компонента фреймворка промежуточных программных компонентов.

Подготовка проекта к интернационализации

Давайте подготовим проект под использование разных языков. Вы создадите английскую и испанскую версии своего магазина. Отредактируйте файл `settings.py` проекта, добавив в него следующий ниже настроечный параметр `LANGUAGES`. Поместите его рядом с настроичным параметром `LANGUAGE_CODE`:

```
LANGUAGES = [  
    ('en', 'English'),  
    ('es', 'Spanish'),  
]
```

Параметр `LANGUAGES` содержит два двухэлементных кортежа, состоящих из кода и названия языка. Коды языков могут быть специфичными для региона, например `en-us` или `en-gb`, либо обобщенными, например `en`. С помощью этого параметра указывается, что приложение будет доступно только на английском и испанском языках. Если не определить конкретно-прикладной параметр `LANGUAGES`, то сайт будет доступен на всех языках, на которые переведен Django.

Приведите настроичный параметр `LANGUAGE_CODE` к следующему виду:

```
LANGUAGE_CODE = 'en'
```

Добавьте `'django.middleware.locale.LocaleMiddleware'` в настроичный параметр `MIDDLEWARE`. Проверьте, чтобы этот параметр стоял после компонента `SessionMiddleware`, поскольку для компонента `LocaleMiddleware` необходимы сеансовые данные. Его также необходимо разместить перед промежуточным компонентом `CommonMiddleware`, потому что последнему нужен активный язык для конвертирования запрошенного URL-адреса. Теперь настроичный параметр `MIDDLEWARE` должен выглядеть следующим образом:

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.locale.LocaleMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```



Порядок следования классов фреймворка промежуточных программных компонентов очень важен, потому что каждый промежуточный компонент может зависеть от данных, устанавливаемых другим промежуточным компонентом, который был исполнен ранее. Промежуточный компонент применяется для запросов в порядке их появления в настроичном параметре MIDDLEWARE и для ответов в обратном порядке.

Внутри главного каталога проекта рядом с файлом `manage.py` создайте следующую ниже каталожную структуру:

```
locale/  
  en/  
  es/
```

Каталог `locale` – это место, где будут находиться файлы сообщений в приложении. Снова откройте файл `settings.py` и добавьте в него следующий ниже настроичный параметр:

```
LOCALE_PATHS = [  
    BASE_DIR / 'locale',  
]
```

Параметр `LOCALE_PATHS` задает каталоги, в которых Django должен искать файлы перевода. Пути `locale`, которые появляются первыми, имеют наивысший приоритет.

При использовании команд `makemessages` из каталога проекта файлы сообщений будут генерироваться в созданном вами пути `locale/`. Однако для приложений, содержащих каталог `locale/`, файлы сообщений будут генерироваться в своем каталоге `locale/`.

Перевод исходного кода Python

Для того чтобы перевести строковые литералы в исходном коде Python, можно пометить строковые литералы для перевода с помощью включенной в `django.utils.translation` функции `gettext()`. Эта функция переводит сообщение и возвращает строковый литерал. По традиции указанная функция импортируется как более короткий псевдоним с именем `_` (символ подчеркивания).

Вся документация о переводах находится на странице <https://docs.djangoproject.com/en/4.1/topics/i18n/translation/>.

Стандартные переводы

Следующий ниже исходный код показывает, как помечать строковый литерал для перевода:

```
from django.utils.translation import gettext as _  
output = _('Text to be translated.')
```

Ленивые переводы

Django содержит ленивые версии всех своих переводных функций, которые имеют суффикс `_lazy()`. При использовании ленивых функций строковые литералы переводятся при доступе к значению, а не при вызове функции (поэтому они переводятся **лениво**). Функции ленивого перевода пригодятся, когда помеченные для перевода строковые литералы будут находиться в путях, которые задействуются при загрузке модулей.



Использование `gettext_lazy()` вместо `gettext()` означает, что строковые литералы переводятся при доступе к значению. Django предлагает ленивую версию всех переводных функций.

Переводы с переменными

Помеченные для перевода строковые литералы могут содержать местозаполнители, чтобы вставлять в переводы переменные.

Следующий ниже исходный код является примером строкового литерала для перевода с местозаполнителем:

```
from django.utils.translation import gettext as _  
month =_('April')  
day = '14'  
output = _('Today is %(month)s %(day)s') % {'month': month,  
                                             'day': day}
```

Используя местозаполнители, можно переупорядочивать текстовые переменные. Например, английский перевод приведенного выше примера будет *Today is April 14*, а испанский – *Hoy es 14 de Abril*. Если в переводном строковом литерале есть более одного параметра, то следует всегда придерживаться строковой интерполяции вместо позиционной. Благодаря этому можно переупорядочивать текст с местозаполнителями.

Формы множественного числа в переводах

Для форм множественного числа можно использовать функции `ngettext()` и `ngettext_lazy()`. Эти функции переводят формы единственного и множественного числа в зависимости от аргумента, указывающего число объектов. В следующем ниже примере показано, как их использовать:

```
output = ngettext('there is %(count)d product',
                  'there are %(count)d products',
                  count) % {'count': count}
```

Теперь, когда вы знаете основы перевода строковых литералов в своем исходном коде Python, самое время применить переводы к проекту.

Перевод собственного исходного кода

Отредактируйте файл `settings.py` проекта, импортировав функцию `gettext_lazy()` и изменив настроечный параметр `LANGUAGES`, как показано ниже, чтобы переводить названия языков:

```
from django.utils.translation import gettext_lazy as _
# ...

LANGUAGES = [
    ('en', _('English')),
    ('es', _('Spanish')),
]
```

Здесь вместо функции `gettext()` используется функция `gettext_lazy()`, чтобы избежать циклического импорта, тем самым переводя названия языков при доступе к ним.

Откройте оболочку и выполните следующую команду из каталога проекта:

```
django-admin makemessages --all
```

Вы должны увидеть такой результат:

```
processing locale es
processing locale en
```

Взгляните на каталог `locale/`. Вы должны увидеть файловую структуру, подобную следующей ниже:

```
en/
LC_MESSAGES/
    django.po
es/
LC_MESSAGES/
    django.po
```

Для каждого языка создан файл сообщений .po. С помощью текстового редактора откройте es/LC_MESSAGES/django.po. В конце файла вы увидите следующее:

```
#: myshop/settings.py:118
msgid "English"
msgstr ""
#: myshop/settings.py:119
msgid "Spanish"
msgstr ""
```

Каждому переводному строковому литералу предшествует комментарий, показывающий детальную информацию о файле и строковый литерал, в котором он был найден. Каждый перевод включает два строковых литерала:

- **msgid**: переводной строковый литерал в том виде, в каком он представлен в исходном коде;
- **msgstr**: языковой перевод, который по умолчанию пуст. Здесь нужно ввести фактический перевод данного строкового литерала.

Заполните переводы msgstr данного строкового литерала msgid, как показано ниже:

```
#: myshop/settings.py:118
msgid "English"
msgstr "Inglés"
#: myshop/settings.py:119
msgid "Spanish"
msgstr "Español"
```

Сохраните видоизмененный файл сообщения, откройте оболочку и выполните такую команду:

```
django-admin compilemessages
```

Если все пройдет хорошо, то вы должны увидеть результат, подобный следующему ниже:

```
processing file django.po in myshop/locale/en/LC_MESSAGES
processing file django.po in myshop/locale/es/LC_MESSAGES
```

Указанный результат показывает информацию о скомпилированных файлах сообщений. Еще раз взгляните на каталог `locale` проекта `myshop`. Вы увидите показанные далее файлы:

```
en/
LC_MESSAGES/
    django.mo
    django.po
es/
LC_MESSAGES/
    django.mo
    django.po
```

Как видите, для каждого языка был скомпилирован файл сообщений `.mo`.

Вы перевели названия языков. Теперь давайте переведем имена полей модели, которые отображаются на сайте. Отредактируйте файл `models.py` приложения `orders`, добавив помеченные для перевода имена в поля модели `Order`, 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)
    # ...
```

Вы добавили имена для полей, которые отображаются, когда пользователь размещает новый заказ. Это поля `first_name`, `last_name`, `email`, `address`, `postal_code` и `city`. Напомним, что для именования полей также можно использовать атрибут `verbose_name`.

Внутри каталога приложения `orders` создайте следующую ниже каталогную структуру:

```
locale/
    en/
    es/
```

При создании каталога `locale` переводные строковые литералы этого приложения будут храниться в файле сообщений в этом каталоге, а не в главном файле сообщений. Благодаря этому можно генерировать отдельные файлы перевода по каждому приложению.

Откройте оболочку из каталога проекта и выполните следующую ниже команду:

```
django-admin makemessages --all
```

Вы должны увидеть такой результат:

```
processing locale es
processing locale en
```

С помощью текстового редактора откройте файл `locale/es/LC_MESSAGES/django.po` приложения `orders`. Вы увидите переводные строковые литералы из модели `Order`. Заполните следующие ниже переводы `msgstr` для строковых литералов `msgid`:

```
#: orders/models.py:12
msgid "first name"
msgstr "nombre"
#: orders/models.py:14
msgid "last name"
msgstr "apellidos"
#: orders/models.py:16
msgid "e-mail"
msgstr "e-mail"
#: orders/models.py:17
msgid "address"
msgstr "dirección"
#: orders/models.py:19
msgid "postal code"
msgstr "código postal"
#: orders/models.py:21
msgid "city"
msgstr "ciudad"
```

Закончив добавлять переводы, сохраните файл.

Помимо текстового редактора, для редактирования переводов еще можно использовать Poedit. Poedit – это программный продукт, служащий для редактирования переводов, в котором используется инструментарий gettext. Он доступен для Linux, Windows и macOS. Программу Poedit можно скачать со страницы с <https://poedit.net/>.

Давайте также переведем формы проекта. Перевод формы `OrderCreateForm` приложения `orders` не требуется. Это связано с тем, что в ней используется класс `ModelForm` и в нем в качестве меток полей формы применяется атрибут `verbose_name` полей модели `Order`. Вы займитесь переводом форм приложений `cart` и `coupons`.

Откройте в редакторе файл `forms.py` внутри каталога приложения `cart` и добавьте атрибут `label` в поле `quantity` формы `CartAddProductForm`. Затем пометьте это поле для перевода, как показано ниже:

```
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'))
    override = forms.BooleanField(required=False,
                                  initial=False,
                                  widget=forms.HiddenInput)
```

Откройте файл `forms.py` приложения `coupons` и переведите 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'))
```

Вы добавили метку в поле `code` и пометили его для перевода.

Перевод шаблонов

Django предлагает теги `{% trans %}` и `{% blocktrans %}` для перевода строковых литералов в шаблонах. Для того чтобы использовать шаблонные теги перевода, необходимо в верхнюю часть шаблона добавить тег `{% load i18n %}`, который будет их загружать.

Шаблонный тег { % trans % }

Шаблонный тег { % trans % } позволяет помечать строковый литерал для перевода. На внутреннем уровне Django выполняет функцию `gettext()` с заданным текстом в качестве параметра. Вот как строковый литерал помечается в шаблоне:

```
{% trans "Text to be translated" %}
```

Притом можно использовать ключевое слово `as`, чтобы сохранять переведенный контент в переменной, которую можно использовать в своем шаблоне.

В следующем ниже примере переведенный текст сохраняется в переменной с именем `greeting`:

```
{% trans "Hello!" as greeting %}
<h1>{{ greeting }}</h1>
```

Тег { % trans % } удобен для простых переводных строковых литералов, но он не способен обрабатывать переводной контент, содержащий переменные.

Шаблонный тег { % blocktrans % }

Шаблонный тег { % blocktrans % } позволяет помечать контент, содержащий строковые литералы и переменные, используя местозаполнители. В следующем ниже примере показано, как использовать тег { % blocktrans % }, содержащий переменную `name` в контенте для перевода:

```
{% blocktrans %}Hello {{ name }}!{% endblocktrans %}
```

Притом можно использовать ключевое слово `with`, чтобы вставлять шаблонные выражения, такие как доступ к объектным атрибутам или применение шаблонных фильтров к переменным. Для них всегда следует использовать местозаполнители. Вместе с тем ни к выражениям, ни к объектным атрибутам внутри блока `blocktrans` обратиться невозможно. В следующем ниже примере показано, как использовать `with` для вставки объектного атрибута, к которому был применен фильтр `capfirst`:

```
{% blocktrans with name=user.name|capfirst %}
Hello {{ name }}!
{% endblocktrans %}
```



Используйте тег `{% blocktrans %}` вместо тега `{% trans %}`, когда в переводной строковый литерал нужно вставлять переменный контент.

Перевод шаблонов магазина

Откройте шаблон `shop/base.html` приложения `shop`. Проверьте, чтобы вверху шаблона был загружен тег `i18n` и были помечены переводные строковые литералы, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
{% 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 total_items > 0 %}
                {% trans "Your cart" %}:
                <a href="{% url "cart:cart_detail" %}">
                    {% blocktrans with total=cart.get_total_price count items=total_items %}
                        {% plural %}
                            {% items %} item, ${% total %}
                        {% plural %}
                            {% items %} items, ${% total %}
                        {% endblocktrans %}
                </a>
            {% elif not order %}
                {% trans "Your cart is empty." %}
            {% endif %}
        {% endwith %}
```

```
</div>
</div>
<div id="content">
    {% block content %}
    {% endblock %}
</div>
</body>
</html>
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк.

Обратите внимание на тег `{% blocktrans %}` для отображения сводной информации о корзине. Сводная информация о корзине ранее была следующей:

```
{{ total_items }} item{{ total_items|pluralize }},
${{ cart.get_total_price }}
```

Он был изменен, и теперь используется тег `{% blocktrans with ... %}`, чтобы установить местозаполнитель `total` со значением `cart.get_total_price` (здесь вызывается объектный метод). Кроме того, используется ключевое слово `count`, которое позволяет устанавливать переменную для подсчета объектов, чтобы Django мог выбирать правильную форму множественного числа. Далее для подсчета объектов со значением из `total_items` задается переменная `items`.

Такой подход позволяет задавать перевод для форм единственного и множественного числа, которые отделяются тегом `{% plural %}` в блоке `{% blocktrans %}`. Результирующий исходный код таков:

```
{% blocktrans with total=cart.get_total_price count items=total_items %}
    {{ items }} item, ${{ total }}
    {% plural %}
        {{ items }} items, ${{ total }}
    {% endblocktrans %}
```

Далее отредактируйте шаблон `shop/product/detail.html` приложения `shop`, загрузив теги `i18n` вверху шаблона, но после тега `{% extends %}`, который всегда должен быть первым тегом в шаблоне:

```
{% extends "shop/base.html" %}
{% load i18n %}
{% load static %}
...
```

Затем найдите следующую ниже строку:

```
<input type="submit" value="Add to cart">
```

и замените ее такой:

```
<input type="submit" value="<% trans "Add to cart" %>">
```

Затем найдите строку

```
<h3>People who bought this also bought</h3>
```

и замените ее следующей:

```
<h3>{% trans "People who bought this also bought" %}</h3>
```

Теперь переведите шаблон приложения `orders`. Отредактируйте шаблон `orders/order/create.html` приложения `orders`, пометив текст перевода, как показано ниже:

```
{% 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 class="neg">- ${{ cart.get_discount|floatformat:2 }}</span>
                </li>
            {% endif %}
        </ul>
        <p>{% trans "Total" %}: ${{ cart.get_total_price_after_discount|floatformat:2 }}</p>
    </div>
```

```
<form method="post" class="order-form">
{{ form.as_p }}
<p><input type="submit" value="{% trans "Place order" %}"></p>
{{ csrf_token }}
</form>
{% endblock %}
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк. Взгляните на следующие ниже файлы в исходном коде, сопровождающем эту главу, чтобы увидеть, как строковые литералы помечены для перевода:

- приложение `shop`: шаблон `shop/product/list.html`;
- приложение `orders`: шаблон `orders/order/pdf.html`;
- приложение `cart`: шаблон `cart/detail.html`;
- приложение `payments`: шаблоны `payment/process.html`, `payment/completed.html` и `payment/canceled.html`.

Напомним, что исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-Example/tree/master/Chapter11>.

Давайте обновим файлы сообщений, чтобы включить новые переводные строковые литералы. Откройте оболочку и выполните следующую ниже команду:

```
django-admin makemessages --all
```

Файлы `.po` находятся в каталоге `locale` проекта `myshop`, и вы увидите, что теперь приложение `orders` содержит все строковые литералы, которые вы пометили для перевода.

Отредактируйте переводные файлы `.po` проекта и приложения `orders` и включите испанские переводы в `msgstr`. Также можно воспользоваться переведенными файлами `.po` в прилагаемом к этой главе исходном коде.

Выполните следующую ниже команду, чтобы скомпилировать файлы перевода:

```
django-admin compilemessages
```

Вы увидите такой результат:

```
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
```

Для каждого файла перевода `.po` был сгенерирован файл `.mo`, содержащий скомпилированные переводы.

Использование интерфейса перевода Rosetta

Rosetta – это стороннее приложение, которое позволяет редактировать переводы, используя тот же интерфейс, что и встроенный в Django сайт администрирования. Приложение Rosetta упрощает редактирование файлов .po и обновляет скомпилированные файлы перевода. Давайте добавим его в проект.

Следующей ниже командой установите приложение Rosetta посредством pip:

```
pip install django-rosetta==0.9.8
```

Затем добавьте 'rosetta' в настроечный параметр INSTALLED_APPS в файле settings.py проекта, как показано ниже:

```
INSTALLED_APPS = [
    # ...
    'rosetta',
]
```

Необходимо добавить URL-адреса приложения Rosetta в главную конфигурацию URL-адресов. Отредактируйте главный файл urls.py проекта, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [
    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')),
]
```

Проверьте, чтобы он был помещен перед шаблоном shop.urls во избежание нежелательного совпадения с шаблоном.

Откройте <http://127.0.0.1:8000/admin/> и войдите в систему с правами суперпользователя. Затем пройдите по URL-адресу <http://127.0.0.1:8000/rosetta/> в своем браузере. В меню Filter (Фильтр) кликните по **THIRD PARTY** (Сторонние файлы), чтобы отобразить все доступные файлы сообщений, включая те, которые принадлежат приложению orders.

Вы должны увидеть список существующих языков, как показано ниже:

The screenshot shows the Rosetta admin interface. At the top, there's a navigation bar with 'Home > Language selection'. Below it is a filter bar with buttons for 'PROJECT' (highlighted in yellow), 'THIRD PARTY', 'DJANGO', and 'ALL'. The main area is divided into two sections: 'English' and 'Spanish'. Each section has a table with columns: APPLICATION, PROGRESS, MESSAGES, TRANSLATED, FUZZY (with a question mark icon), OBSOLETE, and FILE. Under 'English', there are entries for 'Myshop' (0% progress, 37 messages, 0 translated, 0 fuzzy, 0 obsolete, file django.po) and 'Orders' (0% progress, 23 messages, 0 translated, 0 fuzzy, 0 obsolete, file django.po). Under 'Spanish', there are entries for 'Myshop' (100% progress, 37 messages, 37 translated, 0 fuzzy, 0 obsolete, file django.po), 'Orders' (100% progress, 23 messages, 22 translated, 0 fuzzy, 0 obsolete, file django.po), and 'Rosetta' (100% progress, 42 messages, 42 translated, 0 fuzzy, 0 obsolete, file python3.10/site-packages/rosetta/locale/es/LC_MESSAGES/django.po).

Рис. 11.1. Интерфейс администрирования Rosetta

Кликните по ссылке **Myshop** (Мой магазин) в разделе **Spanish** (Испанский), чтобы отредактировать перевод на испанский язык. Вы должны увидеть список переводных строковых литералов, как показано ниже:

The screenshot shows the Rosetta interface for translating strings into Spanish. At the top, there's a search bar with a magnifying glass icon and a 'Go' button. To the right is a 'Display' button followed by four options: 'UNTRANSLATED ONLY' (highlighted in yellow), 'TRANSLATED ONLY', 'FUZZY ONLY', and 'ALL'. Below this is a table with two columns: 'ORIGINAL' and 'SPANISH'. Each row contains a string from the original language and its corresponding translation in Spanish, with a dropdown arrow icon next to each input field. To the right of the table is a column titled 'FUZZY ? OCCURRENCES(S)' with a checkbox and a list of file paths where each string appears. The rows are: 'Quantity' (Original: 'Cantidad', Spanish: 'Cantidad'), 'Your shopping cart' (Original: 'Su carro', Spanish: 'Su carro'), 'Image' (Original: 'Imagen', Spanish: 'Imagen'), 'Product' (Original: 'Producto', Spanish: 'Producto'), 'Remove' (Original: 'Eliminar', Spanish: 'Eliminar'), 'Unit price' (Original: 'Precio unitario', Spanish: 'Precio unitario'), 'Price' (Original: 'Precio', Spanish: 'Precio'), and 'Update' (Original: 'Actualizar', Spanish: 'Actualizar').

Рис. 11.2. Редактирование перевода строковых литералов на испанский язык с помощью Rosetta

Переводы вводятся в столбце **SPANISH** (Испанский). Столбец **OCCURRENCE(S)** (Появление(я)) показывает файлы и строки исходного кода, в которых был найден каждый строковый литерал перевода.

Переводы, содержащие местозаполнители, будут отображаться следующим образом:

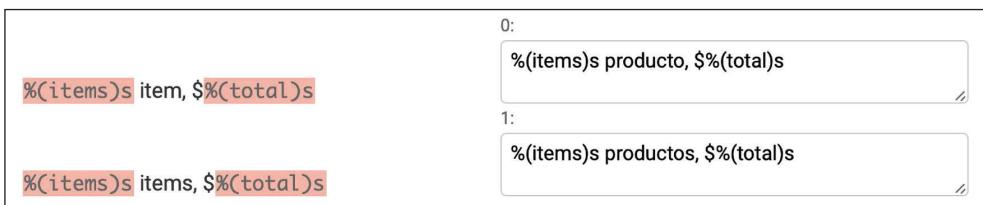


Рис. 11.3. Переводы с местозаполнителями

В приложении Rosetta используется другой цвет фона для отображения местозаполнителей. При переводе контента проверьте, чтобы заполнители не переводились. Например, возьмем следующий ниже строковый литерал:

```
%(items)s items, %(total)s
```

На испанский язык его можно перевести так:

```
%(items)s productos, %(total)s
```

Можете обратиться к прилагаемому к этой главе исходному коду, чтобы применить те же испанские переводы и для своего проекта.

Когда закончите редактирование переводов, кликните по кнопке **Save and translate next block** (Сохранить и перевести следующий блок), чтобы сохранить переводы в файл .po. Приложение Rosetta компилирует файл сообщений при сохранении переводов, поэтому выполнять команду `compilemessages` не требуется. Однако для записи файлов сообщений Rosetta требует доступа с правом записи в каталоги `locale`. Проверьте, чтобы каталоги имели соответствующие разрешения.

Если вы хотите, чтобы другие пользователи могли редактировать переводы, то пройдите по URL-адресу `http://127.0.0.1:8000/admin/auth/group/add/` в своем браузере и создайте новую группу с именем `translators`. Затем зайдите на `http://127.0.0.1:8000/admin/auth/user/`, чтобы изменить пользователей, которым вы хотите предоставить разрешения, и дать им возможность редактировать переводы. При редактировании пользователя в разделе **Permissions** (Разрешения) добавьте группу `translators` в **Chosen Groups** (Выбранные группы) по каждому пользователю. Приложение Rosetta доступно только суперпользователям либо пользователям, входящим в группу `translators`.

Документацию Rosetta можно почитать на странице <https://django-rosetta.readthedocs.io/>.



Если при добавлении новых переводов в производственную среду вы раздаете Django с реальным веб-сервером, то для того чтобы любые изменения вступали в силу, вам придется перезагружать сервер после выполнения команды `compilemessages` либо после сохранения переводов с помощью приложения Rosetta.

При редактировании переводов перевод может быть помечен как *нечеткий*. Давайте посмотрим, что из себя представляют нечеткие переводы.

Нечеткие переводы

При редактировании переводов в приложении Rosetta вы увидите столбец **FUZZY** (Нечеткий). Эта функциональность не принадлежит приложению Rosetta; она предоставляется инструментарием gettext. Если флаг **FUZZY** для перевода активен, то он не будет включаться в скомпилированные файлы сообщений. Этот флаг помечает строковые литералы перевода, которые должны быть проверены переводчиком. Когда файлы .po обновляются новыми переводными строковыми литералами, возможно, что некоторые из них будут автоматически помечены как нечеткие. Это происходит, когда инструментарий gettext находит слегка видоизмененный `msgid`. Инструментарий gettext сопоставляет его со старым переводом и помечает как нечеткий для дальнейшей проверки. Затем переводчик должен просмотреть нечеткие переводы, снять флаг **FUZZY** и снова скомпилировать файл перевода.

Шаблоны URL-адресов для интернационализации

Django предлагает возможности интернационализации URL-адресов. Они предусматривают две главные функциональности для интернационализированных URL-адресов:

- **префикс языка в шаблонах URL-адресов:** добавление префикса языка в URL-адреса для раздачи каждой языковой версии под другим базовым URL-адресом;
- **переведенные шаблоны URL-адресов:** перевод шаблонов URL-адресов таким образом, чтобы каждый URL-адрес по каждому языку отличался.

Одной из причин перевода URL-адресов является оптимизация сайта под поисковые системы. Добавив языковой префикс в свои шаблоны, можно индексировать URL-адрес на каждом языке вместо использования одного URL-адреса для всех. Кроме того, переведя URL-адреса на каждый язык, по-

исковым системам предоставляются URL-адреса, которые будут лучше ранжироваться на каждом языке.

Добавление префикса языка в шаблоны URL-адресов

Django позволяет добавлять префикс языка в шаблоны URL-адресов. Например, англоязычная версия сайта может размещаться по пути, начинающемуся с /en/, а испанская версия – начинающемуся с /es/. В целях применения языков в шаблонах URL-адресов необходимо использовать предоставляемый веб-фреймворком Django класс `LocaleMiddleware`. Django будет использовать его для идентификации текущего языка по запрошенному URL-адресу. Ранее он был добавлен в настроечный параметр `MIDDLEWARE` проекта, так что это не нужно делать сейчас.

Давайте добавим префикс языка в шаблоны URL-адресов. Отредактируйте главный файл `urls.py` проекта `myshop`, добавив `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')),
)
```

Непереводимые стандартные шаблоны URL-адресов и шаблоны можно комбинировать в рамках `i18n_patterns`, чтобы некоторые шаблоны содержали языковой префикс, а другие – нет. Однако лучше использовать только переведенные URL-адреса, чтобы избежать возможности совпадения небрежно переведенного URL-адреса с непереведенным шаблоном URL-адреса.

Запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере. Django выполнит шаги, описанные в разделе «Как Django определяет текущий язык», чтобы определить текущий язык, и перенаправит на запрошенный URL-адрес, содержащий префикс языка. Взгляните на URL-адрес в своем браузере; теперь он должен выглядеть как `http://127.0.0.1:8000/en/`. Текущий язык – это тот, который установлен в заголовке `Accept-Language` браузера, если это испанский либо английский язык; в противном случае это применяемый по умолчанию `LANGUAGE_CODE` (английский), заданный в настроечных параметрах.

Перевод шаблонов URL-адресов

Django поддерживает переведенные строковые литералы в шаблонах URL-адресов. В одном шаблоне URL-адреса можно использовать другой перевод на каждый язык. Шаблоны URL-адресов можно помечать для перевода так же, как и строковые литералы, используя функцию `gettext_lazy()`.

Отредактируйте главный файл `urls.py` проекта `myshop`, добавив переводные строковые литералы в регулярные выражения шаблонов URL-адресов 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`, пометив шаблон URL-адреса `order_create` для перевода следующим образом:

```
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    path(_('create/'), views.order_create, name='order_create'),
    # ...
]
```

Отредактируйте файл `urls.py` приложения `payment`, поменяв исходный код следующим образом:

```
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'),
    path('webhook/', webhooks.stripe_webhook, name='stripe-webhook'),
]
```

Обратите внимание, что эти шаблоны URL-адресов будут содержать префикс языка, поскольку они включены в `i18n_patterns()` в главном файле `urls`.

ру проекта. Это приведет к тому, что каждый шаблон URL-адреса будет иметь свой URI-идентификатор для каждого доступного языка, один из которых будет начинаться с `/en/`, другой – с `/es/` и т. д. Однако требуется единственный URL-адрес, по которому Stripe будет уведомлять о событиях, и нужно избегать языковых префиксов в URL-адресе `webhook`.

Удалите шаблон URL-адреса `webhook` из файла `urls.py` приложения `payment`. Теперь файл должен выглядеть следующим образом:

```
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'),
]
```

Затем добавьте приведенный ниже шаблон URL-адреса `webhook` в главный файл `urls.py` проекта `myshop`. Новый исходный код выделен жирным шрифтом:

```
from django.utils.translation import gettext_lazy as _
from payment import webhooks

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')),
)

urlpatterns += [
    path('payment/webhook/', webhooks.stripe_webhook,
         name='stripe-webhook'),
] 

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                         document_root=settings.MEDIA_ROOT)
```

Мы добавили шаблон URL-адреса `webhook` в шаблоны URL-адресов за пределами `i18n_patterns()` с целью обеспечения поддержки единственного URL-адреса для уведомлений о событиях Stripe.

Переводить шаблоны URL-адресов приложения `shop` не потребуется, поскольку они сформированы с использованием переменных и не содержат никаких других строковых литералов.

Откройте оболочку и выполните следующую ниже команду, чтобы обновить файлы сообщений новыми переводами:

```
django-admin makemessages --all
```

С помощью указанной ниже команды обеспечьте, чтобы сервер разработки был запущен:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/en/rosetta/` в своем браузере и кликните по ссылке **Myshop** в разделе **Spanish** (Испанский). Кликните по **UNTRANSLATED ONLY** (Только непереведенные), чтобы увидеть только те строковые литералы, которые еще не были переведены. Теперь вы увидите шаблоны URL-адресов для перевода, как показано на рис. 11.4:

The screenshot shows the Rosetta interface for translating URLs. At the top, there's a search bar and a 'Go' button. Below that, a 'Display' dropdown menu is set to 'UNTRANSLATED ONLY', with other options like 'TRANSLATED ONLY', 'FUZZY ONLY', and 'ALL' available. The main area lists URL patterns in 'ORIGINAL' and 'SPANISH' columns, each with a checkbox for 'FUZZY' and a link to the file and line number where it appears. The patterns listed are: cart/, orders/, payment/, process/, and completed/. At the bottom, it says 'Displaying: 5/44 messages' and has a 'SAVE AND TRANSLATE NEXT BLOCK' button.

ORIGINAL	SPANISH	FUZZY	OCCURRENCES(S)
cart/		<input type="checkbox"/>	myshop/urls.py:26
orders/		<input type="checkbox"/>	myshop/urls.py:27
payment/		<input type="checkbox"/>	myshop/urls.py:28
process/		<input type="checkbox"/>	payment/urls.py:9
completed/		<input type="checkbox"/>	payment/urls.py:10

Рис. 11.4. Шаблоны URL-адресов для перевода в интерфейсе приложения Rosetta

Добавьте другой строковый литерал перевода для каждого URL-адреса. Не забудьте вставить символ косой черты / в конце каждого URL-адреса, как показано на рис. 11.5:

The screenshot shows the Rosetta application interface for translating URL templates into Spanish. The top navigation bar includes a search bar, a 'Go' button, and filter buttons for 'UNTRANSLATED ONLY', 'TRANSLATED ONLY', 'FUZZY ONLY', and 'ALL'. The main table lists URL patterns and their Spanish equivalents, along with occurrence counts and file paths.

ORIGINAL	SPANISH	FUZZY	OCCURRENCES(S)
cart/	carro/	<input type="checkbox"/>	myshop/urls.py:26
orders/	pedidos/	<input type="checkbox"/>	myshop/urls.py:27
payment/	pago/	<input type="checkbox"/>	myshop/urls.py:28
process/	procesar/	<input type="checkbox"/>	payment/urls.py:9
completed/	completado/	<input type="checkbox"/>	payment/urls.py:10

Displaying: 5/44 messages SAVE AND TRANSLATE NEXT BLOCK

Рис. 11.5. Испанские переводы шаблонов URL-адресов в интерфейсе приложения Rosetta

Когда закончите, кликните по **SAVE AND TRANSLATE NEXT BLOCK** (Сохранить и перевести следующий блок).

Затем кликните по **FUZZY ONLY** (Только нечеткие). Вы увидите переводы, которые были помечены как нечеткие, потому что они были связаны со старым переводом аналогичного исходного строкового литерала. В случае, показанном на рис. 11.6, переводы неверны и должны быть исправлены:

The screenshot shows the Rosetta application interface for translating URL templates into Spanish. The top navigation bar includes a search bar, a 'Go' button, and filter buttons for 'UNTRANSLATED ONLY', 'TRANSLATED ONLY', 'FUZZY ONLY', and 'ALL'. The main table lists URL patterns and their Spanish equivalents, with checkboxes indicating they are fuzzy matches. The 'Fuzzy Only' filter is selected.

ORIGINAL	SPANISH	FUZZY	OCCURRENCES(S)
coupons/	Cupón	<input checked="" type="checkbox"/>	myshop/urls.py:29
canceled/	Pago cancelado	<input checked="" type="checkbox"/>	payment/urls.py:11

Displaying: 2/44 messages SAVE AND TRANSLATE NEXT BLOCK

Рис. 11.6. Нечеткие переводы в интерфейсе приложения Rosetta

Ведите правильный текст для нечетких переводов. При вводе нового текста перевода приложение Rosetta автоматически снимет флагок **FUZZY**. Когда закончите, кликните по **SAVE AND TRANSLATE NEXT BLOCK** (Сохранить и перевести следующий блок):

Рис. 11.7. Исправление нечетких переводов в интерфейсе приложения Rosetta

Теперь можно вернуться к адресу `http://127.0.0.1:8000/en/rosetta/files/third-party/` и так же отредактировать испанский перевод приложения `orders`.

Переключение языка сайта

Поскольку контент сайта раздается на нескольких языках, необходимо предоставить пользователям возможность переключать язык сайта. Вы добавите селектор языка на сайт. Селектор языка будет состоять из списка имеющихся языков, которые будут отображаться с помощью ссылок.

Откройте шаблон `shop/base.html` приложения `shop` и найдите следующие ниже строки:

```
<div id="header">
  <a href="/" class="logo">{% trans "My shop" %}</a>
</div>
```

Замените их таким исходным кодом:

```
<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>
```

```

</a>
</li>
[% endfor %]
</ul>
</div>
</div>

```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк.

Вот как формируется селектор языка.

1. С помощью тега `{% load i18n %}` загружаются теги интернационализации.
2. При помощи тега `{% get_current_language %}` берется текущий язык.
3. Путем тега `{% get_available_languages %}` извлекаются языки, определенные в настроечном параметре `LANGUAGES`.
4. Посредством тега `{% get_language_info_list %}` обеспечивается легкий доступ к атрибутам языка.
5. Для отображения всех имеющихся языков создается HTML-список, и к текущему активному языку добавляется атрибут `class="selected"`.

В исходном коде, связанном с выбором языка, использованы предоставленные в `i18n` шаблонные теги, созданные на основе языков, имеющихся в настройках проекта. Теперь пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере и посмотрите. Вы должны увидеть селектор языка в правом верхнем углу сайта, как показано ниже:

The screenshot shows a web page titled "Mi tienda". At the top right, there is a language selector with "Language: English Español". Below it, a message says "Tu carro está vacío.". The main content area is titled "Productos". On the left, there is a sidebar with "Categorías" and two buttons: "Todos" (which is highlighted in blue) and "Té". The main content area displays three products: "Green tea" (\$30.00), "Red tea" (\$45,50), and "Tea powder" (\$21,20). Each product has an image and a brief description.

Product	Description	Price
Green tea	Photo of a white bowl filled with green tea leaves.	\$30.00
Red tea	Photo of a glass teapot and a small brown teapot.	\$45,50
Tea powder	Photo of a white bowl filled with green tea powder.	\$21,20

Рис. 11.8. Страница списка товаров, содержащая селектор языка в шапке сайта



Изображения в этой главе:

- Зеленый чай: фото Цзя Е на Unsplash;
- Красный чай: фото Манки Ким на Unsplash;
- Чайный порошок: фото Phuong Nguyen на Unsplash.

Теперь пользователи могут легко переключаться на предпочтительный язык, кликнув по нему.

Перевод моделей с помощью модуля django-parler

Django не предоставляет готового технического решения для перевода моделей. Для того чтобы управлять контентом, хранящимся на разных языках, необходимо реализовать свое собственное решение либо задействовать сторонний модуль для перевода модели. Несколько сторонних приложений позволяют переводить поля моделей. В каждом из них используется свой подход к хранению и доступу к переводам. Одним из таких приложений является модуль `django-parler`. Этот модуль предлагает очень эффективный способ перевода моделей и легко интегрируется со встроенным в Django сайтом администрирования.

Модуль `django-parler` по каждой модели создает отдельную таблицу базы данных, содержащую переводы. Эта таблица содержит все переведенные поля и внешний ключ исходного объекта, которому принадлежит перевод. Она также содержит поле языка, поскольку в каждой строке хранится содержимое для одного языка.

Установка модуля django-parler

Следующей ниже командой установите модуль `django-parler` посредством `pip`:

```
pip install django-parler==2.3
```

Отредактируйте файл `settings.py` проекта, добавив '`parler`' в настроечный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [
    # ...
    'parler',
]
```

Также добавьте в настроечные параметры следующий ниже исходный код:

```
# настройки django-parler
PARLER_LANGUAGES = {
    None: (
        {'code': 'en'},
```

```
        {'code': 'es'},
    ),
    'default': {
        'fallback': 'en',
        'hide_untranslated': False,
    }
}
```

Этот параметр определяет доступные для django-parler языки en и es. В нем задается язык, который будет применяться по умолчанию, en, и указывается, что django-parler не должен скрывать непереведенный контент.

Перевод полей моделей

Давайте добавим переводы в каталог товаров. Модуль `django-parler` предоставляет для перевода полей модели модельный класс `TranslatableModel` и обертку `TranslatedFields`.

Откройте в редакторе файл `models.py` внутри каталога приложения `shop` и добавьте следующую ниже инструкцию импорта:

```
from parler.models import TranslatableModel, TranslatedFields
```

Затем видоизмените модель `Category`, чтобы сделать поля `name` и `slug` переводимыми, как показано ниже:

```
class Category(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=200),
        slug = models.SlugField(max_length=200,
                               unique=True),
    )
```

Теперь модель Category наследует от TranslatableModel, а не от models.Model, а поля name и slug включены в обертку TranslatedFields.

Отредактируйте модель `Product`, добавив переводы полей `name`, `slug` и `description`, как показано ниже:

```

    on_delete=models.CASCADE)
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)

```

Модуль `django-parler` управляет переводами, генерируя еще одну модель для каждой переводимой модели. На следующей ниже схеме показаны поля модели `Product` и внешний вид сгенерированной модели `ProductTranslation`:

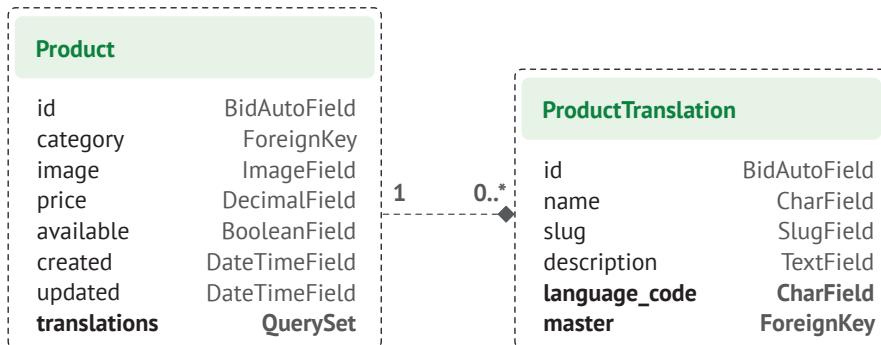


Рис. 11.9. Модель `Product` и связанная с ней модель `ProductTranslation`, сгенерированная модулем `django-parler`

Сгенерированная модулем `django-parler` модель `ProductTranslation` содержит переводимые поля `name`, `slug` и `description`, поле `language_code` и внешний ключ `ForeignKey` для основного объекта `Product`. Здесь имеется взаимосвязь один-ко-многим от `Product` к `ProductTranslation`. Объект `ProductTranslation` будет существовать для каждого доступного языка каждого объекта `Product`.

Поскольку Django использует отдельную таблицу для переводов, есть несколько функциональностей Django, которые невозможно использовать. Невозможно использовать заданный по умолчанию порядок сортировки по переведенному полю. Можно фильтровать по переведенным полям в запросах, но нельзя включать переводимое поле в Meta-опции `ordering`. Кроме того, нельзя использовать индексы для переведенных полей, так как этих полей не будет в изначальной модели – они будут находиться в модели перевода.

Отредактируйте файл `models.py` приложения `shop`, закомментировав атрибуты `ordering` и `indexes` Meta-класса внутри класса `Category`:

```

class Category(TranslatableModel):
    # ...
    class Meta:

```

```
# ordering = ['name']
# indexes = [
#     models.Index(fields=['name']),
# ]
verbose_name = 'category'
verbose_name_plural = 'categories'
```

Кроме того, необходимо закомментировать атрибут порядка сортировки `ordering` Meta-класса внутри класса `Product` и индексы, которые относятся к переведенным полям. Закомментируйте следующие ниже строки Meta-класса внутри класса `Product`:

```
class Product(TranslatableModel):
    ...
class Meta:
    # ordering = ['name']
    indexes = [
        # models.Index(fields=['id', 'slug']),
        # models.Index(fields=['name']),
        models.Index(fields=['-created']),
    ]
```

Подробнее о совместимости модуля `django-parler` с Django можно узнать на странице <https://django-parler.readthedocs.io/en/latest/compatibility.html>.

Интеграция переводов на сайт администрирования

Модуль `django-parler` легко интегрируется со встроенным в Django сайтом администрирования. Он включает в себя класс `TranslatableAdmin`, который переопределяет поставляемый с веб-фреймворком Django класс `ModelAdmin`, и служит для управления переводами моделей.

Отредактируйте файл `admin.py` приложения `shop`, добавив в него следующую ниже инструкцию импорта:

```
from parler.admin import TranslatableAdmin
```

Видоизмените классы `CategoryAdmin` и `ProductAdmin`, чтобы они наследовали от `TranslatableAdmin` вместо `ModelAdmin`. Модуль `django-parler` не поддерживает атрибут `prepopulated_fields`, однако поддерживает метод `get_prepopulated_fields()`, предоставляющий ту же функциональность. Давайте изменим это соответствующим образом. Отредактируйте файл `admin.py`, придав ему следующий вид:

```
from django.contrib import admin
from parler.admin import TranslatableAdmin
from .models import Category, Product

@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',)}
```

Вы адаптировали сайт администрирования для работы с новыми переведенными моделями. Теперь можно синхронизировать базу данных с внешними изменениями в модели.

Создание миграций для переводов моделей

Откройте оболочку и выполните следующую ниже команду, чтобы создать новую миграцию с учетом переводов модели:

```
python manage.py makemigrations shop --name "translations"
```

Вы увидите следующий ниже результат:

```
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 index shop_catego_name_289c7e_idx from category
    - Remove index shop_produc_id_f21274_idx from product
    - Remove index shop_produc_name_a2070e_idx from product
    - Remove field name from category
    - Remove field slug from category
```

- Remove field description from product
- Remove field name from product
- Remove field slug from product
- Add field master to producttranslation
- Add field master to categorytranslation
- Alter unique_together for producttranslation (1 constraint(s))
- Alter unique_together for categorytranslation (1 constraint(s))

Эта миграция автоматически вставляет модели `CategoryTranslation` и `ProductTranslation`, динамически созданные модулем `django-parler`. Важно отметить, что миграция удаляет из моделей существовавшие ранее поля. Это означает, что указанные данные будут утеряны и что после запуска сайта администрирования нужно будет снова задать на нем категории и товары.

Отредактируйте файл `migrations/0002_translations.py` приложения `shop`, заменив два появления строки

```
bases=(parler.models.TranslatedFieldsModelMixin, models.Model),
```

следующей ниже:

```
bases=(parler.models.TranslatableModel, models.Model),
```

Это исправление незначительной проблемы, обнаруженной в используемой версии модуля `django-parler`. Указанное изменение необходимо для предотвращения сбоя миграции при ее применении. Данная проблема связана с созданием переводов для существующих модельных полей и, вероятно, будет исправлена в новых версиях модуля `django-parler`.

Выполните следующую ниже команду, чтобы применить миграцию:

```
python manage.py migrate shop
```

Вы увидите результат, который заканчивается такой строкой:

```
Applying shop.0002_translations... OK
```

Теперь модели синхронизированы с базой данных.

С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/en/admin/shop/category/` в своем браузере. Вы увидите, что существующие категории потеряли свое имя и слаг вследствие удаления этих полей и использования вместо них переводимых моделей, генерированных модулем `django-parler`. Под каждым столбцом будет находиться тире, как на рис. 11.10:

	NAME	SLUG
<input type="checkbox"/>	-	-

1 category

Рис. 11.10. Список категорий на сайте администрирования после создания переводных моделей

Кликните по тире под названием категории, чтобы его отредактировать. Вы увидите, что страница **Change category** (Изменить категорию) содержит две разные вкладки, одну для английского и одну для испанского перевода:

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Shop › Categories › Tea

Change category (English)

HISTORY VIEW ON SITE >

English Spanish

Name: Tea

Slug: tea

Delete Save and add another Save and continue editing SAVE

Рис. 11.11. Форма редактирования категории, содержащая языковые вкладки, добавленные модулем django-parler

Проверьте, чтобы были введены имя и слаг всех существующих категорий. При редактировании категории введите данные на английском языке и кликните по **Save and continue editing** (Сохранить и продолжить редактирование). Затем кликните по **Spanish** (Испанский), добавьте испанский перевод полей и кликните по **SAVE** (Сохранить):

The screenshot shows a Django admin change form for a category. At the top, it says "Change category (Spanish)". There are two tabs: "English" (selected) and "Spanish". Below them are fields for "Name" (containing "Té") and "Slug" (containing "te"). At the bottom, there are buttons: "Delete" (red), "Save and add another" (blue), "Save and continue editing" (blue), and "SAVE" (blue).

Рис. 11.12. Испанский перевод формы для редактирования категории

Перед переключением между языковыми вкладками проверьте, чтобы изменения были сохранены.

После заполнения данных существующих категорий пройдите по URL-адресу `http://127.0.0.1:8000/en/admin/shop/product/` и отредактируйте каждый товар, указав имя на английском и испанском языках, слаг и описание.

Использование переводов с ORM-преобразователем

Далее необходимо адаптировать представления магазина под использование переводных наборов запросов. Выполните следующую ниже команду, чтобы открыть оболочку Python:

```
python manage.py shell
```

Давайте посмотрим, как извлекать и опрашивать переводные поля. С целью получения объекта с переводимыми полями, переведенными на конкретный язык, можно использовать функцию Django `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 предусматривает использование предоставляемого модулем `django-parler` менеджера `language()`, как вы видите ниже:

```
>>> product=Product.objects.language('en').first()
>>> product.name
'Green tea'
```

При доступе к переведенным полям они конвертируются с использованием текущего языка. При этом есть возможность задавать другой текущий язык, чтобы объект получал доступ к этому конкретному переводу, как показано далее:

```
>>> product.set_current_language('es')
>>> product.name
'Té verde'
>>> product.get_current_language()
'es'
```

При выполнении набора запросов `QuerySet` с использованием `filter()` можно выполнять фильтрацию, применяя связанные переводные объекты с синтаксисом `translations__`, как видно ниже:

```
>>> Product.objects.filter(translations__name='Green tea')
<TranslatableQuerySet [<Product: Té verde>]>
```

Адаптация представлений под переводы

Давайте адаптируем представления каталога товаров. Отредактируйте файл `views.py` приложения `shop`, добавив следующий ниже исходный код, выделенный жирным шрифтом, в представление `product_list`:

```
def product_list(request, category_slug=None):
    category = None
    categories = Category.objects.all()
    products = Product.objects.filter(available=True)
    if category_slug:
        language = request.LANGUAGE_CODE
        category = get_object_or_404(Category,
                                     translations__language_code=language,
                                     translations__slug=category_slug)
        products = products.filter(category=category)
    return render(request,
                  'shop/product/list.html',
                  {'category': category,
                   'categories': categories,
                   'products': products})
```

Затем отредактируйте представление `product_detail`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
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})
```

Теперь представления `product_list` и `product_detail` адаптированы под извлечение объектов с использованием переведенных полей.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/es/` в своем браузере. Вы должны увидеть страницу списка товаров, содержащую все товары, переведенные на испанский язык:

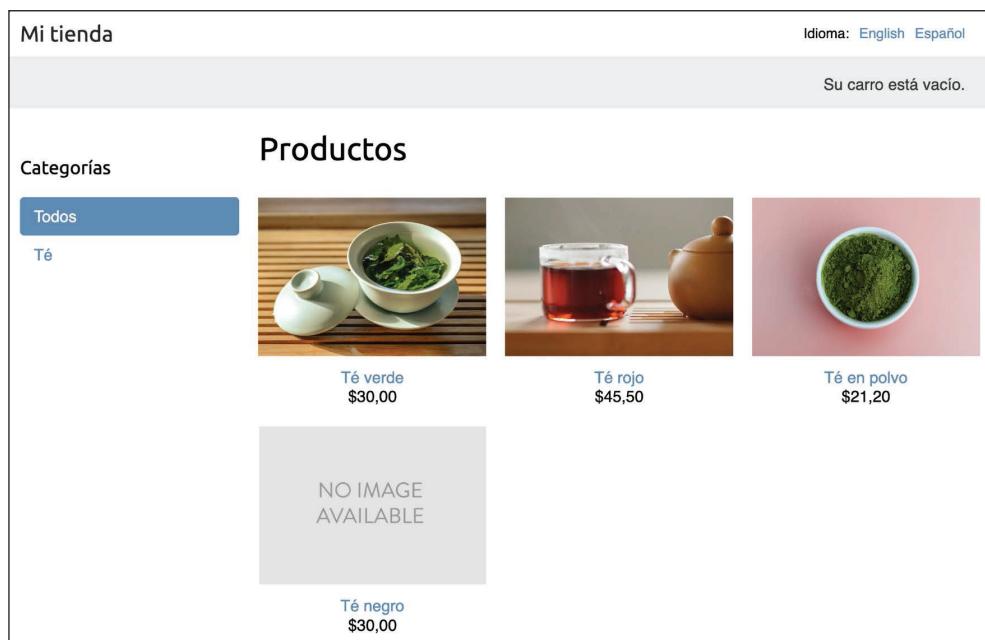


Рис. 11.13. Испанская версия страницы списка товаров

Теперь URL-адрес каждого товара создается с использованием поля `slug`, переведенного на текущий язык. Например, URL-адрес товара на испанском языке – `http://127.0.0.1:8000/es/2/te-gojo/`, а URL-адрес на английском языке – `http://127.0.0.1:8000/en/2/red-tea/`. Если вы перейдете на страницу детальной информации о товаре, то увидите переведенный URL-адрес и содержимое на выбранном языке, как показано в следующем ниже примере:

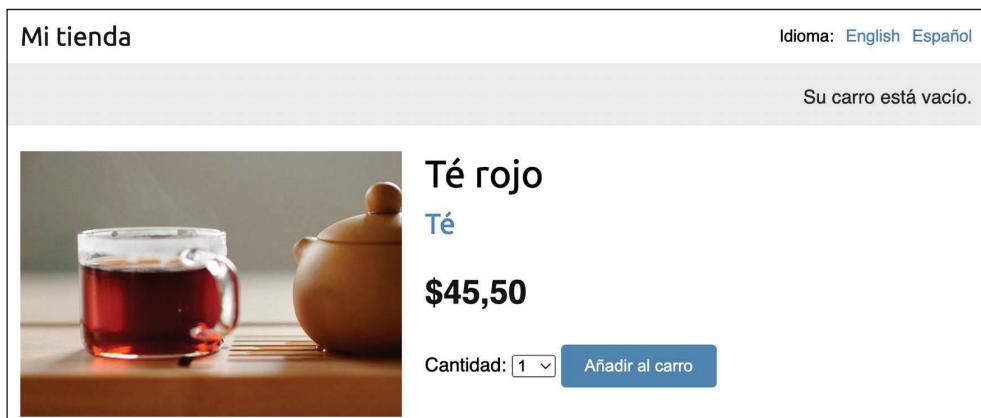


Рис. 11.14. Испанская версия страницы детальной информации о товаре

Если вы хотите узнать о модуле `django-parler` подробнее, то полная документация находится по адресу <https://django-parler.readthedocs.io/en/latest/>.

Вы научились переводить исходный код Python, визуальные шаблоны, шаблоны URL-адресов и поля моделей. В завершение процесса интернационализации и локализации также необходимо применить локализованное форматирование дат, времен и чисел.

Локализация формата

В зависимости от локали пользователя может возникать потребность отображать даты, времена и числа в разных форматах. Локализованное форматирование можно активировать, поменяв значение настроичного параметра `USE_L10N` на `True` в файле `settings.py` проекта.

При активированном параметре `USE_L10N` Django будет пытаться использовать зависящий от локали формат при каждом выводе значения в шаблоне. Вы видите, что разряды десятичных чисел в английской версии сайта отображаются с разделителем в виде точки, в испанской же версии они отображаются в виде запятой. Это связано с форматами локали, заданными в Django для локали `es`. На конфигурацию форматирования на испанском языке можно взглянуть на странице <https://github.com/django/django/blob/stable/4.0.x/django/conf/locale/es/formats.py>.

Обычно значение настроичного параметра USE_L10N устанавливается равным True, что позволяет Django применять локализацию формата для каждой локали. Однако могут возникать ситуации, когда потребность использовать локализованные значения отсутствует. Это особенно актуально при выводе результатов JavaScript или JSON, которые должны обеспечивать машиночитаемый формат.

Django предлагает шаблонный тег `{% localize %}`, который позволяет включать/выключать локализацию для фрагментов шаблона. Такой подход дает возможность контролировать локализованным форматированием. Для того чтобы иметь возможность использовать этот шаблонный тег, нужно загрузить теги `l10n`. Ниже приведен пример включения и выключения локализации в шаблоне:

```
{% load l10n %}

{% localize on %}
  {{ value }}
{% endlocalize %}

{% localize off %}
  {{ value }}
{% endlocalize %}
```

Django также предлагает шаблонные фильтры `localize` и `unlocalize`, чтобы принудительно устанавливать или снимать локализацию со значения. Эти фильтры применяются следующим образом:

```
{{ value|localize }}
{{ value|unlocalize }}
```

Кроме того, существует возможность создавать конкретно-прикладные форматные файлы, чтобы задавать форматирование локали. Дополнительная информация о локализации формата находится на странице <https://docs.djangoproject.com/en/4.1/topics/i18n/formatting/>.

Далее вы научитесь создавать локализованные поля форм.

Использование модуля *django-localflavor* для валидации полей формы

django-localflavor – это сторонний модуль, который содержит набор вспомогательных средств, таких как поля формы или поля модели, специфичные для каждой страны. Это очень удобно при валидации локальных регионов,

локальных телефонных номеров, номеров удостоверений личности, номеров социального страхования и т. д. Пакет организован в виде серии модулей, названных в соответствии с кодами стран ISO 3166.

Следующей ниже командой установите django-localflavor:

```
pip install django-localflavor==3.1
```

Отредактируйте файл `settings.py` проекта, добавив `localflavor` в настроечный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [
    # ...
    'localflavor',
]
```

Вы добавите поле почтового индекса США, чтобы для создания нового заказа требовался допустимый почтовый индекс США.

Отредактируйте файл `forms.py` приложения `orders`, придав ему следующий вид:

```
from django import forms
from localflavor.us.forms import USZipCodeField
from .models import Order

class OrderCreateForm(forms.ModelForm):
    postal_code = USZipCodeField()
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address',
                  'postal_code', 'city']
```

Здесь из пакета `us` модуля `localflavor` импортируется поле `USZipCodeField` и используется для поля `postal_code` формы `OrderCreateForm`.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/en/orders/create/` в своем браузере. Заполните все поля, введите трехбуквенный почтовый индекс и передайте форму на обработку. Вы получите следующую ниже ошибку валидации, вызванную полем `USZipCodeField`:

```
Enter a zip code in the format XXXXX or XXXXX-XXXX.
```

На рис. 11.15 показана ошибка валидации формы:

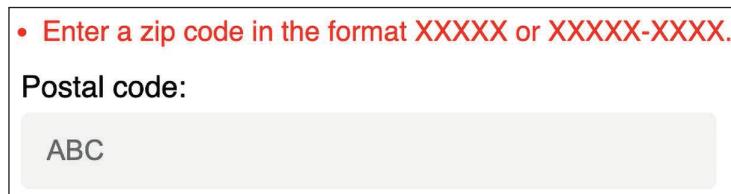


Рис. 11.15. Ошибка валидации недопустимого почтового индекса США

Это всего лишь краткий пример использования конкретно-прикладного поля из модуля `localflavor` для целей валидации в своем собственном проекте. Предоставляемые модулем `localflavor` локальные компоненты очень удобны для адаптации вашего приложения под конкретные страны. Почитать документацию `django-localflavor` и просмотреть все имеющиеся локальные компоненты по каждой стране можно на странице <https://django-localflavor.readthedocs.io/en/latest/>.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе:

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter11>.
- Список допустимых идентификаторов языков: <http://www.i18nguy.com/unicode/language-identifiers.html>.
- Список настроечных параметров интернационализации и локализации: <https://docs.djangoproject.com/en/4.1/ref/settings/#globalization-i18n-l10n>.
- Менеджер пакетов Homebrew: <https://brew.sh/>.
- Установка инструментария gettext в Windows: <https://docs.djangoproject.com/en/4.1/topics/i18n/translation/#gettext-on-windows>.
- Предкомпилированный бинарный установщик инструментария gettext для Windows: <https://mlocati.github.io/articles/gettext-iconv-windows.html>.
- Документация по переводам: <https://docs.djangoproject.com/en/4.1/topics/i18n/translation/>.
- Редактор файлов перевода Poedit: <https://poedit.net/>.
- Документация по приложению Django Rosetta: <https://django-rosetta.readthedocs.io/>.

- Совместимость модуля `django-parler` с Django: <https://django-parler.readthedocs.io/en/latest/compatibility.html>.
- Документация по модулю `django-parler`: <https://django-parler.readthedocs.io/en/latest/>.
- Конфигурация форматирования Django для испанской локали: <https://github.com/django/django/blob/stable/4.0.x/django/conf/locale/es/formats.py>.
- Локализация формата в Django: <https://docs.djangoproject.com/en/4.1/topics/i18n/formatting/>.
- Документация по модулю `django-localflavor`: <https://django-localflavor.readthedocs.io/en/latest/>.

Резюме

В этой главе вы познакомились с основами интернационализации и локализации проектов Django. Вы пометили строковые литералы исходного кода и шаблона для перевода и узнали, как генерировать и компилировать файлы перевода. Вы также установили приложение Rosetta в свой проект, чтобы управлять переводами через его веб-интерфейс. Вы перевели шаблоны URL-адресов и создали селектор языка, позволяющий пользователям переключать язык сайта. Затем вы использовали модуль `django-parler` для перевода моделей и `django-localflavor` для валидации локализованных полей формы.

В следующей главе вы начнете новый проект Django, который будет состоять из платформы электронного обучения. Вы создадите модели приложения и научитесь создавать и применять фикстуры, чтобы предоставлять моделям первоначальные данные. Вы разработаете конкретно-прикладное поле модели и будете использовать его в своих моделях. Вы также разработаете для своего нового приложения представления аутентификации.

12

Разработка платформы электронного обучения

В предыдущей главе вы изучили основы интернационализации и локализации проектов Django. Вы добавили интернационализацию в свой проект интернет-магазина. Вы научились переводить строковые литералы, шаблоны и модели Python. Вы также научились управлять переводами, создали селектор выбора языка и добавили локализованные поля в свои формы.

В этой главе вы начнете новый проект Django, который будет состоять из платформы электронного обучения с вашей собственной системой управления контентом (**CMS**)¹. Платформы онлайнового обучения – отличный пример приложений, в которых необходимо предоставлять инструменты для генерирования контента с учетом гибкости.

В этой главе вы научитесь:

- создавать модели для системы управления контентом;
- создавать фикстуры своих моделей и их применять;
- использовать наследование моделей, чтобы создавать модели данных для полиморфного контента;
- создавать конкретно-прикладные модельные поля;
- упорядочивать содержимое курсов и модулей;
- разрабатывать представления аутентификации для CMS.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter12>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

¹ Англ. Content Management System (CMS). – Прим. перев.

Настройка проекта электронного обучения

Последним практическим проектом в этой книге будет платформа электронного обучения. Сначала следующей ниже командой создайте виртуальную среду нового проекта в каталоге `env/`:

```
python -m venv env/educa
```

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

```
source env/educa/bin/activate
```

Если вы работаете с Windows, то вместо нее примените приведенную далее команду:

```
.\env\educa\Scripts\activate
```

А такой командой установите Django в виртуальной среде:

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

В своем проекте вы будете управлять закачкой изображений, поэтому следующей ниже командой также необходимо установить библиотеку Pillow:

```
pip install Pillow==9.2.0
```

Далее создайте новый проект посредством команды

```
django-admin startproject educa
```

Войдите в новый каталог `educa` и, используя следующие ниже команды, создайте новое приложение:

```
cd educa
django-admin startapp courses
```

Отредактируйте файл `settings.py` проекта `educa`, добавив приложение `courses` в настроечный параметр `INSTALLED_APPS`, как показано ниже. Новая строка выделена жирным шрифтом:

```
INSTALLED_APPS = [
    'courses.apps.CoursesConfig',
    'django.contrib.admin',
    'django.contrib.auth',
```

```
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
]
```

Теперь приложение courses проекта активно. Далее мы подготовим проект к раздаче медиафайлов и сформируем модели курсов и содержимого курсов.

Раздача медиафайлов

Перед тем как создавать модели курсов и содержимого курсов, необходимо подготовить проект к раздаче медиафайлов. Преподаватели курса смогут загружать медиафайлы в содержимое курсов с помощью системы управления контентом CMS, которую мы создадим. Поэтому мы сконфигурируем проект под раздачу медиафайлов.

Отредактируйте файл `settings.py` проекта, добавив следующие ниже строки:

```
MEDIA_URL = 'media/'  
MEDIA_ROOT = BASE_DIR / 'media'
```

Это позволит Django управлять закачкой файлов и раздавать медиафайлы. `MEDIA_URL` – это базовый URL-адрес, используемый для раздачи медиафайлов, закачанных пользователями. `MEDIA_ROOT` – это локальный путь, по которому они находятся. Пути и URL-адреса файлов создаются динамически путем добавления к ним пути проекта или URL-адреса медиафайла в качестве префикса с целью обеспечения переносимости.

Теперь отредактируйте главный файл `urls.py` проекта `educa`, видоизменив исходный код, как показано ниже. Новые строки выделены жирным шрифтом:

Здесь была добавлена вспомогательная функция `static()`, чтобы раздавать медиафайлы встроенным в Django сервером разработки во время разработки (то есть когда значение настроечного параметра `DEBUG` установлено равным `True`).



Напомним, что вспомогательная функция `static()` подходит для разработки, но не для использования в производстве. Django очень неэффективен при раздаче статических файлов. Никогда не раздавайте статические файлы с помощью Django в производственной среде. В главе 17 «Выход в прямой эфир» вы научитесь раздавать статические файлы в производственной среде.

Теперь проект готов к работе с медиафайлами. Давайте создадим модели курсов и содержимого курсов.

Разработка моделей курса

Платформа электронного обучения будет предлагать курсы по различным предметам. Каждый курс будет разделен на конфигурируемое число модулей, и каждый модуль будет содержать конфигурируемый объем содержимого. Содержимое будет разных типов: текст, файлы, изображения или видео. В следующем ниже примере показан внешний вид структуры каталога 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 разработаем модели курса. Отредактируйте файл `models.py` приложения `courses`, добавив в него следующий ниже исходный 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
```

Это первоначальные модели Subject, Course и Module. Поля модели Course таковы:

- `owner`: преподаватель, создавший этот курс;
- `subject`: предмет, к которому относится данный курс. Это поле внешнего ключа (`ForeignKey`), указывающее на модель `Subject`;
- `title`: название курса;
- `slug`: слаг курса. Позже он будет использоваться в URL-адресах;
- `overview`: столбец типа `TextField` для хранения краткого обзора курса;
- `created`: дата и время создания курса. Оно будет устанавливаться веб-фреймворком Django автоматически при создании новых объектов в силу `auto_now_add=True`.

Каждый курс поделен на несколько модулей. Поэтому модель `Module` содержит поле `ForeignKey`, указывающее на модель `Course`.

Откройте оболочку и выполните следующую ниже команду, чтобы создать первоначальную миграцию этого приложения:

```
python manage.py makemigrations
```

Вы увидите такой результат:

```
Migrations for 'courses':  
  courses/migrations/0001_initial.py:  
    - Create model Course
```

- Create model Module
- Create model Subject
- Add field subject to course

Затем выполните следующую ниже команду, чтобы применить все миграции к базе данных:

```
python manage.py migrate
```

Вы должны увидеть результат, содержащий все примененные миграции, включая миграции, относящиеся к Django. Результат будет содержать такую строку:

```
Applying courses.0001_initial... OK
```

Регистрация моделей на сайте администрирования

Давайте добавим модели курса на сайт администрирования. Откройте в редакторе файл `admin.py` внутри каталога приложения `courses` и добавьте в него следующий ниже исходный код:

```
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]
```

Модели приложения `courses` зарегистрированы на сайте администрирования. Напомним, что декоратор `@admin.register()` используется для регистрации моделей на сайте администрирования.

Использование фикстур с целью предоставления моделям первоначальных данных

Иногда возникает потребность предварительно заполнять базу данных жестко заданными данными. Это удобно для автоматического включения первоначальных данных в настройку проекта, чтобы не добавлять их вручную. В Django есть простой способ загрузки и выгрузки данных из базы данных в файлы, именуемые фикстурами. Django поддерживает фикстуры в форматах JSON, XML или YAML. Вы создадите фикстуру, чтобы включить несколько первоначальных объектов `Subject` проекта.

Сначала следующей ниже командой создайте суперпользователя:

```
python manage.py createsuperuser
```

Затем с помощью такой команды запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/courses/subject/` в своем браузере. С помощью сайта администрирования создайте несколько предметов. Страница списка изменений должна выглядеть следующим образом:

TITLE	SLUG
Mathematics	mathematics
Music	music
Physics	physics
Programming	programming

Рис. 12.1. Представление списка изменений предметов на сайте администрирования

Выполните следующую ниже команду из оболочки:

```
python manage.py dumpdata courses --indent=2
```

Вы увидите результат, подобный такому:

```
[  
 {  
     "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` выгружает данные из базы данных в стандартный вывод, по умолчанию сериализуемый в формате JSON. Результирующая структура данных содержит информацию о модели и ее полях, давая Django возможность загрузить ее в базу данных.

Вывод можно ограничивать моделями приложения, указывая имена приложений в команде или отдельные модели с использованием формата `app.Model`. Кроме того, формат можно задать с помощью флага `--format`. По умолчанию `dumpdata` выгружает сериализованные данные в стандартный вывод. Однако с помощью флага `--output` можно указать выходной файл. Флаг `--indent` позволяет задать отступы. Дополнительную информацию об опциях

команды `dumpdata` можно получить, выполнив команду `python manage.py dumpdata --help`.

С помощью следующих ниже команд сохраните этот дамп в файл-фикситру в новом каталоге `fixtures/` приложения `courses`:

```
mkdir courses/fixtures  
python manage.py dumpdata courses --indent=2 --output=courses/fixtures/  
subjects.json
```

Запустите сервер разработки и зайдите на сайт администрирования, чтобы удалить созданные вами предметы, как показано на рис. 12.2:

The screenshot shows the Django Admin 'Select subject to change' page. At the top, there's a toolbar with 'ADD SUBJECT' and a '+' icon. Below it, a 'Action' dropdown is set to 'Delete selected subjects'. A 'Go' button and a status message '4 of 4 selected' are nearby. The main area lists four subjects with checkboxes next to their titles: Mathematics, Music, Physics, and Programming. To the right of each subject is its corresponding 'SLUG' value: mathematics, music, physics, and programming. At the bottom left, it says '4 subjects'.

Рис. 12.2. Удаление всех существующих предметов

После удаления всех предметов загрузите фикстуру в базу данных следующей ниже командой:

```
python manage.py loaddata subjects.json
```

Все включенные в фикстуру объекты `Subject` снова загружаются в базу данных:

This screenshot shows the same 'Select subject to change' page as the previous one, but with a key difference: all four subjects (Mathematics, Music, Physics, Programming) now have their checkboxes unchecked. The rest of the interface, including the toolbar, action dropdown, and table structure, remains identical to the previous screenshot.

Рис. 12.3. Теперь предметы из фикстуры загружены в базу данных

По умолчанию Django ищет файлы в каталоге `fixtures/` каждого приложения, но в команде `loaddata` можно указывать полный путь к файлу-фикстуре. Также можно использовать настроечный параметр `FIXTURE_DIRS`, чтобы сообщать Django о дополнительных каталогах, в которых следует искать фикстуры.



Фикстуры удобны не только для задания первоначальных данных, но и для предоставления приложению образцов данных или данных, необходимых для тестов.

О том, как использовать фикстуры для тестирования, можно почитать по адресу <https://docs.djangoproject.com/en/4.1/topics/testing/tools/#fixture-loading>.

Если вы хотите загружать фикстуры в миграции моделей, то обратитесь к документации Django по миграции данных. Соответствующая документация находится по адресу <https://docs.djangoproject.com/en/4.1/topics/migrations/#data-migrations>.

Вы создали модели для управления предметами курса, курсами и модулями курса. Далее вы создадите модели для управления различными типами содержимого модулей.

Создание моделей полиморфного содержимого

В модули курса планируется добавлять разные типы содержимого, такие как текст, изображения, файлы и видео. Полиморфизм – это предоставление единого интерфейса для сущностей разных типов. Вам нужна разноплановая модель данных, позволяющая хранить разнообразный контент, доступный через единый интерфейс. В главе 7 «Отслеживание действий пользователя» вы узнали об удобстве использования обобщенных отношений для создания внешних ключей, которые могут указывать на объекты любой модели. Вы собираетесь создать модель `Content`, которая будет представлять содержимое модулей, и определите обобщенное отношение, чтобы связывать любой объект с объектом `Content`.

Отредактируйте файл `models.py` приложения `courses`, добавив следующие ниже инструкции импорта:

```
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey
```

Затем добавьте следующий ниже исходный код в конец файла:

```
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 содержит несколько типов содержимого, поэтому в ней определяется поле `ForeignKey`, указывающее на модель `Module`. Также можно установить обобщенное отношение, чтобы связывать объекты из разных моделей, которые представляют разные типы содержимого. Напомним, что для установления обобщенного отношения нужны три разных поля. В модели `Content` они таковы:

- `content_type`: поле `ForeignKey` для модели `ContentType`;
- `object_id`: поле `PositiveIntegerField` для хранения первичного ключа связанного объекта;
- `item`: поле `GenericForeignKey` для связанного объекта, объединяющее два предыдущих поля.

Соответствующий столбец в таблице этой модели в базе данных есть только у полей `content_type` и `object_id`. Поле `item` позволяет извлекать или устанавливать связанный объект напрямую, и его функциональность построена поверх двух других полей.

Вы будете использовать разные модели для каждого типа содержимого. Модели содержимого будут иметь несколько общих полей, но они будут отличаться фактическими данными, которые они могут хранить. Так будет создан единый интерфейс для разных типов содержимого.

Использование модельного наследования

Django поддерживает наследование моделей. Оно работает аналогично стандартному наследованию классов в Python.

Django предлагает следующие три варианта использования наследования моделей:

- **абстрактные модели**: удобны, когда нужно поместить некоторую общую информацию в несколько моделей;
- **наследование многотабличной модели**: применимо, когда каждая модель в иерархии сама по себе считается полной моделью;
- **прокси-модели¹**: удобны, когда нужно изменить поведение модели, например путем включения дополнительных методов, замены применяемого по умолчанию менеджера или использования других методопий.

Давайте рассмотрим каждый из них подробнее.

¹ Син. модели-заместители. – Прим. перев.

Абстрактные модели

Абстрактная модель – это базовый класс, внутри которого определяются поля, подлежащие включению во все дочерние модели. При этом в Django никаких таблиц базы данных для абстрактных моделей не создается. Таблица базы данных создается для каждой дочерней модели, включая поля, унаследованные от абстрактного класса, и поля, определенные в дочерней модели.

Для того чтобы пометить модель как абстрактную, нужно в ее Meta-класс включить атрибут `abstract=True`. Django распознает, что это абстрактная модель, и не будет создавать для нее таблицу базы данных. Для того чтобы создать дочерние модели, просто нужно подклассировать абстрактную модель¹.

В следующем ниже примере показаны абстрактная модель `Content` и дочерняя модель `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 данном случае Django создаст таблицу только для модели `Text`, включая поля `title`, `created` и `body`.

Наследование многотабличной модели

При многотабличном наследовании каждая модель соответствует таблице базы данных. Django создает поле `OneToOneField` для взаимосвязи между дочерней моделью и ее родительской моделью. Для того чтобы использовать многотабличное наследование, необходимо подклассировать существующую модель. Django будет создавать таблицу базы данных как для изначальной модели, так и для подмодели. В следующем ниже примере показано многотабличное наследование:

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)
```

¹ То есть создавать подкласс абстрактной модели. – Прим. перев.

```
class Text(BaseContent):
    body = models.TextField()
```

Django включит автоматически сгенерированное поле `OneToOneField` в модель `Text` и создаст таблицу базы данных для каждой модели.

Прокси-модели

Прокси-модель изменяет поведение модели. Обе модели работают с таблицей изначальной модели в базе данных. Для того чтобы создать прокси-модель, в `Meta`-класс модели следует добавить атрибут `proxy=True`. В следующем ниже примере показано, как создавать прокси-модель:

```
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 определяется модель `OrderedContent`, которая является прокси-моделью для модели `Content`. Эта модель задает применяемый по умолчанию порядок сортировки наборов запросов и обеспечивает дополнительный метод `created_delta()`. Обе модели, `Content` и `OrderedContent`, работают с одной и той же таблицей базы данных, и объекты доступны через ORM-преобразователь посредством любой модели.

Создание моделей Content

Модель `Content` приложения `courses` содержит обобщенное отношение, чтобы ассоциировать с ним разные типы содержимого. По каждому типу содержимого создаются разные модели. Все модели `Content` будут иметь несколько общих полей и дополнительные поля для хранения конкретно-прикладных данных. Вы создадите абстрактную модель, которая будет предоставлять общие поля всем моделям `Content`.

Отредактируйте файл `models.py` приложения `courses`, добавив в него следующий ниже исходный код:

```

class ItemBase(models.Model):
    owner = models.ForeignKey(User,
        related_name='%(class)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 приведенном выше исходном коде определяется абстрактная модель с именем `ItemBase`. Поэтому в ее `Meta`-классе устанавливается свойство `abstract=True`.

В данной модели определяются поля `owner`, `title`, `created` и `updated`. Указанные общие поля будут использоваться для всех типов содержимого.

Поле `owner` позволяет хранить создавшего контент пользователя. Так как это поле определено в абстрактном классе, для каждой подмодели потребуется другое имя `related_name`. Для имени модельного класса Django позволяет указывать местозаполнитель в атрибуте `related_name` в виде `%(class)s`. Благодаря этому `related_name` по каждой дочерней модели будет генерироваться автоматически. Так как в качестве `related_name` используется `'%(class)s_related'`, обратная связь для дочерних моделей будет соответственно `text_related`, `file_related`, `image_related` и `video_related`.

Выше мы определили четыре разные модели `Content`, которые наследуют от абстрактной модели `ItemBase`. Они таковы:

- `Text`: для хранения текстового содержимого;
- `File`: для хранения файлов, например PDF-файлов;
- `Image`: для хранения файлов изображений;
- `Video`: для хранения видео; поле `URLField` используется, чтобы представлять URL-адрес видео для его встраивания в контент.

Каждая дочерняя модель в дополнение к своим собственным полям содержит поля, определенные в классе `ItemBase`. Таблица базы данных будет создана соответственно для моделей `Text`, `File`, `Image` и `Video`. Таблица базы данных, ассоциированная с моделью `ItemBase`, создаваться не будет, поскольку это абстрактная модель.

Отредактируйте созданную ранее модель `Content`, изменив 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'))}
```

Здесь добавлен аргумент `limit_choices_to`; это сделано для того, чтобы ограничивать объекты `ContentType`, которые можно использовать для обобщенного отношения. При этом используется выражение поиска в поле `model__in` с целью фильтрации запроса объектами `ContentType` с атрибутом `model`, то есть `'text'`, `'video'`, `'image'` либо `'file'`.

Давайте создадим миграцию, чтобы вставить новые добавленные вами модели. Выполните следующую ниже команду из командной строки:

```
python manage.py makemigrations
```

Вы увидите такой результат:

```
Migrations for 'courses':
courses/migrations/0002_video_text_image_file_content.py
- Create model Video
- Create model Text
- Create model Image
- Create model File
- Create model Content
```

Затем выполните следующую ниже команду, чтобы применить новую миграцию:

```
python manage.py migrate
```

Вы увидите результат, который должен заканчиваться такой строкой:

```
Applying courses.0002_video_text_image_file_content... OK
```

Вы создали модели, которые подходят для добавления разнообразного контента в модули курса. Однако в этих моделях по-прежнему чего-то не хватает: модули и содержимое курса должны следовать определенному порядку. Требуется поле, которое позволит легко их упорядочивать.

Создание конкретно-прикладных модельных полей

Django поставляется с полной коллекцией модельных полей, которые можно использовать для разработки своих собственных моделей. Вместе с тем существует возможность создавать свои собственные модельные поля, чтобы хранить конкретно-прикладные данные либо изменять поведение существующих полей.

В проекте требуется поле, позволяющее устанавливать порядок следования объектов. Простой подход, позволяющий указывать порядок следования объектов с использованием существующих полей Django, состоит в добавлении поля `PositiveIntegerField` в модели. Используя целые числа, можно легко указывать порядок следования объектов. При этом можно создать конкретно-прикладное поле порядка, которое наследует от `PositiveIntegerField` и обеспечивает дополнительное поведение.

В поле порядка следования будут встроены две соответствующие функциональности:

- **назначать значение порядка следования автоматически, если конкретный порядок не указан:** при сохранении нового объекта без определенного порядка полю должно автоматически назначаться число, следующее за последним существующим упорядоченным объектом. Если имеется два объекта с порядком соответственно 1 и 2, то при сохранении третьего объекта следует автоматически назначать ему порядок 3, если конкретный порядок не указан;
- **упорядочивать объекты по другим полям:** модули курса будут упорядочиваться по курсу, к которому они принадлежат, и по содержимому модуля по отношению к модулю, к которому они принадлежат.

Внутри каталога приложения `courses` создайте новый файл `fields.py`, добавив в него следующий ниже исходный 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().__init__(*args, **kwargs)

    def pre_save(self, model_instance, add):
```

```
if getattr(model_instance, self.attname) is None:
    # текущее значение отсутствует
    try:
        qs = self.model.objects.all()
        if self.for_fields:
            # фильтровать по объектам с
            # одинаковыми значениями полей
            # для полей в "for_fields"
            query = {field: getattr(model_instance, field)}\
                for field in self.for_fields}
            qs = qs.filter(**query)
        # получить порядок последней позиции
        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().pre_save(model_instance, add)
```

Это конкретно-прикладное поле `OrderField`. Оно наследует от предоставляемого веб-фреймворком Django поля `PositiveIntegerField`. Поле `OrderField` принимает опциональный параметр `for_fields`, который позволяет указывать поля, используемые для упорядочения данных.

Указанное выше поле переопределяет метод `pre_save()` поля `PositiveIntegerField`, который исполняется перед сохранением поля в базе данных. В этом методе выполняются следующие действия.

1. Проверяется наличие у этого поля значения в экземпляре модели. При этом используется `self.attname` – имя атрибута, данное полю в модели. Если значение атрибута отличается от `None`, то вычисляется порядок, который ему следует назначить. Это делается следующим образом:
 - 1) формируется набор запросов `QuerySet`, чтобы извлечь все объекты модели поля. Обращаясь к `self.model`, извлекается модельный класс, которому поле принадлежит;
 - 2) если в атрибуте `for_fields` поля есть какие-либо имена полей, то набор запросов `QuerySet` фильтруется по текущему значению модельных полей в `for_fields`. Благодаря такому подходу вычисляется порядок относительно заданных полей;
 - 3) с помощью `last_item = qs.latest(self.attname)` из базы данных извлекается объект с наивысшим порядком. Если объект не найден, то подразумевается, что этот объект является первым, и ему назначается порядок 0;
 - 4) если объект найден, к наивысшему найденному порядку прибавляется 1;
 - 5) с помощью `setattr()` рассчитанный порядок следования задается значению поля в экземпляре модели и затем возвращается.

- Если у текущего поля экземпляра модели есть значение, то вместо вычисления используется это значение.



При создании конкретно-прикладных модельных полей следует делать их обобщенными. При этом следует избегать жесткого привязывания данных, которые зависят от конкретной модели или поля. Конкретно-прикладное поле должно работать в любой модели.

Дополнительная информация о написании конкретно-прикладных полей находится на странице <https://docs.djangoproject.com/en/4.1/howto/custom-model-fields/>.

Добавление упорядочивания в модули и объекты содержимого

Давайте добавим новое поле в созданные ранее модели. Отредактируйте файл `models.py` приложения `courses`, импортировав класс `OrderField` и поле в модель `Module`, как показано ниже:

```
from .fields import OrderField

class Module(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['course'])
```

Здесь задается новый порядок полей и указывается, что порядок вычисляется относительно курса, устанавливая `for_fields=['course']`. Это означает, что порядок нового модуля будет назначаться путем прибавления 1 к последнему модулю того же объекта `Course`.

Теперь можно отредактировать метод `__str__()` модели `Module`, чтобы включить его порядок, как показано ниже:

```
class Module(models.Model):
    # ...
    def __str__(self):
        return f'{self.order}. {self.title}'
```

Содержимое модуля также должно следовать определенному порядку. Добавьте поле `OrderField` в модель `Content`, как показано ниже:

```
class Content(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['module'])
```

На этот раз указывается, что порядок вычисляется относительно поля `module`.

Наконец, давайте добавим порядок, который будет применяться по умолчанию для обеих моделей. Добавьте следующий ниже `Meta`-класс в модели `Module` и `Content`:

```
class Module(models.Model):
    # ...
    class Meta:
        ordering = ['order']

class Content(models.Model):
    # ...
    class Meta:
        ordering = ['order']
```

Теперь модели `Module` и `Content` должны выглядеть следующим 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 f'{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',
                                         'file')})
    object_id = models.PositiveIntegerField()
    item = GenericForeignKey('content_type', 'object_id')
    order = OrderField(blank=True, for_fields=['module'])
```

```
class Meta:  
    ordering = ['order']
```

Давайте создадим новую миграцию моделей, которая будет отражать новые поля порядка следования. Откройте оболочку и выполните следующую ниже команду:

```
python manage.py makemigrations courses
```

Вы увидите такой результат:

```
It is impossible to add a non-nullable field 'order' to content without  
specifying a default. This is because 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 manually define a default value in models.py.  
Select an option:
```

Django сообщает, что для существующих строк в базе данных нужно указать значение нового поля `order`, которое будет применяться по умолчанию. Если поле содержит значение `null=True`, то оно будет принимать нулевые значения, и Django автоматически создаст миграцию, вместо того чтобы запрашивать значение, которое будет применяться по умолчанию. Далее можно указать такое значение либо отменить миграцию и перед созданием миграции добавить атрибут `default` в поле `order` в файле `models.py`.

Ведите 1 и нажмите клавишу **Enter**, чтобы указать применяемое по умолчанию значение для существующих записей. Вы увидите следующий ниже результат:

```
Please enter the default value as valid Python.  
The datetime and django.utils.timezone modules are available, so it is possible  
to provide e.g. timezone.now as a value.  
Type 'exit' to exit this prompt  
>>>
```

Введите 0, чтобы это значение применялось по умолчанию для существующих записей, и нажмите клавишу **Enter**. Django также запросит применяемое по умолчанию значение для модели `Module`. Выберите первый вариант и снова введите 0 в качестве такого значения. Наконец, вы увидите примерно следующее:

```
Migrations for 'courses':  
courses/migrations/0003_alter_content_options_alter_module_options_and_more.py  
  - Change Meta options on content  
  - Change Meta options on module
```

- ```
- Add field order to content
- Add field order to module
```

Затем указанной ниже командой примените новые миграции:

```
python manage.py migrate
```

Результат команды сообщит, что миграция была успешно применена, следующим образом:

```
Applying courses.0003_alter_content_options_alter_module_options_and_more... OK
```

Давайте протестируем новое поле. Следующей ниже командой откройте оболочку:

```
python manage.py shell
```

Создайте новый курс, как показано далее:

```
>>> 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')
```

Вы создали курс в базе данных. Теперь вы будете добавлять модули в курс и смотреть, как автоматически рассчитывается их порядок. Создайте начальный модуль и проверьте его порядок:

```
>>> m1 = Module.objects.create(course=c1, title='Module 1')
>>> m1.order
0
```

Поле `OrderField` устанавливает свое значение равным 0, так как это первый объект `Module`, созданный для данного курса. Создайте второй модуль для того же курса:

```
>>> m2 = Module.objects.create(course=c1, title='Module 2')
>>> m2.order
1
```

Поле `OrderField` вычисляет значение следующего порядка, прибавив 1 к наивысшему порядку для существующих объектов. Давайте создадим третий модуль, принудительно задав определенный порядок:

```
>>> m3 = Module.objects.create(course=c1, title='Module 3', order=5)
>>> m3.order
5
```

Если при создании или сохранении объекта указывается конкретно-прикладной порядок, то `OrderField` будет использовать это значение, не вычисляя порядок.

Давайте добавим четвертый модуль:

```
>>> m4 = Module.objects.create(course=c1, title='Module 4')
>>> m4.order
6
```

Порядок этого модуля был установлен автоматически. Конкретно-прикладное поле `OrderField` не гарантирует, что все значения порядка будут следовать один за другим. Однако оно учитывает существующие значения порядка и всегда назначает следующий порядок на основе самого высокого существующего порядка.

Давайте создадим второй курс и добавим в него модуль:

```
>>> 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
```

Для расчета порядка нового модуля в поле учитываются только существующие модули, принадлежащие к тому же курсу. Поскольку это первый модуль второго курса, то результирующий порядок равен 0. Это вызвано тем, что в поле `order` модели `Module` было указано `for_fields=['course']`.

Поздравляем! Вы успешно создали свое первое конкретно-прикладное модельное поле. Далее вы создадите систему аутентификации для CMS.

## Добавление представлений аутентификации

Теперь, когда вы создали полиморфную модель данных, вы разработаете систему управления контентом (CMS), чтобы управлять курсами и их содержимым. Первым шагом будет добавление системы аутентификации в CMS.

## Добавление системы аутентификации

Вы собираетесь использовать встроенный в Django фреймворк аутентификации с целью аутентификации пользователей на платформе электронного обучения. И преподаватели, и студенты будут экземплярами поставляемой с Django модели `User`, поэтому они смогут входить на сайт, используя представления аутентификации `django.contrib.auth`.

Отредактируйте главный файл `urls.py` проекта `educa`, включив представления входа (`login`) и выхода (`logout`) встретенного в Django фреймворка аутентификации:

```
from django.contrib import admin
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static
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),
]

if settings.DEBUG:
 urlpatterns += static(settings.MEDIA_URL,
 document_root=settings.MEDIA_ROOT)
```

## Создание шаблонов аутентификации

Внутри каталога приложения `courses` создайте следующую ниже файловую структуру:

```
templates/
 base.html
 registration/
 login.html
 logged_out.html
```

Перед тем как создавать шаблоны аутентификации, необходимо подготовить базовый шаблон проекта.

Отредактируйте файл `base.html` шаблона, добавив в него следующее ниже содержимое:

```
{% load static %}
<!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>
 document.addEventListener('DOMContentLoaded', (event) => {
 // DOM-модель загружена
 {% block domready %}
 {% endblock %}
 })
 </script>
</body>
</html>

```

Это базовый шаблон, который будет расширяться остальными шаблонами. В этом шаблоне определены следующие блоки:

- **title**: блок для других шаблонов, куда они будут добавлять собственный заголовок каждой страницы;
- **content**: главный блок содержимого. Все шаблоны, расширяющие базовый шаблон, должны добавлять содержимое в этот блок;
- **domready**: находится внутри прослушивателя событий JavaScript, который отслеживает наступление события `DOMContentLoaded`. Дает возможность исполнять исходный код после завершения загрузки объектной модели документа (**DOM**).

Используемые в этом шаблоне стили CSS расположены в каталоге `static`/приложения `courses` в прилагаемом к этой главе исходном коде. Скопируйте каталог `static/` в тот же каталог проекта, чтобы их использовать. Содержимое каталога находится по адресу <https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter12/educa/courses/static>.

Отредактируйте шаблон `registration/login.html`, добавив в него следующий ниже исходный код:

```

{% 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 входа для встроенного в Django представления `login`.

Отредактируйте шаблон `registration/logout.html`, добавив в него следующий ниже 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 %}
```

Это шаблон, который будет отображаться пользователю после выхода из учетной записи. С помощью следующей команды запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/accounts/login/` в своем браузере. Вы должны увидеть страницу входа в учетную запись:

The screenshot shows the 'Log-in' page of a platform named 'EDUCA'. At the top right is a 'Sign in' link. Below it, the word 'Log-in' is displayed. A message says 'Please, use the following form to log-in:'. There are two input fields: one for 'Username' and one for 'Password', both represented by empty rectangular boxes. At the bottom is a green 'LOG-IN' button.

Рис. 12.4. Страница входа в учетную запись

Пройдите по URL-адресу <http://127.0.0.1:8000/accounts/logout/> в своем браузере. Теперь вы должны увидеть страницу **Logged out** (Зарегистрирован выход из учетной записи), как показано на рис. 12.5:

The screenshot shows the 'Logged out' page of the 'EDUCA' platform. At the top right is a 'Sign in' link. The main area displays the message 'Logged out' in large text. Below it, a smaller message reads 'You have been successfully logged out. You can [log-in again](#)'. The entire page has a light gray background.

Рис. 12.5. Страница выхода из учетной записи

Вы успешно создали систему аутентификации для системы управления контентом (CMS).

## Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter12>.
- Использование фикстур Django для тестирования: <https://docs.djangoproject.com/en/4.1/topics/testing/tools/#fixture-loading>.

- Миграция данных: <https://docs.djangoproject.com/en/4.1/topics/migrations/#data-migrations>.
- Создание конкретно-прикладных модельных полей: <https://docs.djangoproject.com/en/4.1/howto/custom-model-fields/>.
- Каталог `static` для проекта платформы электронного обучения: <https://github.com/PacktPublishing/Django-4-by-Example/tree/main/Chapter12/edu-ca/courses/static>.

## Резюме

В этой главе вы научились использовать фикстуры, чтобы предоставлять моделям первоначальные данные. Используя модельное наследование, вы создали гибкую систему управления различными типами содержимого для модулей курса. Вы также реализовали конкретно-прикладное модельное поле для объектов `order` и создали систему аутентификации для платформы электронного обучения.

В следующей главе вы реализуете функциональность CMS по управлению содержимым курсов с помощью представлений на основе классов. Вы воспользуетесь встроенными в Django группами и системой разрешений, чтобы ограничивать доступ к представлениям, а также реализуете наборы форм, чтобы редактировать содержимое курсов. Вы также разработаете функциональность перетаскивания, чтобы переупорядочивать модули курсов и их содержимое с помощью JavaScript и Django.

# 13

## Создание системы управления контентом

В предыдущей главе вы создали прикладные модели для платформы электронного обучения и научились создавать и применять фикстуры данных к моделям. Вы создали конкретно-прикладное модельное поле, упорядочивать объекты и реализовали аутентификацию пользователей.

В этой главе вы научитесь разрабатывать функциональность, которая обеспечит преподавателей возможностью создавать курсы и управлять содержимым этих курсов разноплановым и эффективным способом.

В этой главе вы научитесь:

- создавать систему управления контентом, используя представления на основе классов и примесей;
- разрабатывать наборы форм и модельные наборы форм, чтобы редактировать модули курсов и содержимое модулей;
- управлять группами и разрешениями;
- реализовывать функциональность перетаскивания, чтобы переупорядочивать модули и содержимое.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter13>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

### Создание CMS

Теперь, когда вы создали разноплановую модель данных, вы разработаете систему управления контентом (CMS). CMS позволит преподавателям создавать курсы и управлять их содержимым. При этом необходимо обеспечить следующую функциональность:

- выводить список созданных преподавателем курсов;
- создавать, редактировать и удалять курсы;
- добавлять модули в курс и их переупорядочивать;
- добавлять разные типы содержимого в каждый модуль;
- переупорядочивать модули и содержимое курса.

Давайте начнем с базовых представлений CRUD<sup>1</sup>.

## Создание представлений на основе классов

Вы разработаете представления создания, редактирования и удаления курсов. Для этого вы будете использовать представления на основе классов. Отредактируйте файл `views.py` приложения `courses`, добавив следующий ниже исходный код:

```
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().get_queryset()
 return qs.filter(owner=self.request.user)
```

Это представление `ManageCourseListView`. Оно наследует от встроенного в Django типового представления `ListView`. Здесь переопределяется метод `get_queryset()` представления, чтобы извлекать только те курсы, которые были созданы текущим пользователем. Для того чтобы запретить пользователям редактировать, обновлять или удалять курсы, которые они не создавали, также потребуется переопределить метод `get_queryset()` в представлениях создания, обновления и удаления. Когда возникает потребность обеспечить определенное поведение нескольких представлений, основанных на классах, рекомендуется использовать *примеси* (примесные классы, или миксины).

## Использование примесей для представлений на основе классов

Примеси – это особый для класса вид множественного наследования. Их можно использовать для предоставления общей дискретной функциональности, которая при добавлении к другим примесям позволяет формировать поведение класса. Примеси применяются главным образом в двух ситуациях:

<sup>1</sup> От англ. Create, Update and Delete. – Прим. перев.

- когда нужно предоставить классу несколько опциональных функциональностей;
- когда нужно использовать ту или иную функциональность в нескольких классах.

Django поставляется с несколькими примесями, которые предоставляют дополнительную функциональность представлениям на основе классов. Подробнее о примесях можно узнать на странице <https://docs.djangoproject.com/en/4.1/topics/class-based-views/mixins/>.

Вы реализуете общее поведение для нескольких представлений в примесных классах и будете его использовать в представлениях курса. Отредактируйте файл `views.py` приложения `courses`, видоизменив его, как показано ниже:

```
from django.views.generic.list import ListView
from django.views.generic.edit import CreateView, \
 UpdateView, DeleteView
from django.urls import reverse_lazy
from .models import Course

class OwnerMixin:
 def get_queryset(self):
 qs = super().get_queryset()
 return qs.filter(owner=self.request.user)

class OwnerEditMixin:
 def form_valid(self, form):
 form.instance.owner = self.request.user
 return super().form_valid(form)

class OwnerCourseMixin(OwnerMixin):
 model = Course
 fields = ['subject', 'title', 'slug', 'overview']
 success_url = reverse_lazy('manage_course_list')

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin):
 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'
```

В приведенном выше исходном коде создаются примесные классы `OwnerMixin` и `OwnerEditMixin`. Эти примеси будут использоваться вместе со встроенными в Django представлениями `ListView`, `CreateView`, `UpdateView` и `DeleteView`. Примесный класс `OwnerMixin` реализует метод `get_queryset()`, который используется представлениями для получения базового набора запросов `QuerySet`. Указанный примесный класс будет переопределять этот метод с целью фильтрации объектов по атрибуту `owner`, чтобы извлекать объекты, принадлежащие текущему пользователю (`request.user`).

Примесный класс `OwnerEditMixin` реализует метод `form_valid()`, который используется представлениями, использующими примесный класс `DjangoModelFormMixin`, то есть представлениями с формами или модельными формами, такими как `CreateView` и `UpdateView`. Метод `form_valid()` исполняется, когда переданная на обработку форма является валидной.

По умолчанию поведение этого метода состоит в сохранении экземпляра (в случае модельных форм) и перенаправлении пользователя на адрес `Success_url`. Указанный метод переопределяется для того, чтобы автоматически устанавливать текущего пользователя в атрибуте `owner` сохраняемого объекта. Тем самым при сохранении объекта автоматически устанавливается его владелец.

Примесный класс `OwnerMixin` можно использовать в представлениях, которые взаимодействуют с любой моделью, содержащей атрибут `owner`.

Далее определяется примесный класс `OwnerCourseMixin`, который наследует от `OwnerMixin` и предоставляет следующие атрибуты дочерним представлениям:

- `model`: модель, используемая для наборов запросов; этот атрибут используется всеми представлениями;
- `fields`: поля модели, служащие для компоновки модельной формы представлений `CreateView` и `UpdateView`;
- `Success_url`: используется представлениями `CreateView`, `UpdateView` и `DeleteView`, чтобы перенаправлять пользователя после успешной передачи формы на обработку или удаления объекта. При этом используется URL-адрес с именем `manage_course_list`, который будет создан чуть позже.

Далее определяется примесный класс `OwnerCourseEditMixin` со следующим атрибутом:

- `template_name`: шаблон, который будет использоваться для представлений `CreateView` и `UpdateView`.

Наконец, создаются следующие ниже представления, которые являются подклассами примесного класса `OwnerCourseMixin`:

- `ManageCourseListView`: выводит список созданных пользователем курсов. Указанное представление наследует от `OwnerCourseMixin` и `ListView` и определяет специальный атрибут `template_name` для шаблона, который будет выводить список курсов;
- `CourseCreateView`: использует модельную форму для создания нового объекта `Course`. В указанном представлении используются поля, определенные в примесном классе `OwnerCourseMixin`, чтобы компоновать мо-

дельную форму; это представление также является подклассом класса `CreateView`. Оно использует шаблон, определенный в примесном классе `OwnerCourseEditMixin`;

- `CourseUpdateView`: обеспечивает возможность редактировать существующий объект `Course`. В указанном представлении используются поля, определенные в примесном классе `OwnerCourseMixin`, чтобы компоновать модельную форму; это представление также является подклассом класса `UpdateView`. Оно использует шаблон, определенный в примесном классе `OwnerCourseEditMixin`;
- `CourseDeleteView`: наследует от `OwnerCourseMixin` и типового `DeleteView`. В указанном представлении определяется специальный атрибут `template_name` для шаблона, который будет подтверждать удаление курса.

На данный момент были созданы базовые представления для управления курсами. Далее вы займетесь аутентификационными группами и разрешениями Django, чтобы ограничивать доступ к этим представлениям.

## Работа с группами и разрешениями

В настоящее время любой пользователь может получать доступ к представлениям, чтобы управлять курсами. Нужно ограничить эти представления, чтобы только преподаватели имели право создавать курсы и управлять ими.

Встроенный в Django фреймворк аутентификации содержит в своем составе систему разрешений, которая позволяет назначать разрешения пользователям и группам. Вы создадите группу для пользователей-преподавателей и назначите разрешения на создание, обновление и удаление курсов.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/auth/group/add/` в своем браузере, чтобы создать новый объект `Group`. Добавьте имя `Instructors` и выберите все разрешения приложения `courses`, кроме разрешений модели `Subject`, как показано на рис. 13.1.

Как видите, для каждой модели существует четыре разных разрешения: можно просматривать, добавлять, изменять и удалять. После выбора разрешений для этой группы кликните по кнопке **SAVE** (Сохранить).

Django создает разрешения для моделей автоматически, но помимо этого можно создавать конкретно-прикладные разрешения. Вы научитесь создавать конкретно-прикладные разрешения в главе 15 «Разработка API». Подробнее о добавлении конкретно-прикладных разрешений можно узнать по адресу <https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#custom-permissions>.



Рис. 13.1. Разрешения группы *Instructors*

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/auth/user/add/> и создайте нового пользователя. Отредактируйте пользователя, добавив его в группу *Instructors*, как показано ниже:



Рис. 13.2. Выбор группы пользователей

Пользователи наследуют разрешения групп, к которым они принадлежат, но помимо этого с помощью сайта администрирования можно добавлять индивидуальные разрешения одному пользователю. Пользователи, значение параметра `is_superuser` которых задано равным `True`, получают все разрешения автоматически.

## Ограничение доступа к представлениям, основанным на классах

Вы собираетесь ограничить доступ к представлениям, чтобы только пользователи с соответствующими разрешениями имели возможность добавлять, изменять или удалять объекты курса. С целью ограничения доступа к представлениям вы будете использовать следующие два примесных класса, предоставленных фреймворком django.contrib.auth:

- `LoginRequiredMixin`: воспроизводит функциональность декоратора `login_required`;
- `PermissionRequiredMixin`: предоставляет доступ к представлению пользователям с конкретным разрешением. Напомним, что суперпользователи получают все разрешения автоматически.

Отредактируйте файл `views.py` приложения `courses`, добавив следующую ниже инструкцию импорта:

```
from django.contrib.auth.mixins import LoginRequiredMixin, \
 PermissionRequiredMixin
```

Придайте примесному классу `OwnerCourseMixin` соответствующий вид, чтобы он наследовал от `LoginRequiredMixin` и `PermissionRequiredMixin`, как показано ниже:

```
class OwnerCourseMixin(OwnerMixin,
 LoginRequiredMixin,
 PermissionRequiredMixin):
 model = Course
 fields = ['subject', 'title', 'slug', 'overview']
 success_url = reverse_lazy('manage_course_list')
```

Затем добавьте атрибут `permission_required` в представления курса, как показано ниже:

```
class ManageCourseListView(OwnerCourseMixin, ListView):
 template_name = 'courses/manage/course/list.html'
 permission_required = 'courses.view_course'

class CourseCreateView(OwnerCourseEditMixin, CreateView):
 permission_required = 'courses.add_course'

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):
 permission_required = 'courses.change_course'

class CourseDeleteView(OwnerCourseMixin, DeleteView):
 template_name = 'courses/manage/course/delete.html'
 permission_required = 'courses.delete_course'
```

Примесный класс `PermissionRequiredMixin` проверяет, чтобы обращающийся к представлению пользователь имел разрешение, указанное в атрибуте `permission_required`. Созданные представления теперь доступны только пользователям с соответствующими разрешениями.

Давайте создадим URL-адреса этим представлениям. Внутри каталога приложения `courses` создайте новый файл и назовите его `urls.py`. Добавьте в него следующий ниже исходный 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-адресов вывода списка, создания, редактирования и удаления курсов. Параметр `pk` относится к полю первичного ключа. Напомним, что `pk` – это сокращение от англ. primary key (первичный ключ). Каждая модель Django имеет поле, которое служит первичным ключом. По умолчанию первичным ключом является автоматически генерируемое поле `id`. Типовые представления Django для одиночных объектов извлекают объект по его полю `pk`. Отредактируйте главный файл `urls.py` проекта `educa`, вставив шаблоны URL-адресов приложения `courses`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static
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),
 path('course/', include('courses.urls')),
]

if settings.DEBUG:
 urlpatterns += static(settings.MEDIA_URL,
 document_root=settings.MEDIA_ROOT)
```

Для этих представлений необходимо создать шаблоны. Внутри каталога templates/ приложения courses создайте следующие ниже каталоги и файлы:

```
courses/
 manage/
 course/
 list.html
 form.html
 delete.html
```

Отредактируйте шаблон courses/manage/course/list.html, добавив в него следующий ниже исходный 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 %}
```

Это шаблон представления `ManageCourseListView`. В данном шаблоне перечисляются курсы, созданные текущим пользователем. При этом вставляются ссылки на редактирование или удаление каждого курса, а также ссылка на создание новых курсов.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/accounts/login/?next=/course/mine/` в своем браузере и войдите на сайт под именем пользователя, принадлежащего к группе `Instructors`. После входа вы будете перенаправлены на URL-адрес `http://127.0.0.1:8000/course/mine/` и должны будете увидеть следующую ниже страницу:



Рис. 13.3. Страница курсов преподавателей,  
пока что без курсов

На этой странице будут отображаться все курсы, созданные текущим пользователем.

Давайте создадим шаблон, который отображает форму для создания и обновления представлений курса. Откройте в редакторе шаблон `courses/manage/course/form.html` и напишите следующий ниже исходный 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 method="post">
 {{ form.as_p }}
 {% csrf_token %}
 <p><input type="submit" value="Save course"></p>
 </form>
</div>
{% endblock %}

```

Шаблон `form.html` используется для представлений `CourseCreateView` и `CourseUpdateView`. В этом шаблоне делается проверка на наличие переменной `object` в контексте. Если `object` в контексте существует, то это означает, что обновляется существующий курс, и он используется в шапке страницы. В противном случае создается новый объект `Course`.

Пройдите по URL-адресу `http://127.0.0.1:8000/course/mine/` в своем браузере и кликните по кнопке **CREATE NEW COURSE** (Создать новый курс). Вы увидите страницу, показанную на рис. 13.4.

Заполните форму и кликните по кнопке **SAVE COURSE** (Сохранить курс). Курс будет сохранен, и вы будете перенаправлены на страницу списка курсов. Она должна выглядеть так, как показано на рис. 13.5.

Затем кликните по ссылке **Edit** (Редактировать) под только что созданным курсом. Вы снова увидите форму, но на этот раз вы редактируете существующий объект `Course`, а не создаете его.

Наконец, отредактируйте шаблон `courses/manage/course/delete.html`, добавив в него следующий ниже исходный код:

```

{% 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" value="Confirm">
 </form>
 </div>
{% endblock %}

```

### Create a new course

Course info

Subject:

Title:

Slug:

Overview:

**SAVE COURSE**

Рис. 13.4. Форма для создания нового курса

## EDUCA

### My courses

Django course

Edit Delete

**CREATE NEW COURSE**

Рис. 13.5. Страница курсов преподавателей с одним курсом

Это шаблон представления `CourseDeleteView`. Указанное представление наследует от встроенного в Django представления `DeleteView`, которое ожидает от пользователя подтверждения на удаление объекта.

Откройте список курсов в браузере и кликните по ссылке **Delete** (Удалить) курса. Вы должны увидеть следующую ниже страницу подтверждения:

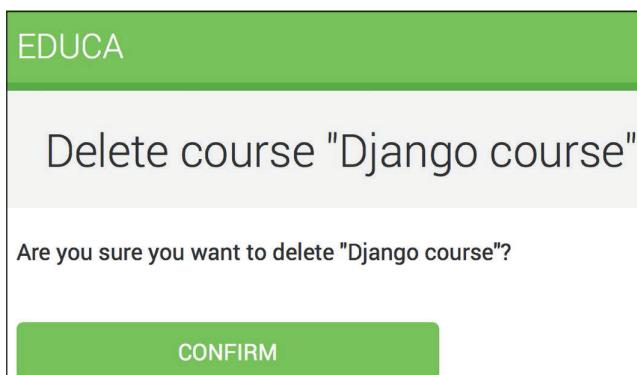


Рис. 13.6. Страница подтверждения на удаление курса

Кликните по кнопке **CONFIRM** (Подтвердить). Курс будет удален, и вы снова будете перенаправлены на страницу списка курсов.

Преподаватели теперь могут создавать, редактировать и удалять курсы. Далее необходимо предоставить им CMS-функциональности по добавлению модулей курса и их содержимого. Вы начнете с управления модулями курса.

## Управление модулями курса и их содержимым

Вы разработаете систему управления модулями курса и их содержимым. Вам нужно будет создать формы, которые можно использовать для управления несколькими модулями курса и разными типами содержимого по каждому модулю. И модули, и их содержимое должны подчиняться определенному порядку, и вы должны иметь возможность их переупорядочивать с помощью CMS.

## Использование наборов форм для модулей курса

Django поставляется со слоем абстракции, предназначенным для работы с несколькими формами на одной странице. Эти группы форм называются *наборами форм* (*formsets*). Наборы форм управляют несколькими экземпля-

рами определенного класса `Form` или `ModelForm`. Все формы передаются на обработку одновременно, и набор форм отвечает за первоначальное число отображаемых форм, ограничивая максимальное число форм, которые могут передаваться на обработку, и выполняя валидацию всех форм.

Наборы форм содержат метод `is_valid()` для одновременной валидации всех форм. Кроме того, формам можно указывать первоначальные данные и число дополнительных пустых форм, подлежащих отображению. Подробнее о наборах форм можно узнать на странице <https://docs.djangoproject.com/en/4.1/topics/forms/formsets/>, а о модельных наборах форм<sup>1</sup> – по адресу <https://docs.djangoproject.com/en/4.1/topics/forms/modelformsets/#model-formsets>.

Так как курс поделен на переменное число модулей, для управления ими имеет смысл использовать наборы форм. Внутри каталога приложения `courses` создайте файл `forms.py`, добавив в него следующий ниже исходный 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`. Он создается с помощью предоставляемой веб-фреймворком Django функции `inlineformset_factory()`. Внутристочные наборы форм – это небольшая абстракция поверх наборов форм, упрощающая работу со связанными объектами. Указанная функция позволяет создавать модельный набор форм динамически для объектов `Module`, связанных с объектом `Course`.

Для компоновки набора форм используются следующие параметры:

- `fields`: поля, которые будут включены в каждую форму набора форм;
- `extra`: позволяет устанавливать число пустых дополнительных форм для отображения в наборе форм;
- `can_delete`: если значение этого параметра устанавливается равным `True`, то Django будет вставлять булево поле для каждой формы, которая будет прорисовываться в виде флагка. Этим обеспечивается возможность помечать объекты, которые требуется удалить.

Отредактируйте файл `views.py` приложения `courses`, добавив в него следующий ниже исходный код:

<sup>1</sup> Модельные наборы форм (model formsets) состоят из нескольких расширенных классов наборов форм, чтобы делать работу с моделями Django удобнее. – Прим. перев.

```
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().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})
```

Представление `CourseModuleUpdateView` обрабатывает набор форм, служащий для добавления, обновления и удаления модулей определенного курса. Это представление наследует от следующих примесей и представлений:

- `TemplateResponseMixin`: этот примесный класс отвечает за прорисовку шаблонов и возврат HTTP-ответа. Для него требуется атрибут `template_name`, указывающий на подлежащий прорисовке шаблон и предоставляющий метод `render_to_response()`, чтобы передавать ему контекст и прорисовывать шаблон;
- `View`: предоставляемое веб-фреймворком Django базовое представление на основе класса.

В указанном представлении реализованы следующие методы:

- `get_formset()`: этот метод определяется для того, чтобы избегать повторения исходного кода компоновки набора форм. Для заданного объекта `Course` создается объект `ModuleFormSet` с опциональными данными;
- `dispatch()`: этот метод предоставляется классом `View`. Он принимает HTTP-запрос и его параметры и пытается делегировать его более низкоуровневому методу, который отыскивает совпадение используемому HTTP-методу. Запрос методом `GET` делегируется методу `get()`, а запрос методом `POST` – соответственно методу `post()`. В этом методе используется функция сокращенного доступа `get_object_or_404()`, чтобы получить объект `Course` для заданного параметра `id`, который принадлежит текущему пользователю. Этот исходный код вставляется в метод `dispatch()`, потому что требуется извлекать курс как для запросов `GET`, так и для запросов `POST`. Он сохраняется в атрибуте `course` представления, чтобы сделать его доступным для других методов;
- `get()`: выполняется для запросов методом `GET`. При этом создается пустой набор форм `ModuleFormSet` и прорисовывается по шаблону вместе с текущим объектом `Course`, используя предоставленный примесным классом `TemplateResponseMixin` метод `render_to_response()`;
- `post()`: выполняется для запросов методом `POST`.

В этом методе выполняются следующие действия.

1. Создается экземпляр `ModuleFormSet`, используя отправленные на обработку данные.
2. Выполняется метод `is_valid()` набора форм для валидации всех его форм.
3. Если набор форм валиден, то он сохраняется, вызывая метод `save()`. На этом этапе любые сделанные изменения, такие как добавление, обновление или пометка модулей на удаление, применяются к базе данных. Затем пользователи перенаправляются на URL-адрес `manage_course_list`. Если набор форм невалиден, то шаблон прорисовывается, чтобы отобразить ошибки.

Отредактируйте файл `urls.py` приложения `courses`, добавив в него следующий ниже шаблон URL-адреса:

```
path('<pk>/module/',
 views.CourseModuleUpdateView.as_view(),
 name='course_module_update'),
```

Внутри каталога шаблонов `courses/manage/` создайте новый каталог и назовите его `module`. Создайте шаблон `courses/manage/module/formset.html` и добавьте в него следующий ниже исходный код:

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

{% block title %}
 Edit "{{ course.title }}"
{% endblock %}

{% block content %}
 <h1>Edit "{{ course.title }}"</h1>
 <div class="module">
 <h2>Course modules</h2>
 <form method="post">
 {{ formset }}
 {{ formset.management_form }}
 {{ csrf_token }}
 <input type="submit" value="Save modules">
 </form>
 </div>
{% endblock %}
```

В этом шаблоне создается HTML-элемент `<form>`, в который вставляется набор форм. Также вставляется управляющая форма для набора форм с переменной `{{ formset.management_form }}`. Управляющая форма вставляет скрытые поля, чтобы контролировать первоначальное, общее, минимальное и максимальное числа форм. Как видите, создавать набор форм очень просто.

Отредактируйте шаблон `courses/manage/course/list.html`, добавив следующую ниже ссылку на URL-адрес `course_module_update` под ссылками редактирования (**Edit**) и удаления (**Delete**) курса:

```
Edit
Delete
Edit modules
```

Вы включили ссылку на редактирование модулей курса.

Пройдите по URL-адресу `http://127.0.0.1:8000/course/mine/` в своем браузере. Создайте курс и кликните по ссылке **Edit modules** (Редактировать модули). Вы должны увидеть набор форм, как показано на рис. 13.7.

Набор форм содержит форму по каждому объекту модуля, содержащемуся в курсе. После них отображаются две пустые дополнительные формы, потому что для `ModuleFormSet` было установлено `extra=2`. При сохранении набора форм Django добавит еще два дополнительных поля для добавления новых модулей.

Edit "Django course"

Course modules

Title:

Description:

Delete:

Title:

Description:

Delete:

**SAVE MODULES**

Рис. 13.7. Страница редактирования курса, содержащая набор форм для модулей курса

## Добавление содержимого в модули курса

Теперь нужен способ добавления содержимого в модули курса. Имеется четыре разных типа содержимого: текст, видео, изображение и файл. Можно рассмотреть возможность создания четырех разных представлений для создания содержимого, по одному для каждой модели. Однако вы будете использовать более типовой подход и создадите представление, которое обрабатывает создание либо обновление объектов любой модели содержимого.

Отредактируйте файл `views.py` приложения `courses`, добавив в него следующий ниже исходный 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().dispatch(request, module_id, model_name, id)
```

Это первая часть представления `ContentCreateUpdateView`. Оно позволит создавать и обновлять содержимое разных моделей. В указанном представлении определены следующие методы:

- `get_model()`: здесь делается проверка на принадлежность данного имени одной из четырех моделей содержимого: `Text`, `Video`, `Image` либо `File`. Затем используется встроенный в Django модуль `apps`, чтобы получить фактический класс для данного имени модели. Если данное имя модели не является одним из допустимых, то возвращается `None`;

- `get_form()`: используя функцию `modelform_factory()` фреймворка форм, создается динамическая форма. Поскольку форма будет компоноваться для моделей `Text`, `Video`, `Image` и `File`, тут используется параметр `exclude`, чтобы указать общие поля, которые нужно исключить из формы, и разрешить автоматическое включение всех остальных атрибутов. Благодаря этому не нужно знать, какие поля включать в зависимости от модели;
- `dispatch()`: получает следующие параметры URL-адреса и сохраняет соответствующий модуль, модель и объект содержимого в качестве атрибутов класса:
  - `module_id`: ИД модуля, с которым ассоциировано / будет ассоциировано содержимое;
  - `model_name`: имя модели содержимого, которое нужно создать/обновить;
  - `id`: ИД обновляемого объекта. Он равен `None`, если создаются новые объекты.

Добавьте следующие методы `get()` и `post()` в представление `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:
 # new content
 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})
```

Приведенные выше методы работают следующим образом:

- `get()`: выполняется при получении запроса методом GET. Модельная форма компонуется для обновляемого экземпляра `Text`, `Video`, `Image` или `File`. В противном случае экземпляр для создания нового объекта не передается, поскольку если ИД не указан, то `self.obj` имеет значение `None`;

- `post()`: выполняется при получении запроса методом POST. Модельная форма компонуется, передавая ей любые отправленные на обработку данные и файлы. Затем она валидируется. Если форма валидна, то создается новый объект, и `request.user` назначается его владельцем перед сохранением в базе данных. Затем проверяется наличие параметра `id`. Если ИД не указан, то это означает, что пользователь создает новый объект, а не обновляет существующий. Если это новый объект, то для данного модуля создается объект `Content` и с ним связывается новое содержимое.

Отредактируйте файл `urls.py` приложения `courses`, добавив в него следующие ниже шаблоны URL-адресов:

```
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'),
```

Новые шаблоны URL-адресов таковы:

- `module_content_create`: для создания новых объектов текста, видео, изображения или файла и добавления их в модуль. Он содержит параметры `module_id` и `model_name`. Первый позволяет связывать новый объект содержимого с данным модулем. Последний задает модель содержимого, для которой создается форма;
- `module_content_update`: для обновления существующего объекта текста, видео, изображения или файла. Он содержит параметры `module_id` и `model_name`, а также параметр `id` для идентификации обновляемого содержимого.

Внутри каталога шаблонов `courses/manage/` создайте новый каталог и назовите его `content`. Создайте шаблон `courses/manage/content/form.html` и добавьте в него следующий ниже исходный код:

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

{% block title %}
 {% if object %}
 Edit content "{{ object.title }}"
 {% else %}
 Add new content
 {% endif %}
{% endblock %}
```

```
{% block content %}
 <h1>
 {% if object %}
 Edit content "{{ object.title }}"
 {% else %}
 Add 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 представления ContentCreateUpdateView. В указанном шаблоне делается проверка на наличие переменной `object` в контексте. Если `object` в контексте существует, то существующий объект обновляется. В противном случае создается новый объект.

Далее в HTML-элемент `<form>` вставляется `enctype="multipart/form-data"`, поскольку для моделей содержимого `File` и `Image` форма содержит закачку файла на сайт.

Запустите сервер разработки, пройдите по URL-адресу `http://127.0.0.1:8000/course/mine/`, кликните по **Edit modules** (Редактировать модули) рядом с существующим курсом и создайте модуль.

Затем следующей ниже командой откройте оболочку Python:

```
python manage.py shell
```

Получите ИД самого последнего созданного модуля, как показано ниже:

```
>>> from courses.models import Module
>>> Module.objects.latest('id').id
6
```

Запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/course/module/6/content/image/create/` в своем браузере, заменив ИД модуля на тот, который вы получили ранее. Вы увидите форму для создания объекта `Image`, как показано ниже:

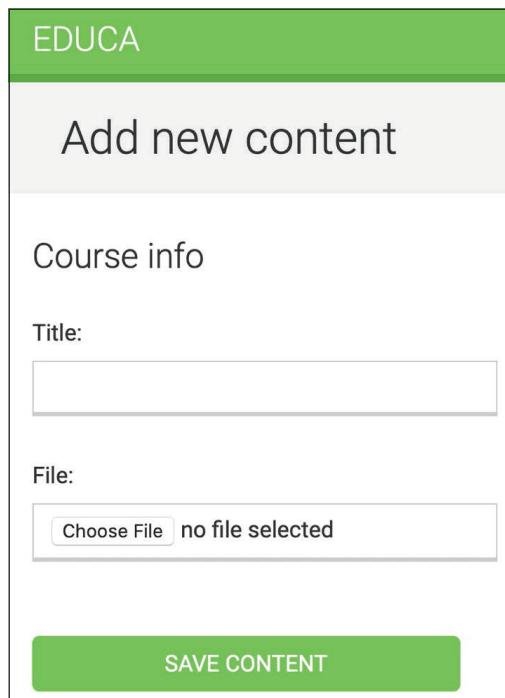


Рис. 13.8. Форма для добавления нового содержимого в виде изображения в курс

Пока не передавайте форму на обработку. Если вы попытаетесь это сделать, то у вас ничего не получится, потому что вы еще не определили URL-адрес `module_content_list`. Вы создадите его чуть позже.

Кроме того, еще потребуется представление удаления содержимого. Отредактируйте файл `views.py` приложения `courses`, добавив следующий ниже исходный код:

```
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)
```

Класс `ContentDeleteView` извлекает объект `Content` с заданным ИД. Он удаляет связанный текст, видео, изображение или файл. Наконец, он удаляет объект `Content` и перенаправляет пользователя на URL-адрес `module_content_list`, чтобы вывести список другого содержимого модуля.

Отредактируйте файл `urls.py` приложения `courses`, добавив в него следующий ниже шаблон URL-адреса:

```
path('content/<int:id>/delete/',
 views.ContentDeleteView.as_view(),
 name='module_content_delete'),
```

Теперь преподаватели могут легко создавать, обновлять и удалять содержимое.

## Управление модулями и их содержимым

Вы создали представления создания, редактирования и удаления модулей курса и их содержимого. Далее вам потребуется представление отображения всех модулей курса и вывода списка содержимого конкретного модуля.

Отредактируйте файл `views.py` приложения `courses`, добавив в него следующий ниже исходный 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 представление `ModuleContentListView`. Указанное представление получает объект `Module` с заданным ИД, который принадлежит текущему пользователю, и прорисовывает шаблон с данным модулем.

Отредактируйте файл `urls.py` приложения `courses`, добавив в него следующий ниже шаблон URL:

```
path('module/<int:module_id>',
 views.ModuleContentListView.as_view(),
 name='module_content_list'),
```

Внутри каталога `templates/courses/manage/module/` создайте новый шаблон и назовите его `content_list.html`. Добавьте в него такой исходный 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 %}
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк.

Это шаблон, который отображает все модули курса и содержимое выбранного модуля. Модули курса прокручиваются в цикле, чтобы отобразить их на боковой панели. Содержимое модуля прокручивается в цикле, обращаясь к `content.item`, чтобы получить соответствующий объект `Text`, `Video`, `Image` либо `File`. Также вставляются ссылки на создание нового типа содержимого в формате текста, видео, изображения или файла.

Далее, необходимо знать, к какому типу относится каждый объект `item`: `Text`, `Video`, `Image` либо `File`. Для того чтобы сформировать URL-адрес редактирования объекта, требуется имя модели. Помимо этого, каждый элемент `item` в шаблоне можно отображать по-разному в зависимости от типа содержимого. Модельное имя для объекта можно получать из Meta-класса модели, обращаясь к атрибуту `_meta` объекта. Однако при этом Django не разрешает доступ в шаблонах к переменным и атрибутам, начинающимся со знака подчеркивания, чтобы предотвратить получение закрытых атрибутов и вызов закрытых методов. Эта проблема решается путем написания конкретно-прикладного шаблонного фильтра.

Внутри каталога приложения `courses` создайте следующую ниже файловую структуру:

```
templatetags/
 __init__.py
 course.py
```

Отредактируйте модуль course.py, добавив в него следующий ниже исходный код:

```
from django import template

register = template.Library()

@register.filter
def model_name(obj):
 try:
 return obj._meta.model_name
 except AttributeError:
 return None
```

Это шаблонный фильтр `model_name`. Его можно применять в шаблонах как `object|model_name`, чтобы получать для объекта его модельное имя.

Отредактируйте шаблон templates/courses/manage/module/content\_list.html, добавив следующую ниже строку под шаблонным тегом `{% extends %}`:

```
{% load course %}
```

Указанный тег загрузит шаблонные теги курса. Затем найдите следующие ниже строки:

```
<p>{{ item }}</p>
Edit
```

И замените их такими:

```
<p>{{ item }} ({{ item|model_name }})</p>

 Edit

```

В приведенном выше исходном коде отображается модельное имя элемента `item` в шаблоне; модельное имя также используется для формирования ссылки на редактирование объекта.

Отредактируйте шаблон courses/manage/course/list.html, добавив ссылку на URL-адрес `module_content_list`, как показано ниже:

```
Edit modules
{% if course.modules.count > 0 %}

```

```
Manage contents

{% endif %}
```

Новая ссылка позволяет пользователям обращаться к содержимому первого модуля курса, если таковые имеются.

Остановите сервер разработки и с помощью указанной ниже команды снова его запустите:

```
python manage.py runserver
```

Остановив и запустив сервер разработки, вы вызовите загрузку файла шаблонных тегов course.

Пройдите по URL-адресу <http://127.0.0.1:8000/course/mine/> и кликните по ссылке **Manage contents** (Управление содержимым) на курс, который содержит хотя бы один модуль. Вы увидите страницу, похожую на приведенную ниже:

The screenshot shows a web interface for managing course content. At the top, there's a green header bar with the word 'EDUCA'. Below it, the main title is 'Course "Django course"'. On the left, there's a dark sidebar with the heading 'Modules'. Under 'MODULE 1', it lists 'Introduction to Django'. At the bottom of the sidebar, there's a green link labeled 'Edit modules'. The main content area on the right has a title 'Module 1: Introduction to Django'. Below it, under 'Module contents:', it says 'This module has no contents yet.' There's also a section for adding new content with buttons for 'Text', 'Image', 'Video', and 'File'.

Рис. 13.9. Страница управления содержимым модуля курса

При нажатии на модуль на левой боковой панели его содержимое отображается в главной области. Шаблон также содержит ссылки на добавление нового текста, видео, изображения либо файла отображаемого модуля.

Добавьте в модуль пару разных типов содержимого и посмотрите на результат. Содержимое модуля появится под надписью **Module contents** (Содержимое модуля):

The screenshot shows a user interface for managing course modules. At the top, there's a green header bar with the word 'EDUCA'. Below it, the title 'Course "Django course"' is displayed. On the left, a sidebar lists 'Modules' and two specific sections: 'MODULE 1 Introduction to Django' and 'MODULE 2 Configuring Django'. A button 'Edit modules' is also present in this sidebar. The main content area is titled 'Module 2: Configuring Django'. It contains a section 'Module contents:' with two items: 'Setting up Django (text)' and 'Example settings.py (image)'. Each item has 'Edit' and 'Delete' buttons next to it. Below this, there's a section 'Add new content:' with buttons for 'Text', 'Image', 'Video', and 'File'.

Рис. 13.10. Управление различным содержимым модуля

Далее мы предоставим преподавателям курса возможность переупорядочивать модули и содержимое модулей с помощью простой функциональности перетаскивания.

## Переупорядочивание модулей и их содержимого

Мы выполним реализацию функциональности перетаскивания на JavaScript, чтобы предоставить преподавателям курсов возможность переупорядочивать модули курса путем их перетаскивания.

С целью реализации этой функциональности мы будем использовать библиотеку HTML5 Sortable, которая упрощает процесс создания сортируемых списков с помощью нативного для HTML5 API перетаскивания (Drag and Drop).

По завершении пользователями перетаскивания модуля вы будете работать с интерфейсом Fetch API на JavaScript для отправки асинхронного HTTP-запроса на сервер, на котором хранится новый порядок следования модулей.

Более подробную информацию об HTML5 API Drag and Drop можно почитать на странице [https://www.w3schools.com/html/html5\\_draganddrop.asp](https://www.w3schools.com/html/html5_draganddrop.asp). Созданные с помощью библиотеки HTML5 Sortable примеры находятся на странице <https://lukasoppermann.github.io/html5sortable/>. Документация по библиотеке HTML5 Sortable доступна на странице <https://github.com/lukasoppermann/html5sortable>.

## Использование примесей из модуля django-braces

`django-braces` – это сторонний модуль, содержащий коллекцию типовых примесных классов для Django. Эти примеси обеспечивают дополнительные возможности для представлений на основе классов. Список всех предоставляемых модулем `django-braces` примесных классов можно увидеть на странице <https://django-braces.readthedocs.io/>.

Вы будете использовать следующие примесные классы модуля `django-braces`:

- `CsrfExemptMixin`: используется, чтобы избежать проверки токена подделки межсайтовых запросов (CSRF) в запросах методом POST. Это необходимо для выполнения AJAX-запросов методом POST без необходимости передачи токена `csrf_`;
- `JsonRequestResponseMixin`: конвертирует данные запроса в формат JSON, а также сериализует ответ в формат JSON и возвращает HTTP-ответ с типом содержимого `application/json`.

Следующей ниже командой установите модуль `django-braces` посредством `pip`:

```
pip install django-braces==1.15.0
```

Теперь требуется представление, которое получает новый порядок следования кодированных в формате JSON идентификаторов модулей и соответствующим образом обновляет порядок. Отредактируйте файл `views.py` приложения `courses`, добавив в него следующий ниже исходный 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 представление `ModuleOrderView`, которое позволяет обновлять порядок следования модулей курса.

Теперь можно разработать аналогичное представление, чтобы упорядочивать содержимое модуля. Добавьте следующий ниже исходный код в файл `views.py`:

```
class ContentOrderView(CsrftokenExemptMixin,
 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'})
```

Теперь отредактируйте файл `urls.py` приложения `courses`, добавив в него следующие ниже шаблоны URL-адресов:

```
path('module/order/',
 views.ModuleOrderView.as_view(),
 name='module_order'),
path('content/order/',
 views.ContentOrderView.as_view(),
 name='content_order'),
```

Наконец, необходимо реализовать функциональность перетаскивания в шаблоне. Мы будем использовать библиотеку HTML5 Sortable, которая упрощает создание сортируемых элементов с помощью стандартного API Drag and Drop на HTML.

Отредактируйте шаблон `base.html`, расположенный в каталоге `templates`/приложения `courses`, добавив следующий ниже блок, выделенный жирным шрифтом:

```
{% load static %}
<!DOCTYPE html>
<html>
 <head>
 # ...
 </head>
 <body>
 <div id="header">
 # ...
 </div>
 <div id="content">
 {% block content %}
 {% endblock %}
 </div>
```

```
{% block include_js %}
{% endblock %}
<script>
 document.addEventListener('DOMContentLoaded', (event) => {
 // DOM-модель загружена
 {% block domready %}
 {% endblock %}
 })
</script>
</body>
</html>
```

Приведенный выше новый блок с именем `include_js` позволит вставлять файлы JavaScript в любой шаблон, расширяющий шаблон `base.html`.

Далее отредактируйте шаблон `courses/manage/module/content_list.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом, в нижнюю часть шаблона:

```
...
{% block content %}
 #
{% endblock %}

{% block include_js %}
<script src="https://cdnjs.cloudflare.com/ajax/libs/html5sortable/0.13.3/
html5sortable.min.js"></script>
{% endblock %}
```

В приведенном выше исходном коде из общедоступной сети доставки контента (CDN) загружается библиотека HTML5 Sortable. Напомним, что вы уже загружали библиотеку JavaScript из CDN-сети ранее в главе 6 «Распространение контента на веб-сайте».

Теперь добавьте следующий ниже блок `domready`, выделенный жирным шрифтом, в шаблон `courses/manage/module/content_list.html`:

```
...
{% block content %}
 #
{% endblock %}

{% block include_js %}
<script src="https://cdnjs.cloudflare.com/ajax/libs/html5sortable/0.13.3/
html5sortable.min.js"></script>
{% endblock %}

{% block domready %}
var options = {
```

```

 method: 'POST',
 mode: 'same-origin'
 }
const moduleOrderUrl = '{% url "module_order" %}';
{% endblock %}

```

В приведенных выше новых строках в блок `{% block domready %}`, определенный в прослушивателе события `DOMContentLoaded` в шаблоне `base.html`, добавляется исходный код JavaScript. Этим гарантируется, что исходный код JavaScript будет исполняться после загрузки страницы. С помощью него определяются опции HTTP-запроса для переупорядочивания модулей, которые будут реализованы дальше. Запрос методом `POST` будет отправляться посредством Fetch API, чтобы обновлять порядок модулей. Путь URL-адреса `module_order` создается и сохраняется в константе JavaScript `moduleOrderUrl`.

Добавьте в блок `domready` следующий ниже исходный код, выделенный жирным шрифтом:

```

{% block domready %}
var options = {
 method: 'POST',
 mode: 'same-origin'
}

const moduleOrderUrl = '{% url "module_order" %}';

sortable('#modules', {
 forcePlaceholderSize: true,
 placeholderClass: 'placeholder'
});
{% endblock %}

```

В новом исходном коде определяется элемент `sortable` для HTML-элемента с `id="modules"`, который представляет собой список модулей на боковой панели. Напомним, что CSS-селектор `#` используется для выбора элемента с заданным `id`. Когда пользователь начинает перетаскивать элемент, библиотека HTML5 Sortable создает элемент-местозаполнитель, который позволяет легко видеть место, куда этот элемент будет размещен.

Значение параметра `forcePlaceholderSize` устанавливается равным `true`, чтобы элемент-местозаполнитель имел высоту, и используется `placeholderClass`, чтобы определить CSS-класс для элемента-местозаполнителя. Здесь применяется класс с именем `placeholder`, определенный в статическом файле `css/base.css`, загруженному в шаблон `base.html`.

Пройдите по URL-адресу `http://127.0.0.1:8000/course/mine/` в своем браузере и кликните по **Manage contents** (Управление содержимым) любого курса. Теперь вы сможете перетаскивать модули курса на левой боковой панели, как показано на рис. 13.11:

Course "Django Course"

Modules

MODULE 2  
Configuring Django

MODULE 1  
Introduction to Django

Edit modules

Module 2: Introduction to Django

Module contents:

Setting up Django (text)

Example settings.py (image)

Add new content:

Text      Image      Video      File

Рис. 13.11. Переупорядочивание модулей с помощью функциональности перетаскивания

При перетаскивании элемента вы увидите созданный библиотекой Sortable элемент-местозаполнитель, который имеет пунктирную границу. Элемент-местозаполнитель позволяет указывать позицию, в которую будет перенесен перетаскиваемый элемент.

При перетаскивании модуля в другое место нужно отправить HTTP-запрос на сервер, чтобы сохранить новый порядок. Это делается путем прикрепления обработчика событий к сортируемому элементу и отправки запроса на сервер с помощью Fetch API.

Отредактируйте блок domready шаблона `courses/manage/module/content_list.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
{% block domready %}
var options = {
 method: 'POST',
 mode: 'same-origin'
}

const moduleOrderUrl = '{% url "module_order" %}';

sortable('#modules', {
 forcePlaceholderSize: true,
});
```

```

placeholderClass: 'placeholder'
})[0].addEventListener('sortupdate', function(e) {

 modulesOrder = {};
 var modules = document.querySelectorAll('#modules li');
 modules.forEach(function (module, index) {
 // обновить индекс модуля
 modulesOrder[module.dataset.id] = index;
 // обновить индекс в HTML-элементе
 module.querySelector('.order').innerHTML = index + 1;
 // добавить новый порядок в опции HTTP-запроса
 options['body'] = JSON.stringify(modulesOrder);

 // отправить HTTP-запрос
 fetch(moduleOrderUrl, options)
 });
});
{%
 endblock %}

```

В новом исходном коде создается прослушиватель события `sortupdate` сортируемого элемента. Событие `sortupdate` запускается, когда элемент перетаскивается в другую позицию. В функции события выполняется следующая работа.

1. Создается пустой словарь `modulesOrder`. Ключами этого словаря являются идентификаторы модулей, а значениями – индекс каждого модуля.
2. Элементы списка HTML-элемента `#modules` выбираются с помощью `document.querySelectorAll()` с использованием CSS-селектора `#modules li`.
3. Метод `forEach()` используется для прокручивания каждого элемента списка в цикле.
4. Новый индекс каждого модуля сохраняется в словаре `modulesOrder`. ИД каждого модуля извлекается из HTML-атрибута `data-id` путем доступа к `module.dataset.id`. В качестве ключа словаря `modulesOrder` используется ИД, а в качестве значения – новый индекс модуля.
5. Отображаемый порядок каждого модуля обновляется путем выбора элемента с CSS-классом `order`. Поскольку индекс отсчитывается от нуля, а нужно отображать индекс от единицы, к индексу прибавляется 1.
6. Ключ с именем `body` добавляется в словарь опций с новым порядком, содержащимся в `modulesOrder`. Метод `JSON.stringify()` конвертирует объект JavaScript в строковый литерал JSON. Это тело HTTP-запроса для обновления порядка следования модулей.
7. Далее используется Fetch API, чтобы создать HTTP-запрос `fetch()` и обновить порядок следования модулей. Представление `ModuleOrderView`, соответствующее URL-адресу `module_order`, обеспечивает обновление порядка следования модулей.

Теперь вы можете перетаскивать модули. Когда вы закончите перетаскивать модуль, на URL-адрес `module_order` будет отправлен HTTP-запрос на обновление порядка следования модулей. Если вы обновите страницу, то

последний порядок следования модулей будет сохранен, поскольку он был обновлен в базе данных. Рисунок 13.12 иллюстрирует другой порядок следования модулей на боковой панели после их сортировки с помощью перетаскивания:



Рис. 13.12. Новый порядок следования модулей  
после их переупорядочивания с помощью перетаскивания

При возникновении каких-либо проблем не забудьте использовать инструменты браузера для разработчика, чтобы выполнить отладку JavaScript и HTTP-запросов. Для этого обычно нужно кликнуть правой кнопкой мыши в любом месте веб-сайта, чтобы открыть контекстное меню, и нажать **Inspect** (Проинспектировать) или **Inspect Element** (Проинспектировать элемент), чтобы получить доступ к браузерным инструментам веб-разработчика.

Давайте добавим ту же самую функциональность перетаскивания, чтобы преподаватели курсов имели возможность сортировать содержимое модулей.

Отредактируйте блок `domready` шаблона `courses/manage/module/content_list.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
{% block domready %}

// ...

const contentOrderUrl = '{% url "content_order" %}';
```

```

sortable('#module-contents', {
 forcePlaceholderSize: true,
 placeholderClass: 'placeholder'
})[0].addEventListener('sortupdate', function(e) {

 contentOrder = {};
 var contents = document.querySelectorAll('#module-contents div');
 contents.forEach(function (content, index) {
 // обновить индекс контента
 contentOrder[content.dataset.id] = index;
 // добавить новый порядок в опции HTTP-запроса
 options['body'] = JSON.stringify(contentOrder);

 // отправить HTTP-запрос
 fetch(contentOrderUrl, options)
 });
});

{%

```

В данном случае вместо URL-адреса `module_order` используется URL-адрес `content_order` и создается сортируемая функциональность для HTML-элемента с ИД `module-contents`. Функциональность в основном такая же, как и при упорядочивании модулей курса. В данном же случае обновлять нумерацию содержимого не требуется, потому что она не имеет никакого видимого индекса.

Теперь вы можете перетаскивать как модули, так и их содержимое, как показано на рис. 13.13:

The screenshot shows the EDUCA application interface. At the top, there's a green header bar with the text "EDUCA". Below it, the main title is "Course "Django Course"".

On the left side, there's a sidebar with a dark background. It contains a list of modules: "MODULE 1 Configuring Django" and "MODULE 2 Introduction to Django". Below this list is a button labeled "Edit modules".

The main content area has a light gray background. It displays "Module 2: Introduction to Django". Underneath the module title, there's a section titled "Module contents:" which lists "Setting up Django (text)". Below this item are two buttons: "Edit" and "Delete".

Further down, there's another section titled "Example settings.py (image)". This section also includes "Edit" and "Delete" buttons. A dashed line indicates that this content item can be moved.

At the bottom of the main content area, there's a section titled "Add new content:" with four buttons: "Text", "Image", "Video", and "File".

*Рис. 13.13. Переупорядочивание содержимого модуля с помощью функциональности перетаскивания*

Отлично! Вы создали очень разноплановую систему управления контентом для преподавателей курсов.

## Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе:

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter13>.
- Документация по примесным классам Django: <https://docs.djangoproject.com/en/4.1/topics/class-based-views/mixins/>.
- Создание конкретно-прикладных разрешений: <https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#custom-permissions>.
- Наборы форм Django: <https://docs.djangoproject.com/en/4.1/topics/forms/formsets/>.
- Модельные наборы форм Django: <https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/#model-formsets>.
- API HTML5 для перетаскивания: [https://www.w3schools.com/html/html5\\_draganddrop.asp](https://www.w3schools.com/html/html5_draganddrop.asp).
- Документация по библиотеке HTML5 Sortable: <https://github.com/lukasoppermann/html5sortable>.
- Примеры библиотеки HTML5 Sortable: <https://lukasoppermann.github.io/html5sortable/>.
- Документация по модулю django-braces: <https://django-braces.readthedocs.io/>.

## Резюме

В этой главе вы научились использовать представления на основе классов и примеси для создания системы управления контентом. Вы также поработали с группами и разрешениями, чтобы ограничить доступ к представлениям. Вы научились применять наборы форм и модельные наборы форм, чтобы управлять модулями курса и их содержимым. Вы также разработали функциональность перетаскивания с помощью JavaScript, чтобы переупорядочивать модули курса и их содержимое.

В следующей главе вы создадите систему регистрации студентов и будете управлять зачислением студентов на курсы. Вы также научитесь отображать разные виды содержимого и кэшировать содержимое с помощью кеш-фреймворка Django.

# 14

## Прорисовка и кеширование контента

В предыдущей главе вы использовали наследование моделей и обобщенные отношения, чтобы создавать гибкие модели содержимого курса. Вы реализовали конкретно-прикладное модельное поле и разработали систему управления курсами, используя представления на основе классов. Наконец, вы создали функциональность перетаскивания с использованием JavaScript и асинхронных HTTP-запросов, чтобы упорядочивать модули курса и их содержимое.

В этой главе вы разработаете функциональность доступа к содержимому курса, создадите систему регистрации студентов и будете управлять зачислением студентов на курсы. Вы также научитесь кешировать данные, используя кеш-фреймворк Django.

В этой главе вы:

- создадите общедоступные представления, чтобы отображать информацию о курсе;
- разработаете систему регистрации студентов;
- будете управлять зачислением студентов на курсы;
- будете прорисовывать разнообразное содержимое модулей курса;
- установите и сконфигурируете кеш-бэкенд Memcached;
- будете кешировать содержимое с помощью кеш-фреймворка Django;
- задействуете кеш-бэкенды Memcached и Redis;
- будете отслеживать свой сервер Redis на сайте администрирования.

Начнем с создания каталога курсов, чтобы студенты имели возможность просматривать существующие курсы и записываться на них.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter14>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по

установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

## Отображение курсов

Для вашего каталога курсов необходимо создать следующие функциональности:

- вывод списка всех доступных курсов, при необходимости отфильтрованных по предмету;
- отображение краткого обзора одного курса.

Отредактируйте файл `views.py` приложения `courses`, добавив следующий ниже исходный 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 представление `CourseListView`. Оно наследует от примесного класса `TemplateResponseMixin` и представления `View`. В данном представлении выполняется следующая работа.

1. Используя метод ORM-преобразователя `annotate()` с функцией агрегирования `Count()`, извлекаются все предметы, чтобы вставить общее число курсов по каждому предмету.
2. Извлекаются все доступные курсы, включая общее число модулей, содержащихся в каждом курсе.
3. Если задан URL-параметр слага предмета, то извлекается соответствующий объект `subject`, и запрос ограничивается курсами, принадлежащими данному предмету.

4. Для прорисовки объектов по шаблону и возврата HTTP-ответа используется предоставляемый примесным классом `TemplateResponseMixin` метод `render_to_response()`.

Давайте создадим представление детальной информации, чтобы отображать краткий обзор одного курса. Добавьте следующий ниже исходный код в файл `views.py`:

```
from django.views.generic.detail import DetailView

class CourseDetailView(DetailView):
 model = Course
 template_name = 'courses/course/detail.html'
```

Это представление наследует от предоставляемого веб-фреймворком Django типового представления `DetailView`. В нем указываются атрибуты `model` и `template_name`. Django `DetailView` ожидает, что первичный ключ (`pk`) или URL-параметр `slug` будет извлекать один объект данной модели. Представление прорисовывает шаблон, указанный в `template_name`, включая объект `Course` в переменной `object` контекста шаблона.

Отредактируйте главный файл `urls.py` проекта `educa`, добавив в него следующий ниже шаблон URL-адреса:

```
from courses.views import CourseListView

urlpatterns = [
 # ...
 path('', CourseListView.as_view(), name='course_list'),
]
```

Шаблон URL-адреса `course_list` добавляется в главный файл `urls.py` проекта, поскольку по URL-адресу `http://127.0.0.1:8000/` планируется отображать список курсов, а все остальные URL-адреса приложения `courses` имеют префикс `/course/`.

Отредактируйте файл `urls.py` приложения `courses`, добавив следующие ниже шаблоны URL-адресов:

```
path('subject/<slug:subject>/',
 views.CourseListView.as_view(),
 name='course_list_subject'),
path('<slug:slug>',
 views.CourseDetailView.as_view(),
 name='course_detail'),
```

Здесь определяются следующие шаблоны URL-адресов:

- `course_list_subject`: для отображения всех курсов по предмету;
- `course_detail`: для отображения краткого обзора одного курса.

Давайте создадим шаблоны представлений `CourseListView` и `CourseDetailView`.

Внутри каталога `templates/courses/` приложения `courses` создайте следующую ниже файловую структуру:

```
course/
 list.html
 detail.html
```

Отредактируйте шаблон `courses/course/list.html` приложения `courses`, написав следующий ниже исходный код:

```
{% 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 }}

 {% endfor %}

</div>

```

```


 {{ s.total_courses }} course{{ s.total_courses|pluralize }}

{% 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 %}
```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк.

Это шаблон списка имеющихся курсов. В нем создается HTML-список для отображения всех объектов `Subject` и создается ссылка на URL-адрес `course_list_subject` по каждому из них. Также вставляется общее число курсов по каждому предмету и используется шаблонный фильтр `pluralize`, чтобы добавлять суффикс множественного числа к слову `course`, когда число отличается от 1, показывая *0 courses*, *1 course*, *2 courses* и т. д. Далее добавляется HTML-класс `selected`, чтобы выделять текущий предмет, если предмет выбран. Каждый объект `Course` прокручивается в цикле, и отображается общее число модулей и имя преподавателя.

Запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере. Вы должны увидеть страницу, похожую на следующую:

The screenshot shows a user interface for managing courses. On the left, a sidebar titled 'Subjects' lists categories: 'All', 'Mathematics' (1 COURSES), 'Music' (0 COURSES), 'Physics' (0 COURSES), and 'Programming' (2 COURSES). The 'Programming' category is currently selected. The main content area displays two course entries under 'Programming': 'Django course' by Antonio Melé and 'Python for beginners' by Laura Marlon.

Рис. 14.1. Страница списка курсов

Левая боковая панель содержит все предметы, включая общее число курсов по каждому из них. По любому предмету можно кликнуть, чтобы отфильтровать отображаемые курсы.

Отредактируйте шаблон `courses/course/detail.html`, добавив в него следующий ниже исходный код:

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

{% block title %}
 {{ object.title }}
{% endblock %}

{% block content %}
 {% with subject=object.subject %}
 <h1>
 {{ object.title }}
 </h1>
 <div class="module">
 <h2>Overview</h2>
 <p>

 {{ subject.title }}.
 {{ object.modules.count }} modules.
 Instructor: {{ object.owner.get_full_name }}
 </p>
 </div>
 {% endwith %}
{% endblock %}
```

```

</p>
{{ object.overview|linebreaks }}
</div>
{% endwith %}
{% endblock %}

```

В приведенном выше шаблоне отображается краткий обзор и детальная информация об одном курсе. Пройдите по URL-адресу <http://127.0.0.1:8000/> в своем браузере и кликните по одному из курсов. Вы должны увидеть страницу со следующей ниже структурой:

The screenshot shows a web browser window with a green header bar containing the text 'EDUCA' on the left and 'Sign out' on the right. Below the header, the main content area has a light gray background. At the top of this area, the title 'Django course' is displayed. Below the title, the word 'Overview' is centered. Underneath 'Overview', there is a section with the text 'Programming. 2 modules. Instructor: Antonio Melé'. At the bottom of this section, a descriptive paragraph reads: 'Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.' On the far right edge of the browser window, there is a vertical scrollbar.

*Рис. 14.2. Страница краткого обзора курса*

Вы создали общедоступную область отображения курсов. Далее необходимо разрешить пользователям регистрироваться в качестве студентов и записываться на курсы.

## Добавление регистрации студентов

Следующей ниже командой создайте новое приложение:

```
python manage.py startapp students
```

Отредактируйте файл `settings.py` проекта `educa`, добавив новое приложение в настроечный параметр `INSTALLED_APPS`, как показано ниже:

```

INSTALLED_APPS = [
 ...
 'students.apps.StudentsConfig',
]

```

## Создание представления регистрации студентов

Отредактируйте файл `views.py` приложения `students`, написав следующий ниже исходный код:

```
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().form_valid(form)
 cd = form.cleaned_data
 user = authenticate(username=cd['username'],
 password=cd['password1'])
 login(self.request, user)
 return result
```

Это представление, которое позволяет студентам регистрироваться на сайте. Здесь используется типовое представление `CreateView`, которое предоставляет функциональность создания модельных объектов. В указанном представлении требуются следующие атрибуты:

- `template_name`: путь к шаблону, применяемому для прорисовки этого представления;
- `form_class`: форма для создания объектов, которая должна быть `ModelForm`. В качестве регистрационной формы для создания объектов `User` используется встроенная в Django форма `UserCreationForm`;
- `Success_url`: URL-адрес перенаправления пользователя после успешной передачи формы на обработку. URL-адрес с именем `student_course_list`, который будет создан в разделе «Доступ к содержимому курсов», переворачивается (`reverse_lazy`), чтобы выводить список курсов, на которые зачислены студенты.

Метод `form_valid()` исполняется при отправке валидных данных формы. Он должен возвращать HTTP-ответ. Этот метод переопределяется, чтобы допускать пользователя на сайт после его успешной регистрации.

Внутри каталога приложения `students` создайте новый файл и назовите его `urls.py`. Добавьте в него следующий ниже исходный код:

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

```
urlpatterns = [
 path('register/',
 views.StudentRegistrationView.as_view(),
 name='student_registration'),
]
```

Затем отредактируйте главный файл `urls.py` проекта `educa`, вставив URL-адреса приложения `students`, добавив следующий ниже шаблон в конфигурацию URL-адресов:

```
urlpatterns = [
 # ...
 path('students/', include('students.urls')),
]
```

Внутри каталога приложения `students` создайте такую файловую структуру:

```
templates/
 students/
 student/
 registration.html
```

Отредактируйте шаблон `student/student/registration.html`, добавив в него следующий ниже исходный код:

```
{% 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 method="post">
 {{ form.as_p }}
 {% csrf_token %}
 <p><input type="submit" value="Create my account"></p>
 </form>
 </div>
{% endblock %}
```

Запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/students/register/` в своем браузере. Вы должны увидеть вот такую регистрационную форму:

The screenshot shows a registration form titled "Sign up". The form asks for details to create an account. It includes fields for "Username" (with a note that it's required and can have up to 150 characters), "Password", and "Password confirmation". There is also a note about password requirements. A green button at the bottom says "CREATE MY ACCOUNT".

Sign up

Enter your details to create an account:

**Username:** Required. 150 characters or fewer. Letters, digits and @/./+/-/\_ only.

**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.

**Password confirmation:** Enter the same password as before, for verification.

**CREATE MY ACCOUNT**

Рис. 14.3. Форма для регистрации студента

Обратите внимание, что указанный в атрибуте `success_url` представления `StudentRegistrationView` URL-адрес `student_course_list` пока еще не существует. Если отправить форму на обработку, то Django не найдет URL-адрес перенаправления после успешной регистрации. Как уже упоминалось, этот URL-адрес будет создан в разделе «Доступ к содержимому курсов».

## Зачисление на курсы

После того как пользователи создадут учетную запись, они смогут зачисляться на курсы. Для хранения зачислений необходимо создать взаимосвязь многие-ко-многим между моделями `Course` и `User`.

Отредактируйте файл `models.py` приложения `courses`, добавив следующее ниже поле в модель `Course`:

```
students = models.ManyToManyField(User,
 related_name='courses_joined',
 blank=True)
```

В оболочке исполните следующую ниже команду, чтобы создать миграцию для этого изменения:

```
python manage.py makemigrations
```

Вы увидите примерно такой результат:

```
Migrations for 'courses':
 courses/migrations/0004_course_students.py
 - Add field students to course
```

Затем исполните показанную далее команду, чтобы применить намеченные миграции:

```
python manage.py migrate
```

Вы должны увидеть результат, который заканчивается такой строкой:

```
Applying courses.0004_course_students... OK
```

Теперь можно связывать студентов с курсами, на которые они зачислены. Давайте создадим функциональность зачисления студентов на курсы.

Внутри каталога приложения `students` создайте новый файл и назовите его `forms.py`. Добавьте в него следующий ниже исходный код:

```
from django import forms
from courses.models import Course

class CourseEnrollForm(forms.Form):
 course = forms.ModelChoiceField(
 queryset=Course.objects.all(),
 widget=forms.HiddenInput)
```

Приведенная выше форма будет использоваться для зачисления студентов на курсы. Поле `course` предназначено для курса, на который пользователь будет зачислен; следовательно, это поле `ModelChoiceField`. При этом используется виджет `HiddenInput`, потому что это поле не будет показываться пользователю. Указанная форма будет использоваться в представлении `CourseDetailview`, чтобы отображать кнопку зачисления.

Отредактируйте файл `views.py` приложения `students`, добавив следующий ниже исходный код:

```
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().form_valid(form)

 def get_success_url(self):
 return reverse_lazy('student_course_detail',
 args=[self.course.id])
```

Это представление `StudentEnrollCourseView`. Оно обслуживает зачисляемых на курсы студентов. Указанное представление наследует от примеси `LoginRequiredMixin`, поэтому получать доступ к данному представлению могут только вошедшие на сайт пользователи. Оно также наследует от встроенного в Django представления `FormView`, так как в его функции входит передача формы на обработку. Форма `CourseEnrollForm` используется для атрибута `form_class`. Помимо этого, для хранения данного объекта `Course` определяется атрибут `course`. Если форма валидна, то текущий пользователь добавляется к зачисленным на курс студентам.

Метод `get_success_url()` возвращает URL-адрес, на который пользователь будет перенаправлен, если форма была успешно передана на обработку. Этот метод эквивалентен атрибуту `success_url`. Затем URL-адрес с именем `student_course_detail` переворачивается.

Отредактируйте файл `urls.py` приложения `students`, добавив в него следующий ниже шаблон URL:

```
path('enroll-course/',
 views.StudentEnrollCourseView.as_view(),
 name='student_enroll_course'),
```

Давайте добавим форму с кнопкой зачисления на страницу краткого обзора курса. Отредактируйте файл `views.py` приложения `courses`, видоизменив класс `CourseDetailView` так, чтобы он выглядел следующим образом:

```
from students.forms import CourseEnrollForm
```

```
class CourseDetailView(DetailView):
 model = Course
 template_name = 'courses/course/detail.html'

 def get_context_data(self, **kwargs):
 context = super().get_context_data(**kwargs)
 context['enroll_form'] = CourseEnrollForm(
 initial={'course': self.object})
 return context
```

Метод `get_context_data()` используется для того, чтобы вставлять форму для зачисления в контекст прорисовки шаблонов. Скрытое поле `course` формы инициализируется текущим объектом `Course`, чтобы его можно было передавать на обработку напрямую.

Откройте в редакторе шаблон `courses/course/detail.html` и найдите следующую ниже строку:

```
 {{ object.overview|linebreaks }}
```

Замените ее таким исходным кодом:

```
 {{ object.overview|linebreaks }}
 {% if request.user.is_authenticated %}
 <form action="{% url "student_enroll_course" %}" method="post">
 {{ enroll_form }}
 {% csrf_token %}
 <input type="submit" value="Enroll now">
 </form>
 {% else %}

 Register to enroll

 {% endif %}
```

Это кнопка зачисления на курсы. Если пользователь аутентифицирован, то отображается кнопка зачисления, содержащая скрытую форму, указывающую на URL-адрес `student_enroll_course`. Если пользователь не аутентифицирован, отображается ссылка на регистрацию на платформе.

Проверьте, чтобы сервер разработки был запущен, пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере и кликните по курсу. Если вы вошли на платформу, то должны увидеть кнопку **ENROLL NOW** (Записаться сейчас), расположенную под кратким обзором курса, как показано ниже:

## Overview

Programming. 2 modules. Instructor: Antonio Melé

Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

ENROLL NOW

Рис. 14.4. Страница краткого обзора курса, включающая кнопку ENROLL NOW

Если вы не вошли на платформу, то вместо этого увидите кнопку REGISTER TO ENROLL (Зарегистрироваться, чтобы записаться).

## Доступ к содержимому курсов

Далее требуется представление для отображения курсов, на которые зачислены студенты, и представление для доступа к фактическому содержимому курсов. Отредактируйте файл `views.py` приложения `students`, добавив в него следующий ниже исходный 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().get_queryset()
 return qs.filter(students__in=[self.request.user])
```

Это представление для просмотра курсов, на которые зачислены студенты. Оно наследует от примесного класса `LoginRequiredMixin`, чтобы обеспечивать доступ к представлению только вошедшим на платформу пользователям. А также наследует от типового представления `ListView`, чтобы отображать список объектов `Course`. Далее переопределяется метод `get_queryset()`, чтобы извлекать только те курсы, на которые студент зачислен; для этого выполняется фильтрация набора запросов `QuerySet` по полю `ManyToManyField` студента.

Затем добавьте следующий ниже исходный код в файл `views.py` приложения `students`:

```
from django.views.generic.detail import DetailView

class StudentCourseDetailView(DetailView):
 model = Course
 template_name = 'students/course/detail.html'

 def get_queryset(self):
 qs = super().get_queryset()
 return qs.filter(students__in=[self.request.user])

 def get_context_data(self, **kwargs):
 context = super().get_context_data(**kwargs)
 # получить объект Course
 course = self.get_object()
 if 'module_id' in self.kwargs:
 # взять текущий модуль
 context['module'] = course.modules.get(
 id=self.kwargs['module_id'])
 else:
 # взять первый модуль
 context['module'] = course.modules.all()[0]
 return context
```

Это представление `StudentCourseDetailView`. В нем переопределяется метод `get_queryset()`, чтобы ограничивать базовый набор запросов `QuerySet` курсами, на которые зачислен студент. Кроме того, переопределяется метод `get_context_data()`, чтобы устанавливать модуль курса в контекст, если задан URL-параметр `module_id`. В противном случае задается первый модуль курса. Благодаря этому студенты смогут перемещаться по модулям внутри курса.

Отредактируйте файл `urls.py` приложения `students`, добавив в него следующие ниже шаблоны URL-адресов:

```
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'),
```

Внутри каталога `templates/students/` приложения `students` создайте следующую ниже файловую структуру:

```
course/
 detail.html
 list.html
```

Отредактируйте шаблон `student/course/list.html`, добавив в него такой исходный 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>
 Access contents</p>
 </div>
 {% empty %}
 <p>
 You are not enrolled in any courses yet.
 Browse courses
 to enroll on a course.
 </p>
 {% endfor %}
</div>
{% endblock %}
```

Этот шаблон отображает курсы, на которые зачислен студент. Напомним, что когда новый студент успешно регистрируется на платформе, он перенаправляется на URL-адрес `student_course_list`. Давайте также будем перенаправлять студентов на этот URL-адрес при входе на платформу.

Отредактируйте файл `settings.py` проекта `educa`, добавив в него следующий ниже исходный код:

```
from django.urls import reverse_lazy

LOGIN_REDIRECT_URL = reverse_lazy('student_course_list')
```

Это настроечный параметр, используемый модулем `auth` для перенаправления студента после успешного входа на платформу, если в запросе нет параметра `next`. После успешного входа студент будет перенаправлен на URL-адрес `student_course_list`, чтобы просмотреть курсы, на которые он зачислен.

Отредактируйте шаблон `student/course/detail.html`, добавив в него следующий ниже исходный код:

```

{% 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 %}

```

Проверьте, чтобы ни один шаблонный тег не был разбит на несколько строк. Это шаблон доступа зачисленных студентов к содержимому курса. Сперва создается HTML-список, содержащий все модули курса и выделяющий текущий модуль. Затем содержимое текущего модуля прокручивается в цикле, и, используя {{ item.render }}, извлекается каждый элемент содержимого, чтобы его отобразить. Далее к моделям содержимого добавляется метод render(). Этот метод служит для надлежащей прорисовки содержимого.

Теперь можно обратиться по адресу <http://127.0.0.1:8000/students/register/>, зарегистрировать новую учетную запись студента и записаться на любой курс.

## Прорисовка разных типов содержимого

Для того чтобы отобразить содержимое курса, необходимо прорисовывать созданные вами типы содержимого: *текст, изображение, видео и файл*.

Отредактируйте файл `models.py` приложения `courses`, добавив следующий ниже метод `render()` в модель `ItemBase`:

```
from django.template.loader import render_to_string

class ItemBase(models.Model):
 # ...
 def render(self):
 return render_to_string(
 f'courses/content/{self._meta.model_name}.html',
 {'item': self})
```

В приведенном выше методе используется функция `render_to_string()`, которая прорисовывает шаблон и возвращает прорисованное содержимое в виде строкового литерала. Каждый вид содержимого прорисовывается с использованием шаблона, названного по имени модели содержимого. Параметр `self._meta.model_name` используется для того, чтобы динамически создавать соответствующее имя шаблона для каждой модели содержимого. Метод `render()` предоставляет общий интерфейс для прорисовки разнoplанового контента.

Внутри каталога `templates/courses/` приложения `courses` создайте следующую ниже файловую структуру:

```
content/
 text.html
 file.html
 image.html
 video.html
```

Откройте в редакторе шаблон `courses/content/text.html` и напишите следующий ниже исходный код:

```
 {{ item.content|linebreaks }}
```

Это шаблон для прорисовки текстового содержимого. Шаблонный фильтр `linebreaks` заменяет разрывы строк в обычном тексте разрывами строк в формате HTML.

Отредактируйте шаблон `courses/content/file.html`, добавив в него следующее:

```
<p>
 Download file
</p>
```

Это шаблон прорисовки файлов. Здесь генерируется ссылка на скачивание файла.

Откройте шаблон `courses/content/image.html` и напишите:

```
<p>

</p>
```

Это шаблон для прорисовки изображений.

Кроме того, необходимо создать шаблон прорисовки объектов `Video`. Для встраивания видеоконтента вы будете использовать `django-embed-video`. Это стороннее приложение Django, которое позволяет вставлять видео в шаблоны из таких источников, как YouTube и Vimeo, просто предоставляя их общедоступный URL-адрес.

Следующей ниже командой установите пакет:

```
pip install django-embed-video==1.4.4
```

Отредактируйте файл `settings.py` проекта, добавив приложение в настроенный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [
 # ...
 'embed_video',
]
```

Документация по приложению `django-embed-video` находится на странице <https://django-embed-video.readthedocs.io/en/latest/>.

Откройте шаблон `courses/content/video.html` и напишите следующий ниже исходный код:

```
{% load embed_video_tags %}
{% video item.url "small" %}
```

Это шаблон прорисовки видео.

Теперь запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/course/mine/` в своем браузере. Получите доступ к сайту с пользователем, принадлежащим к группе `Instructors`, и добавьте несколько элементов содержимого в курс. Для того чтобы вставить видеоконтент, можно просто скопировать любой URL-адрес YouTube, например <https://www.youtube.com/watch?v=bgV39DlmZ2U>, и вставить его в поле `url` формы.

После добавления содержимого в курс пройдите по URL-адресу `http://127.0.0.1:8000/`, выберите курс и кликните по кнопке **ENROLL NOW** (Зарегистрироваться сейчас). Вы должны быть зачислены на курс и перенаправлены на URL-адрес `student_course_detail`. На рис. 14.5 показан пример страницы содержимого курса:

The screenshot shows a web application interface for 'EDUCA'. At the top, there's a green header bar with the word 'EDUCA' on the left and 'Sign out' on the right. Below the header, the title 'Introduction to Django' is displayed. On the left side, there's a sidebar titled 'Modules' containing four items: 'MODULE 1 Introduction to Django' (which is highlighted in grey), 'MODULE 2 Configuring Django', 'MODULE 3 Your first Django project', and 'MODULE 4 Django URLs'. The main content area on the right is titled 'Why Django?' and contains a brief introduction: 'Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.' Below this text is the title 'Django video'. A video player window is embedded, showing a presentation slide with the text 'In the background...' and a bulleted list: '• All aliases in a QuerySet can be changed at once  
– T1, T2, T3, ... -> U1, U2, U3, ...  
– for nested queries  
• QuerySets can be merged  
• Same table can appear with different aliases'. The video player has a play button and other controls. In the bottom right corner of the video player, there's a small 'New Relic' logo.

Рис. 14.5. Страница содержимого курса

Отлично! Вы создали общий интерфейс для прорисовки курсов с разными типами содержимого.

## Использование кеш-фреймворка

Обработка HTTP-запросов к веб-приложению обычно влечет за собой доступ к базе данных, манипулирование данными и прорисовку шаблона. Это намного дороже с точки зрения обработки, чем просто раздача статического веб-сайта. Накладные расходы в некоторых запросах бывают значительными в ситуациях, когда сайт начинает получать все больше и больше трафика. Именно здесь ценность кеширования проявляется себя ярче всего. Кеширование запросов, результатов вычислений или прорисованного содержимого в HTTP-запросе позволяет избегать дорогостоящих операций в последующих запросах, которые должны возвращать те же данные. Это приводит к сокращению времени отклика и уменьшению объема обработки на стороне сервера.

Django содержит устойчивую систему кеширования, которая позволяет кешировать данные с разными уровнями гранулярности. Можно кешировать

один запрос, результат конкретного представления, части содержимого про-рисованного шаблона либо весь сайт. Элементы хранятся в системе кеширо-вания в течение предустановленного времени, при этом при кешировании данных тайм-аут можно указывать.

Ниже приведено типовое применение кеш-фреймворка, когда приложе-ние обрабатывает HTTP-запрос.

1. Попытаться найти запрошенные данные в кеше.
2. Если они найдены, то вернуть кешированные данные.
3. Если они не найдены, то выполнить следующие действия:
  - 1) выполнить запрос к базе данных либо обработку, необходимые для генерирования данных;
  - 2) сохранить сгенерированные данные в кеше;
  - 3) вернуть данные.

Подробную информацию о встроенной в Django системе кеширования можно найти на странице <https://docs.djangoproject.com/en/4.1/topics/cache/>.

## Доступные кеш-бэкенды

Django поставляется со следующими кеш-бэкендами<sup>1</sup>:

- `backends.memcached.PyMemcacheCache` или `backends.memcached.PyLibMCCache`: бэкенды Memcached. Memcached – это быстрый и эффективный ре-зидентный кеш-сервер. Используемый бэкенд зависит от выбранных привязок Python к Memcached;
- `backends.redis.RedisCache`: бэкенд кеширования Redis. Этот бэкенд был добавлен в Django 4.0;
- `backends.db.DatabaseCache`: использует базу данных в качестве системы кеширования;
- `backends.filebased.FileBasedCache`: использует файловую систему хране-ния; сериализует и сохраняет каждое значение кеша в виде отдельного файла;
- `backends.locmem.LocMemCache`: кеш-бэкенд на основе локальной памяти. Этот кеш-бэкенд используется по умолчанию;
- `backends.dummy.DummyCache`: фиктивный кеш-бэкенд, предназначенный только для разработки. Он реализует кеш-интерфейс без фактического кеширования чего-либо. Этот кеш является попроцессным и потоко-безопасным.



В целях достижения оптимальной производительности следует использовать кеш-бэкенд на основе памяти, такой как бэкенды Memcached или Redis.

<sup>1</sup> Син. фоновый кеш-сервер. – Прим. перев.

## Установка резидентного кеш-сервера Memcached

Memcached – популярный высокопроизводительный резидентный кеш-сервер. Вы будете использовать кеш-сервер Memcached и бэкенд Memcached PyMemcacheCache.

### Установка образа Memcached платформы Docker

Запустите следующую ниже команду из оболочки, чтобы получить Docker-образ Memcached:

```
docker pull memcached
```

Она скачает образ Memcached платформы Docker на локальный компьютер. Если вы не хотите использовать Docker, то можете загрузить Memcached со страницы <https://memcached.org/downloads>.

Следующей ниже командой запустите контейнер Memcached платформы Docker:

```
docker run -it --rm --name memcached -p 11211:11211 memcached -m 64
```

По умолчанию Memcached работает на порту 11211. Опция `-p` используется для публикации порта 11211 на тот же порт интерфейса хоста. Опция `-m` используется для ограничения памяти контейнера до 64 Мб. Memcached работает резидентно, то есть в памяти, и ему выделяется определенный объем оперативной памяти. Когда выделенная оперативная память заполнена, Memcached начинает удалять самые старые данные, чтобы сохранять новые. Если вы хотите запустить команду в автономном режиме (в фоновом режиме терминала), то можете использовать параметр `-d`.

Дополнительная информация о Memcached находится на странице <https://memcached.org>.

### Установка привязки Python к Memcached

После установки Memcached необходимо установить привязку Python к Memcached. Мы установим быстрый клиент Memcached на чистом Python `rutmemcache`. Выполните следующую ниже команду в оболочке:

```
pip install pymemcache==3.5.2
```

Подробнее о библиотеке `rutmemcache` можно почитать на странице <https://github.com/pinterest/pymemcache>.

## Настроочные параметры кеша

Django предоставляет следующие настроочные параметры кеша:

- CACHES: словарь, содержащий все доступные проекту кеши;
- CACHE\_MIDDLEWARE\_ALIAS: используемый для хранения псевдоним кеша;
- CACHE\_MIDDLEWARE\_KEY\_PREFIX: префикс ключей кеша. Префикс устанавливается во избежание конфликтов ключей, если один и тот же кеш используется на нескольких сайтах;
- CACHE\_MIDDLEWARE\_SECONDS: принятое по умолчанию число секунд кеширования страниц.

Систему кеширования в проекте можно конфигурировать с помощью настроочного параметра CACHES. Этот параметр позволяет указывать конфигурацию нескольких кешей. Каждый включенный в словарь CACHES кеш может указывать следующие данные:

- BACKEND: подлежащий использованию кеш-бэкенд;
- KEY\_FUNCTION: строковый литерал, содержащий точечный путь к вызываемому объекту, который принимает префикс, версию и ключ в качестве аргументов и возвращает окончательный ключ кеша;
- KEY\_PREFIX: строковый префикс для всех ключей кеша во избежание конфликтов;
- LOCATION: местоположение кеша. В зависимости от кеша-бэкенда это может быть каталог, хост и порт или имя резидентного бэкенда;
- OPTIONS: любые дополнительные параметры, которые должны передаваться в кеш-бэкенд;
- TIMEOUT: принятый по умолчанию тайм-аут в секундах для хранения ключей кеша. По умолчанию равен 300 секундам, то есть 5 минутам. Если установлено значение None, то срок действия ключей кеша не истекает;
- VERSION: принятый по умолчанию номер версии ключей кеша. Удобен для управления версиями кеша.

## Добавление кеш-сервера Memcached в проект

Давайте сконфигурируем кеш под ваш проект. Отредактируйте файл settings.py проекта educa, добавив в него следующий ниже исходный код:

```
CACHES = {
 'default': {
 'BACKEND': 'django.core.cache.backends.memcached.PyMemcacheCache',
 'LOCATION': '127.0.0.1:11211',
 }
}
```

Здесь используется бэкенд PyMemcacheCache. При этом его местоположение задается посредством нотации адрес:порт. Если есть несколько экземпляров кеш-сервера Memcached, то в LOCATION можно использовать список.

Вы настроили резидентный кеш-сервер Memcached под свой проект. Начнем кешировать данные!

## Уровни кеша

Django предоставляет следующие уровни кеша, перечисленные здесь в порядке возрастания гранулярности:

- **низкоуровневый API кеша**: обеспечивает высочайшую степень гранулярности. Позволяет кешировать конкретные запросы или вычисления;
- **кеш шаблона**: позволяет кешировать фрагменты шаблона;
- **кеш представлений**: обеспечивает кеширование отдельных представлений;
- **сайтовый кеш**: кеш самого высокого уровня. Он кеширует весь сайт.



Перед тем как внедрять кеширование, следует подумать о своей стратегии кеширования. Сначала сосредоточьтесь на дорогостоящих запросах или вычислениях, которые не рассчитываются для каждого пользователя.

Давайте начнем и первым делом научимся использовать низкоуровневый API кеша в исходном коде Python.

## Использование низкоуровневого API кеша

Низкоуровневый API кеша позволяет хранить объекты в кеше с любой степенью гранулярности. Он находится в `django.core.cache` и импортируется следующим образом:

```
from django.core.cache import cache
```

Он использует предустановленный (или дефолтный) кеш. Это эквивалентно `caches['default']`. Доступ к конкретному кешу также возможен по его псевдониму:

```
from django.core.cache import caches
my_cache = caches['alias']
```

Давайте взглянем на работу API кеша. Следующей ниже командой откройте оболочку Django:

```
python manage.py shell
```

Исполните такой исходный код:

```
>>> from django.core.cache import cache
>>> cache.set('musician', 'Django Reinhardt', 20)
```

Вы обращаетесь к используемому по умолчанию кеш-бэкенду и применяете `set(key, value, timeout)`, чтобы сохранить и удерживать ключ с именем `'musician'` со значением в виде строкового литерала `'Django Reinhardt'` в течение 20 секунд. Если вы не укажете тайм-аут, то Django будет использовать тайм-аут, заданный по умолчанию, который указан для кеш-бэкенда в настроичном параметре `CACHES`. Теперь исполните следующий ниже исходный код:

```
>>> cache.get('musician')
'Django Reinhardt'
```

Вы извлекаете ключ из кеша. Подождите 20 секунд и исполните тот же исходный код:

```
>>> cache.get('musician')
```

На этот раз значение не возвращается. Срок действия ключа `'musician'` кеша истек, и метод `get()` возвращает `None`, потому что ключа больше в кеше нет.



Всегда избегайте хранения значения `None` в ключе кеша, потому что таким образом вы не сможете отличать фактическое значение от непопадания в кеш<sup>1</sup>.

Давайте с помощью следующего ниже исходного кода занесем в кеш набор запросов `QuerySet`:

```
>>> from courses.models import Subject
>>> subjects = Subject.objects.all()
>>> cache.set('my_subjects', subjects)
```

Здесь набор запросов `QuerySet` исполняется на модели `Subject`, и возвращенные объекты сохраняются в ключе `my_subjects`. Давайте извлечем кешированные данные:

```
>>> cache.get('my_subjects')
<QuerySet [<Subject: Mathematics>, <Subject: Music>, <Subject: Physics>,
<Subject: Programming>]>
```

<sup>1</sup> Англ. `cache miss`. – Прим. перев.

Вы планируете кешировать запросы в представлениях. Отредактируйте файл `views.py` приложения `courses`, добавив следующую ниже инструкцию импорта:

```
from django.core.cache import cache
```

В методе `get()` объекта `CourseListView` найдите следующие ниже строки:

```
subjects = Subject.objects.annotate(
 total_courses=Count('courses'))
```

Замените эти строки на такие:

```
subjects = cache.get('all_subjects')
if not subjects:
 subjects = Subject.objects.annotate(
 total_courses=Count('courses'))
 cache.set('all_subjects', subjects)
```

В приведенном выше исходном коде делается попытка получить ключ `all_students` из кеша с помощью метода `cache.get()`. Этот метод возвращает `None`, если данный ключ не найден. Если ключ не найден (еще не кеширован или кеширован, но истек тайм-аут), то выполняется запрос, чтобы извлечь все объекты `Subject` и их число курсов, и результат кешируется с помощью метода `cache.set()`.

## Проверка запросов к кешу с помощью меню отладочных инструментов Django Debug Toolbar

Давайте добавим меню отладочных инструментов Django Debug Toolbar в проект, чтобы проверять запросы к кешу. Вы научились использовать меню отладочных инструментов Django в главе 7 «Отслеживание действий пользователя».

Сначала следующей ниже командой установите меню отладочных инструментов Django Debug Toolbar:

```
pip install django-debug-toolbar==3.6.0
```

Отредактируйте файл `settings.py` проекта, добавив `debug_toolbar` в настроекный параметр `INSTALLED_APPS`, как показано ниже. Новая строка выделена жирным шрифтом:

```
INSTALLED_APPS = [
 # ...
```

```
'debug_toolbar',
]
```

В том же файле добавьте в настроечный параметр MIDDLEWARE следующую ниже строку, выделенную жирным шрифтом:

```
MIDDLEWARE = [
 'debug_toolbar.middleware.DebugToolbarMiddleware',
 'django.middleware.security.SecurityMiddleware',
 'django.contrib.sessions.middleware.SessionMiddleware',
 'django.middleware.common.CommonMiddleware',
 'django.middleware.csrf.CsrfViewMiddleware',
 'django.contrib.auth.middleware.AuthenticationMiddleware',
 'django.contrib.messages.middleware.MessageMiddleware',
 'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Напомним, что промежуточный компонент DebugToolbarMiddleware необходимо размещать перед любым другим промежуточным компонентом, за исключением промежуточного компонента, кодирующего содержимое ответа, например GZipMiddleware, который, если он присутствует, должен стоять первым.

В конец файла `settings.py` добавьте такие строки:

```
INTERNAL_IPS = [
 '127.0.0.1',
]
```

Меню отладочных инструментов Django будет отображаться только в том случае, если ваш IP-адрес соответствует записи в настроечном параметре `INTERNAL_IPS`.

Отредактируйте главный файл `urls.py` проекта, добавив следующий ниже шаблон URL-адреса в `urlpatterns`:

```
path('__debug__/', include('debug_toolbar.urls'))]
```

Запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере. Теперь вы должны увидеть меню отладочных инструментов Django в правой части страницы. Кликните по **Cache** (Кеш) в боковом меню. Вы увидите следующую ниже панель:

**Cache calls from 1 backend**

Total calls	Total time	Cache hits	Cache misses
2	103.26123237609863 ms	0	1

**Commands**

add	get	set	get_or_set	touch	delete	clear	get_many	set_many	delete_many	has_key	incr	decr	incr_version	decr_version
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

**Calls**

Time (ms)	Type	Arguments	Keyword arguments	Backend
76.8130	get	('all_subjects',)	0	<django.core.cache.backends.memcached.PyMemcacheCache object at 0x107f633a0>
26.4482	set	('all_subjects', <QuerySet [<Subject: Mathematics>, <Subject: Music>, <Subject: Physics>, <Subject: Programming>]>)	0	<django.core.cache.backends.memcached.PyMemcacheCache object at 0x107f633a0>

Рис. 14.6. Панель **Cache** меню отладочных инструментов Django, включающая запросы к кешу для представления *CourseListView* при непопадании в кеш

В разделе **Total calls** (Всего вызовов) вы должны увидеть 2. При первом исполнении представления *CourseListView* есть два запроса к кешу. В разделе **Commands** (Команды) вы увидите, что команда *get* была исполнена один раз и что команда *set* тоже была исполнена один раз. Команда *get* соответствует вызову, который извлекает ключ *all\_subjects* кеша. Это первый вызов, отображаемый в разделе **Calls** (Вызовы). При первом исполнении представления происходит непопадание в кеш, потому что данные еще не кешированы. Именно по этой причине в **Cache misses** (Непопадания в кеш) стоит 1. Затем команда *set* используется для сохранения в кеше результатов набора запросов на получение предметов с использованием ключа *all\_subjects* кеша. Это второй вызов, отображаемый в разделе **Calls**.

В пункте **SQL** меню отладочных инструментов Django вы увидите общее число SQL-запросов, исполненных в этом запросе. Сюда входит запрос на получение всех предметов, которые затем сохраняются в кеше:



Рис. 14.7. SQL-запросы, исполненные для представления *CourseListView* при непопадании в кеш

Перезагрузите страницу в браузере и кликните по **Cache** в боковом меню:

The screenshot shows the 'Cache' panel of the Django Debug Toolbar. It has a yellow header bar with the title 'Cache calls from 1 backend'. Below it is a 'Summary' section with a table:

Total calls	Total time	Cache hits	Cache misses
1	23.750782012939453 ms	1	0

Below the summary is a 'Commands' section with a table:

add	get	set	get_or_set	touch	delete	clear	get_many	set_many	delete_many	has_key	incr	decr	incr_version	decr_version
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Finally, there is a 'Calls' section with a table:

Time (ms)	Type	Arguments	Keyword arguments	Backend
23.7508	get	('all_subjects',)	0	<django.core.cache.backends.memcached.PyMemcacheCache object at 0x107cbbc10>

Рис. 14.8. Панель **Cache** меню отладочных инструментов Django, включающая запросы к кешу для представления *CourseListView* при попадании в кеш

Теперь есть только один запрос к кешу. В разделе **Total calls** (Всего вызовов) вы должны увидеть 1, а в разделе **Commands** (Команды) – что запрос к кешу соответствует команде get. В данном случае происходит попадание в кеш (см. **Cache hits** – Попадания в кеш), а не непопадание в кеш, поскольку данные были найдены в кеше. В разделе **Calls** (Вызовы) вы увидите запрос на получение ключа all\_subjects кеша.

Проверьте пункт **SQL** меню отладочных инструментов. Вы должны увидеть, что в этом запросе на один SQL-запрос меньше. Сохраняется всего один SQL-запрос, потому что представление находит данные в кеше и не требует их извлечения из базы данных:

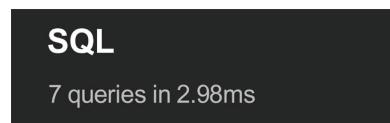


Рис. 14.9. SQL-запросы, выполненные для представления *CourseListView* при попадании в кеш

В этом примере на один запрос на извлечение элемента из кеша требуется больше времени, чем время, сэкономленное на дополнительном SQL-запросе. Однако когда к сайту будет обращаться большое число пользователей, вы обнаружите значительное сокращение времени за счет извлечения данных из кеша, без обращения к базе данных, и вы сможете раздавать сайт большему числу concurrentных пользователей.

Поочередные запросы к одному и тому же URL-адресу будут извлекать данные из кеша. Поскольку при кешировании данных с помощью cache.

`set('all_subjects', subject)` в представлении `CourseListView` тайм-аут указан не был, будет использоваться тайм-аут, принятый по умолчанию (300 секунд, что составляет 5 минут). По истечении тайм-аута следующий запрос к URL-адресу вызовет непопадание в кеш, будет выполнен набор запросов `QuerySet`, и данные будут кешированы еще на 5 минут. При этом в элементе `TIMEOUT` настроечного параметра `CACHES` можно определять другой тайм-аут, который будет использоваться по умолчанию.

## Кеширование на основе динамических данных

Нередко приходится кешировать что-то, основанное на динамических данных. В этих случаях необходимо создавать динамические ключи, содержащие всю информацию, необходимую для уникальной идентификации кешированных данных.

Отредактируйте файл `views.py` приложения `courses`, видоизменив представление `CourseListView`, приведя его к следующему виду:

```
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 = f'subject_{subject.id}_courses'
 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})
```

В данном случае кешируются как все курсы, так и курсы, отфильтрованные по предмету. Ключ `all_courses` кеша используется для хранения всех курсов,

если предмет не указан. Если предмет есть, то ключ создается динамически посредством инструкции `f'subject_{subject.id}_courses'`.

Важно отметить, что использовать кешированный набор запросов `QuerySet` для создания других наборов запросов невозможно, поскольку то, что кешируется, на самом деле является результатом набора запросов `QuerySet`. И поэтому нельзя сделать следующее:

```
courses = cache.get('all_courses')
courses.filter(subject=subject)
```

Вместо этого нужно создать базовый набор запросов `QuerySet Course.objects.annotate(total_modules=Count('modules'))`, который не будет исполняться до тех пор, пока он не будет принудительно выполнен, и использовать его для дальнейшего ограничения набора запросов с помощью `all_courses.filter(subject=subject)` в случае, если данные не были найдены в кеше.

## Кеширование фрагментов шаблона

Кеширование фрагментов шаблона – это подход более высокого уровня. Для этого необходимо загрузить шаблонные теги кеша в шаблон с помощью `{% load cache %}`, а затем использовать шаблонный тег `{% cache %}`, чтобы кешировать те или иные фрагменты шаблона. Шаблонный тег обычно применяется следующим образом:

```
{% cache 300 fragment_name %}
...
{% endcache %}
```

Шаблонный тег `{% cache %}` имеет два обязательных аргумента: тайм-аут в секундах и имя фрагмента. Если нужно кешировать содержимое в зависимости от динамических данных, то это можно сделать, передав шаблонному тегу `{% cache %}` дополнительные аргументы, чтобы уникально идентифицировать фрагмент.

Откройте в редакторе файл `/students/course/detail.html` приложения `students`. Добавьте следующий ниже исходный код в верхнюю его часть, сразу после тега `{% extends %}`:

```
{% load cache %}
```

Затем найдите такие строки:

```
{% for content in module.contents.all %}
 {% with item=content.item %}
 <h2>{{ item.title }}</h2>
 {{ item.render }}
```

```
{% endwith %}
{% endfor %}
```

Замените их следующими:

```
{% cache 600 module_contents module %}
 {% for content in module.contents.all %}
 {% with item=content.item %}
 <h2>{{ item.title }}</h2>
 {{ item.render }}
 {% endwith %}
 {% endfor %}
 {% endcache %}
```

Приведенный выше фрагмент шаблона кешируется, используя имя `module_contents`, при этом ему передается текущий объект `Module`. Таким образом, фрагмент идентифицируется однозначно. Очень важно избегать кеширования содержимого модуля и раздачи неправильного содержимого при запросе другого модуля.

Если значение настроечного параметра `USE_I18N` установлено равным `True`, то сайтовый кеш в промежуточном компоненте будет учитывать активный язык. Если используется шаблонный тег `{% cache %}`, то для достижения того же результата необходимо использовать одну из имеющихся в шаблонах переводных переменных, например `{% cache 600 name request.LANGUAGE_CODE %}`.

## Кеширование представлений

С помощью декоратора `cache_page`, расположенного в `django.views.decorators.cache`, можно кешировать результат отдельных представлений. Декоратору требуется аргумент `timeout` (в секундах).

Давайте применим его в ваших представлениях. Отредактируйте файл `urls.py` приложения `students`, добавив в него следующую ниже инструкцию импорта:

```
from django.views.decorators.cache import cache_page
```

Затем примените декоратор `cache_page` к шаблонам URL-адресов `student_course_detail` и `student_course_detail_module`, 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 полный контент, возвращаемый представлением `StudentCourseDetailview`, кешируется на 15 минут.



В кеше по каждому представлению для формирования ключа кеша используется URL-адрес. Несколько URL-адресов, указывающих на одно и то же представление, будут кешироваться отдельно.

## Использование сайтового кеша

Это кеш наивысшего уровня. Он позволяет кешировать весь сайт. Для того чтобы разрешить сайтовый кеш, отредактируйте файл `settings.py` проекта, добавив классы `UpdateCacheMiddleware` и `FetchFromCacheMiddleware` в настроочный параметр `MIDDLEWARE`, как показано ниже:

```
MIDDLEWARE = [
 'debug_toolbar.middleware.DebugToolbarMiddleware',
 'django.middleware.security.SecurityMiddleware',
 'django.contrib.sessions.middleware.SessionMiddleware',
 'django.middleware.cache.UpdateCacheMiddleware',
 'django.middleware.common.CommonMiddleware',
 'django.middleware.cache.FetchFromCacheMiddleware',
 'django.middleware.csrf.CsrfViewMiddleware',
 'django.contrib.auth.middleware.AuthenticationMiddleware',
 'django.contrib.messages.middleware.MessageMiddleware',
 'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Напомним, что промежуточные программные компоненты исполняются в заданном порядке в фазе запроса и в обратном порядке в фазе ответа. Компонент `UpdateCacheMiddleware` размещается перед компонентом `CommonMiddleware`, потому что он запускается во время ответа, когда промежуточные компоненты исполняются в обратном порядке. Компонент `FetchFromCacheMiddleware` намеренно помещается после компонента `CommonMiddleware`, поскольку ему необходимо получить доступ к данным запроса, устанавливаемым последним из них.

Далее добавьте следующие ниже настроочные параметры в файл `settings.py`:

```
CACHE_MIDDLEWARE_ALIAS = 'default'
CACHE_MIDDLEWARE_SECONDS = 60 * 15 # 15 минут
CACHE_MIDDLEWARE_KEY_PREFIX = 'educa'
```

В приведенных выше настроочных параметрах промежуточного компонента кеша используется предустановленный (или дефолтный) кеш и устанавливается значение тайм-аута глобального кеша, равное 15 минутам.

Кроме того, во избежание коллизий, если один и тот же бэкенд Memcached используется в нескольких проектах, то для всех ключей кеша задается префикс. Теперь сайт будет кешироваться и возвращать кешированный контент для всех запросов методом GET.

С помощью меню отладочных инструментов Django можно обращаться к различным страницам и проверять запросы к кешу. Следует отметить, что сайтовый кеш непригоден для многих сайтов, поскольку он влияет на все представления, даже те, которые вы, возможно, не хотите кешировать, например управляющие представления, в которых подразумевается, что возвращаемые из базы данных данные будут отражать последние изменения.

В нашем проекте наилучшим подходом является кеширование шаблонов или представлений, используемых для отображения содержимого курса студентам, при этом представления управления содержимым для преподавателей лучше оставить без какого-либо кеширования.

Давайте отключим сайтовый кеш. Отредактируйте файл `settings.py` проекта, закомментировав классы `UpdateCacheMiddleware` и `FetchFromCacheMiddleware` в настроичном параметре `MIDDLEWARE`, как показано ниже:

```
MIDDLEWARE = [
 'debug_toolbar.middleware.DebugToolbarMiddleware',
 'django.middleware.security.SecurityMiddleware',
 'django.contrib.sessions.middleware.SessionMiddleware',
 # 'django.middleware.cache.UpdateCacheMiddleware',
 # 'django.middleware.common.CommonMiddleware',
 # 'django.middleware.cache.FetchFromCacheMiddleware',
 'django.middleware.csrf.CsrfViewMiddleware',
 'django.contrib.auth.middleware.AuthenticationMiddleware',
 'django.contrib.messages.middleware.MessageMiddleware',
 'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Вы увидели общий обзор разных методов, предоставляемых веб-фреймворком Django для кеширования данных. Следует разумно определять свою стратегию кеширования, принимая во внимание дорогостоящие наборы запросов или вычисления, данные, которые не будут изменяться часто, и данные, к которым будут обращаться многие пользователи в конкретном режиме.

## Использование кеш-бэкенда Redis

Django 4.0 ввел в свой состав кеш-бэкенд Redis. Давайте изменим настройки, чтобы использовать Redis вместо Memcached в качестве кеш-бэкенда проекта. Напомним, что вы уже использовали Redis в главе 7 «Отслеживание действий пользователя» и в главе 10 «Расширение магазина».

С помощью следующей команды установите redis-ру в своей среде:

```
pip install redis==4.3.4
```

Затем отредактируйте файл `settings.py` проекта `educa`, видоизменив настоечный параметр `CACHES`, как показано ниже:

```
CACHES = {
 'default': {
 'BACKEND': 'django.core.cache.backends.redis.RedisCache',
 'LOCATION': 'redis://127.0.0.1:6379',
 }
}
```

Теперь в проекте будет использоваться кеш-бэкенд `RedisCache`. Его местоположение определяется в формате `redis://[хост]:[порт]`. При этом используется адрес `127.0.0.1`, чтобы указывать на локальный хост, и порт `6379`, который для Redis используется по умолчанию.

Следующей ниже командой инициализируйте контейнер Redis платформы Docker:

```
docker run -it --rm --name redis -p 6379:6379 redis
```

Если вы хотите выполнить команду в фоновом (раздельном) режиме, то можете воспользоваться опцией `-d`.

Запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере. Проверьте запросы к кешу на панели **Cache** меню отладочных инструментов Django. Теперь в качестве кеш-бэкенда проекта вы используете Redis вместо Memcached.

## Отслеживание сервера Redis с помощью приложения Django Redisboard

Свой сервер Redis можно контролировать с помощью приложения Django Redisboard. Приложение Django Redisboard добавляет статистику сервера Redis на сайт администрирования. Дополнительная информация о Django Redisboard находится на странице <https://github.com/ionelmc/django-redisboard>.

Следующей ниже командой установите пакет `django-redisboard` в своей среде:

```
pip install django-redisboard==8.3.0
```

Посредством указанной далее команды установите используемую пакетом `django-redisboard` библиотеку Python `attrs` в своей среде:

```
pip install attrs
```

Отредактируйте файл `settings.py` проекта, добавив приложение в настроекий параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [
 # ...
 'redisboard',
]
```

Выполните следующую ниже команду из каталога проекта, чтобы запустить миграцию приложения Django Redisboard:

```
python manage.py migrate redisboard
```

Запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/admin/redisboard/redisserver/add/` в своем браузере, чтобы добавить сервер Redis и начать его отслеживать. Под надписью **Label** (Метка) введите `redis`, а под надписью **URL** введите `redis://localhost:6379/0`, как показано на рис. 14.10:

The screenshot shows a form titled 'Add Redis Server'. It has three fields: 'Label' with the value 'redis', 'URL' with the value 'redis://localhost:6379/0', and 'Password' which is masked. Below the URL field, there is a note: 'IANA-compliant URL. Examples: redis://[[username]:[password]]@localhost:6379/0, rediss://[[username]:[password]]@localhost:6379/0, unix://[[username]:[password]]@/path/to/socket.sock?db=0'. A note at the bottom of the password field says: 'You can also specify the password here (the field is masked)'.

Рис. 14.10. Форма для добавления сервера Redis для приложения Django Redisboard на сайте администрирования

Мы будем отслеживать экземпляр сервера Redis, работающий на локальном хосте, который работает на порту 6379 и использует базу данных Redis под номером 0. Кликните по **SAVE** (Сохранить). Информация будет сохранена в базе данных, и вы сможете увидеть конфигурацию и метрики сервера Redis на сайте администрирования:

Select Redis Server to change									
Action:	NAME	STATUS	MEMORY	CLIENTS	DETAILS	CPU UTILIZATION	SLOWLOG	TOOLS	
<input type="checkbox"/>	redis	UP	1.29M (peak: 1.34M)	7	redis version redis mode os multiplexing api atomicvar api gcc version used memory used memory rss used memory peak total system memory human used memory lua used memory vm used memory scripts human maxmemory human maxmemory policy expired keys	7.0.4 standalone Linux 5.10.104- linuxkit aarch64 epoll c11-builtin 10.2.1 1.29M 8.14M 1.34M 7.76G 31.00K 63.00K 184B 0B noeviction 32	cpu utilization 0.002% used cpu sys 2318.116634 13.7ms INFO used cpu sys children 0.123416 13.6ms INFO used cpu user 1732.266923 11.2ms ZUNIONSTORE tmp_34 2 product:3:purchased_with product:4:purchased_with children	Total: 3 items 13.7ms INFO 13.6ms INFO 11.2ms ZUNIONSTORE tmp_34 2 product:3:purchased_with product:4:purchased_with children	Inspect Details

Рис. 14.11. Отслеживание сервера Redis приложением Django Redisboard на сайте администрирования

Поздравляем! Вы успешно реализовали кеширование в своем проекте.

## Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter14>.
- Документация по django-embed-video: <https://django-embed-video.readthedocs.io/en/last/>.
- Документация по кеш-фреймворку Django: <https://docs.djangoproject.com/en/4.1/topics/cache/>.
- Файлы кеш-бэкенда Memcached для скачивания: <https://memcached.org/downloads>.
- Официальный сайт кеш-бэкенда Memcached: <https://memcached.org>.
- Исходный код библиотеки pymemcache: <https://github.com/pinterest/pymemcache>.
- Исходный код приложения Django Redisboard: <https://github.com/ionelmc/django-redisboard>.

## Резюме

В этой главе вы реализовали общедоступные представления для каталога курсов. Вы создали систему регистрации и зачисления студентов на курсы. Вы также создали функциональность отображения различных типов содержимого модулей курса. Наконец, вы научились использовать кеш-фреймворк Django и использовали кеш-бэнды Memcached и Redis в своем проекте.

В следующей главе вы разработаете RESTful API к своему проекту с помощью фреймворка Django REST framework и будете использовать его посредством библиотеки Python `requests`.

# 15

## Разработка API

В предыдущей главе вы разработали систему регистрации и зачисления студентов на курсы. Вы создали представления, чтобы отображать содержимое курсов, и научились использовать кеш-фреймворк Django.

В этой главе вы создадите API в архитектурном стиле RESTful к своей платформе электронного обучения. API позволяет разрабатывать общее ядро, которое можно использовать на нескольких платформах, таких как веб-сайты, мобильные приложения, плагины и т. д. Например, можно создать API, который будет использоваться мобильным приложением платформы электронного обучения. Если предоставить API третьим сторонам, то они смогут потреблять информацию и работать с вашим приложением в программном режиме. API позволяет разработчикам автоматизировать действия на платформе и интегрировать ваш сервис с другими приложениями или онлайновыми сервисами. Вы разработаете полнофункциональный API к своей платформе электронного обучения.

В этой главе вы:

- установите фреймворк Django REST framework;
- создадите сериализаторы для своих моделей;
- разработаете RESTful API;
- создадите вложенные сериализаторы;
- разработаете конкретно-прикладные представления API;
- организуете аутентификацию по API;
- добавите разрешения для представлений API;
- создадите конкретно-прикладное разрешение;
- реализуете наборы представлений ViewSets и маршрутизаторы;
- будете использовать библиотеку `requests`, чтобы потреблять API.

Начнем с настройки API.

Исходный код этой главы можно найти по адресу <https://github.com/Packt-Publishing/Django-4-by-example/tree/main/Chapter15>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

# Разработка RESTful API

При создании API существует несколько способов структурирования его конечных точек и действий, но рекомендуется следовать принципам REST. Термин **REST** происходит от англ. **Representational State Transfer** – передача представительного (или самоописывающего) состояния. API в архитектурном стиле RESTful основаны на ресурсах; модели представляют ресурсы, а HTTP-методы, такие как GET, POST, PUT или DELETE, используются для извлечения, создания, обновления или удаления объектов. Коды HTTP-ответа также используются в этом контексте. Разные коды HTTP-ответа возвращаются, чтобы указывать результат HTTP-запроса, например коды 2XX ответа относятся к успеху, 4XX относятся к ошибкам и т. д.

Наиболее распространенными форматами обмена данными в RESTful API являются JSON и XML. Вы разработаете RESTful API с сериализацией JSON к своему проекту. Ваш API будет обеспечивать следующую функциональность:

- извлекать предметы;
- извлекать имеющиеся курсы;
- извлекать содержимое курса;
- записывать на курс.

API можно разработать с нуля с помощью Django, создав конкретно-прикладные представления. Однако несколько сторонних модулей упрощают создание API к проекту; наиболее популярным среди них является фреймворк Django REST framework<sup>1</sup>.

## Установка фреймворка Django REST framework

Фреймворк Django REST framework позволяет легко разрабатывать RESTful API к проектам. Вся информация о фреймворке Django REST framework находится по адресу <https://www.django-rest-framework.org/>.

Следующей ниже командой откройте оболочку и установите фреймворк:

```
pip install djangorestframework==3.13.1
```

Отредактируйте файл `settings.py` проекта `educa`, добавив `rest_framework` в настроочный параметр `INSTALLED_APPS`, чтобы активировать приложение, как показано ниже:

```
INSTALLED_APPS = [
 # ...
 'rest_framework',
]
```

<sup>1</sup> Данный фреймворк представляет собой гибкий инструментарий для разработки веб-API. – Прим. перев.

Затем добавьте следующий ниже исходный код в файл `settings.py`:

```
REST_FRAMEWORK = {
 'DEFAULT_PERMISSION_CLASSES': [
 'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
]
}
```

Для своего API можно указать конкретную конфигурацию, используя настроечный параметр `REST_FRAMEWORK`. Фреймворк REST предлагает широкий спектр настроечных параметров, чтобы конфигурировать настройки стандартных видов поведения. Настроечный параметр `DEFAULT_PERMISSION_CLASSES` указывает стандартные разрешения на чтение, создание, обновление или удаление объектов. Класс `DjangoModelPermissionsOrAnonReadOnly` задается в качестве единственного стандартного класса разрешений. Этот класс опирается на встроенную в Django систему разрешений, позволяющую пользователям создавать, обновлять или удалять объекты, предоставляя анонимным пользователям доступ только для чтения. Подробнее о разрешениях вы узнаете позже, в разделе «Добавление разрешений в представления».

Полный список имеющихся настроечных параметров для фреймворка REST находится на странице <https://www.django-rest-framework.org/api-guide/settings/>.

## Определение сериализаторов

После установки фреймворка REST необходимо задать способ сериализации данных. Выходные данные должны сериализоваться в конкретный формат, а входные данные – десериализоваться для обработки. Фреймворк предоставляет следующие классы, чтобы компоновать сериализаторы для одиночных объектов:

- `Serializer`: обеспечивает сериализацию обычных экземпляров класса Python;
- `ModelSerializer`: обеспечивает сериализацию экземпляров модели;
- `HyperlinkedModelSerializer`: тот же, что и `ModelSerializer`, но представляет объектные взаимосвязи со ссылками, а не с первичными ключами.

Давайте скомпонуем первый сериализатор. Внутри каталога приложения `courses` создайте следующую ниже файловую структуру:

```
api/
 __init__.py
 serializers.py
```

В целях поддержания исходного кода в хорошо организованном состоянии вы создадите всю функциональность API внутри каталога `api`. Отредактируйте файл `serializers.py`, добавив следующий ниже исходный код:

```
from rest_framework import serializers
from courses.models import Subject

class SubjectSerializer(serializers.ModelSerializer):
 class Meta:
 model = Subject
 fields = ['id', 'title', 'slug']
```

Это сериализатор модели `Subject`. Сериализаторы определяются аналогично классам Django `Form` и `ModelForm`. `Meta`-класс позволяет указывать подлежащую сериализации модель и поля, которые необходимо включать в сериализацию. Если не задать атрибут `fields`, то будут включены все поля модели.

Давайте испытаем сериализатор. Откройте командную оболочку и с помощью следующей команды запустите оболочку Django:

```
python manage.py shell
```

Выполните такой исходный код:

```
>>> 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 приведенном выше примере извлекается объект `Subject`, создается экземпляр `SubjectSerializer` и берутся сериализованные данные. Вы видите, что модельные данные транслируются в нативные для Python типы данных.

## Что такое парсер и рендерер

Перед тем как возвращать сериализованные данные в HTTP-ответе, они должны быть прорисованы в определенный формат. Аналогичным образом, перед тем как приступать к работе с входящими данными при получении HTTP-запроса, они должны быть разобраны и десериализованы. Фреймворк REST содержит служащие для этого рендереры (прорисовщики или трансляторы)<sup>1</sup> и парсеры (разборщики или синтаксико-структурные анализаторы).

Давайте посмотрим, как разбирать входящие данные. Выполните следующий ниже исходный код в оболочке Python:

<sup>1</sup> В качестве напоминания, в терминологии Django в рамках архитектуры MVP под рендерингом понимается взятие промежуточного представления шаблона вместе с контекстом и его превращение в итоговый поток байтов, иными словами: прорисовка шаблона, а в рамках API – трансляция данных в другой формат. – Прим. перев.

```
>>> 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'}
```

Получив входные данные в виде строкового литерала JSON, можно воспользоваться поставляемым с фреймворком REST классом `JSONParser`, чтобы конвертировать его в объект Python.

Фреймворк REST также содержит классы `Renderer`, которые позволяют форматировать API-ответы. Фреймворк определяет подлежащий использованию рендерер посредством согласования контента, обследуя заголовок `Accept` запроса, чтобы определить ожидаемый тип контента в ответе. При необходимости рендерер определяется форматным суффиксом URL-адреса. Например, URL-адрес `http://127.0.0.1:8000/api/data.json` может быть конечной точкой, которая запускает `JSONRenderer` для того, чтобы вернуть JSON-ответ.

Вернитесь в оболочку и исполните следующий ниже исходный код, чтобы оттранслировать объект `serializer` из предыдущего примера сериализатора:

```
>>> from rest_framework.renderers import JSONRenderer
>>> JSONRenderer().render(serializer.data)
```

Вы увидите такой результат:

```
b'{"id":4,"title":"Programming","slug":"programming"}'
```

Класс `JSONRenderer` применяется для трансляции сериализованных данных в JSON. По умолчанию фреймворк REST использует два разных рендерера: `JSONRenderer` и `BrowsableAPIRenderer`. Последний предоставляет веб-интерфейс для удобного просмотра API. С помощью опции `DEFAULT_RENDERER_CLASSES` настроекного параметра `REST_FRAMEWORK` можно менять классы-рендереры, которые применяются по умолчанию.

Более подробную информацию о рендерерах и парсерах можно найти соответственно на страницах <https://www.django-rest-framework.org/api-guide/renderers/> и <https://www.django-rest-framework.org/api-guide/parsers/>.

Далее вы научитесь разрабатывать представления API и использовать сериализаторы в представлениях.

## Разработка представлений списка и детальной информации

Фреймворк REST поставляется с набором типовых представлений и примесных классов, которые можно использовать для разработки представлений API. Они предоставляют функциональность извлечения, создания, обновления или удаления модельных объектов. Все типовые поставляемые с фрейм-

ворком REST примесные классы и представления находятся по адресу <https://www.djangoproject.org/api-guide/generic-views/>.

Давайте создадим представления списка и детальной информации, чтобы извлекать объекты `Subject`. Внутри каталога `courses/api` создайте новый файл и назовите его `views.py`. Добавьте в него следующий ниже исходный код:

```
from rest_framework import generics
from courses.models import Subject
from courses.api.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 приведенном выше исходном коде используются типовые представления `ListAPIView` и `RetrieveAPIView` фреймворка REST. URL-параметр `pk` вставлен в представление детальной информации, чтобы извлекать объект по данному первичному ключу.

Оба представления имеют следующие атрибуты:

- `queryset`: базовый набор запросов `QuerySet`, используемый для извлечения объектов;
- `serializer_class`: класс сериализации объектов.

Давайте добавим шаблоны URL-адресов представлений. Внутри каталога `courses/api` создайте новый файл, назовите его `urls.py` и придайте ему следующий вид:

```
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'),
]
```

Отредактируйте главный файл `urls.py` проекта `educa`, вставив шаблон URL-адреса API, как показано ниже:

```
urlpatterns = [
 # ...
 path('api/', include('courses.api.urls', namespace='api')),
]
```

Первоначальные конечные точки API теперь готовы к использованию.

## Потребление API

Для URL-адресов API используется именное пространство `api`. Следующей ниже командой проверьте, чтобы сервер работал:

```
python manage.py runserver
```

Для потребления API мы будем использовать инструмент командной строки `curl`, который позволяет передавать данные на сервер и с сервера назад клиенту. Если вы используете Linux, macOS или Windows 10/11, то инструмент `curl`, скорее всего, уже содержится в системе. Однако указанный инструмент можно скачать со страницы <https://curl.se/download.html>.

Откройте оболочку и извлеките URL-ресурс `http://127.0.0.1:8000/api/subjects/` с помощью `curl`, как показано ниже:

```
curl http://127.0.0.1:8000/api/subjects/
```

Вы получите ответ, аналогичный такому:

```
[
 {
 "id":1,
 "title":"Mathematics",
 "slug":"mathematics"
 },
 {
 "id":2,
 "title":"Music",
 "slug":"music"
 },
 {
 "id":3,
 "title":"Physics",
 "slug":"physics"
 },
 {
 "id":4,
 "title":"Programming",
```

```
 "slug": "programming"
 }
]
```

С целью получения более удобочитаемого JSON-ответа с правильным отступом можно воспользоваться утилитой `json_pp` вместе с `curl`, как показано ниже:

```
curl http://127.0.0.1:8000/api/subjects/ | json_pp
```

HTTP-ответ содержит список объектов `Subject` в формате JSON.

Вместо инструмента `curl` также можно задействовать любой другой инструмент отправки конкретно-прикладных HTTP-запросов, включая браузерное расширение, такое как Postman, которое можно получить по адресу <https://www.getpostman.com/>.

Пройдите по URL-адресу `http://127.0.0.1:8000/api/subjects/` в своем браузере. Вы увидите доступный для просмотра API-интерфейс фреймворка REST, как показано ниже:

The screenshot shows the 'Subject List' collection in Postman. At the top, there are buttons for 'OPTIONS', 'GET', and a dropdown menu. Below that is a search bar with the text 'GET /api/subjects/'. Underneath is a detailed response section:

**HTTP 200 OK**

**Allow:** GET, HEAD, OPTIONS  
**Content-Type:** application/json  
**Vary:** Accept

```
[
 {
 "id": 1,
 "title": "Mathematics",
 "slug": "mathematics"
 },
 {
 "id": 2,
 "title": "Music",
 "slug": "music"
 },
 {
 "id": 3,
 "title": "Physics",
 "slug": "physics"
 },
 {
 "id": 4,
 "title": "Programming",
 "slug": "programming"
 }
]
```

Рис. 15.1. Страница списка предметов  
в просматриваемом API-интерфейсе фреймворка REST

Этот HTML-интерфейс предоставляется рендерером `BrowsableAPIRenderer`. Он отображает результирующие заголовки и содержимое и позволяет выполнить запросы. Кроме того, есть возможность обращаться к представлению детальной информации об объекте `Subject`, включив его ИД в URL-адрес.

Пройдите по URL-адресу `http://127.0.0.1:8000/api/subjects/1/` в своем браузере. Вы увидите один объект `Subject`, транслированный в формат JSON.

```
GET /api/subjects/1/
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
 "id": 1,
 "title": "Mathematics",
 "slug": "mathematics"
}
```

Рис. 15.2. Страница детальной информации о предмете в просматриваемом API-интерфейсе фреймворка REST

Это ответ для представления `SubjectDetailView`. Далее мы остановимся на сериализаторах моделей подробнее.

## Создание вложенных сериализаторов

Мы собираемся создать сериализатор модели `Course`. Отредактируйте файл `api/serializers.py` приложения `courses`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from courses.models import Subject, Course

class CourseSerializer(serializers.ModelSerializer):
 class Meta:
 model = Course
 fields = ['id', 'subject', 'title', 'slug',
 'overview', 'created', 'owner',
 'modules']
```

Давайте посмотрим на сериализацию объекта `Course`. Откройте оболочку и выполните следующую ниже команду:

```
python manage.py shell
```

Выполните такой исходный код:

```
>>> 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)
```

Вы получите объект JSON с полями, которые вы включили в сериализатор `CourseSerializer`. Вы увидите, что связанные объекты менеджера `modules` сериализованы в виде списка первичных ключей следующим образом:

```
"modules": [6, 7, 9, 10]
```

Однако вы хотите получать больше информации о каждом модуле, и поэтому необходимо сериализовывать объекты `modules` и их вкладывать. Видоизмените приведенный выше исходный код файла `api/serializers.py` приложения `courses`, придав ему следующий вид:

```
from rest_framework import serializers
from courses.models import Subject, Course, 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']
```

В новом исходном коде определяется класс `ModuleSerializer`, чтобы обеспечивать сериализацию модели `Module`. Затем в класс `CourseSerializer` добавляется атрибут `modules`, чтобы вложить сериализатор `ModuleSerializer`. При этом задается параметр `many=True`, указывая на то, что сериализуется несколько объектов. Параметр `read_only` сообщает, что это поле доступно только для

чтения и не должно включаться ни в какие входные данные для создания или обновления объектов.

Откройте оболочку и снова создайте экземпляр класса `CourseSerializer`. Прорисуйте атрибут `data` сериализатора с помощью `JSONRenderer`. На этот раз перечисленные модули сериализуются с помощью вложенного сериализатора `ModuleSerializer`, как показано ниже:

```
"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."
 },
 ...
]
```

Подробнее о сериализаторах можно узнать по адресу <https://www.django-rest-framework.org/api-guide/serializers/>.

Типовые представления API очень удобны для создания REST API-интерфейсов на основе собственных моделей и сериализаторов. Однако, возможно, вам также понадобится реализовывать собственные представления с конкретно-прикладной логикой. Давайте познакомимся с тем, как создавать конкретно-прикладное представление API.

## Разработка конкретно-прикладных представлений API

Фреймворк REST предоставляет класс `APIView`, который создает функциональность API поверх встроенного в Django класса `View`. Класс `APIView` отличается от `View` использованием конкретно-прикладных объектов `Request` и `Response` фреймворка REST и обработкой исключений `APIException`, чтобы возвращать надлежащие HTTP-ответы. Он также имеет встроенную систему аутентификации и авторизации, чтобы управлять доступом к представлениям.

Вы создадите представление зачисления пользователей на курсы. Отредактируйте файл `api/views.py` приложения `courses`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.shortcuts import get_object_or_404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import generics
```

```
from courses.models import Subject, Course
from courses.api.serializers import SubjectSerializer

...

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})
```

Представление `CourseEnrollView` обрабатывает зачисление пользователей на курсы. Приведенный выше исходный код выполняет следующую работу.

1. Создается конкретно-прикладное представление как подкласс `APIView`.
2. Определяется метод `post()` для действий POST. В этом представлении никакой другой HTTP-метод не разрешен.
3. Ожидается URL-параметр `pk`, содержащий ИД курса. Курс извлекается по заданному параметру `pk`, и если он не найден, то вызывается исключение 404.
4. Текущий пользователь добавляется во взаимосвязь `students` многих-ко-многим объекта `Course`, и возвращается успешный ответ.

Отредактируйте файл `api.urls.py`, добавив следующий ниже шаблон URL-адреса представления `CourseEnrollView`:

```
path('courses/<pk>/enroll/',
 views.CourseEnrollView.as_view(),
 name='course_enroll'),
```

Теоретически теперь можно выполнять запрос методом POST, чтобы зачислять текущего пользователя на курс. Однако еще необходимо иметь возможность идентифицировать пользователя и предотвращать доступ неавторизованных пользователей к этому представлению. Давайте посмотрим на принцип работы API-аутентификации и разрешений.

## Обработка аутентификации

Фреймворк REST предоставляет классы аутентификации, чтобы идентифицировать выполняющего запрос пользователя. Если аутентификация проходит успешно, то фреймворк устанавливает аутентифицированный объект `User` в `request.user`. Если ни один пользователь не аутентифицирован, то вместо этого устанавливается экземпляр класса `Django AnonymousUser`.

Фреймворк REST предоставляет следующие бэкенды аутентификации:

- `BasicAuthentication`: это базовая HTTP-аутентификация. Пользователь и пароль отправляются клиентом в HTTP-заголовке `Authorization` в кодировке Base64. Подробнее об этом можно узнать на странице [https://en.wikipedia.org/wiki/Basic\\_access\\_authentication](https://en.wikipedia.org/wiki/Basic_access_authentication);

- `TokenAuthentication`: это аутентификация на основе токенов. Модель `Token` используется для хранения пользовательских токенов. Пользователи вставляют подлежащий аутентификации токен в HTTP-заголовок `Authorization`;
- `SessionAuthentication`: для аутентификации используется сеансовый бэкенд Django. Этот бэкенд удобен при выполнении аутентифицированных AJAX-запросов к API из фронтенда сайта<sup>1</sup>;
- `RemoteUserAuthentication`: позволяет делегировать аутентификацию веб-серверу, который устанавливает переменную `REMOTE_USER` среды.

Следует отметить, что можно разработать конкретно-прикладной бэкенд аутентификации. Это делается путем подклассирования предоставленного фреймворком REST класса `BaseAuthentication` и переопределения метода `authentication()`.

Кроме того, можно задавать аутентификацию для каждого представления либо устанавливать ее глобально с помощью настроичного параметра `DEFAULT_AUTHENTICATION_CLASSES`.



Аутентификация идентифицирует только пользователя, который выполняет запрос. Она не будет разрешать или запрещать доступ к представлениям. В целях ограничения доступа к представлениям следует использовать разрешения.

Вся информация об аутентификации находится на странице <https://www.djangoproject-rest-framework.org/api-guide/authentication/>.

Давайте добавим `BasicAuthentication` в представление. Отредактируйте файл `api/views.py` приложения `courses`, добавив атрибут `authentication_classes` в представление `CourseEnrollView`, как показано ниже:

```
...
from rest_framework.authentication import BasicAuthentication

class CourseEnrollView(APIView):
 authentication_classes = [BasicAuthentication]
 # ...
```

Пользователи будут идентифицироваться по учетным данным, установленным в заголовке `Authorization` HTTP-запроса.

## Добавление разрешений в представления

Фреймворк REST содержит систему разрешений, служащую для ограничения доступа к представлениям. Несколько встроенных в фреймворк REST разрешений таковы:

<sup>1</sup> Син. клиентская часть веб-приложения. – Прим. перев.

- `AllowAny`: неограниченный доступ, независимо от того, аутентифицирован пользователь или нет;
- `IsAuthenticated`: разрешает доступ только аутентифицированным пользователям;
- `IsAuthenticatedOrReadOnly`: полный доступ для аутентифицированных пользователей. Анонимным пользователям разрешено выполнять методы только для чтения, такие как `GET`, `HEAD` или `OPTIONS`;
- `DjangoModelPermissions`: разрешения, привязанные к `django.contrib.auth`. В представлении требуется атрибут `queryset`. Разрешение предоставляется только аутентифицированным пользователям, чьим назначенным модельным разрешениям предоставлено разрешение;
- `DjangoObjectPermissions`: разрешения Django на пообъектной основе.

Если пользователям отказано в разрешении, то они обычно получают один из следующих HTTP-кодов ошибок:

- HTTP 401: неавторизован;
- HTTP 403: отказано в доступе.

Более подробную информацию о разрешениях можно почитать на странице <https://www.djangoproject.org/api-guide/permissions/>.

Отредактируйте файл `api/views.py` приложения `courses`, добавив атрибут `permission_classes` в класс `CourseEnrollView`, как показано ниже:

```
...
from rest_framework.authentication import BasicAuthentication
from rest_framework.permissions import IsAuthenticated

class CourseEnrollView(APIView):
 authentication_classes = [BasicAuthentication]
 permission_classes = [IsAuthenticated]
 # ...
```

Здесь вставлено разрешение `IsAuthenticated`. Оно будет предотвращать доступ анонимных пользователей к представлению. Теперь можно выполнять запрос методом `POST` к новому методу API.

Проверьте, чтобы сервер разработки был запущен. Откройте оболочку и выполните следующую ниже команду:

```
curl -i -X POST http://127.0.0.1:8000/api/courses/1/enroll/
```

Вы получите такой ответ:

```
HTTP/1.1 401 Unauthorized
...
{"detail": "Authentication credentials were not provided."}
```

Как и ожидалось, вы получили HTTP-код 401, поскольку вы не аутентифицированы. Давайте применим базовую аутентификацию с одним из

ваших пользователей. Выполните следующую ниже команду, заменив `student:password` учетными данными существующего пользователя:

```
curl -i -X POST -u student:password http://127.0.0.1:8000/api/courses/1/enroll/
```

Вы получите следующий ниже ответ:

```
HTTP/1.1 200 OK
```

```
...
```

```
{"enrolled": true}
```

Вы можете обратиться к сайту администрирования и убедиться, что пользователь зачислен на курс.

Далее вы узнаете другой способ разработки общих представлений – с помощью наборов представлений `ViewSet`.

## Создание наборов представлений и маршрутизаторов

Наборы представлений `ViewSet` позволяют определять взаимодействия API и предоставляют фреймворку REST возможность создавать URL-адреса динамически с помощью объекта `Router`. Использование наборов представлений `ViewSet` позволяет избегать повторения логики в нескольких представлениях. Наборы представлений `ViewSet` содержат действия для следующих стандартных операций:

- операция создания: `create()`;
- операция извлечения: `list()` и `retrieve()`;
- операция обновления: `update()` и `partial_update()`;
- операция удаления: `destroy()`.

Давайте создадим `ViewSet` для модели `Course`. Отредактируйте файл `api/views.py`, добавив в него следующий ниже исходный код:

```
...
from rest_framework import viewsets
from courses.api.serializers import SubjectSerializer,
 CourseSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
 queryset = Course.objects.all()
 serializer_class = CourseSerialize
```

Здесь создается подкласс класса `ReadOnlyModelViewSet`, который предоставляет действия только для чтения `list()` и `retrieve()` для обоих списковых объектов или извлекает один объект.

Отредактируйте файл `api/urls.py`, создав маршрутизатор для вашего набора представлений `ViewSet`, как показано ниже:

```
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)),
]
```

В приведенном выше исходном коде создается объект-маршрутизатор `DefaultRouter` и регистрируется `ViewSet` с префиксом `courses`. Маршрутизатор берет на себя ответственность за автоматическое генерирование URL-адресов для набора представлений `ViewSet`.

Пройдите по URL-адресу `http://127.0.0.1:8000/api/` в своем браузере. Вы увидите, что маршрутизатор выводит список всех `ViewSet` по своему базовому URL-адресу, как показано на рис. 15.3:



Рис. 15.3. Корневая страница API просматриваемого API-интерфейса фреймворка REST

Вы можете обратиться к `http://127.0.0.1:8000/api/courses/` и получить список курсов.

Подробнее о наборах представлений `ViewSet` можно узнать по адресу <https://www.djangoproject.org/api-guide/viewsets/>, а дополнительная информация о маршрутизаторах находится на странице <https://www.djangoproject.org/api-guide/routers/>.

## Добавление дополнительных действий в наборы представлений

В наборы представлений `ViewSet` можно добавлять дополнительные действия. Давайте поменяем приведенное ранее представление `CourseEnrollView` на конкретно-прикладное действие `ViewSet`. Отредактируйте файл `api/views.py`, видоизменив класс `CourseViewSet` и придав ему следующий вид:

```
...
from rest_framework.decorators import action

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
 queryset = Course.objects.all()
 serializer_class = CourseSerializer

 @action(detail=True,
 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})
```

В приведенном выше исходном коде добавляется конкретно-прикладной метод `enroll()`, который представляет дополнительное действие этого `ViewSet`. Указанный исходный код выполняет следующую работу.

1. Используется декоратор `action` фреймворка с параметром `detail=True`, указывая на то, что это действие должно выполняться над одним объектом.
2. Декоратор позволяет добавлять действию конкретно-прикладные атрибуты. При этом указывается, что для этого представления разрешен только метод `post()`, и задаются классы аутентификации и разрешений.
3. Для извлечения объекта `Course` используется `self.get_object()`.
4. Во взаимосвязь `students` многие-ко-многим добавляется текущий пользователь и возвращается конкретно-прикладной ответ об успехе.

Отредактируйте файл `api/urls.py`, удалив либо закомментировав следующий ниже URL-адрес, так как он больше не нужен:

```
path('courses/<pk>/enroll/',
 views.CourseEnrollView.as_view(),
 name='course_enroll'),
```

Затем отредактируйте файл `api/views.py`, удалив либо закомментировав класс `CourseEnrollView`.

URL-адрес зачисления на курсы теперь автоматически генерируется маршрутизатором. URL-адрес остается прежним, поскольку он создается динамически с использованием имени `enroll` действия.

После зачисления студентов на курс им нужно получать доступ к содержимому курса. Далее вы научитесь обеспечивать доступ к курсу только зарегистрированным студентам.

## Создание конкретно-прикладных разрешений

Вы хотите, чтобы студенты имели доступ к содержимому курсов, на которые они зачислены. Доступ к содержимому курса должны иметь только те студенты, которые были зачислены на курс. В такой ситуации самым лучшим подходом будет применение конкретно-прикладного класса разрешений. Фреймворк REST предоставляет класс `BasePermission`, который позволяет определять следующие методы:

- `has_permission()`: проверка разрешений на уровне представления;
- `has_object_permission()`: проверка разрешений на уровне экземпляра.

Эти методы должны возвращать `True`, разрешая доступ, либо `False` в противном случае.

Внутри каталога `courses/api/` создайте новый файл и назовите его `permissions.py`. Добавьте в него следующий ниже исходный 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()
```

Здесь подклассируется класс `BasePermission` и переопределяется метод `has_object_permission()`. Выполняется проверка, что делающий запрос пользователь присутствует во взаимосвязи `students` объекта `Course`. Далее вы примените разрешение `IsEnrolled`.

## Сериализация содержимого курса

Вам необходимо сериализовать содержимое курса. Модель `Content` имеет в своем составе типовой внешний ключ, который позволяет связывать объекты разных моделей содержимого. Тем не менее в предыдущей главе во все модели содержимого был добавлен общий метод `render()`. Этот метод можно использовать для передачи прорисованного содержимого API-интерфейсу.

Отредактируйте файл `api/serializers.py` приложения `courses`, добавив в него следующий ниже исходный код:

```
from courses.models import Subject, Course, Module, 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 приведенном выше исходном коде определяется конкретно-прикладное поле путем подклассирования поставляемого с фреймворком REST поля-сериализатора `RelatedField` и переопределения метода `to_representation()`. Для модели `Content` определяется сериализатор `ContentSerializer`, и для типового внешнего ключа `item` используется конкретно-прикладное поле.

Теперь нужен альтернативный сериализатор для модели `Module`, включающий его содержимое, а также расширенный сериализатор `Course`. Отредактируйте файл `api/serializers.py`, добавив в него следующий ниже исходный 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 создадим представление, которое имитирует поведение действия `retrieve()`, но включает в себя содержимое курса.

Отредактируйте файл `api/views.py`, добавив в класс `CourseViewSet` следующий ниже метод:

```
from courses.api.permissions import IsEnrolled
from courses.api.serializers import CourseWithContentsSerializer
```

```
class CourseViewSet(viewsets.ReadOnlyModelViewSet):
 # ...
 @action(detail=True,
 methods=['get'],
 serializer_class=CourseWithContentsSerializer,
 authentication_classes=[BasicAuthentication],
 permission_classes=[IsAuthenticated, IsEnrolled])
 def contents(self, request, *args, **kwargs):
 return self.retrieve(request, *args, **kwargs)
```

Работа данного метода описывается следующим образом.

1. Декоратор `action` используется с параметром `detail=True`, чтобы задавать действие, которое выполняется над одним объектом.
2. Затем указывается, что для этого действия разрешен только метод GET.
3. Далее используется новый класс-сериализатор `CourseWithContentsSerializer`, который включает прорисованное содержимое курса.
4. Используются как разрешения `IsAuthenticated`, так и конкретно-прикладные разрешения `IsEnrolled`. Тем самым обеспечивается, чтобы доступ к содержимому курса могли получать только те пользователи, которые были зачислены на курс.
5. В конце применяется существующее действие `retrieve()`, чтобы возвращать объект `Course`.

Пройдите по URL-адресу `http://127.0.0.1:8000/api/courses/1/contents/` в своем браузере. Если вы обратитесь к представлению с правильными учетными данными, то увидите, что каждый модуль курса включает прорисованный HTML содержимого курса, как показано ниже:

```
{
 "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 создали простой API, который позволяет другим сервисам программно обращаться к приложению `courses`. Фреймворк REST также позволяет создавать и редактировать объекты с помощью класса `ModelViewSet`. Мы охватили главные аспекты фреймворка Django REST framework, но дополнительная информация о его функциональных возможностях находится в обширной документации по адресу <https://www.django-rest-framework.org/>.

## Потребление RESTful API

Теперь, когда вы реализовали API, вы можете потреблять его в программном режиме из других приложений. С API можно взаимодействовать, используя JavaScript Fetch API во фронтенде вашего приложения, аналогично функциональностям, которые вы создали в главе 6 «Распространение контента на веб-сайте». API также можно использовать из приложений, созданных с помощью Python или любого другого языка программирования.

Здесь вы создадите простое приложение Python, которое использует RESTful API для извлечения всех имеющихся курсов, а затем зачисляет студента на все эти курсы. Вы научитесь проходить аутентификацию в API, используя базовую HTTP-аутентификацию, и выполнять запросы методами GET и POST.

Для потребления API мы будем использовать библиотеку Python `requests`. Мы работали с этой библиотекой в главе 6 «Распространение контента на веб-сайте», чтобы получать изображения по их URL-адресу. Библиотека `requests` абстрагируется от сложности работы с HTTP-запросами и предоставляет очень простой интерфейс, чтобы потреблять HTTP-сервисы. Документация для библиотеки `requests` находится по адресу <https://requests.readthedocs.io/en/master/>.

Следующей ниже командой откройте оболочку и установите библиотеку `requests`:

```
pip install requests==2.28.1
```

Рядом с каталогом проекта `educa` создайте новый каталог и назовите его `api_examples`. Внутри каталога `api_examples/` создайте новый файл и назовите его `enroll_all.py`. Теперь файловая структура должна выглядеть так:

```
api_examples/
 enroll_all.py
educa/
 ...
```

Отредактируйте файл `enroll_all.py`, добавив в него следующий ниже исходный код:

```
import requests

base_url = 'http://127.0.0.1:8000/api/'
```

```
извлечь все курсы
r = requests.get(f'{base_url}courses/')
courses = r.json()

available_courses = ', '.join([course['title'] for course in courses])
print(f'Available courses: {available_courses}')
```

В приведенном выше исходном коде выполняются следующие действия.

1. Импортируется библиотека `requests`, и определяется базовый URL-адрес API.
2. Применяется метод `request.get()`, чтобы извлечь данные из API путем отправки запроса методом GET на URL-адрес `http://127.0.0.1:8000/api/courses/`. Эта конечная точка API общедоступна и поэтому не требует никакой аутентификации.
3. Применяется метод `json()` объекта ответа, чтобы декодировать возвращаемые API-интерфейсом данные JSON.
4. Печатается атрибут `title` каждого курса.

Следующей ниже командой запустите сервер разработки из каталога проекта `educa`:

```
python manage.py runserver
```

В еще одной оболочке выполните следующую ниже команду из каталога `api_examples/`:

```
python enroll_all.py
```

Вы увидите результат со списком всех названий курсов. Например:

```
Available courses: Introduction to Django, Python for beginners, Algebra basics
```

Это первый автоматический вызов вашего API.

Отредактируйте файл `enroll_all.py`, изменив его так, чтобы он выглядел следующим образом:

```
import requests

username = ''
password = ''
base_url = 'http://127.0.0.1:8000/api/'

извлечь все курсы
r = requests.get(f'{base_url}courses/')
courses = r.json()

available_courses = ', '.join([course['title'] for course in courses])
print(f'Available courses: {available_courses}')
```

```
for course in courses:
 course_id = course['id']
 course_title = course['title']
 r = requests.post(f'{base_url}courses/{course_id}/enroll/',
 auth=(username, password))
 if r.status_code == 200:
 # успешный запрос
 print(f'Successfully enrolled in {course_title}')
```

Замените значения переменных `username` и `password` учетными данными существующего пользователя.

Приведенный выше исходный код выполняет следующие действия.

1. Задается пользовательское имя (`username`) и пароль студента, которого нужно зачислить на курсы.
2. Полученные из API доступные курсы прокручиваются в цикле.
3. В переменной `course_id` сохраняется атрибут ИД курса, в переменной `course_title` – атрибут `title`.
4. Применяется метод `request.post()`, чтобы отправить запрос POST на URL-адрес `http://127.0.0.1:8000/api/courses/[id]/enroll/` по каждому курсу. Этот URL-адрес соответствует представлению API `CourseEnrollView`, которое позволяет зачислять пользователя на курс. Используя переменную `course_id`, формируется URL-адрес каждого курса. Представление `CourseEnrollView` требует проведения аутентификации. В нем используется разрешение `IsAuthenticated` и аутентификационный класс `BasicAuthentication`. Библиотека `requests` поддерживает базовую HTTP-аутентификацию прямо «из коробки». Параметр `auth` используется для передачи кортежа с пользовательским именем и паролем, чтобы аутентифицировать пользователя посредством базовой HTTP-аутентификации.
5. Если статусный код ответа равен 200 OK, то печатается сообщение, указывающее на то, что пользователь успешно зачислен на курс.

С библиотекой `requests` можно использовать разные виды аутентификации. Дополнительная информация об аутентификации с помощью библиотеки `requests` находится по адресу <https://requests.readthedocs.io/en/master/user/authentication/>.

Выполните следующую ниже команду из каталога `api_examples/`:

```
python enroll_all.py
```

Теперь вы увидите вот такой результат:

```
Available courses: Introduction to Django, Python for beginners, Algebra basics
Successfully enrolled in Introduction to Django
Successfully enrolled in Python for beginners
Successfully enrolled in Algebra basics
```

Отлично! Вы успешно зачислили пользователя на все имеющиеся курсы с помощью API. Вы получили сообщение об успешном зачислении `Successfully enrolled` по каждому курсу на платформе. Как видите, использовать API из любого другого приложения очень просто. На основе API можно легко создавать иные функциональности и позволять другим интегрировать ваш API в свои приложения.

## Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter15>.
- Веб-сайт фреймворка REST framework: <https://www.django-rest-framework.org/>.
- Настройки фреймворка REST framework: <https://www.django-rest-framework.org/api-guide/settings/>.
- Рендереры фреймворка REST framework: <https://www.django-rest-framework.org/api-guide/renderers/>.
- Парсеры фреймворка REST framework: <https://www.django-rest-framework.org/api-guide/parsers/>.
- Типовые примесные классы и представления фреймворка REST framework: <https://www.django-rest-framework.org/api-guide/generic-views/>.
- Файл для скачивания инструмента curl: <https://curl.se/download.html>.
- API-платформа Postman: <https://www.getpostman.com/>.
- Сериализаторы фреймворка REST framework: <https://www.django-rest-framework.org/api-guide/serializers/>.
- Базовая HTTP-аутентификация: [https://en.wikipedia.org/wiki/Basic\\_access\\_authentication](https://en.wikipedia.org/wiki/Basic_access_authentication).
- Аутентификация в фреймворке REST framework: <https://www.django-rest-framework.org/api-guide/authentication/>.
- Разрешения в фреймворке REST framework: <https://www.django-rest-framework.org/api-guide/разрешения/>.
- Наборы представлений `ViewSet` фреймворка REST framework: <https://www.django-rest-framework.org/api-guide/viewsets/>.
- Маршрутизаторы фреймворка REST framework: <https://www.django-rest-framework.org/api-guide/routers/>.
- Документация по библиотеке Python `requests`: <https://requests.readthedocs.io/en/master/>.
- Аутентификация с помощью библиотеки `requests`: <https://requests.readthedocs.io/en/master/user/authentication/>.

## Резюме

В этой главе вы научились применять фреймворк Django REST framework, чтобы разработать RESTful API к своему проекту. Вы создали сериализаторы и представления для моделей, а также разработали конкретно-прикладные представления API. Вы также добавили аутентификацию в свой API и ограничили доступ к представлениям API с помощью разрешений. Далее вы научились создавать конкретно-прикладные разрешения и реализовали наборы представления `ViewSet` и маршрутизаторы. Наконец, вы применили библиотеку `requests`, чтобы потреблять API из внешнего скрипта Python.

В следующей главе вы научитесь разрабатывать чат-сервер с использованием каналов Django. Вы реализуете асинхронную связь с помощью WebSockets и будете использовать резидентное хранилище Redis для установления канального слоя.

# 16

## Разработка чат-сервера

В предыдущей главе вы создали RESTful API к своему проекту. В этой главе вы разработаете чат-сервер для студентов, используя каналы на основе приложения Django Channels. Студенты смогут обращаться к отдельной чат-комнате по каждому курсу, на который они зачислены. С целью создания чат-сервера вы научитесь раздавать свой проект Django через интерфейс шлюза асинхронного сервера (ASGI) и реализуете асинхронную связь.

В этой главе вы:

- добавите приложение Channels в проект;
- разработаете WebSocket-потребителя и соответствующую маршрутизацию;
- реализуете WebSocket-клиента;
- активируете канальный слой с резидентным хранилищем Redis;
- сделаете потребителя полностью асинхронным.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter16>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

### Создание приложения для ведения чата

Вы собираетесь реализовать чат-сервер, чтобы предоставлять студентам чат-комнату по каждому курсу. Зачисленные на курс студенты смогут обращаться к чат-комнате курса и обмениваться сообщениями в режиме реального вре-

мени. Для создания этой функциональности вы будете использовать Channels. Channels – это приложение Django, которое расширяет Django в части оперирования протоколами, требующими длительных соединений, такими как веб-сокеты, чат-боты или MQTT (облегченный транспорт сообщений публикации/подписки, широко применяемый в проектах интернета вещей (IoT<sup>1</sup>)).

Используя приложение Channels в своем проекте, можно легко реализовывать реально-временные или асинхронные функциональности в дополнение к стандартным синхронным HTTP-представлениям. Вы начнете с добавления нового приложения в проект. Новое приложение будет содержать логику чат-сервера.

Документацию по приложению Django Channels можно почитать по адресу <https://channels.readthedocs.io/>.

Приступим к реализации чат-сервера. Из каталога проекта `educa` выполните следующую ниже команду, чтобы создать новую файловую структуру приложения:

```
django-admin startapp chat
```

Откройте файл `settings.py` проекта `educa` в редакторе, активируйте приложение `chat` в своем проекте, отредактировав настроечный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [
 # ...
 'chat',
]
```

Теперь новое приложение `chat` является в проекте активным.

## Реализация представления чат-комнаты

Вы будете предоставлять студентам отдельную чат-комнату для ведения дискуссии по каждому курсу. Вам необходимо создать студентам представление присоединения к чат-комнате данного курса. К чат-комнате курса смогут получить доступ только те студенты, которые были зачислены на курс.

Отредактируйте файл `views.py` приложения `newchat`, добавив в него следующий ниже исходный код:

```
from django.shortcuts import render, get_object_or_404
from django.http import HttpResponseRedirect
from django.contrib.auth.decorators import login_required
```

<sup>1</sup> Англ. Internet of Things. – Прим. перев.

```
@login_required
def course_chat_room(request, course_id):
 try:
 # извлечь курс с заданным id, к которому
 # присоединился текущий пользователь
 course = request.user.courses_joined.get(id=course_id)
 except:
 # пользователь не является студентом курса либо
 # курс не существует
 return HttpResponseRedirect('/chat/room')
 return render(request, 'chat/room.html', {'course': course})
```

Это представление `course_chat_room`. В данном представлении используется декоратор `@login_required`, чтобы предотвращать доступ любого неавтентифицированного пользователя к представлению. Представление получает обязательный параметр `course_id`, который используется для извлечения курса с заданным `id`.

Доступ к курсам, на которые пользователь зачислен, обеспечивается по-средством взаимосвязи `Courses_Joined`, и из этого подмножества курсов извлекается курс с заданным `id`. Если курс с указанным `id` не существует либо пользователь на него не зачислен, то возвращается ответ `HttpResponseForbidden`, который транслируется в HTTP-ответ со статусом 403.

Если курс с заданным `id` существует и пользователь на него зачислен, то прорисовывается шаблон `chat/room.html` с переданным в контекст шаблона объектом `course`.

Теперь необходимо добавить шаблон URL-адреса этого представления. Внутри каталога приложения `chat` создайте новый файл и назовите его `urls.py`. Добавьте в него следующий ниже исходный код:

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

app_name = 'chat'

urlpatterns = [
 path('room/<int:course_id>', views.course_chat_room,
 name='course_chat_room'),
]
```

Это первоначальный файл шаблонов URL-адресов приложения `chat`. Здесь определяется шаблон URL-адреса `course_chat_room`, включая параметр `course_id` с префиксом `int`, так как ожидается только целочисленное значение.

Вставьте новые шаблоны URL-адресов приложения `chat` в главные шаблоны URL-адресов проекта. Отредактируйте главный файл `urls.py` проекта `educa`, добавив в него следующую ниже строку:

```
urlpatterns = [
 # ...
 path('chat/', include('chat.urls', namespace='chat')),
]
```

Шаблоны URL-адресов приложения `chat` добавляются в проект в рамках пути `chat/`.

Теперь нужно создать шаблон представления `course_chat_room`. Этот шаблон будет содержать область визуализации сообщений, которыми обмениваются в чате, и поле ввода текста с кнопкой `submit`, чтобы отправлять текстовые сообщения в чат.

Внутри каталога приложения `chat` создайте следующую ниже файловую структуру:

```
templates/
 chat/
 room.html
```

Откройте в редакторе шаблон `chat/room.html` и добавьте в него следующий ниже исходный код:

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

{% block title %}Chat room for "{{ course.title }}"{% endblock %}

{% block content %}
<div id="chat">
</div>
<div id="chat-input">
 <input id="chat-message-input" type="text">
 <input id="chat-message-submit" type="submit" value="Send">
</div>
{% endblock %}

{% block include_js %}
{% endblock %}

{% block domready %}
{% endblock %}
```

Это шаблон чат-комнаты курса. В данном шаблоне расширяется шаблон `base.html` проекта и заполняется его блок `content`. В шаблоне определяется HTML-элемент `<div>` с ИД чата, который будет использоваться для отображения сообщений чата, отправляемых пользователем и другими студентами. Также определяется второй элемент `<div>` с полем ввода текста и кнопкой `submit`, которая позволит пользователю отправлять

сообщения. Кроме того, добавляются блоки `include_js` и `domready`, определенные в шаблоне `base.html`, который будет реализован позже, чтобы устанавливать соединение с WebSocket и отправлять или получать сообщения.

Запустите сервер разработки и пройдите по URL-адресу `http://127.0.0.1:8000/chat/room/1/` в своем браузере, заменив 1 на `id` существующего в базе данных курса. Обратитесь к чат-комнате с вошедшим на платформу пользователем, который был зачислен на этот курс. Вы увидите следующий ниже экран:



Рис. 16.1. Страница чат-комнаты курса

Это экран чат-комнаты курса, который студенты будут использовать для обсуждения тем в рамках курса.

## Реально-временной Django на основе Channels

Вы разрабатываете чат-сервер, чтобы предоставлять студентам чат-комнату по каждому курсу. Зачисленные на курс студенты смогут обращаться к чат-комнате курса и обмениваться сообщениями. Эта функциональность требует реально-временной связи между сервером и клиентом. Клиент должен иметь возможность подсоединяться к чату и отправлять либо получать данные в любое время. Эта функциональность реализуется несколькими способами, используя опрашивание через AJAX или длительное опрашивание в сочетании с сохранением сообщений в базе данных или резидентном хранилище Redis. Однако эффективного способа реализовать чат-сервер с помощью стандартного синхронного веб-приложения нет. Вы построите чат-сервер, используя асинхронную связь через ASGI.

## Асинхронные приложения с использованием ASGI

Django обычно развертывается с использованием интерфейса шлюза веб-сервера (**WSGI**)<sup>1</sup>, то есть стандартного для приложений Python интерфейса для оперирования HTTP-запросами. Однако для работы с асинхронными приложениями необходимо использовать другой интерфейс под названием «Интерфейс шлюза асинхронного сервера» (**ASGI**)<sup>2</sup>, который вдобавок может оперировать WebSocket-запросами. ASGI – это новый стандарт Python для асинхронных веб-серверов и приложений.

Введение в ASGI находится по адресу <https://asgi.readthedocs.io/en/latest/introduction.html>.

Django поставляется с поддержкой работы асинхронного Python посредством ASGI. Написание асинхронных представлений поддерживается в связи с тем, что в Django 3.1 и Django 4.1 вводятся асинхронные обработчики представлений на основе классов. Приложение Channels строится как обертка<sup>3</sup> поверх имеющейся в Django нативной поддержки ASGI и предоставляет дополнительные функциональности оперирования протоколами, требующими длительных соединений, такими как веб-сокеты, протоколы IoT и чат-протоколы.

Веб-сокеты обеспечивают полнодуплексную связь, устанавливая постоянное открытое двунаправленное соединение по протоколу управления передачей (**TCP**)<sup>4</sup> между серверами и клиентами. Вы будете использовать веб-сокеты для реализации чат-сервера.

Дополнительную информацию о развертывании Django с ASGI можно найти по адресу <https://docs.djangoproject.com/ru/4.1/howto/deployment/asgi/>.

Более подробная информация о поддержке веб-фреймворком Django написания асинхронных представлений находится на странице <https://docs.djangoproject.com/en/4.1/topics/async/>, о поддержке асинхронных представлений на основе классов можно почитать на странице <https://docs.djangoproject.com/en/4.1/topics/class-based-views/#async-class-based-views>.

## Цикл запроса/ответа с использованием приложения Channels

Важно понимать различия между стандартным синхронным циклом запроса/ответа и циклом с реализацией каналов на основе приложения Channels. На рис. 16.2 показан цикл запроса/ответа в синхронной конфигурации Django.

Когда HTTP-запрос отправляется браузером на веб-сервер, Django обрабатывает запрос и передает объект `HttpRequest` соответствующему представлению. Представление обрабатывает запрос и возвращает объект `HttpResponse`,

<sup>1</sup> Англ. Web Server Gateway Interface. – Прим. перев.

<sup>2</sup> Англ. Asynchronous Server Gateway Interface. – Прим. перев.

<sup>3</sup> Подробнее о приложении Channels как обертке можно узнать на странице <https://channels.readthedocs.io/en/stable/introduction.html>. – Прим. перев.

<sup>4</sup> Англ. Transmission Control Protocol. – Прим. перев.

который отправляется обратно в браузер в качестве HTTP-ответа. Без соответствующего HTTP-запроса не существует механизма поддержания открытого соединения или отправки данных в браузер.

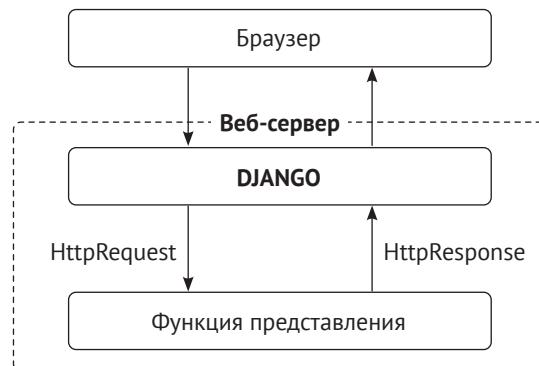


Рис. 16.2. Цикл запроса/ответа Django

На следующей ниже схеме показан цикл запроса/ответа проекта Django с использованием приложения-обертки Channels вместе с веб-сокетами:



Рис. 16.3. Цикл запросов/ответов Django с приложением Channels

Приложение-обертка Channels заменяет цикл запроса/ответа Django сообщениями, которые отправляются по каналам. HTTP-запросы по-прежнему маршрутизируются функциям представления с использованием Django, но теперь они маршрутизируются по каналам. Такой подход также позволяет оперировать сообщением веб-сокетов, когда есть производители и потребители, которые обмениваются сообщениями в канальном слое. Приложение-обертка Channels сохраняет синхронную архитектуру Django, позволяя выбирать между написанием синхронного и асинхронного исходного кода или их комбинаций.

## Установка приложения-обертки Channels

Вы добавите в проект приложение-обертку Channels и настроите необходимую базовую маршрутизацию ASGI-приложения, чтобы управлять HTTP-запросами.

Следующей ниже командой установите приложение `channels` в виртуальной среде:

```
pip install channels==3.0.5
```

Отредактируйте файл `settings.py` проекта `educa`, добавив `channels` в настроечный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [
 # ...
 'channels',
]
```

Теперь приложение `channels` активировано в проекте.

Приложение-обертка Channels ожидает, что будет определено одно корневое приложение, которое будет исполняться для всех запросов. Корневое приложение определяется путем добавления настроечного параметра `ASGI_APPLICATION` в проект. Данный параметр похож на параметр `ROOT_URLCONF`, который указывает на базовые шаблоны URL-адресов проекта. Корневое приложение можно размещать в любом месте проекта, но рекомендуется его помещать в файл уровня проекта. Конфигурацию маршрутизации корня можно добавлять непосредственно в файл `asgi.py`, в котором будет определено ASGI-приложение.

Откройте файл `asgi.py` внутри каталога проекта `educa` и добавьте следующий ниже исходный код, выделенный жирным шрифтом:

```
import os
```

```
from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'educa.settings')

django_asgi_app = get_asgi_application()

application = ProtocolTypeRouter({
 'http': django_asgi_app,
})
```

В приведенном выше исходном коде определяется главное ASGI-приложение, которое будет исполняться во время раздачи проекта Django через ASGI. При этом в качестве главной точки входа в систему маршрутизации используется поставляемый с Channels класс `ProtocolTypeRouter`. В указанном классе используется словарь, который соотносит типы связи, такие как `http` или `websocket`, с ASGI-приложениями. Экземпляр этого класса создается с приложением, изначально работающим по HTTP-протоколу. Позже будет добавлен протокол для WebSocket.

Добавьте следующую ниже строку в файл `settings.py` проекта:

```
ASGI_APPLICATION = 'educa.routing.application'
```

Параметр `ASGI_APPLICATION` используется приложением `Channels`, чтобы определять местонахождение конфигурации маршрутизации корня.

Когда приложение-обертка `Channels` добавлено в настроечный параметр `INSTALLED_APPS`, оно берет на себя управление командой `runserver`, подменяя стандартный сервер разработки Django. Помимо обработки маршрутизации URL-адресов к представлениям Django для синхронных запросов, встроенный в `Channels` сервер разработки также управляет маршрутами к WebSocket-потребителям.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Вы увидите результат, подобный представленному далее:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
May 30, 2022 - 08:02:57
Django version 4.0.4, using settings 'educa.settings'
Starting ASGI/Channels version 3.0.4 development server at
http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Проверьте, чтобы этот результат содержал строку `Starting ASGI/Channels version 3.0.4 development server`. Указанная строка подтверждает, что вместо встроенного в Django стандартного сервера разработки используется сервер разработки, встроенный в Channels, способный управлять синхронными и асинхронными запросами. HTTP-запросы продолжают вести себя так же, как и раньше, но они маршрутизируются по каналам.

Теперь, когда приложение-обертка Channels установлено в проекте, можно разработать чат-сервер для ведения дискуссий по курсам. Для того чтобы реализовать чат-сервер в проекте, необходимо предпринять следующие шаги.

- Настроить потребителя:** потребители – это отдельные фрагменты кода, которые могут оперировать веб-сокетами в ключе, очень похожими на традиционные HTTP-представления. Вы создадите потребителя для чтения сообщений из канала связи и записи сообщений в канал.
- Сконфигурировать маршрутизацию:** приложение-обертка Channels предоставляет классы маршрутизации, которые позволяют комбинировать и ставить потребителей в стек. Вы сконфигурируете маршрутизацию URL-адресов для своего чата-потребителя.
- Реализовать WebSocket-клиент:** при обращении студента к чат-комнате происходит подсоединение к веб-сокету из браузера и с помощью JavaScript отправляются либо приходят сообщения.
- Активировать канальный слой:** канальные слои позволяют обмениваться данными между разными экземплярами приложения. Они являются полезной частью создания распределенного реально-временного приложения. Вы установите канальный слой с использованием резидентного хранилища Redis.

Давайте начнем с написания своего собственного потребителя, чтобы оперировать соединением с веб-сокетом, получением и отправкой сообщений и отсоединением.

## Написание потребителя

Потребители являются эквивалентом представлений Django, но предназначенные для асинхронных приложений. Как уже упоминалось, они оперируют веб-сокетами очень похоже на то, как традиционные представления оперируют HTTP-запросами. Потребители – это ASGI-приложения, которые могут оперировать сообщениями, уведомлениями и другими вещами. В отличие от представлений Django, потребители предназначены для длительной связи. URL-адреса соотносятся с потребителями через классы маршрутизации, которые позволяют комбинировать и ставить потребителей в стек.

Давайте реализуем базового потребителя, который может принимать WebSocket-соединения и отражать эхом каждое сообщение, которое он получает от веб-сокета, обратно ему. Эта первоначальная функциональность позволит студенту отправлять сообщения потребителю и получать отправленные им сообщения обратно.

Внутри каталога приложения `chat` создайте новый файл и назовите его `consumers.py`. Добавьте в него следующий ниже исходный код:

```
import json
from channels.generic.websocket import WebsocketConsumer

class ChatConsumer(WebsocketConsumer):
 def connect(self):
 # принять соединение
 self.accept()

 def disconnect(self, close_code):
 pass

 # получить сообщение из WebSocket
 def receive(self, text_data):
 text_data_json = json.loads(text_data)
 message = text_data_json['message']
 # отправить сообщение в WebSocket
 self.send(text_data=json.dumps({'message': message}))
```

Это потребитель `ChatConsumer`. Приведенный выше класс наследует от встроенного в Channels класса `WebsocketConsumer`, чтобы реализовать базового WebSocket-потребителя. В указанном потребителе реализуются следующие методы:

- `connect()`: вызывается при получении нового соединения. Любая связь принимается с помощью `self.accept()`. Кроме того, соединение можно отклонить, вызвав `self.close()`;
- `disconnect()`: вызывается при закрытии сокета. Здесь используется `pass`, потому что при закрытии соединения клиентом никаких действий выполнять не нужно;
- `receive()`: вызывается при получении данных. При этом ожидается, что будет получен текст в виде `text_data` (в случае двоичных данных он поступает в виде `binary_data`). Полученные текстовые данные трактуются как JSON. Поэтому для загрузки полученных JSON-данных в словарь Python используется метод `json.loads()`. Выполняется обращение к ключу `message`, который ожидаемо будет присутствовать в полученной структуре JSON. Для того чтобы отразить сообщение назад, оно отправляется обратно в веб-сокет с помощью `self.send()`, снова преобразовывая его в формат JSON посредством метода `json.dumps()`.

Первоначальная версия потребителя `ChatConsumer` принимает любое WebSocket-соединение и отражает WebSocket-клиенту каждое полученное им сообщение назад. Обратите внимание, что потребитель еще не делает широковещательной рассылки сообщений другим клиентам. Вы создадите эту функциональность чуть позже, реализовав канальный слой.

## Маршрутизация

Теперь необходимо определить URL-адрес, чтобы маршрутизировать соединения к созданному потребителю `ChatConsumer`. Приложение-обертка `Channels` предоставляет классы маршрутизации, которые позволяют комбинировать и ставить потребителей в стек, чтобы выполнять диспетчеризацию в зависимости от типа соединения. Указанные классы можно трактовать как встроенную в `Django` систему маршрутизации URL-адресов, но только для асинхронных приложений.

Внутри каталога приложения `chat` создайте новый файл и назовите его `routing.py`. Добавьте в него следующий ниже исходный код:

```
from django.urls import re_path
from . import consumers

websocket_urlpatterns = [
 re_path(r'ws/chat/room/(?P<course_id>\d+)/$',
 consumers.ChatConsumer.as_asgi()),
]
```

В приведенном выше исходном коде шаблон URL-адреса соотносится с классом `ChatConsumer`, который был определен в файле `chat/consumers.py`. Функция `Django` `re_path` используется для определения пути с помощью регулярных выражений. Она применяется вместо общей функции `path` из-за ограничений маршрутизации URL-адресов в `Channels`. URL-адрес содержит целочисленный параметр с именем `course_id`. Указанный параметр будет доступен в области видимости потребителя и позволит определять чат-комнату курса, к которой пользователь подсоединяется. Затем вызывается метод `as_asgi()` класса `consumer`, чтобы получить ASGI-приложение, которое будет создавать экземпляр потребителя для каждого пользовательского соединения. Это поведение похоже на метод `Django` `as_view()` для представлений на основе классов.



К URL-адресам веб-сокетов рекомендуется добавлять префикс `/ws/`, чтобы отличать их от URL-адресов, используемых для стандартных синхронных HTTP-запросов. Такой подход также упрощает работу в производственной среде, когда HTTP-сервер направляет запросы, основываясь на путях.

Отредактируйте глобальный файл `asgi.py`, расположенный рядом с файлом `settings.py`, придав ему следующий вид:

```
import os

from django.core.asgi import get_asgi_application
```

```
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack
import chat.routing

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'educa.settings')

django_asgi_app = get_asgi_application()

application = ProtocolTypeRouter({
 'http': django_asgi_app,
 'websocket': AuthMiddlewareStack(
 URLRouter(chat.routing.websocket_urlpatterns)
),
})
```

В приведенном выше исходном коде добавляется новый маршрут для протокола `websocket`. Маршрутизатор `URLRouter` используется для соотнесения соединений `websocket` с шаблонами URL-адресов, определенными в списке `websocket_urlpatterns` файла `routing.py` приложения `chat`. Помимо этого, используется предоставляемый приложением-оберткой `Channels` класс `AuthMiddlewareStack`, который поддерживает встроенную в Django стандартную аутентификацию, при которой детальная информация о пользователе хранится в сеансе. Позже вы будете обращаться к экземпляру пользователя в области видимости потребителя, чтобы идентифицировать отправляющего сообщение пользователя.

## Реализация WebSocket-клиента

На данный момент вы создали представление `course_chat_room` и соответствующий ему шаблон, чтобы студенты могли обращаться к чат-комнате курса. Вы реализовали WebSocket-потребителя для чат-сервера и связали его с маршрутизацией URL-адресов. Теперь требуется разработать WebSocket-клиента, чтобы устанавливать соединение с веб-сокетом в шаблоне чат-комнаты курса и иметь возможность отправлять/получать сообщения.

Вы реализуете WebSocket-клиента с помощью JavaScript, чтобы открывать и поддерживать соединение в браузере. Вы будете взаимодействовать с объектной моделью документа (**DOM**) с помощью JavaScript.

Отредактируйте шаблон `chat/room.html` приложения `chat`, видоизменив блоки `include_js` и `domready`, как показано ниже:

```
{% block include_js %}
{{ course.id|json_script:"course-id" }}
{% endblock %}

{% block domready %}
```

```

const courseId = JSON.parse(
 document.getElementById('course-id').textContent
);
const url = 'ws://' + window.location.host +
 '/ws/chat/room/' + courseId + '/';
const chatSocket = new WebSocket(url);
{% endblock %}

```

В блоке `include_js` используется шаблонный фильтр `json_script`, чтобы безопасно использовать значение `course.id` с помощью JavaScript. Предоставляемый веб-фреймворком Django шаблонный фильтр `json_script` выводит объект Python в формате JSON, заключенный в тег `<script>`, чтобы безопасно его использовать с JavaScript. Исходный код `{{ course.id|json_script:"course-id" }}` транслируется как `<script id="course-id" type="application/json">6</script>`. Затем это значение извлекается в блоке `domready` путем структурного разбора содержимого элемента с `id="course-id"` с помощью `JSON.parse()`. Это безопасный способ использования объектов Python в JavaScript.

Дополнительная информация о шаблонном фильтре `json_script` находится по адресу <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#json-script>.

В блоке `domready` определяется URL-адрес с протоколом WebSocket, который выглядит как `ws://` (или `wss://` для безопасных веб-сокетов, подобно `https://`). Далее создается URL-адрес, используя текущее местоположение браузера, которое извлекается из `window.location.host`. Остальная часть URL-адреса формируется с помощью пути шаблона URL-адреса чат-комнаты, который был определен в файле `routing.py` приложения `chat`.

Здесь URL-адрес пишется вами вместо его формирования с помощью револьвера, потому что приложение-обертка `Channels` не предоставляет способа переворачивания URL-адресов. Вы используете ИД текущего курса, чтобы генерировать URL-адрес текущего курса и сохранять URL-адрес в новой константе с именем `url`.

Затем, применяя инструкцию `new WebSocket(url)`, открывается WebSocket-соединение с сохраненным URL-адресом. Инстанцированный экземпляр объекта WebSocket-клиента назначается новой константе `chatSocket`.

Вы создали WebSocket-потребителя, включили для него маршрутизацию и реализовали базового WebSocket-клиента. Давайте испытаем первоначальную версию чата.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/chat/room/1/` в своем браузере, заменив 1 на `id` существующего в базе данных курса. Взгляните на результат в консоли. Помимо HTTP-запросов страницы и ее статических файлов методом GET, вы должны увидеть две строки, включая WebSocket HANDSHAKING и WebSocket CONNECT, как показано ниже:

```
HTTP GET /chat/room/1/ 200 [0.02, 127.0.0.1:57141]
HTTP GET /static/css/base.css 200 [0.01, 127.0.0.1:57141]
WebSocket HANDSHAKING /ws/chat/room/1/ [127.0.0.1:57144]
WebSocket CONNECT /ws/chat/room/1/ [127.0.0.1:57144]
```

Встроенный в Channels сервер разработки прослушивает входящие сокет-соединения, используя стандартный TCP-сокет. Рукопожатие – это мост от HTTP к веб-сокетам. При рукопожатии согласовываются детали соединения, и любая из сторон может закрыть соединение до его завершения. Напомним, что для принятия любого соединения используется `self.accept()` в методе `connect()` класса `ChatConsumer`, реализованном в файле `consumers.py` приложения `chat`. Соединение принято, и поэтому в консоли появилось сообщение `WebSocket CONNECT`.

Если для отслеживания сетевых соединений вы используете браузерные инструменты для разработчика, то вы тоже можете просматривать информацию об установленном WebSocket-соединении.

Она должна выглядеть так, как показано на рис. 16.4:

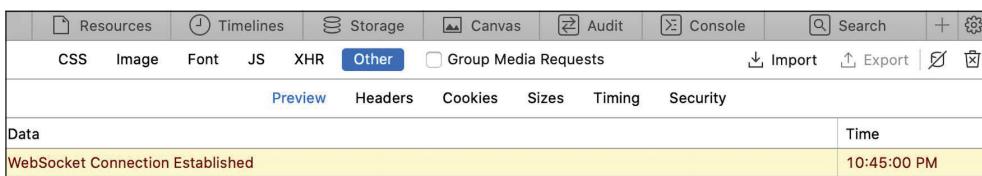


Рис. 16.4. Браузерные инструменты для разработчика, показывающие информацию об установленном WebSocket-соединении

Теперь, когда вы можете подсоединяться к веб-сокету, самое время начать с ним взаимодействовать. Вы реализуете методы оперирования общими событиями, такими как получение сообщения и закрытие соединения. Отредактируйте шаблон `chat/room.html` приложения `chat`, видоизменив блок `domready`, как показано ниже:

```
{% block domready %}
const courseId = JSON.parse(
 document.getElementById('course-id').textContent
);
const url = 'ws://' + window.location.host +
 '/ws/chat/room/' + courseId + '/';
const chatSocket = new WebSocket(url);

chatSocket.onmessage = function(event) {
 const data = JSON.parse(event.data);
 const chat = document.getElementById('chat');
 chat.innerHTML += '<div class="message">' +
 data.message + '</div>';
```

```

chat.scrollTop = chat.scrollHeight;
};

chatSocket.onclose = function(event) {
 console.error('Chat socket closed unexpectedly');
};
{% endblock %}

```

В приведенном выше исходном коде для WebSocket-клиента определяются следующие события:

- **onmessage**: запускается при получении данных через веб-сокет. Выполняется структурный разбор сообщения, которое ожидается в формате JSON, и берется его атрибут `message`. Затем в HTML-элемент с ИД чата добавляется новый элемент `<div>` с полученным сообщением. Таким путем будут добавляться новые сообщения в журнал чата, при этом сохраняя все предыдущие добавленные в журнал сообщения. Журнал чата `<div>` прокручивается вниз, чтобы новое сообщение стало видимым. Это достигается путем прокручивания до полной высоты журнала чата, которую можно получить, обратившись к его атрибуту `scrollHeight`;
- **onclose**: срабатывает при закрытии соединения с веб-сокетом. Закрытие соединения не ожидается, и, следовательно, в случае если это произойдет, в журнал консоли пишется ошибка `Chat socket closed unexpectedly`.

Вы реализовали действие по отображению сообщения при получении нового сообщения. Помимо этого, еще необходимо реализовать функциональность отправки сообщений в сокет.

Отредактируйте шаблон `chat/room.html` приложения `chat`, добавив следующий ниже исходный код JavaScript в нижнюю часть блока `domready`:

```

const input = document.getElementById('chat-message-input');
const submitButton = document.getElementById('chat-message-submit');

submitButton.addEventListener('click', function(event) {
 const message = input.value;
 if(message) {
 // отправить сообщение в формате JSON
 chatSocket.send(JSON.stringify({'message': message}));
 // очистить поле ввода
 input.innerHTML = '';
 input.focus();
 }
});

```

В приведенном выше исходном коде определяется прослушиватель события `click` кнопки передачи на обработку, которая идентифицируется по ее ИД `chat-message-submit`. При нажатии кнопки выполняются следующие действия:

1. Из поля ввода текста с ИД `chat-message-input` читается введенное пользователем сообщение.
2. С помощью булева выражения `if(message)` проверяется наличие у сообщения какого-либо содержимого.
3. Если пользователь ввел сообщение, то с помощью метода `JSON.stringify()` формируется содержимое JSON, например `{'message': 'string entered by the user'}`.
4. Содержимое JSON отправляется через веб-сокет, вызывая метод `send()` клиента `chatSocket`.
5. Содержимое поля ввода текста очищается, устанавливая его равным пустому строковому значению посредством инструкции `input.innerHTML = ''`.
6. С помощью метода `input.focus()` фокус возвращается к полю ввода текста, чтобы пользователь мог сразу написать новое сообщение.

Теперь пользователь может отправлять сообщения, используя поле ввода текста и кликая по кнопке передачи на обработку `submit`.

В целях улучшения взаимодействия пользователя со страницей поле ввода текста будет получать фокус сразу после загрузки страницы, чтобы пользователь мог вводить текст непосредственно в него. Помимо этого, будут улавливаться события нажатия клавиш клавиатуры, чтобы выявлять клавишу **Enter** и активировать событие `click` на кнопке `submit`. Для того чтобы отправлять сообщение, пользователь сможет кликать по кнопке либо нажимать клавишу **Enter**.

Отредактируйте шаблон `chat/room.html` приложения `chat`, добавив следующий ниже исходный код JavaScript в нижнюю часть блока `domready`:

```
input.addEventListener('keypress', function(event) {
 if (event.key === 'Enter') {
 // отменить стандартное действие,
 // если необходимо
 event.preventDefault();
 // запустить событие нажатия клавиши
 submitButton.click();
 }
});
input.focus();
```

В приведенном выше исходном коде также определяется функция события `keypress` в элементе `input`. Любая нажатая пользователем клавиша пропускается через проверку на нажатие клавиши **Enter**. При этом с помощью метода `event.preventDefault()` предотвращается стандартное для этой клавиши поведение. Если нажата клавиша **Enter**, то запускается событие `click` на кнопке `submitButton`, чтобы отправить сообщение в веб-сокет.

Вне обработчика событий, в главном исходном коде JavaScript блока `domready`, фокус передается полю ввода текста с помощью метода `input.focus()`. Таким образом, сразу после загрузки DOM фокус будет устанавливаться на элемент ввода текста, чтобы пользователь мог вводить сообщение.

Блок `domready` шаблона `chat/room.html` теперь должен выглядеть следующим образом:

```
{% block domready %}
const courseId = JSON.parse(
 document.getElementById('course-id').textContent
);
const url = 'ws://' + window.location.host +
 '/ws/chat/room/' + courseId + '/';
const chatSocket = new WebSocket(url);

chatSocket.onmessage = function(event) {
 const data = JSON.parse(event.data);
 const chat = document.getElementById('chat');
 chat.innerHTML += '<div class="message">' +
 data.message + '</div>';
 chat.scrollTop = chat.scrollHeight;
};

chatSocket.onclose = function(event) {
 console.error('Chat socket closed unexpectedly');
};

const input = document.getElementById('chat-message-input');
const submitButton = document.getElementById('chat-message-submit');

submitButton.addEventListener('click', function(event) {
 const message = input.value;
 if(message) {
 // отправить сообщение в формате JSON
 chatSocket.send(JSON.stringify({'message': message}));
 // очистить поле ввода
 input.value = '';
 input.focus();
 }
});

input.addEventListener('keypress', function(event) {
 if (event.key === 'Enter') {
 // отменить стандартное действие,
 // если необходимо
 event.preventDefault();
 // запустить событие нажатия клавиши
 submitButton.click();
 }
});

input.focus();
{% endblock %}
```

Пройдите по URL-адресу `http://127.0.0.1:8000/chat/room/1/` в своем браузере, заменив 1 на `id` существующего в базе данных курса. С вошедшим в систему пользователем, который зачислен на курс, введите текст в поле ввода и кликните по кнопке **SEND** (Отправить) либо нажмите клавишу **Enter**.

Вы увидите, что ваше сообщение появилось в журнале чата:

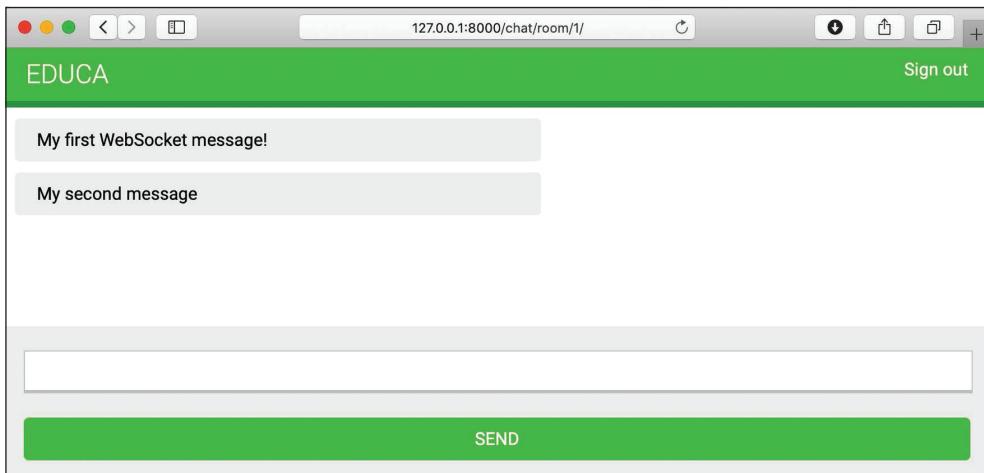


Рис. 16.5. Страница чат-комнаты, включающая сообщения, отправленные через веб-сокет

Отлично! Сообщение было отправлено через веб-сокет, а потребитель `ChatConsumer` получил сообщение и отправил его обратно через веб-сокет. Клиент `chatSocket` получил событие `message`, и была запущена функция `onmessage`, добавив сообщение в журнал чата.

Вы реализовали функциональность с WebSocket-потребителем и WebSocket-клиентом, чтобы устанавливать клиент-серверную связь и отправлять или получать события. Однако чат-сервер пока не может рассыпать сообщения другим клиентам в широковещательном режиме. Если открыть вторую вкладку браузера и ввести сообщение, то оно не появится на первой вкладке. Для того чтобы наладить связь между пользователями, необходимо активировать канальный слой.

## Активирование канального слоя

Канальные слои позволяют обмениваться данными между разными экземплярами приложения. Канальный уровень – это транспортный механизм, который позволяет нескольким экземплярам-потребителям общаться друг с другом и с другими частями Django.

На чат-сервере вы планируете иметь несколько экземпляров-потребителей `ChatConsumer` одной и той же чат-комнаты курса. Каждый присоединившийся

к чат-комнате студент будет создавать экземпляр WebSocket-клиента в своем браузере, а тот будет открывать соединение с экземпляром WebSocket-потребителя. Вам нужен общий канальный слой, чтобы распространять сообщения между потребителями.

## Каналы и группы

Канальные слои предоставляют две абстракции по управлению связью: каналы и группы:

- **канал**: канал можно трактовать как папку «Входящие», куда можно отправлять сообщения, либо как очередь заданий. У каждого канала есть имя. Сообщения отправляются в канал любым, кто знает имя канала, а затем передаются потребителям, прослушивающим этот канал;
- **группа**: несколько каналов могут быть сгруппированы в группу. Каждая группа имеет имя. Канал может быть добавлен или удален из группы любым, кто знает имя группы. Используя имя группы, также можно отправлять сообщение всем каналам в группе.

Вы будете работать с группами каналов, чтобы реализовать чат-сервер. Создав группу каналов для чат-комнаты каждого курса, экземпляры ChatConsumer смогут общаться друг с другом.

## Установление канального слоя с использованием Redis

Резидентное хранилище Redis является предпочтительным вариантом для канального слоя, хотя приложение-обертка Channels поддерживает другие типы канальных слоев. Redis работает для канального слоя как хранилище связи. Напомним, что вы уже использовали Redis в главе 7 «Отслеживание действий пользователя», главе 10 «Расширение магазина» и главе 14 «Прорисовка и кеширование контента».

Если вы еще не установили хранилище Redis, то можете найти инструкции по его установке в главе 7 «Отслеживание действий пользователя».

Для того чтобы использовать Redis в качестве канального слоя, необходимо установить пакет `channels-redis`. С помощью следующей команды установите пакет `channels-redis` в виртуальной среде:

```
pip install channels-redis==3.4.1
```

Отредактируйте файл `settings.py` проекта `educa`, добавив в него следующий ниже исходный код:

```
CHANNEL_LAYERS = {
 'default': {
 'BACKEND': 'channels_redis.core.RedisChannelLayer',
```

```
'CONFIG': {
 'hosts': [('127.0.0.1', 6379)],
},
}
```

Параметр `CHANNEL_LAYERS` задает конфигурацию доступных для проекта канальных слоев. Определяется канальный слой, который будет использоваться по умолчанию, используя предоставляемый пакетом `channels-redis` бэкенд `RedisChannelLayer`, и указывается хост `127.0.0.1` и порт `6379`, на котором работает Redis.

Давайте испытаем канальный слой. Следующей ниже командой инициализируйте контейнер Redis платформы Docker:

```
docker run -it --rm --name redis -p 6379:6379 redis
```

Если вы хотите выполнить команду в фоновом (раздельном) режиме, то можете воспользоваться опцией `-d`.

Откройте оболочку Django из каталога проекта командой

```
python manage.py shell
```

С целью подтверждения способности канального слоя общаться с Redis напишите следующий ниже исходный код, чтобы отправить сообщение на тестовый канал с именем `test_channel` и получить его обратно:

```
>>> import channels.layers
>>> from asgiref.sync import async_to_sync
>>> channel_layer = channels.layers.get_channel_layer()
>>> async_to_sync(channel_layer.send)('test_channel', {'message': 'hello'})
>>> async_to_sync(channel_layer.receive)('test_channel')
```

Вы должны получить такой результат:

```
{'message': 'hello'}
```

В приведенном выше исходном коде сообщение отправляется в тестовый канал через канальный слой, а затем извлекается из канального слоя. Канальный слой успешно общается с Redis.

## Обновление потребителя с целью широковещательной рассылки сообщений

Давайте отредактируем потребителя `ChatConsumer`, чтобы задействовать канальный слой. Вы будете использовать группу каналов для чат-комнаты каждого курса, и поэтому для формирования имени группы будете использовать

`id` курса. Экземпляры `ChatConsumer` будут знать имя группы и смогут общаться друг с другом.

Откройте в редакторе файл `consumers.py` приложения `chat`, импортируйте функцию `async_to_sync()` и видоизмените метод `connect()` класса `ChatConsumer`, как показано ниже:

```
import json
from channels.generic.websocket import WebsocketConsumer
from asgiref.sync import async_to_sync

class ChatConsumer(WebsocketConsumer):
 def connect(self):
 self.id = self.scope['url_route']['kwargs']['course_id']
 self.room_group_name = f'chat_{self.id}'
 # присоединиться к группе чат-комнаты
 async_to_sync(self.channel_layer.group_add)(
 self.room_group_name,
 self.channel_name
)
 # принять соединение
 self.accept()
 # ...
```

В приведенном выше исходном коде импортируется вспомогательная функция `async_to_sync()`, чтобы обернуть вызовы асинхронных методов канального слоя. `ChatConsumer` – это синхронный потребитель `WebsocketConsumer`, но ему необходимо вызывать асинхронные методы канального слоя.

В новом методе `connect()` выполняется следующая работа.

1. Из области видимости извлекается `id` курса, чтобы выяснить, с каким курсом связана чат-комната. При этом параметр `course_id` извлекается из URL-адреса посредством инструкции доступа `self.scope['url_route'][ 'kwargs' ][ 'course_id' ]`. Каждый потребитель имеет область видимости с информацией о его соединении, переданных в URL-адресе аргументах и аутентифицированном пользователе, если таковой имеется.
2. Формируется имя группы с `id` курса, которому группа соответствует. Напомним, что у вас будет группа каналов для чат-комнаты каждого курса. Имя группы сохраняется в атрибуте `room_group_name` потребителя.
3. Выполняется присоединение к группе, добавляя текущий канал в группу. Из атрибута `channel_name` потребителя берется имя канала. Метод `group_add` канального слоя используется для добавления канала в группу. Обертка `async_to_sync()` применяется для использования асинхронного метода канального слоя.
4. Вызов `self.accept()` сохраняется, чтобы принять WebSocket-соединение.

При получении потребителем `ChatConsumer` нового WebSocket-соединения он добавляет канал в группу, связанную с курсом в его области видимости. Теперь потребитель может получать любые отправляемые группе сообщения.

В том же файле `consumers.py` видоизмените метод `disconnect()` класса `ChatConsumer`, как показано ниже:

```
class ChatConsumer(WebSocketConsumer):
 # ...
 def disconnect(self, close_code):
 # покинуть группу чат-комнаты
 async_to_sync(self.channel_layer.group_discard)(
 self.room_group_name,
 self.channel_name
)
 # ...
```

При закрытии соединения вызывается метод `group_discard()` канального слоя, чтобы покинуть группу. Оболочка `async_to_sync()` применяется для использования асинхронного метода канального слоя.

В том же файле `consumers.py` видоизмените метод `receive()` класса `ChatConsumer`, как показано ниже:

```
class ChatConsumer(WebSocketConsumer):
 # ...
 # получить сообщение из веб-сокета
 def receive(self, text_data):
 text_data_json = json.loads(text_data)
 message = text_data_json['message']
 # отправить сообщение в группу чат-комнаты
 async_to_sync(self.channel_layer.group_send)(
 self.room_group_name,
 {
 'type': 'chat_message',
 'message': message,
 }
)
```

При получении сообщения из WebSocket-соединения вместо отправки сообщения в ассоциированный канал отправляется сообщение группе. Это делается путем вызова метода `group_send()` канального слоя. Оболочка `async_to_sync()` применяется для использования асинхронного метода канального слоя. В отправляемом группе событии передается следующая информация:

- `type`: тип события. Это специальный ключ, соответствующий имени метода, который следует вызывать с потребителями, получающими событие. В потребителе можно реализовать метод с таким же именем, как у типа сообщения, чтобы он выполнялся всякий раз при получении сообщения с этим конкретным типом;
- `message`: фактическое отправляемое сообщение.

В том же файле `consumers.py` добавьте метод `newchat_message()` в класс `ChatConsumer`, как показано ниже:

```
class ChatConsumer(WebSocketConsumer):
 # ...
 # получить сообщение из группы чат-комнаты
 def chat_message(self, event):
 # отправить сообщение в веб-сокет
 self.send(text_data=json.dumps(event))
```

Этот метод называется `chat_message()`, чтобы он соответствовал ключу `type`, который отправляется в канальную группу при получении сообщения из веб-сокета. При отправке в группу сообщения с типом `chat_message` все подписанные на группу потребители будут получать это сообщение и исполнять метод `chat_message()`. В методе `chat_message()` полученное сообщение о событии отправляется в веб-сокет.

Полный файл `consumers.py` теперь должен выглядеть так:

```
import json
from channels.generic.websocket import WebSocketConsumer
from asgiref.sync import async_to_sync

class ChatConsumer(WebSocketConsumer):
 def connect(self):
 self.id = self.scope['url_route']['kwargs']['course_id']
 self.room_group_name = f'chat_{self.id}'
 # присоединиться к группе чат-комнаты
 async_to_sync(self.channel_layer.group_add)(
 self.room_group_name,
 self.channel_name
)
 # принять соединение
 self.accept()

 def disconnect(self, close_code):
 # покинуть группу чат-комнаты
 async_to_sync(self.channel_layer.group_discard)(
 self.room_group_name,
 self.channel_name
)

 # получить сообщение из веб-сокета
 def receive(self, text_data):
 text_data_json = json.loads(text_data)
 message = text_data_json['message']
 # отправить сообщение в группу чат-комнаты
```

```
async_to_sync(self.channel_layer.group_send)(
 self.room_group_name,
 {
 'type': 'chat_message',
 'message': message,
 }
)

получить сообщение из группы чат-комнаты
def chat_message(self, event):
 # отправить сообщение в веб-сокет
 self.send(text_data=json.dumps(event))
```

Вы реализовали канальный слой в потребителе `ChatConsumer`, позволяющий потребителям широковещательно рассылать сообщения и общаться друг с другом.

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/chat/room/1/` в своем браузере, заменив 1 на `id` существующего в базе данных курса. Напишите сообщение и отправьте его. Затем откройте второе окно браузера и обратитесь к тому же URL-адресу. Отправьте сообщение из каждого окна браузера.

Результат должен выглядеть так:



Рис. 16.6. Страница чат-комнаты с сообщениями, отправленными из разных окон браузера

Вы увидите, что первое сообщение отображается только в первом окне браузера. Но, открыв второе окно браузера, увидите, что сообщения, отправляемые в любом окне браузера, отображаются в них обоих. При открытии нового окна браузера и обращении к URL-адресу чат-комнаты между WebSocket-клиентом на JavaScript в браузере и WebSocket-потребителем на сервере устанавливается новое WebSocket-соединение. Каждый канал добавляется в связанную с ИД курса группу и передается потребителю через URL-адрес. Сообщения отправляются группе и принимаются всеми пользователями.

## Добавление контекста в сообщения

Теперь, когда между всеми пользователями в чат-комнате можно обмениваться сообщениями, вы, вероятно, захотите отображать, кто какое сообщение отправил и когда оно было отправлено. Давайте добавим в сообщения контекст.

Отредактируйте файл `consumers.py` приложения `chat`, внеся следующие ниже изменения:

```
import json
from channels.generic.websocket import WebsocketConsumer
from asgiref.sync import async_to_sync
from django.utils import timezone

class ChatConsumer(WebsocketConsumer):
 def connect(self):
 self.user = self.scope['user']
 self.id = self.scope['url_route']['kwargs']['course_id']
 self.room_group_name = f'chat_{self.id}'
 # присоединиться к группе чат-комнаты
 async_to_sync(self.channel_layer.group_add)(
 self.room_group_name,
 self.channel_name
)
 # принять соединение
 self.accept()

 def disconnect(self, close_code):
 # покинуть группу чат-комнаты
 async_to_sync(self.channel_layer.group_discard)(
 self.room_group_name,
 self.channel_name
)

 # получить сообщение из веб-сокета
 def receive(self, text_data):
 text_data_json = json.loads(text_data)
```

```

message = text_data_json['message']
now = timezone.now()
отправить сообщение в группу чат-комнаты
async_to_sync(self.channel_layer.group_send)(
 self.room_group_name,
 {
 'type': 'chat_message',
 'message': message,
 'user': self.user.username,
 'datetime': now.isoformat(),
 }
)

получить сообщение из группы чат-комнаты
def chat_message(self, event):
 # отправить сообщение в веб-сокет
 self.send(text_data=json.dumps(event))

```

Теперь импортируется предоставляемый веб-фреймворком Django модуль `timezone`. В методе `connect()` потребителя посредством инструкции `self.scope['user']` из области видимости извлекается текущий пользователь и сохраняется в новом атрибуте `user` потребителя. При получении потребителем сообщения через веб-сокет он получает текущее время с помощью `timezone.now()` и передает текущего `user` и `datetime` в формате ISO 8601 вместе с сообщением в событии, отправляемом канальной группе.

Отредактируйте шаблон `chat/room.html` приложения `chat`, добавив в блок `include_js` следующую ниже строку, выделенную жирным шрифтом:

```

{% block include_js %}
{{ course.id|json_script:"course-id" }}
{{ request.user.username|json_script:"request-user" }}
{% endblock %}

```

Используя шаблон `json_script`, безопасно печатается пользовательское имя (`username`) пользователя запроса, чтобы применять его с JavaScript.

В блок `domready` шаблона `chat/room.html` добавьте следующие ниже строки, выделенные жирным шрифтом:

```

{% block domready %}
const courseId = JSON.parse(
 document.getElementById('course-id').textContent
);
const requestUser = JSON.parse(
 document.getElementById('request-user').textContent
);
...
{% endblock %}

```

В новом исходном коде безопасно выполняется структурный разбор данных элемента с ИД `request-user`, и результат сохраняется в константе `requestUser`. Затем в блоке `domready` найдите следующие ниже строки:

```
const data = JSON.parse(e.data);
const chat = document.getElementById('chat');

chat.innerHTML += '<div class="message">' +
 data.message + '</div>';
chat.scrollTop = chat.scrollHeight;
```

Замените эти строки таким исходным кодом:

```
const data = JSON.parse(e.data);
const chat = document.getElementById('chat');

const dateOptions = {hour: 'numeric', minute: 'numeric', hour12: true};
const datetime = new Date(data.datetime).toLocaleString('en', dateOptions);
const isMe = data.user === requestUser;
const source = isMe ? 'me' : 'other';
const name = isMe ? 'Me' : data.user;

chat.innerHTML += '<div class="message ' + source + '">' +
 '' + name + '' +
 '' + datetime + '
' +
 data.message + '</div>';
chat.scrollTop = chat.scrollHeight;
```

В приведенном выше исходном коде реализуются следующие изменения.

- Полученное в сообщении `datetime` конвертируется в объект JavaScript `Date` и форматируется с определенной локалью.
- Полученное в сообщении пользовательское имя (`username`) сравнивается с двумя разными константами в качестве помощников, чтобы идентифицировать пользователя.
- Константа `source` получает значение `me`, если отправляющий сообщение пользователь является текущим пользователем, либо `other` в противном случае.
- Константа `name` получает значение `Me`, если отправляющий сообщение пользователь является текущим пользователем, либо имя отправляющего сообщение пользователя в противном случае. Оно используется для отображения имени отправляющего сообщение пользователя.
- Значение `source` используется как `class` главного элемента `<div>`, содержащего сообщение, чтобы отличать сообщения, отправленные текущим пользователем, от сообщений, отправленных другими. Различные CSS-стили применяются на основе атрибута `class`. Указанные CSS-стили объявлены в статическом файле `css/base.css`.
- В сообщении, которое добавляется в журнал чата, используются пользовательское имя (`username`) и `datetime`.

Пройдите по URL-адресу `http://127.0.0.1:8000/chat/room/1/` в своем браузере, заменив 1 на `id` существующего в базе данных курса. С вошедшим в систему пользователем, который зачислен на курс, напишите сообщение и отправьте его.

Затем откройте второе окно браузера в режиме инкогнито, чтобы предотвратить использование того же сеанса. Войдите под другим пользователем, тоже зачисленным на тот же курс, и отправьте сообщение.

Вы сможете обмениваться сообщениями, используя двух разных пользователей, и видеть пользователя и время, с четким различием между сообщениями, отправленными пользователем, и сообщениями, отправленными другими. Разговор между двумя пользователями должен выглядеть примерно так:



Рис. 16.7. Страница чат-комнаты с сообщениями из двух разных пользовательских сеансов

Отлично! Вы создали функциональное реально-временное приложение для ведения чата с использованием приложения-обертки Channels. Далее вы узнаете, как улучшить потребителя чата, сделав его полностью асинхронным.

## Видоизменение потребителя с целью обеспечения полной асинхронности

Реализованный вами `ChatConsumer` наследует от базового класса `WebSocketConsumer`, который является синхронным. Синхронные потребители удобны для доступа к моделям Django и вызова обычных функций синхронного ввода-

вывода. Однако асинхронные потребители работают лучше, так как при оперировании запросами им не требуются дополнительные потоки. Поскольку вы используете асинхронные функции канального слоя, то можете легко переписать класс `ChatConsumer`, сделав его асинхронным.

Отредактируйте файл `consumers.py` приложения `chat`, внеся следующие ниже изменения:

```
import json
from channels.generic.websocket import AsyncWebSocketConsumer
from asgiref.sync import async_to_sync
from django.utils import timezone

class ChatConsumer(AsyncWebSocketConsumer):
 async def connect(self):
 self.user = self.scope['user']
 self.id = self.scope['url_route']['kwargs']['course_id']
 self.room_group_name = 'chat_%s' % self.id
 # присоединиться к группе чат-комнаты
 await self.channel_layer.group_add(
 self.room_group_name,
 self.channel_name
)
 # принять соединение
 await self.accept()

 async def disconnect(self, close_code):
 # покинуть группу чат-комнаты
 await self.channel_layer.group_discard(
 self.room_group_name,
 self.channel_name
)

 # получить сообщение из веб-сокета
 async def receive(self, text_data):
 text_data_json = json.loads(text_data)
 message = text_data_json['message']
 now = timezone.now()
 # отправить сообщение в группу чат-комнаты
 await self.channel_layer.group_send(
 self.room_group_name,
 {
 'type': 'chat_message',
 'message': message,
 'user': self.user.username,
 'datetime': now.isoformat(),
 }
)
```

```
получить сообщение из группы чат-комнаты
async def chat_message(self, event):
 # отправить сообщение в веб-сокет
 await self.send(text_data=json.dumps(event))
```

Здесь были внесены следующие изменения:

- 1) потребитель `ChatConsumer` теперь наследует от класса `AsyncWebSocketConsumer`, чтобы реализовать асинхронные вызовы;
- 2) определение всех методов было заменено с `def` на `async def`;
- 3) для вызова асинхронных функций, выполняющих операции ввода-вывода, используется `await`;
- 4) вспомогательная функция `async_to_sync()` больше не используется при вызове методов канального слоя.

Снова пройдите по URL-адресу `http://127.0.0.1:8000/chat/room/1/` в двух разных окнах браузера и убедитесь, что чат-сервер по-прежнему работает. Чат-сервер теперь полностью асинхронный!

## Интеграция приложения для ведения чата с существующими представлениями

Чат-сервер теперь полностью реализован, и зачисленные на курс студенты могут общаться друг с другом. Давайте добавим ссылку на присоединение студентов к чат-комнате по каждому курсу.

Отредактируйте шаблон `student/course/detail.html` приложения `students`, добавив следующий ниже исходный код HTML-элемента `<h3>` внизу элемента `<div class="contents">`:

```
<div class="contents">
...
<h3>

 Course chat room

</h3>
</div>
```

Откройте браузер и обратитесь к любому курсу, на который студент зачислен, чтобы просмотреть его содержимое. Боковая панель теперь будет содержать ссылку **Course chat room** (Чат-комната курса), которая указывает на представление чат-комнаты курса. Если вы по ней кликнете, то попадете в чат-комнату:

The screenshot shows a web application interface for a course titled 'Introduction to Django' on the EDUCA platform. On the left, a sidebar lists four modules: 'Introduction to Django', 'Configuring Django', 'Your first Django project', and 'Django URLs'. The main content area has a heading 'Why Django?' followed by a text block: 'Meet Django. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.' Below this is a section titled 'Django video' featuring a thumbnail for a video from 'DjangoCon 2012' by Malcolm Tredinnick. The thumbnail includes a play button and a list of bullet points: 'In the background...', 'All aliases in a QuerySet can be changed at once', 'T1, T2, T3, ... -> U1, U2, U3, ...', 'for nested queries', 'QuerySets can be merged', and 'Same table can appear with different aliases...'. A 'New Relic' logo is visible in the bottom right corner of the video thumbnail.

Рис. 16.8. Страница детальной информации о курсе, содержащая ссылку на чат-комнату курса

Поздравляем! Вы успешно создали свое первое асинхронное приложение с использованием приложения-обертки Django Channels.

## Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе:

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter16>.
- Введение в ASGI: <https://asgi.readthedocs.io/en/latest/introduction.html>.
- Поддержка веб-фреймворком Django асинхронных представлений: <https://docs.djangoproject.com/en/4.1/topics/async/>.
- Поддержка веб-фреймворком Django асинхронных представлений на основе классов: <https://docs.djangoproject.com/en/4.1/topics/class-based-views/#async-class-based-views>.
- Документация по приложению-обертке Django Channels: <https://channels.readthedocs.io/>.

- Развёртывание Django с ASGI: <https://docs.djangoproject.com/en/4.1/how-to/deployment/asgi/>.
- Использование шаблонного фильтра `json_script`: <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#json-script>.

## Резюме

В этой главе вы научились создавать чат-сервер, используя каналы на основе приложения Django Channels. Вы реализовали WebSocket-потребителя и WebSocket-клиента. Вы также активировали связь между потребителями, используя канальный слой с резидентным хранилищем Redis, и видоизменили потребителя, сделав его полностью асинхронным.

В следующей главе вы научитесь создавать производственную среду для проекта Django с использованием веб-сервера NGINX, сервера приложений uWSGI и асинхронного веб-сервера Daphne с помощью инструмента Docker Compose. Вы также научитесь реализовывать конкретно-прикладные промежуточные программные компоненты и создавать конкретно-прикладные команды управления.

# 17

## Выход в прямой эфир

В предыдущей главе вы разработали реально-временной чат-сервер для студентов на основе приложения-обертки Django Channels. Теперь, когда вы создали полнофункциональную платформу электронного обучения, вам необходимо настроить производственную среду, чтобы к ней обращаться через интернет. До сего момента вы работали в среде разработки, используя для работы сайта встроенный в Django сервер разработки. В этой главе вы научитесь настраивать производственную среду, способную безопасно и эффективно раздавать проект Django.

В этой главе будут рассмотрены следующие темы:

- конфигурирование настроек параметров Django под несколько сред;
- использование инструмента Docker Compose для запуска нескольких служб;
- настройка веб-сервера с сервером приложений uWSGI и Django;
- раздача данных PostgreSQL и Redis с помощью Docker Compose;
- использование встроенного в Django фреймворка проверки системы;
- раздача веб-сервера NGINX с помощью платформы Docker;
- раздача статических ресурсов через веб-сервер NGINX;
- защита соединений через TLS/SSL;
- использование ASGI-сервера Daphne для приложения Django Channels;
- создание конкретно-прикладных промежуточных программных компонентов Django;
- реализация конкретно-прикладных команд управления Django.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter17>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

# Создание производственной среды

Пришло время развернуть свой проект Django в производственной среде. Вы начнете с конфигурирования настроек параметров Django под несколько сред, а затем настройте производственную среду.

## Управление настроек параметрами для нескольких сред

В реальных проектах вам придется иметь дело с несколькими средами. Обычно у вас будет как минимум локальная среда для разработки и производственная среда для раздачи вашего приложения по интернету. У вас могут быть и другие среды, такие как тестовая среда и/или промежуточная среда.

Некоторые настройки проекта будут общими для всех сред, другие же будут специфическими для каждой среды. Обычно вы будете использовать базовый файл, определяющий общие настроек параметры, и файл настроек параметров по каждой среде, в котором переопределяются все необходимые настроек параметры и определяются дополнительные.

Мы будем управлять следующими средами:

- `local`: локальная среда для запуска и управления проектом на своем компьютере;
- `prod`: среда для развертывания проекта на производственном сервере.

Создайте каталог `settings/` рядом с файлом `settings.py` проекта `educa`. Переименуйте файл `settings.py` в `base.py` и переместите его в новый каталог `settings/`.

Создайте следующие ниже дополнительные файлы внутри папки `settings/`, чтобы новый каталог выглядел следующим образом:

```
settings/
 __init__.py
 base.py
 local.py
 prod.py
```

Ниже приведено описание этих файлов:

- `base.py`: файл базовых настроек, содержащий общие настроек параметры (ранее это был файл `settings.py`);
- `local.py`: конкретно-прикладные настройки под свою локальную среду;
- `prod.py`: конкретно-прикладные настройки под производственную среду.

Вы переместили файлы настроек параметров в каталог на один уровень ниже, поэтому теперь необходимо обновить параметр `BASE_DIR` в файле `settings/base.py`, чтобы он указывал на главный каталог проекта.

При оперировании несколькими средами следует создавать файл базовых настроек и файл настроек под каждую среду. Файлы настроек параметров среды должны наследовать общие настроочные параметры и переопределять специфичные для среды параметры.

Отредактируйте файл `settings/base.py`, заменив строку

```
BASE_DIR = Path(__file__).resolve().parent.parent
```

следующей ниже:

```
BASE_DIR = Path(__file__).resolve().parent.parent.parent
```

Добавив `.parent` к пути `BASE_DIR`, вы указываете на один каталог выше. Давайте сконфигурируем настроочные параметры локальной среды.

## ***Настроочные параметры локальной среды***

Вместо использования стандартной конфигурации настроек параметров `DEBUG` и `DATABASES` вы определите их в явной форме для каждой среды. Эти настройки будут специфичными для каждой среды. Отредактируйте файл `educa/settings/local.py`, добавив следующие ниже строки:

```
from .base import *

DEBUG = True

DATABASES = {
 'default': {
 'ENGINE': 'django.db.backends.sqlite3',
 'NAME': BASE_DIR / 'db.sqlite3',
 }
}
```

Это файл настроек параметров вашей локальной среды. В данном файле импортируются все настройки, определенные в файле `base.py`, и определяются настроочные параметры `DEBUG` и `DATABASES` этой среды. Параметры `DEBUG` и `DATABASES` остаются такими же, какими они использовались для разработки.

Теперь удалите настроочные параметры `DATABASES` и `DEBUG` из файла настроек `base.py`.

Встроенные в Django команды управления не будут автоматически обнаруживать подлежащий использованию файл настроек, потому что файл настроек проекта не является системно-заданным файлом `settings.py`. При выполнении команд управления необходимо указывать используемый модуль настроек, добавляя параметр `--settings`, как показано ниже:

```
python manage.py runserver --settings=educa.settings.local
```

Далее мы проведем валидацию проекта и конфигурации локальной среды.

## Запуск локальной среды

Давайте запустим локальную среду, используя новую структуру настроек. Проверьте, чтобы сервер Redis был запущен, либо следующей ниже командой запустите контейнер Redis платформы Docker в оболочке:

```
docker run -it --rm --name redis -p 6379:6379 redis
```

В еще одной оболочке выполните следующую ниже команду управления из каталога проекта:

```
python manage.py runserver --settings=educa.settings.local
```

Пройдите по URL-адресу `http://127.0.0.1:8000/` в своем браузере и убедитесь, что сайт загружается корректно. Теперь вы раздаете свой сайт, используя настройки локальной среды `local`.

Если вы не хотите передавать параметр `--settings` при каждом выполнении команды управления, то можете определить средовую переменную `DJANGO_SETTINGS_MODULE`. Django будет использовать ее для идентификации используемого модуля настроек. Если вы используете Linux или macOS, то можете определить средовую переменную, выполнив следующую ниже команду в оболочке:

```
export DJANGO_SETTINGS_MODULE=educa.settings.local
```

Если вы работаете с Windows, то выполните показанную далее команду в оболочке:

```
set DJANGO_SETTINGS_MODULE=educa.settings.local
```

Любая команда управления, которую вы будете выполнять позже, будет использовать настройки, определенные в средовой переменной `DJANGO_SETTINGS_MODULE`.

Остановите встроенный в Django сервер разработки из оболочки, нажав клавиши **Ctrl+C**, и остановите контейнер Redis платформы Docker из оболочки, также нажав клавиши **Ctrl+C**.

Локальная среда работает без проблем. Давайте подготовим настройки производственной среды.

## Настройки производственной среды

Начнем с добавления первоначальных настроек производственной среды. Откройте в редакторе файл `educa/settings/prod.py` и придавайте ему следующий вид:

```
from .base import *

DEBUG = False

ADMINS = [
 ('Antonio M', 'email@mydomain.com'),
]

ALLOWED_HOSTS = ['*']

DATABASES = {
 'default': {
 }
}
```

Ниже перечислены настроочные параметры производственной среды:

- DEBUG: установка значения `False` параметру `DEBUG` необходима для любой производственной среды. Если этого не сделать, то информация об обратной трассировке и чувствительные конфигурационные данные станут доступны всем;
- ADMINS: если значение параметра `DEBUG` равно `False` и представление вызывает исключение, то вся информация будет отправлена по электронной почте людям, указанным в настроичном параметре `ADMINS`. Проследите, чтобы кортеж имя / адрес электронной почты был заменен своей информацией;
- ALLOWED\_HOSTS: из соображений безопасности Django разрешает раздачу проекта только теми хостами, которые включены в этот список. На данный момент вы разрешаете все хосты, используя символ звездочки `*`. Позже вы ограничите число хостов, которые можно использовать для раздачи проекта;
- DATABASES: вы оставляете предустановленные (`default`) настроочные параметры базы данных пустыми, поскольку сконфигурируете производственную базу данных чуть позже.

В следующих далее разделах этой главы вы создадите файл настроек своей производственной среды.

Вы успешно организовали настройки для оперирования несколькими средами. Теперь вы создадите полную производственную среду, настроив различные службы с помощью Docker.

# Использование инструмента Docker Compose

Платформа Docker позволяет создавать, развертывать и запускать контейнеры приложений. Контейнер платформы Docker объединяет исходный код приложения с библиотеками операционной системы и зависимостями, необходимыми для запуска приложения. Используя контейнеры приложений, можно улучшать переносимость приложений. Вы уже применяете образ Redis платформы Docker для раздачи Redis в локальной среде. Указанный образ платформы Docker содержит все необходимое для запуска сервера Redis и позволяет запускать его без проблем на своем компьютере. В производственной среде вы будете использовать инструмент Docker Compose, чтобы создавать и запускать различные контейнеры платформы Docker.

Docker Compose – это инструмент, служащий для определения и выполнения многоконтейнерных приложений. При этом он позволяет создавать конфигурационный файл с определениями различных служб и использовать одну команду, чтобы запускать все службы из своей конфигурации. Информация об инструменте Docker Compose находится по адресу <https://docs.docker.com/compose/>.

Для производственной среды вы создадите распределенное приложение, которое работает в нескольких контейнерах платформы Docker. В каждом контейнере будет работать отдельная служба. Сначала определяются следующие три службы, а в последующих разделах добавляются дополнительные службы:

- веб-служба: веб-сервер, который раздает проект Django;
- служба базы данных: служба базы данных, которая обеспечивает работу PostgreSQL;
- служба кеша: служба, которая обеспечивает работу сервера Redis.

Начнем с установки инструмента Docker Compose.

## Установка инструмента Docker Compose

Инструмент Docker Compose можно запускать в macOS, 64-битной версии Linux и Windows. Самый быстрый способ установить инструмент Docker Compose – установить настольное приложение Docker Desktop. Установка включает Docker Engine, интерфейс командной строки и плагин инструмента Docker Compose.

Установите настольное приложение Docker Desktop, следуя инструкциям на странице <https://docs.docker.com/compose/install/compose-desktop/>.

Откройте настольное приложение Docker Desktop и кликните по **Containers** (Контейнеры). Результат будет выглядеть следующим образом:



Рис. 17.1. Интерфейс настольного приложения Docker Desktop

После установки инструмента Docker Compose нужно создать Docker-образ проекта Django.

## Создание файла Dockerfile

Теперь необходимо создать Docker-образ для выполнения проекта Django. Dockerfile – это текстовый файл, содержащий команды Docker по сборке Docker-образа. Вы подготовите Dockerfile с помощью команд, чтобы сформировать Docker-образ проекта Django.

Рядом с каталогом проекта `educa` создайте новый файл и назовите его `Dockerfile`. Добавьте в новый файл следующий ниже исходный код:

```
Взять официальный базовый образ Python платформы Docker
FROM python:3.10.6

Задать переменные среды
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

Задать рабочий каталог
WORKDIR /code

Установить зависимости
RUN pip install --upgrade pip
```

Приведенный выше исходный код выполняет следующую работу.

1. Используется родительский образ Python 3.10.6 платформы Docker. Официальный образ Python платформы Docker находится по адресу [https://hub.docker.com/\\_/python](https://hub.docker.com/_/python).

2. Задаются следующие ниже средовые переменные:
  - a) PYTHONDONTWRITEBYTECODE: запрещает Python писать файлы рус;
  - b) PYTHONUNBUFFERED: обеспечивает, чтобы потоки Python `stdout` и `stderr` отправлялись прямо в терминал без предварительной буферизации.
3. Команда `WORKDIR` используется для определения рабочего каталога образа.
4. Выполняется апгрейд пакета `pip` образа.
5. Файл `requirements.txt` копируется в каталог `code` родительского образа Python.
6. Пакеты Python в файле `requirements.txt` устанавливаются в образ с помощью `pip`.
7. Исходный код проекта Django копируется из локального каталога в каталог `code` образа.

С помощью этого файла `Dockerfile` вы определили порядок сборки Docker-образа для раздачи Django.

Справочник по файлу `Dockerfile` находится по адресу <https://docs.docker.com/engine/reference/builder/>.

## Добавление требующихся пакетов Python

Файл `requirements.txt` используется в созданном вами файле `Dockerfile`, чтобы установить все необходимые для проекта пакеты Python.

Рядом с каталогом проекта `educa` создайте новый файл и назовите его `requirements.txt`. Возможно, вы уже создали этот файл раньше и скопировали содержимое файла `requirements.txt` со страницы <https://github.com/PacktPublishing/Django-4-by-example/blob/main/Chapter17/requirements.txt>. Если вы этого не сделали, то добавьте следующие ниже строки в только что созданный файл `requirements.txt`:

```
asgiref==3.5.2
Django~=4.1
Pillow==9.2.0
sqlparse==0.4.2
django-braces==1.15.0
django-embed-video==1.4.4
pymemcache==3.5.2
django-debug-toolbar==3.6.0
redis==4.3.4
django-redisboard==8.3.0
djangorestframework==3.13.1
requests==2.28.1
channels==3.0.5
channels-redis==3.4.1
psycopg2==2.9.3
uwsgi==2.0.20
daphne==3.0.2
```

В дополнение к пакетам Python, которые вы установили в предыдущих главах, в файл requirements.txt включены следующие ниже пакеты:

- psycopg2: адаптер PostgreSQL. В производственной среде будет использоваться PostgreSQL;
- uwsgi: веб-сервер на основе WSGI. Позже вы сконфигурируете этот веб-сервер под раздачу Django в производственной среде;
- daphne: веб-сервер на основе ASGI. Позже вы будете использовать этот веб-сервер для раздачи приложения Django Channels.

Начнем с настройки приложения платформы Docker в инструменте Docker Compose. Мы создадим файл Compose платформы Docker с определением служб веб-сервера, базы данных и Redis.

## Создание файла Compose платформы Docker

Для того чтобы определить службы, которые будут работать в разных контейнерах платформы Docker, мы будем использовать файл Compose платформы Docker. Файл Compose – это текстовый файл в формате YAML, в котором для приложения Docker определены службы, сети и тома данных. YAML – это человекочитаемый язык сериализации данных. Пример файла YAML можно увидеть по адресу <https://yaml.org/>.

Рядом с каталогом проекта educa создайте новый файл и назовите его docker-compose.yml. Добавьте в него следующий ниже исходный код:

```
services:

 web:
 build: .
 command: python /code/educa/manage.py runserver 0.0.0.0:8000
 restart: always
 volumes:
 - .:/code
 ports:
 - "8000:8000"
 environment:
 - DJANGO_SETTINGS_MODULE=educa.settings.prod
```

В приведенном выше файле определяется веб-служба. Ниже приводится описание разделов, в которых указанная служба определяется:

- build: определяет требования к сборке образа контейнера службы. Это может быть отдельная строка, определяющая путь к контексту, либо подробное определение сборки. Относительный путь задается одной точкой ., указывая на тот же каталог, где находится файл Compose. Инструмент Docker Compose будет искать файл Dockerfile в этом месте. Подробнее о разделе build можно почитать на странице <https://docs.docker.com/compose/compose-file/build/>;

- `command`: переопределяет команду контейнера, используемую по умолчанию. Встроенный в Django сервер разработки запускается с помощью команды управления `runserver`. Проект раздается на хосте `0.0.0.0`, то есть по стандартному IP-адресу Docker, на порту `8000`;
- `restart`: определяет политику перезапуска контейнера. Используя `always`, контейнер перезапускается всегда, в случае если он останавливается. Это удобно в условиях производственной среды, где требуется свести к минимуму время простоя. Подробнее о политике перезапуска можно почитать по адресу <https://docs.docker.com/config/Containers/start-containers-automatically/>;
- `volumes`: данные в контейнерах платформы Docker не являются постоянными. Каждый контейнер имеет виртуальную файловую систему, которая заполняется файлами образа и уничтожается при остановке контейнера. Тома представляют собой предпочтительный метод поддержания постоянства данных, создаваемых и используемых контейнерами. В этом разделе локальный каталог `.code` монтируется в каталог `/code` образа. Подробнее о томах Docker можно узнать по адресу <https://docs.docker.com/storage/volumes/>;
- `ports`: открывает порты контейнеров. Порт `8000` хоста соотносится с портом `8000` контейнера, на котором работает встроенный в Django сервер разработки;
- `environment`: определяет средовые переменные. Средовая переменная `DJANGO_SETTINGS_MODULE` устанавливается для использования файла производственных настроек `Django educa.settings.prod`.

Обратите внимание, что в определении Docker-файла Compose для раздачи приложения используется встроенный в Django сервер разработки. Указанный сервер разработки не подходит для использования в производственной среде, поэтому позже вы замените его веб-сервером Python на основе WSGI.

Информация о спецификации Docker Compose находится по адресу <https://docs.docker.com/compose/compose-file/>.

На этом этапе, исходя из того, что родительский каталог называется `Chapter17`, файловая структура должна выглядеть следующим образом:

```
Chapter17/
 Dockerfile
 docker-compose.yml
 educa/
 manage.py
 ...
 requirements.txt
```

Откройте оболочку в родительском каталоге, в котором находится файл `docker-compose.yml`, и выполните следующую ниже команду:

```
docker compose up
```

Она запустит приложение платформы Docker, определенное в Docker-файле Compose. Вы увидите результат, который содержит такие строки:

```
chapter17-web-1 | Performing system checks...
chapter17-web-1 |
chapter17-web-1 | System check identified no issues (0 silenced).
chapter17-web-1 | July 19, 2022 - 15:56:28
chapter17-web-1 | Django version 4.1, using settings 'educa.settings.prod'
chapter17-web-1 | Starting ASGI/Channels version 3.0.5 development server at
http://0.0.0.0:8000/
chapter17-web-1 | Quit the server with CONTROL-C.
```

Docker-контейнер вашего проекта Django запущен!

Пройдите по URL-адресу <http://localhost:8000/admin/> в своем браузере. Вы должны увидеть форму для входа на встроенный в Django сайт администрирования. Она должна выглядеть так, как показано на рис. 17.2:



Рис. 17.2. Форма для входа на встроенный в Django сайт администрирования

Стили CSS не загружены. Вы используете DEBUG=False, поэтому шаблоны URL-адресов для раздачи статических файлов не включены в главный файл urls.py проекта. Напомним, что встроенный в Django сервер разработки не подходит для раздачи статических файлов. Позже в этой главе вы сконфигурируете сервер под раздачу статических файлов.

Если вы обратитесь к любому другому URL-адресу своего сайта, то получите ошибку HTTP 500, потому что база данных производственной среды еще не сконфигурирована.

Взгляните на настольное приложение Docker Desktop. Вы увидите следующие ниже контейнеры:

	NAME	IMAGE	STATUS	PORT(S)	STARTED	
□	chapter17 1 container	-	Running (1/1) -			⏪ ■ 🗑
□	web-1 a9d7ca5be970	chapter17_web	Running	8000	37 seconds ago	⏪ ■ 🗑

Рис. 17.3. Приложение Chapter17 и контейнер web-1 в настольном приложении Docker Desktop

Приложение Chapter17 платформы Docker функционирует и имеет один контейнер с именем web-1, который работает на порту 8000. Имя приложения платформы Docker генерируется динамически с использованием имени каталога, в котором находится Docker-файл Compose, в данном случае Chapter17.

Далее в приложение платформы Docker будут добавлены служба PostgreSQL и служба Redis.

## Конфигурирование службы PostgreSQL

В этой книге вы в основном использовали базу данных SQLite, которая отличается простотой использования и быстрой настройки, но для производственной среды понадобится более мощная база данных, такая как PostgreSQL, MySQL или Oracle. Вы научились устанавливать PostgreSQL в главе 3 «Расширение приложения для ведения блога». Вместо этого для производственной среды мы будем использовать образ PostgreSQL платформы Docker. Информация об официальном образе PostgreSQL платформы Docker находится по адресу [https://hub.docker.com/\\_/postgres](https://hub.docker.com/_/postgres).

Отредактируйте файл docker-compose.yml, добавив в него следующие ниже строки, выделенные жирным шрифтом:

```
services:
 db:
 image: postgres:14.5
 restart: always
 volumes:
 - ./data/db:/var/lib/postgresql/data
 environment:
 - POSTGRES_DB=postgres
 - POSTGRES_USER=postgres
 - POSTGRES_PASSWORD=postgres

 web:
 build: .
 command: python /code/educa/manage.py runserver 0.0.0.0:8000
```

```

restart: always
volumes:
- .:/code
ports:
- "8000:8000"
environment:
- DJANGO_SETTINGS_MODULE=educa.settings.prod
- POSTGRES_DB=postgres
- POSTGRES_USER=postgres
- POSTGRES_PASSWORD=postgres
depends_on:
- db

```

С помощью этих изменений определяется служба с именем db со следующими ниже подразделами:

- `image`: служба использует базовый образ `postgres` платформы Docker;
- `restart`: значение политики перезапуска задано равным `always`;
- `volumes`: каталог `./data/db` монтируется в каталог образа `/var/lib/postgresql/.data` для выгрузки базы данных, чтобы хранящиеся в базе данных данные сохранялись после остановки приложения платформы Docker. Он создаст локальный путь `data/db/`;
- `environment`: переменные `POSTGRES_DB` (имя базы данных), `POSTGRES_USER` и `POSTGRES_PASSWORD` используются с предустановленными значениями.

Определение веб-службы теперь содержит средовые переменные PostgreSQL для Django. Зависимость службы создается с помощью `depend_on`, чтобы веб-служба запускалась после службы базы данных. Такой подход будет гарантировать порядок инициализации контейнера, но не будет гарантировать, что база данных PostgreSQL будет полностью инициирована до запуска веб-сервера Django. В целях решения этой проблемы необходимо использовать скрипт, который будет ждать доступности хоста базы данных и его TCP-порта. Для управления инициализацией контейнера Docker рекомендуется использовать инструмент ожидания `wait-for-it`.

Скачайте Bash-скрипт `wait-for-it.sh` с URL-адреса <https://github.com/vish-nubob/wait-for-it/blob/master/wait-for-it.sh> и сохраните файл рядом с файлом `docker-compose.yml`. Затем отредактируйте файл `docker-compose.yml`, видоизменив определение веб-службы, как показано ниже. Новый исходный код выделен жирным шрифтом:

```

web:
 build: .
 command: ["/./wait-for-it.sh", "db:5432", "--",
 "python", "/code/educa/manage.py", "runserver",
 "0.0.0.0:8000"]
 restart: always
 volumes:

```

```
- ./code
environment:
- DJANGO_SETTINGS_MODULE=educa.settings.prod
- POSTGRES_DB=postgres
- POSTGRES_USER=postgres
- POSTGRES_PASSWORD=postgres
depends_on:
- db
```

В этом определении службы используется Bash-скрипт `wait-for-it.sh`, чтобы дожидаться готовности хоста базы данных и приема соединений через используемый для PostgreSQL по умолчанию порт 5432, перед тем как запускать встроенный в Django сервер разработки. Подробнее о порядке запуска служб в Compose можно почитать на странице <https://docs.docker.com/compose/startup-order/>.

Давайте отредактируем настройки Django. Отредактируйте файл `educa/settings/prod.py`, добавив в него следующий ниже исходный код, выделенный жирным шрифтом:

```
import os
from .base import *

DEBUG = False

ADMINS = [
 ('Antonio M', 'email@mydomain.com'),
]

ALLOWED_HOSTS = ['*']

DATABASES = {
 'default': {
 'ENGINE': 'django.db.backends.postgresql',
 'NAME': os.environ.get('POSTGRES_DB'),
 'USER': os.environ.get('POSTGRES_USER'),
 'PASSWORD': os.environ.get('POSTGRES_PASSWORD'),
 'HOST': 'db',
 'PORT': 5432,
 }
}
```

В файле производственных настроек используются следующие ниже настроечные параметры:

- **ENGINE**: используется бэкенд базы данных PostgreSQL;
- **NAME**, **USER** и **PASSWORD**: для извлечения средовых переменных `POSTGRES_DB` (имя базы данных), `POSTGRES_USER` и `POSTGRES_PASSWORD` используется ме-

тод `os.environ.get()`. Указанные средовые переменные были заданы в Docker-файле Compose;

- `HOST`: используется `db`, то есть хост-имя контейнера службы базы данных, определенной в Docker-файле Compose. Хост-имя контейнера по умолчанию соответствует ИД контейнера в Docker. Именно по этой причине используется хост-имя `db`;
- `PORT`: используется значение `5432`, то есть порт, который для PostgreSQL задан по умолчанию.

Остановите приложение платформы Docker из оболочки, нажав клавиши **Ctrl+C** либо кнопку остановки в настольном приложении Docker Desktop. Затем снова запустите Compose командой

```
docker compose up
```

Первое выполнение после добавления службы `db` в Docker-файл Compose займет больше времени, потому что системе PostgreSQL необходимо инициализировать базу данных. Результат будет содержать следующие ниже две строки:

```
chapter17-db-1 | database system is ready to accept connections
...
chapter17-web-1 | Starting ASGI/Channels version 3.0.5 development server at
http://0.0.0.0:8000/
```

И база данных PostgreSQL, и приложение Django готовы. Производственная база данных пуста, поэтому необходимо применить миграцию базы данных.

## Применение миграции базы данных и создание суперпользователя

Откройте еще одну оболочку в родительском каталоге, в котором находится файл `docker-compose.yml`, и выполните следующую ниже команду:

```
docker compose exec web python /code/educa/manage.py migrate
```

Команда `docker compose exec` позволяет выполнять команды в контейнере. Эта команда используется для исполнения команды управления `migrate` в контейнере `web` платформы Docker.

Наконец, приведенной далее командой создайте суперпользователя:

```
docker compose exec web python /code/educa/manage.py createsuperuser
```

К базе данных применены миграции, и вы создали суперпользователя. Теперь можно обратиться к URL-адресу `http://localhost:8000/admin/` с учет-

ными данными суперпользователя. CSS-стили по-прежнему не будут загружены, потому что раздача статических файлов еще не сконфигурирована.

Вы определили службы для раздачи Django и PostgreSQL с помощью инструмента Docker Compose. Далее вы добавите службу раздачи Redis в производственной среде.

## Конфигурирование службы Redis

Добавьте службу Redis в Docker-файл Compose. С этой целью вы будете использовать официальный образ Redis платформы Docker. Информация об официальном образе Redis платформы Docker находится по адресу [https://hub.docker.com/\\_/redis](https://hub.docker.com/_/redis).

Отредактируйте файл `docker-compose.yml`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
services:

 db:
 # ...

 cache:
 image: redis:7.0.4
 restart: always
 volumes:
 - ./data/cache:/data

 web:
 # ...
 depends_on:
 - db
 - cache
```

В приведенном выше исходном коде определяется служба `cache` со следующими ниже подразделами:

- `image`: служба использует базовый образ Redis платформы Docker;
- `restart`: значение политики перезапуска задано равным `always`;
- `volumes`: каталог `./data/cache` монтируется в каталог образа `/data`, куда будут сбрасываться любые результаты операции записи Redis. Он создаст локальный путь `data/cache/`.

В определении веб-службы добавляется служба `cache` в качестве зависимости, чтобы веб-служба запускалась после службы `cache`. Сервер Redis инициализируется быстро, поэтому в данном случае использовать инструмент ожидания `wait-for-it` не потребуется.

Отредактируйте файл `educa/settings/prod.py`, добавив в него следующие ниже строки:

```
REDIS_URL = 'redis://cache:6379'
CACHES['default']['LOCATION'] = REDIS_URL
CHANNEL_LAYERS['default']['CONFIG']['hosts'] = [REDIS_URL]
```

В приведенных выше настроеках используется хост-имя `cache`, которое автоматически генерируется инструментом Docker Compose с использованием имени службы `cache` и используемого хранилищем Redis порта 6379. Настроек параметр Django `CACHE` и используемый приложением Channels параметр `CHANNEL_LAYERS` видоизменяются, чтобы применять производственный URL-адрес Redis.

Остановите приложение платформы Docker из оболочки, нажав клавиши **Ctrl+C** либо кнопку остановки в настольном приложении Docker Desktop. Затем снова запустите Compose командой

```
docker compose up
```

Откройте настольное приложение Docker Desktop. Теперь вы должны увидеть приложение `chapter17` платформы Docker, в котором работает контейнер для каждой службы, определенной в Docker-файле Compose: `db`, `cache` и `web`:

	NAME	IMAGE	STATUS	PORT(S)	STARTED	
□	chapter17 3 containers	-	Running (3/3)	-		⏴ ■ ━
□	db-1 9d0b376f0547	postgres	Running	-	5 minutes ago	☒    ⏴ ■ ━
□	web-1 087abe150782	chapter17_web	Running	8000	5 minutes ago	☒    ⏴ ⏵ ■ ━
□	cache-1 0600a3fbf3cd	redis	Running	-	5 minutes ago	☒    ⏴ ■ ━

Рис. 17.4. Приложение `Chapter17` с контейнерами `db-1`, `web-1` и `cache-1` в настольном приложении Docker Desktop

Вы по-прежнему раздаете Django с помощью встроенного в Django сервера разработки, который не подходит для использования в производстве. Давайте заменим его веб-сервером Python на основе WSGI.

## Раздача Django через WSGI и NGINX

Первичной платформой развертывания Django является WSGI. Аббревиатура **WSGI** означает Web Server Gateway Interface, то есть интерфейс шлюза веб-сервера, и является стандартом для раздачи приложений Python в веб.

При создании нового проекта с помощью команды `startproject` Django создает файл `wsgi.py` внутри каталога проекта. Этот файл содержит вызы-

ваемое приложение WSGI, которое является точкой доступа к вашему приложению.

WSGI используется как для выполнения вашего проекта на сервере разработки Django, так и для развертывания вашего приложения на сервере по вашему выбору в производственной среде. Подробнее о WSGI можно узнать по адресу <https://wsgi.readthedocs.io/en/latest/>.

## Использование сервера приложений uWSGI

На протяжении всей этой книги вы использовали встроенный в Django сервер разработки, чтобы выполнять проекты в локальной среде. Однако вам нужен стандартный веб-сервер для развертывания приложения в производственной среде.

uWSGI – это чрезвычайно быстрый сервер приложений Python. Он взаимодействует с приложением Python, используя спецификацию WSGI. uWSGI транслирует веб-запросы в формат, который может обрабатываться вашим проектом Django.

Давайте сконфигурируем сервер приложений uWSGI для раздачи проекта Django. Вы уже добавили uwsgi==2.0.20 в файл requirements.txt проекта, поэтому uWSGI уже установлен в образе веб-службы платформы Docker.

Отредактируйте файл docker-compose.yml, видоизменив определение веб-службы, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
web:
 build: .
 command: ["./wait-for-it.sh", "db:5432", "--",
 "uwsgi", "--ini", "/code/config/uwsgi/uwsgi.ini"]
 restart: always
 volumes:
 - ./code
 environment:
 - DJANGO_SETTINGS_MODULE=educa.settings.prod
 - POSTGRES_DB=postgres
 - POSTGRES_USER=postgres
 - POSTGRES_PASSWORD=postgres
 depends_on:
 - db
 - cache
```

Проследите, чтобы раздел ports был удален. Сервер приложений uWSGI будет доступен через сокет, поэтому открывать порт в контейнере не потребуется.

Новый раздел command с командой для образа запускает uwsgi, передавая ему конфигурационный файл /code/config/uwsgi/uwsgi.ini. Давайте создадим конфигурационный файл для uWSGI.

## Конфигурирование сервера приложений uWSGI

Сервер приложений uWSGI позволяет определять конкретно-прикладную конфигурацию в файле `.ini`. Рядом с файлом `docker-compose.yml` создайте путь к файлу и сам файл `config/uwsgi/uwsgi.ini`. Исходя из того, что родительский каталог называется `Chapter17`, файловая структура должна выглядеть следующим образом:

```
Chapter17/
 config/
 uwsgi/
 uwsgi.ini
 Dockerfile
 docker-compose.yml
 educa/
 manage.py
 ...
 requirements.txt
```

Откройте файл `config/uwsgi/uwsgi.ini` и добавьте в него следующий ниже исходный код:

```
[uwsgi]
socket=/code/educa/uwsgi_app.sock
chdir = /code/educa/
module=educa.wsgi:application
master=true
chmod-socket=666
uid=www-data
gid=www-data
vacuum=true
```

В файле `uwsgi.ini` определены такие опции:

- `socket`: UNIX/TCP-сокет для привязки сервера;
- `chdir`: путь к каталогу проекта, чтобы сервер uWSGI переходил в этот каталог перед загрузкой приложения Python;
- `module`: подлежащий использованию модуль WSGI. Он устанавливается для вызываемого приложения, содержащегося в модуле `wsgi` проекта;
- `master`: активировать ведущий процесс;
- `chmod-socket`: права доступа к файлу сокета. В данном случае используется `666`, чтобы NGINX мог читать/писать сокет;
- `uid`: ИД пользователя процесса после его запуска;
- `gid`: ИД группы процесса после его запуска;
- `vacuum`: наличие значения `true` сообщает серверу uWSGI, что следует очистить все временные файлы или UNIX-сокеты, которые он создает.

Опция `socket` предназначена для связи с каким-либо сторонним маршрутизатором, например NGINX. Вы будете выполнять сервер приложений uWS-

GI, используя сокет, и вы сконфигурируете NGINX в качестве веб-сервера, который будет общаться с uWSGI через сокет.

Список доступных параметров сервера приложения uWSGI находится по адресу <https://uwsgi-docs.readthedocs.io/en/latest/Options.html>.

Теперь вы не сможете обращаться к своему экземпляру сервера приложений uWSGI из браузера, так как он работает через сокет. Давайте завершим настройку производственной среды.

## Использование веб-сервера NGINX

При раздаче веб-сайта обычно раздается динамический контент, но вам также необходимо раздавать статические файлы, такие как таблицы стилей CSS, файлы JavaScript и изображения. Хотя сервер приложений uWSGI способен раздавать статические файлы, он добавляет ненужные накладные расходы к HTTP-запросам, и поэтому рекомендуется перед ним настраивать веб-сервер, такой как NGINX.

NGINX – это веб-сервер, ориентированный на высокую конкурентность, производительность и низкое использование памяти. NGINX также действует как обратный прокси-сервер, получая HTTP- и WebSocket-запросы и маршрутизируя их в разные бэкенды.

Как правило, вы будете использовать веб-сервер, такой как NGINX, перед сервером приложений uWSGI, чтобы эффективно раздавать статические файлы, и будете перенаправлять динамические запросы работникам сервера uWSGI. С помощью NGINX также можно применять различные правила и пользоваться выгодами от возможностей обратного прокси-сервера.

Мы добавим службу NGINX в Docker-файл Compose, используя официальный образ NGINX платформы Docker. Информация об официальном образе NGINX платформы Docker находится по адресу [https://hub.docker.com/\\_/nginx](https://hub.docker.com/_/nginx).

Отредактируйте файл `docker-compose.yml`, добавив в него следующие ниже строки, выделенные жирным шрифтом:

```
services:

 db:
 # ...

 cache:
 # ...

 web:
 # ...

 nginx:
 image: nginx:1.23.1
 restart: always
 volumes:
 - ./config/nginx:/etc/nginx/templates
```

```
- ./code
ports:
- "80:80"
```

Здесь было добавлено определение службы nginx со следующими ниже подразделами:

- `image`: служба использует базовый образ nginx платформы Docker;
- `restart`: значение политики перезапуска задано равным `always`;
- `volumes`: том `./config/nginx` монтируется в каталог `/etc/nginx/templates` образа платформы Docker. Здесь веб-сервер NGINX будет искать применяемый по умолчанию шаблон конфигурации. Кроме того, локальный каталог `.` монтируется в каталог `/code` образа, чтобы обеспечить веб-серверу NGINX доступ к статическим файлам;
- `ports`: предоставляется порт 80, который соотносится с портом 80 контейнера. Этот порт используется для HTTP по умолчанию.

Давайте сконфигурируем веб-сервер NGINX.

## Конфигурирование веб-сервера NGINX

В каталоге `config/` создайте следующий ниже путь к файлу и сам файл, выделенные жирным шрифтом:

```
config/
 uwsgi/
 uwsgi.ini
 nginx/
 default.conf.template
```

Откройте файл `nginx/default.conf.template` и добавьте в него следующий ниже исходный код<sup>1</sup>:

```
входной поток для uWSGI
upstream uwsgi_app {
 server unix:/code/educa/uwsgi_app.sock;
}

server {
 listen 80;
 server_name www.educaproject.com educaproject.com;
 error_log stderr warn;
 access_log /dev/stdout main;
```

<sup>1</sup> В терминологии NGINX входной поток (`upstream`) – это место, откуда данные при текают (например, HTTP-запрос), а выходной поток (`downstream`) – место, куда они попадают (например, обслуживающая запрос базовая система). – *Прим. перев.*

```

location / {
 include /etc/nginx/uwsgi_params;
 uwsgi_pass uwsgi_app;
}

```

Это базовая конфигурация веб-сервера NGINX. В данной конфигурации настраивается вышестоящее приложение с именем `uwsgi_app`, которое указывает на сокет, созданный сервером приложений uWSGI. Блок `server` используется со следующей ниже конфигурацией:

- веб-серверу NGINX указывается, что нужно прослушивать порт 80;
- имя сервера задается равным [www.educaproject.com](http://www.educaproject.com) и [educaproject.com](http://educaproject.com). NGINX будет раздавать входящие запросы для обоих доменов;
- для директивы `error_log` используется `stderr`, чтобы журналы ошибок записывались в стандартный файл ошибок. Второй параметр определяет уровень журналирования. Для получения предупреждений и ошибок более высокого уровня серьезности используется `warn`;
- `access_log` указывает на стандартный вывод с помощью `/dev/stdout`;
- указывается, что любой запрос в рамках пути `/` должен маршрутизоваться в сокет `uwsgi_app` к серверу приложений uWSGI;
- включаются заранее заданные параметры конфигурации uWSGI, поставляемые с NGINX. Они расположены в `/etc/nginx/uwsgi_params`.

Теперь NGINX настроен. Документация по NGINX находится по адресу <https://nginx.org/en/docs/>.

Остановите приложение платформы Docker из оболочки, нажав клавиши **Ctrl+C** либо с помощью кнопки остановки в настольном приложении Docker Desktop. Затем снова запустите Compose командой

```
docker compose up
```

Пройдите по URL-адресу `http://localhost/` в своем браузере. Добавлять порт к URL-адресу не требуется, потому что вы обращаетесь к хосту через стандартный HTTP-порт 80. Вы должны увидеть страницу списка курсов без стилей CSS, как на рис. 17.5.

На рис. 17.6 показан цикл запроса/ответа в производственной среде, которую вы настроили.

Когда клиентский браузер отправляет HTTP-запрос, происходит следующее.

1. Веб-сервер NGINX получает HTTP-запрос.

[Educa](#)

- [Sign in](#)

## All courses

### Subjects

- [All](#)

Рис. 17.5. Страница списка курсов, раздаваемая веб-сервером NGINX и сервером приложений uWSGI

2. Веб-сервер NGINX делегирует запрос серверу приложений uWSGI через сокет.
3. Сервер приложений uWSGI передает запрос Django на обработку.
4. Django возвращает HTTP-ответ, который передается обратно в веб-сервер NGINX, который, в свою очередь, передает его браузеру клиента.



Рис. 17.6. Цикл запроса/ответа производственной среды

Если вы сверитесь с настольным приложением Docker Desktop, то увидите, что запущено 4 контейнера:

- служба db, работающая под управлением базы данных PostgreSQL;
- служба cache, работающая под управлением хранилища Redis;
- служба web, работающая под управлением сервера приложений uWSGI + Django;
- служба nginx, работающая под управлением веб-сервера NGINX.

Давайте продолжим настройку производственной среды. Вместо доступа к проекту, используя `localhost`, мы сконфигурируем проект под использование хост-имени `educaproject.com`.

## Использование хост-имени

Для сайта будет применяться хост-имя `educaproject.com`. Поскольку вы используете пробное доменное имя, необходимо перенаправлять его на локальный хост.

Если вы используете Linux или macOS, то отредактируйте файл `/etc/hosts`, добавив в него следующую ниже строку:

```
127.0.0.1 educaproject.com www.educaproject.com
```

Если же вы используете Windows, то отредактируйте файл `C:\Windows\System32\drivers\etc`, добавив ту же строку.

Поступая таким образом, вы будете маршрутизировать хост-имена `educaproject.com` и `www.educaproject.com` на свой локальный сервер. На производственном сервере этого делать не нужно, так как у вас будет фиксированный IP-адрес и вы укажете свое хост-имя на свой сервер в DNS-конфигурации вашего домена.

Пройдите по URL-адресу <http://educaproject.com/> в своем браузере. Вы должны увидеть свой сайт без загруженных статических ресурсов. Ваша производственная среда почти готова.

Теперь вы можете ограничить хосты, которые могут раздавать ваш проект Django. Отредактируйте файл производственных настроек `educa/settings/prod.py` проекта, изменив настроечный параметр `ALLOWED_HOSTS` следующим образом:

```
ALLOWED_HOSTS = ['educaproject.com', 'www.educaproject.com']
```

Django будет раздавать ваше приложение только в том случае, если оно работает под любым из этих хост-имен. Подробнее о настроичном параметре `ALLOWED_HOSTS` можно узнать на странице <https://docs.djangoproject.com/en/4.1/ref/settings/#allowed-hosts>.

Производственная среда почти готова. Давайте продолжим и сконфигурируем веб-сервер NGINX под раздачу статических файлов.

## Раздача статических и мультимедийных ресурсов

Сервер приложений uWSGI способен безупречно раздавать статические файлы, но не так быстро и эффективно, как веб-сервер NGINX. В целях достижения наилучшей производительности раздачи статических файлов в производственной среде вы будете использовать NGINX. Вы настроите NGINX для раздачи как статических файлов приложения (таблиц стилей CSS, файлов JavaScript и изображений), так и мультимедийных файлов, закачанных преподавателями в качестве содержимого курсов.

Отредактируйте файл `settings/base.py`, добавив следующую ниже строку сразу после настроичного параметра `STATIC_URL`:

```
STATIC_ROOT = BASE_DIR / 'static'
```

Это корневой каталог всех статических файлов проекта. Далее вы собираетесь собрать статические файлы из разных приложений Django в общий каталог.

### Сбор статических файлов

Каждое приложение в проекте Django может содержать статические файлы в каталоге `static/`. Django предоставляет команду для сбора статических файлов из всех приложений в одном месте. За счет этого упрощается конфигурирование под раздачу статических файлов в производственной среде. Команда `collectstatic` собирает статические файлы из всех приложений проекта, размещая их по пути, определенному настроичным параметром `STATIC_ROOT`.

Остановите приложение платформы Docker из оболочки, нажав клавиши `Ctrl+C` либо кнопку остановки в настольном приложении Docker Desktop. Затем снова запустите Compose командой

```
docker compose up
```

Откройте еще одну оболочку в родительском каталоге, в котором находится файл docker-compose.yml, и выполните следующую ниже команду:

```
docker compose exec web python /code/educa/manage.py collectstatic
```

Обратите внимание, что помимо этого можно выполнить такую команду в оболочке из каталога educa/project:

```
python manage.py collectstatic --settings=educa.settings.local
```

Обе команды будут иметь одинаковый эффект, поскольку базовый локальный каталог монтируется в Docker-образ. Django спросит, не хотите ли вы переопределить какие-либо существующие файлы в корневом каталоге. Наберите yes и нажмите **Enter**. Вы увидите следующий результат:

```
171 static files copied to '/code/educa/static'.
```

Файлы, расположенные в каталоге static/ каждого приложения, присутствующего в настроичном параметре INSTALLED\_APPS, были скопированы в глобальный каталог /educa/static/ проекта.

## **Раздача статических файлов с помощью веб-сервера NGINX**

Отредактируйте файл config/nginx/default.conf.template, добавив в блок server следующие ниже строки, выделенные жирным шрифтом:

```
server {
...

location / {
 include /etc/nginx/uwsgi_params;
 uwsgi_pass uwsgi_app;
}

location /static/ {
 alias /code/educa/static/;
}
location /media/ {
 alias /code/educa/media/;
}
}
```

Эти директивы сообщают веб-серверу NGINX, что статические файлы, расположенные по путям /static/ и /media/, нужно раздавать напрямую.

Вот описание этих путей:

- /static/: соответствует пути настроичного параметра STATIC\_URL. Целевой путь соответствует значению параметра STATIC\_ROOT. Он использу-

ется для раздачи статических файлов приложения из каталога, смонтированного в образ NGINX платформы Docker;

- `/media/`: соответствует путям настроек параметра `MEDIA_URL`, а его цепевой путь соответствует значению параметра `MEDIA_ROOT`. Он используется для раздачи медиафайлов, закачанных в содержимое курсов из каталога, смонтированного в образ NGINX платформы Docker.

Схема производственной среды теперь выглядит так:

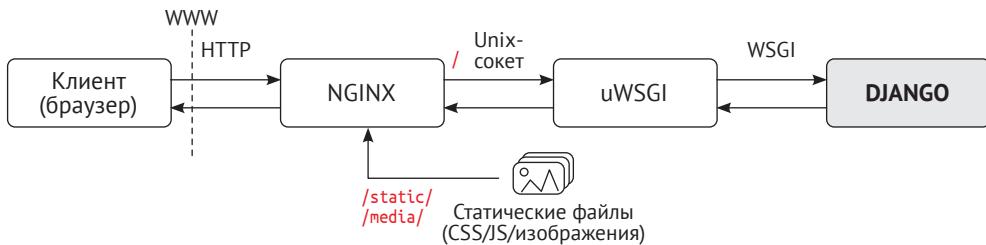


Рис. 17.7. Цикл запроса/ответа производственной среды, включающий статические файлы

Файлы по путям `/static/` и `/media/` теперь раздаются веб-сервером NGINX напрямую, без перенаправления на сервер приложений uWSGI. Запросы к любому другому пути по-прежнему передаются веб-сервером NGINX на сервер приложений uWSGI через UNIX-сокет.

Остановите приложение Docker из оболочки, нажав клавиши **Ctrl+C** либо с помощью кнопки остановки в настольном приложении Docker Desktop. Затем снова запустите Compose командой:

```
docker compose up
```

Пройдите по URL-адресу <http://educaproject.com/> в своем браузере. Вы должны увидеть следующий ниже экран:

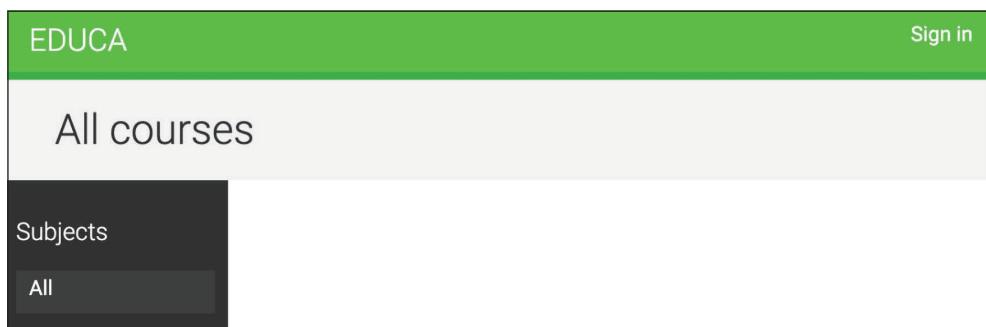


Рис. 17.8. Страница списка курсов, раздаваемая веб-сервером NGINX и сервером приложений uWSGI

Статические ресурсы, такие как таблицы стилей CSS и изображения, теперь загружаются правильно. HTTP-запросы к статическим файлам раздаются непосредственно веб-сервером NGINX, не перенаправляя их на сервер приложений uWSGI.

Вы успешно сконфигурировали веб-сервер NGINX под раздачу статических файлов. Далее вы собираетесь проверить свой проект Django, чтобы развернуть его в производственной среде, и вы начнете раздавать свой сайт по протоколу HTTPS.

## Обеспечение защиты сайта с помощью SSL/TLS

Криптографический протокол защищенного соединения транспортного слоя (**TLS**) является стандартом для раздачи веб-сайтов через защищенное соединение. Предшественником TLS является криптографический протокол слоя защищенных сокетов (**SSL**). Хотя SSL в настоящее время устарел, во многих библиотеках и онлайновой документации можно найти ссылки как на TLS, так и на SSL. Настоятельно рекомендуется раздавать свои веб-сайты через HTTPS.

В этом разделе вы собираетесь проверить свой проект Django на предмет готовности к развертыванию в производстве и подготовить проект к раздаче через HTTPS. Затем вы сконфигурируете SSL/TLS-сертификат в веб-сервере NGINX, чтобы безопасно раздавать свой сайт.

## Проверка готовности проекта к работе в производственной среде

Django содержит фреймворк проверки системы, служащий для валидации проекта в любое время. Фреймворк проверки инспектирует приложения, установленные в проекте Django, и обнаруживает распространенные проблемы. Проверки запускаются неявно при выполнении таких команд управления, как `runserver` и `migrate`. Однако существует возможность инициировать проверки в явной форме командой управления `check`.

Подробнее о встроенным в Django фреймворке проверки системы можно узнать по адресу <https://docs.djangoproject.com/en/4.1/topics/checks/>.

Давайте получим подтверждение, что фреймворк проверки не сигнализирует о каких-либо проблемах в вашем проекте. Откройте оболочку в каталоге проекта `educa` и выполните следующую ниже команду, чтобы проверить свой проект:

```
python manage.py check --settings=educa.settings.prod
```

Вы увидите такой результат:

```
System check identified no issues (0 silenced).
```

Фреймворк проверки системы не выявил никаких проблем. Если применить опцию `--deploy`, то фреймворк проверки системы выполнит дополнительные проверки, необходимые для производственного развертывания.

Выполните следующую ниже команду из каталога проекта `educa`:

```
python manage.py check --deploy --settings=educa.settings.prod
```

Вы увидите результат, подобный приведенному ниже:

```
System check identified some issues:
```

**WARNINGS:**

```
(security.W004) You have not set a value for the SECURE_HSTS_SECONDS setting.
...
(security.W008) Your SECURE_SSL_REDIRECT setting is not set to True...
(security.W009) Your SECRET_KEY has less than 50 characters, less than 5 unique
characters, or it's prefixed with 'django-insecure-'...
(security.W012) SESSION_COOKIE_SECURE is not set to True. ...
(security.W016) You have 'django.middleware.csrf.CsrfViewMiddleware' in your
MIDDLEWARE, but you have not set CSRF_COOKIE_SECURE ...
```

```
System check identified 5 issues (0 silenced).
```

Фреймворк проверки выявил пять проблем (0 ошибок, 5 предупреждений). Все предупреждения связаны с настройками обеспечения безопасности.

Давайте исправим проблему `security.W009`. Откройте файл `educa/settings/base.py` и измените настроечный параметр `SECRET_KEY`, удалив префикс `django-insecure-` и добавив дополнительные случайные символы, чтобы генерировать строковый литерал длиной не менее 50 символов.

Выполните команду `check` еще раз и убедитесь, что проблема `security.W009` больше не возникает. Остальные предупреждения связаны с конфигурацией SSL/TLS. Мы рассмотрим их далее.

## Конфигурирование проекта Django под SSL/TLS

Django поставляется со специальными настроечными параметрами, предназначеными для поддержки SSL/TLS. Вы собираетесь отредактировать производственные настройки, чтобы раздавать свой сайт по протоколу HTTPS.

Отредактируйте файл настроек `educa/settings/prod.py`, добавив в него следующие ниже настроечные параметры:

```
Безопасность
CSRF_COOKIE_SECURE = True
SESSION_COOKIE_SECURE = True
SECURE_SSL_REDIRECT = True
```

Ниже приводится описание этих параметров:

- `CSRF_COOKIE_SECURE`: использовать безопасный cookie-файл с целью защиты от подделки межсайтовых запросов (**CSRF**). При наличии значения `True` браузеры будут передавать cookie-файлы только по протоколу HTTPS;
- `SESSION_COOKIE_SECURE`: использовать безопасный сеансовый cookie-файл. При наличии значения `True` браузеры будут передавать cookie-файлы только по протоколу HTTPS;
- `SECURE_SSL_REDIRECT`: должны ли HTTP-запросы перенаправляться на HTTPS.

Теперь Django будет перенаправлять HTTP-запросы на HTTPS; сеансовые cookie-файлы и cookie-файлы CSRF будут отправляться только через HTTPS.

Выполните следующую ниже команду из главного каталога проекта:

```
python manage.py check --deploy --settings=educa.settings.prod
```

Остается только одно предупреждение, `security.W004`:

```
(security.W004) You have not set a value for the SECURE_HSTS_SECONDS setting.
...
```

Это предупреждение связано с политикой обеспечения строгой безопасности передачи информации по протоколу HTTP (**HSTS**)<sup>1</sup>. Политика HSTS не позволяет пользователям обходить предупреждения и подсоединяться к сайту с просроченным, самозаверенным или иным образом недействительным SSL-сертификатом. В следующем разделе мы будем использовать самозаверенный сертификат, поэтому сейчас проигнорируем это предупреждение. Если у вас есть настоящий домен, то можете подать заявку в доверяемый центр сертификации (CA) на выдачу для сайта SSL/TLS-сертификата, чтобы браузеры могли подтверждать его подлинность. В таком случае вы сможете указать значение `SECURE_HSTS_SECONDS` выше 0, то есть используемого по умолчанию значения. Подробнее о политике HSTS можно узнать по адресу <https://docs.djangoproject.com/en/4.1/ref/middleware/#http-strict-transport-security>.

Вы успешно устранили возникшие в рамках проверки остальные проблемы. Подробнее о контрольном списке развертывания Django можно почитать на странице <https://docs.djangoproject.com/en/4.1/howto/deployment/checklist/>.

<sup>1</sup> Англ. HTTP Strict Transport Security. – Прим. перев.

## Создание SSL/TLS-сертификата

Создайте новый каталог внутри каталога проекта `educa` и назовите его `ssl`. Затем следующей ниже командой сгенерируйте SSL/TLS-сертификат из командной строки:

```
openssl req -x509 -newkey rsa:2048 -sha256 -days 3650 -nodes \
-keyout ssl/educa.key -out ssl/educa.crt \
-subj '/CN=*.educaproject.com' \
-addext 'subjectAltName=DNS:*.educaproject.com'
```

Она сгенерирует приватный ключ и 2048-битный SSL/TLS-сертификат, действительный в течение 10 лет. Указанный сертификат выдан хост-имени `*.educaproject.com`. Это групповой сертификат; при использовании подстановочного знака `*` в доменном имени сертификат можно использовать для любого поддомена `educaproject.com`, например [www.educaproject.com](http://www.educaproject.com) или [djangoproject.educaproject.com](http://djangoproject.educaproject.com). После создания сертификата каталог `educa/ssl/` будет содержать два файла: `educa.key` (приватный ключ) и `educa.crt` (сертификат).

Для того чтобы использовать опцию `-addext`, понадобится криптографическая библиотека, как минимум OpenSSL 1.1.1 либо LibreSSL 3.1.0. Местоположение OpenSSL на своем компьютере можно проверить с помощью команды `openssl`, а версию – с помощью команды `openssl version`.

В качестве альтернативы можно использовать SSL/TLS-сертификат, предоставленный в исходном коде к этой главе. Данный сертификат находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/blob/main/Chapter17/educa/ssl/>. Обратите внимание, что необходимо сгенерировать приватный ключ и не использовать этот сертификат в производстве.

## Конфигурирование веб-сервера NGINX под использование SSL/TLS

Отредактируйте файл `docker-compose.yml`, добавив следующую ниже строку, выделенную жирным шрифтом:

```
services:
 # ...

 nginx:
 #...
 ports:
 - "80:80"
 - ""443:443"
```

Хост-контейнер NGINX будет доступен через порт 80 (HTTP) и порт 443 (HTTPS). Хост-порт 443 соотносится с контейнерным портом 443.

Откройте файл config/nginx/default.conf.template проекта educa и отредактируйте блок server, вставив SSL/TLS, как показано ниже:

```
server {
 listen 80;
 listen 443 ssl;
 ssl_certificate /code/educa/ssl/educa.crt;
 ssl_certificate_key /code/educa/ssl/educa.key;
 server_name www.educaproject.com educaproject.com;
 # ...
}
```

С помощью приведенного выше исходного кода теперь веб-сервер NGINX прослушивает HTTP через порт 80 и HTTPS через порт 443. Путь к SSL/TLS-сертификату указывается с помощью `ssl_certificate`, а ключ сертификата – с помощью `ssl_certificate_key`.

Остановите приложение платформы Docker из оболочки, нажав клавиши **Ctrl+C** либо кнопку остановки в настольном приложении Docker Desktop. Затем снова запустите Compose командой

```
docker compose up
```

Пройдите по URL-адресу <https://educaproject.com/> в своем браузере. Вы должны увидеть предупреждающее сообщение, похожее на следующее ниже:



Рис. 17.9. Предупреждение о недействительном сертификате

Этот экран может отличаться в зависимости от браузера. Он предупреждает о том, что ваш сайт не использует ни доверяющий, ни действительный сертификат; браузер не может верифицировать подлинность сайта. Это вызвано тем, что вы заверили свой сертификат сами, а не получили его от доверяющего центра сертификации (CA). Если у вас есть настоящий домен, то вы можете подать заявку в доверяющий центр сертификации на выдачу для него SSL/TLS-сертификата, чтобы браузеры могли подтверждать его подлинность. Если вы хотите получить доверяющий сертификат для настоящего домена, то можете обратиться в проект Let's Encrypt, созданный фондом Linux Foundation. Это некоммерческий центр сертификации, который упрощает бесплатное получение и обновление доверяющих SSL/TLS-сертификатов. Дополнительная информация находится по адресу <https://letsencrypt.org>.

Кликните по ссылке либо кнопке, предоставляющей дополнительную информацию, и выберите посещение веб-сайта, игнорируя предупреждения. Браузер может предложить добавить для этого сертификата исключение или подтвердить, что вы ему доверяете. Если вы используете Chrome, то, возможно, не увидите никаких вариантов перехода на веб-сайт. Если это так, то наберите `thisisunsafe` и нажмите **Enter** прямо в Chrome на странице предупреждения. Затем Chrome загрузит веб-сайт. Обратите внимание, что вы делаете это с собственноручно выпущенным сертификатом; не доверяйте никакому неизвестному сертификату и не обходите браузерную проверку SSL/TLS-сертификата других доменов.

При входе на сайт браузер отобразит значок замка рядом с URL-адресом, как показано на рис. 17.10:



Рис. 17.10. Адресная строка браузера, включающая значок замка безопасного соединения

Другие браузеры могут отображать предупреждение о том, что сертификат не является доверяющим, как показано на рис. 17.11:



Рис. 17.11. Адресная строка браузера с предупреждающим сообщением

Если кликнуть по значку замка или значку предупреждения, то детальная информация о SSL/TLS-сертификате будет отображена следующим образом:

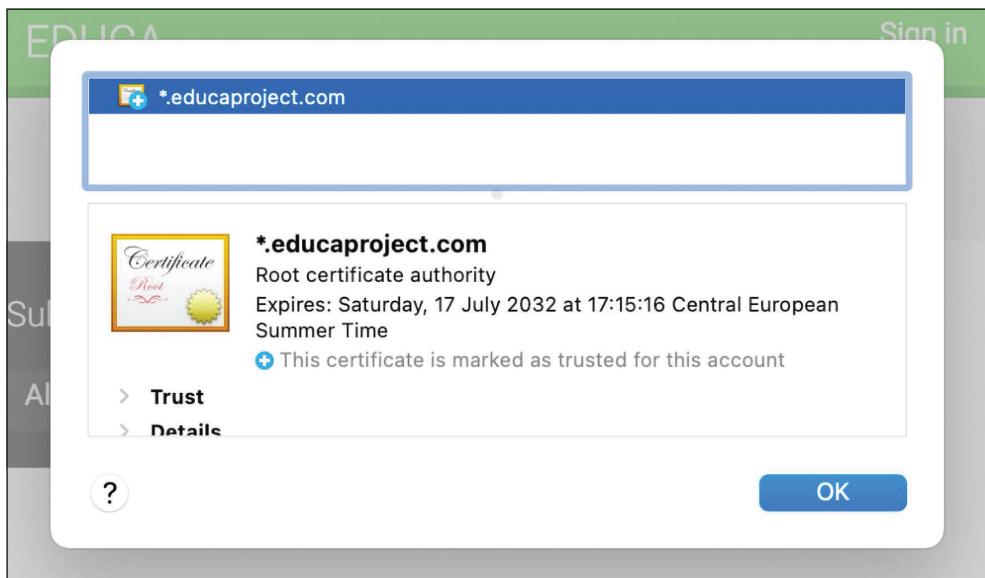


Рис. 17.12. Детальная информация о TLS/SSL-сертификате

В детальной информации о сертификате будет показано, что это самозаверенный сертификат, и вы увидите дату истечения срока его действия. Ваш браузер может пометить сертификат как небезопасный, но вы используете его только для тестирования. Теперь вы безопасно раздаете свой сайт через HTTPS.

## Перенаправление HTTP-трафика на HTTPS

Перенаправление HTTP-запросов на HTTPS осуществляется с помощью Django, применяя настроочный параметр SECURE\_SSL\_REDIRECT. Любой запрос посредством `http://` перенаправляется на тот же URL-адрес с использованием `https://`. Однако это решается эффективнее при помощи веб-сервера NGINX.

Отредактируйте файл `config/nginx/default.conf.template`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
входной поток для uWSGI
upstream uwsgi_app {
 server unix:/code/educa/uwsgi_app.sock;
}

server {
 listen 80;
 server_name www.educaproject.com educaproject.com;
 return 301 https://$host$request_uri;
```

```
}

server {
 listen 443 ssl;
 ssl_certificate /code/educa/ssl/educa.crt;
 ssl_certificate_key /code/educa/ssl/educa.key;
 server_name www.educaproject.com educaproject.com;
 # ...
}
```

В приведенном выше исходном коде из изначального блока `server` удаляется директива `listen 80;`, чтобы платформа была доступна только через HTTPS (порт 443). Поверх изначального блока `server` добавляется дополнительный блок `server`, который прослушивает только порт 80 и перенаправляет все HTTP-запросы на HTTPS. Для этого возвращается код 301 HTTP-ответа (постоянное перенаправление), который переадресует на `https://`-версию запрошенного URL-адреса с использованием переменных `$host` и `$request_uri`.

Откройте оболочку в родительском каталоге, в котором находится файл `docker-compose.yml`, и выполните следующую ниже команду, чтобы перезагрузить веб-сервер NGINX:

```
docker compose exec nginx nginx -s reload
```

Она выполняет команду `nginx -s reload` в контейнере `nginx`. Теперь вы перенаправляете весь HTTP-трафик на HTTPS с помощью веб-сервера NGINX.

Сейчас ваша среда защищена с помощью TLS/SSL. В целях завершения настройки производственной среды еще необходимо настроить асинхронный веб-сервер для приложения-обертки Django Channels.

## Использование Daphne для приложения Django Channels

В главе 16 «Разработка чат-сервера» вы использовали приложение-обертку Django Channels для создания чат-сервера с помощью веб-сокетов. Сервер приложений uWSGI подходит для управления приложениями Django или любыми другими WSGI-приложениями, но не поддерживает асинхронную связь с использованием асинхронного интерфейса серверного шлюза (**ASGI**)<sup>1</sup> или веб-сокетов. В целях обеспечения работы приложения Channels в производственной среде нужен веб-сервер на основе ASGI, способный управлять веб-сокетами.

Daphne – это HTTP-, HTTP2- и WebSocket-сервер на основе ASGI, разработанный для раздачи каналов. Веб-сервер Daphne можно выполнять вместе

<sup>1</sup> Англ. Asynchronous Server Gateway Interface. – Прим. перев.

с сервером приложений uWSGI, чтобы эффективно раздавать ASGI- и WSGI-приложения. Более подробная информация об асинхронном веб-сервере Daphne находится по адресу <https://github.com/django/daphne>.

Вы уже добавили daphne==3.0.2 в файл requirements.txt проекта. Давайте создадим новую службу в Docker-файле Compose, чтобы запускать веб-сервер Daphne.

Отредактируйте файл docker-compose.yml, добавив следующие ниже строки:

```
daphne:
 build: .
 working_dir: /code/educa/
 command: ["/../.wait-for-it.sh", "db:5432", "--",
 "daphne", "-u", "/code/educa/daphne.sock",
 "educa.asgi:application"]
 restart: always
 volumes:
 - .:/code
 environment:
 - DJANGO_SETTINGS_MODULE=educa.settings.prod
 - POSTGRES_DB=postgres
 - POSTGRES_USER=postgres
 - POSTGRES_PASSWORD=postgres
 depends_on:
 - db
 - cache
```

Определение службы daphne очень похоже на веб-службу. Образ службы daphne также создается с помощью Dockerfile, который ранее был создан для веб-службы. Главные отличия в следующем:

- work\_dir меняет рабочий каталог образа на /code/educa/;
- command запускает приложение educa.asgi:application, определенное в файле educa/asgi.py, в котором daphne использует UNIX-сокет. В ней также применяется Bash-скрипт wait-for-it для ожидания готовности базы данных PostgreSQL перед инициализацией веб-сервера.

Поскольку Django используется вами в производстве, при получении HTTP-запросов Django проверяет настроенный параметр ALLOWED\_HOSTS. Мы выполним реализацию той же проверки для WebSocket-соединений.

Отредактируйте файл educa/asgi.py проекта, добавив следующие ниже строки, выделенные жирным шрифтом:

```
import os

from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from channels.auth import AuthMiddlewareStack
```

```
import chat.routing

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'educa.settings')

django_asgi_app = get_asgi_application()

application = ProtocolTypeRouter({
 'http': django_asgi_app,
 'websocket': AllowedHostsOriginValidator(
 AuthMiddlewareStack(
 URLRouter(chat.routing.websocket_urlpatterns)
)
),
})
```

Теперь конфигурация приложения Channels готова к работе.

## Использование безопасных соединений для веб-сокетов

Вы сконфигурировали веб-сервер NGINX под использование безопасных соединений с SSL/TLS. Вам необходимо поменять WebSocket-соединения (`ws`), чтобы использовать протокол `wss` (WebSocket Secure) точно так же, как HTTP-соединения теперь обслуживаются через HTTPS.

Откройте в редакторе шаблон `chat/room.html` приложения `chat` и в блоке `domready` найдите следующую ниже строку:

```
const url = 'ws://' + window.location.host +
```

Замените эту строку такой:

```
const url = 'wss://' + window.location.host +
```

Используя `wss://` вместо `ws://`, вы в явной форме подсоединяетесь к защищенному веб-сокету.

## Включение веб-сервера Daphne в конфигурацию веб-сервера NGINX

В своей производственной среде вы будете выполнять веб-сервер Daphne на UNIX-сокете и использовать перед ним веб-сервер NGINX. NGINX будет передавать запросы веб-серверу Daphne на основе запрошенного пути. Вы

будете выставлять асинхронный веб-сервер Daphne веб-серверу NGINX через интерфейс UNIX-сокета, как и при настройке сервера приложений uwsgi.

Отредактируйте файл config/nginx/default.conf.template, придав ему следующий вид:

```
входной поток для uwsgi
upstream uwsgi_app {
 server unix:/code/educa/uwsgi_app.sock;
}

входной поток для Daphne
upstream daphne {
 server unix:/code/educa/daphne.sock;
}

server {
 listen 80;
 server_name www.educaproject.com educaproject.com;
 return 301 https://$host$request_uri;

 server {
 listen 443 ssl;
 ssl_certificate /code/educa/ssl/educa.crt;
 ssl_certificate_key /code/educa/ssl/educa.key;
 server_name www.educaproject.com educaproject.com;
 error_log stderr warn;
 access_log /dev/stdout main;

 location / {
 include /etc/nginx/uwsgi_params;
 uwsgi_pass uwsgi_app;
 }

 location /ws/ {
 proxy_http_version 1.1;
 proxy_set_header Upgrade $http_upgrade;
 proxy_set_header Connection "upgrade";
 proxy_redirect off;
 proxy_pass http://daphne;
 }

 location /static/ {
 alias /code/educa/static/;
 }
 location /media/ {
 alias /code/educa/media/;
 }
 }
}
```

В этой конфигурации настраивается новый вышестоящий поток с именем `daphne`, который указывает на UNIX-сокет, созданный веб-сервером Daphne. В блоке `server` конфигурируется местоположение `/ws/`, чтобы переадресовывать запросы веб-серверу Daphne. Директива `proxy_pass` используется для передачи запросов веб-серверу Daphne, при этом включаются некоторые дополнительные прокси-директивы.

При такой конфигурации веб-сервер NGINX будет передавать любой URL-запрос, начинающийся с префикса `/ws/`, асинхронному веб-серверу Daphne, а остальные – серверу приложений uWSGI, за исключением файлов с путями `/static/` или `/media/`, которые будут обслуживаться непосредственно веб-сервером NGINX.

Производственная среда, включая веб-сервер Daphne, теперь выглядит так:



Рис. 17.13. Цикл запроса/ответа производственной среды, включающей сервер Daphne

Веб-сервер NGINX работает перед сервером приложений uWSGI и веб-сервером Daphne в качестве обратного прокси-сервера. NGINX выходит в интернет и передает запросы серверу приложений uWSGI или веб-серверу Daphne на основе их префикса пути. Помимо этого, NGINX также раздает статические файлы и перенаправляет незащищенные запросы на защищенные. Такая настройка сокращает время простоя, потребляет меньше серверных ресурсов и обеспечивает более высокую производительность и безопасность.

Остановите приложение Docker из оболочки, нажав клавиши **Ctrl+C** либо кнопку остановки в настольном приложении Docker Desktop. Затем снова запустите Compose командой

```
docker compose up
```

Используйте свой браузер, чтобы создать образец курса с пользователем-преподавателем, войдите на платформу с пользователем, который зачислен на курс, и пройдите по URL-адресу <https://educaproject.com/chat/room/1/> в сво-

ем браузере. Вы должны иметь возможность отправлять и получать сообщения, подобно следующему ниже примеру:

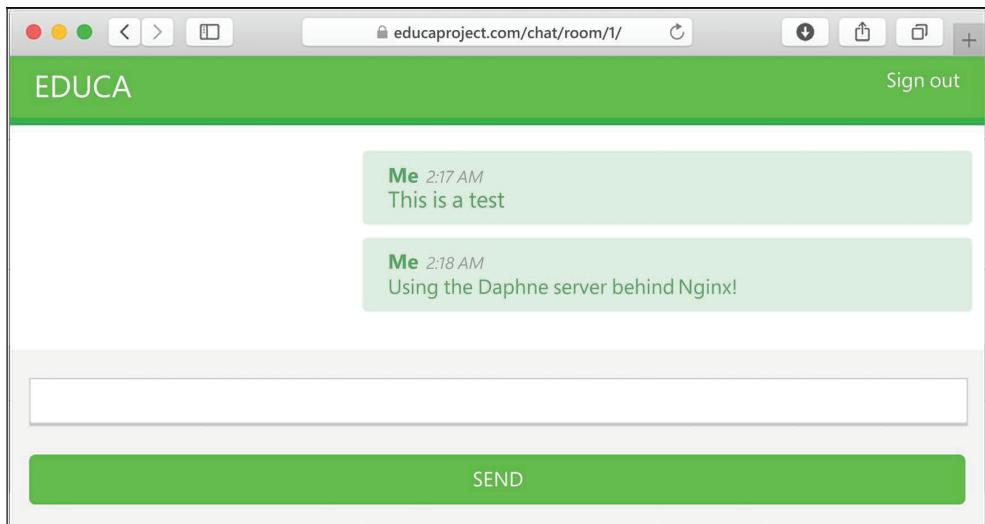


Рис. 17.14. Сообщения в чат-комнате курса, раздаваемые серверами NGINX и Daphne

Daphne работает правильно, и NGINX передает ему WebSocket-запросы. Все соединения защищены SSL/TLS.

Поздравляем! Вы создали конкретно-прикладной готовый к эксплуатации в производственной среде стек с использованием веб-сервера NGINX и серверов uWSGI и Daphne. Теперь можно заняться дальнейшей оптимизации с целью повышения производительности и безопасности, используя для этого настроочные параметры конфигурации в NGINX, uWSGI и Daphne. Тем не менее такая производственная среда представляет собой отличное начало!

Вы использовали инструмент Docker Compose для определения и запуска служб в нескольких контейнерах. Обратите внимание, что Docker Compose можно использовать как для локальных сред разработки, так и для производственных сред. Дополнительная информация об использовании инструмента Docker Compose в производственной среде находится по адресу <https://docs.docker.com/compose/production/>.

Для более сложных производственных сред потребуется распределять контейнеры динамически по варьирующемуся числу машин. Для этого вместо Docker Compose понадобится оркестровщик, такой как режим Docker Swarm или Kubernetes. Информация о режиме Docker Swarm находится по адресу <https://docs.docker.com/engine/swarm/>, а о Kubernetes – по адресу <https://kubernetes.io/docs/home/>.

# Создание конкретно-прикладных промежуточных программных компонентов

Вы уже знаете настроенный параметр MIDDLEWARE, который содержит промежуточные программные компоненты проекта. Их можно трактовать как низкоуровневую систему плагинов, позволяющую реализовывать перехватчики, которые исполняются в процессе запроса/ответа. Каждый промежуточный программный компонент системы отвечает за определенное действие, которое будет исполняться для всех HTTP-запросов или ответов.

Следует избегать добавления дорогостоящей обработки в промежуточные программные компоненты, поскольку они исполняются в каждом отдельном запросе.

При получении HTTP-запроса промежуточные программные компоненты исполняются в порядке их появления в настроичном параметре MIDDLEWARE. После того как Django сгенерировал HTTP-ответ, ответ пропускается через все промежуточные компоненты в обратном порядке.

Промежуточный программный компонент можно написать в виде функции следующим образом<sup>1</sup>:

```
def my_middleware(get_response):
 def middleware(request):
 # Исходный код, исполняемый для каждого
 # запроса до вызова представления
 # (и позже промежуточного компонента).
 response = get_response(request)
 # Исходный код, исполняемый для каждого
 # запроса/ответа после вызова представления.
 return response
 return middleware
```

Фабрика промежуточных программных компонентов – это вызываемый объект, который принимает вызываемый объект `get_response` и возвращает промежуточный компонент. Промежуточный программный компонент – это вызываемый объект, который принимает запрос и возвращает ответ, как и представление. Вызываемый объект `get_response` может быть следующим промежуточным компонентом в цепочке или фактическим представлением в случае последнего промежуточного компонента в списке.

Если какой-либо промежуточный компонент возвращает ответ, не вызывая его вызываемый объект `get_response`, то он замыкает процесс в себе; никакой

<sup>1</sup> Функция в таком виде, в сущности, представляет собой декоратор. – Прим. перев.

дополнительный промежуточный компонент не исполняется (и представление тоже), и ответ возвращается через те же слои, через которые прошел запрос.

Порядок следования промежуточных программных компонентов в настроичном параметре MIDDLEWARE очень важен, поскольку промежуточный компонент может зависеть от данных, устанавливаемых в запросе другим исполненным ранее промежуточным компонентом.

При добавлении нового промежуточного компонента в настроичный параметр MIDDLEWARE проверьте, чтобы он был помещен в правильное место. Промежуточный компонент исполняется в порядке появления в настроичном параметре в фазе запроса и в обратном порядке для ответов.

Дополнительная информация о промежуточных программных компонентах находится по адресу <https://docs.djangoproject.com/en/4.1/topics/http/middleware/>.

## Создание поддоменного промежуточного компонента

Вы создадите конкретно-прикладной промежуточный программный компонент, позволяющий обеспечивать доступ к курсам через конкретно-прикладной поддомен. Каждый URL-адрес детальной информации о курсе, который выглядит как <https://educaproject.com/course/django/>, также будет доступен через поддомен, в котором используется слаг курса, например <https://djangoproject.com/>. Для того чтобы обращаться к детальной информации о курсе, пользователи смогут использовать поддомен в качестве аббревиатуры. Любые запросы к поддоменам будут перенаправляться на каждый соответствующий URL-адрес детальной информации о курсе.

Промежуточный компонент может находиться в любом месте проекта. Однако рекомендуется создавать файл middleware.py в каталоге приложения.

Внутри каталога приложения courses создайте новый файл и назовите его middleware.py. Добавьте в него следующий ниже исходный код:

```
from django.urls import reverse
from django.shortcuts import get_object_or_404, redirect
from .models import Course

def subdomain_course_middleware(get_response):
 """
 Поддомены курсов
 """

 def middleware(request):
 host_parts = request.get_host().split('.')
 if len(host_parts) > 2 and host_parts[0] != 'www':
 # получить курс для заданного поддомена
 course = get_object_or_404(Course, slug=host_parts[0])
 course_url = reverse('course_detail',
 kwargs={'slug': course.slug})
 return redirect(course_url)
 else:
 return get_response(request)
```

```
args=[course.slug])
перенаправить текущий запрос к
представлению course_detail
url = '{}://{}'.format(request.scheme,
 '.').join(host_parts[1:]),
course_url)
return redirect(url)
response = get_response(request)
return response
return middleware
```

При получении HTTP-запроса выполняется следующая работа.

1. Берется используемое в запросе хост-имя и разбивается на части. Например, если пользователь обращается к mycourse.educaproject.com, то создается список ['mycourse', 'educaproject', 'com'].
2. Выполняется проверка наличия поддомена в хост-имени, проверяя наличие более двух генерированных разбивкой элементов. Если хост-имя содержит поддомен, и это не www, то делается попытка получить курс со слагом, предоставленным в поддомене.
3. Если курс не найден, то создается исключение HTTP 404. В противном случае браузер перенаправляется на URL-адрес детальной информации о курсе.

Отредактируйте файл settings/base.py проекта и добавьте 'courses.middleware.SubdomainCourseMiddleware' внизу списка настроичного параметра MIDDLEWARE, как показано ниже:

```
MIDDLEWARE = [
 # ...
 'courses.middleware.subdomain_course_middleware',
]
```

Промежуточный компонент теперь будет исполняться при каждом запросе.

Напомним, что хост-имена, которым разрешено раздавать проект Django, указаны в настроичном параметре ALLOWED\_HOSTS. Давайте изменим этот параметр, чтобы любой возможный поддомен домена educaproject.com мог раздавать приложение.

Отредактируйте файл educa/settings/prod.py, видоизменив настроичный параметр ALLOWED\_HOSTS следующим образом:

```
ALLOWED_HOSTS = ['.educaproject.com']
```

Значение, которое начинается с точки, используется как подстановочный знак поддомена; '.educaproject.com' будет сочетаться с educaproject.com и любым поддоменом этого домена, например course.educaproject.com и django.educaproject.com.

## Раздача нескольких поддоменов с помощью веб-сервера NGINX

Вам нужен веб-сервер NGINX, чтобы иметь возможность раздавать свой сайт с любым возможным поддоменом. Отредактируйте файл config/nginx/default.conf.template, заменив два вхождения строки

```
server_name www.educaproject.com educaproject.com;
```

следующей ниже

```
server_name *.educaproject.com educaproject.com;
```

При использовании звездочки это правило применяется ко всем поддоменам домена educaproject.com. Для того чтобы протестировать ваш промежуточный программный компонент локально, необходимо добавить в /etc/hosts любые поддомены, которые вы хотите протестировать. Для тестирования промежуточного компонента с объектом Course со слагом django добавьте следующую ниже строку в файл /etc/hosts:

```
127.0.0.1 django.educaproject.com
```

Остановите приложение Docker из оболочки, нажав клавиши **Ctrl+C** либо кнопку остановки в настольном приложении Docker Desktop. Затем снова запустите Compose командой

```
docker compose up
```

Далее пройдите по URL-адресу <https://django.educaproject.com/> в своем браузере. Промежуточный компонент найдет курс по поддомену и перенаправит браузер на <https://educaproject.com/course/django/>.

## Реализация конкретно-прикладных команд управления

Django предоставляет приложениям возможность регистрировать конкретно-прикладные команды управления, применяемые с утилитой manage.py. Например, в главе 11 «Добавление интернационализации в магазин» вы использовали команды управления makemessages и compilemessages, чтобы создавать и компилировать файлы перевода.

Команда управления состоит из модуля Python, содержащего класс Command, который наследует от django.core.management.base.BaseCommand или одного из его подклассов. При этом можно создавать простые команды или наделять

их возможностью принимать позиционные и опциональные аргументы в качестве входных данных.

Django ищет команды управления в каталоге `management/commands/` по каждому активному приложению в настроичном параметре `INSTALLED_APPS`. Каждый найденный модуль регистрируется как команда управления, названная по его имени.

Подробнее о конкретно-прикладных командах управления можно узнать по адресу <https://docs.djangoproject.com/en/4.1/howto/custom-management-commands/>.

Вы собираетесь создать конкретно-прикладную команду управления, чтобы напоминать студентам о необходимости записаться хотя бы на один курс. Команда будет отправлять напоминание по электронной почте пользователям, которые зарегистрированы дольше указанного периода и еще не зачислены ни на один курс.

Внутри каталога приложения `students` создайте следующую ниже файловую структуру:

```
management/
 __init__.py
 commands/
 __init__.py
 enroll_reminder.py
```

Откройте файл `enroll_reminder.py` и добавьте в него такой исходный 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
from django.utils import timezone

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 = timezone.now().today() - \
 datetime.timedelta(days=options['days'] or 0)
 users = User.objects.annotate(course_count=\

```

```

 Count('courses_joined'))\|
 .filter(course_count=0,
 date_joined__date__lte=date_joined)

for user in users:
 message = """Dear {},
 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 ваша команда `enroll_reminder`. Приведенный выше исходный код выполняет следующую работу:

- класс `Command` наследует от `BaseCommand`;
- вставляется атрибут `help`. Этот атрибут содержит краткое описание команды и печатается при выполнении команды `python manage.py help enroll_reminder`;
- используется метод `add_arguments()`, чтобы добавить именованный аргумент `--days`. Этот аргумент применяется для указания минимального числа дней, прошедших с момента регистрации пользователя, который не записался ни на один курс, чтобы начать получать напоминание;
- команда `handle()` содержит фактическую команду. Из командной строки извлекается атрибут `days`. Если он не задан, то используется 0, чтобы напоминание отправлялось всем пользователям, которые не записались на курс, независимо от даты регистрации пользователя. При этом применяется предоставляемая веб-фреймворком Django утилита `timezone`; она позволяет получать текущую дату с учетом часового пояса с помощью `timezone.now().date()`. (Часовой пояс своего проекта можно установить посредством настроичного параметра `TIME_ZONE`.) Далее извлекаются пользователи, которые были зарегистрированы более указанного числа дней и еще не зачислены ни на один курс. Это достигается путем аннотирования набора запросов `QuerySet` общим числом курсов, на которые зачислен каждый пользователь. Создается электронное письмо с напоминанием каждому пользователю и добавляется в список электронных писем. Наконец, электронные письма отправляются с помощью функции `send_mass_mail()`, которая оптимизирована под открытие одного SMTP-соединения для отправки всех электронных писем вместо открытия одного соединения для каждого отправляемого электронного письма.

Вы создали свою первую команду управления. Откройте оболочку и выполните свою команду:

```
docker compose exec web python /code/educa/manage.py \
enrollReminder --days=20 --settings=educa.settings.prod
```

Если у вас нет работающего локального SMTP-сервера, то можете пролистать главу 2 «Усовершенствование блога за счет продвинутых функциональностей», в которой излагалась тема конфигурирования настроек параметров SMTP для своего первого проекта Django. Кроме того, можно добавить следующий ниже настроек параметр в файл `settings.py`, чтобы Django выводил электронные письма в стандартный вывод во время разработки:

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

Django также содержит утилиту вызова команд управления с помощью Python. Команды управления можно выполнять из своего исходного кода следующим образом:

```
from django.core import management
management.call_command('enroll_reminder', days=20)
```

Поздравляем! Теперь вы можете создавать конкретно-прикладные команды управления для своих приложений.

## Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter17>.
- Обзор инструмента Docker Compose: <https://docs.docker.com/compose/>.
- Установка настольного приложения Docker Desktop: <https://docs.docker.com/compose/install/compose-desktop/>.
- Официальный образ Python платформы Docker: [https://hub.docker.com/\\_/python](https://hub.docker.com/_/python).
- Справочник по Docker-файлам: <https://docs.docker.com/engine/reference/builder/>.
- Файл `requirements.txt` к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/blob/main/Chapter17/requirements.txt>.
- Пример файла YAML: <https://yaml.org/>.
- Раздел `build` файла Dockerfile: <https://docs.docker.com/compose/compose-file/build/>.
- Политика перезапуска Docker: <https://docs.docker.com/config/containers/start-containers-automatically/>.
- Тома Docker: <https://docs.docker.com/storage/volumes/>.
- Спецификация инструмента Docker Compose: <https://docs.docker.com/compose/compose-file/>.
- Официальный образ PostgreSQL платформы Docker: [https://hub.docker.com/\\_/postgres](https://hub.docker.com/_/postgres).

- Bash-скрипт wait-for-it.sh для Docker: <https://github.com/vishnubob/wait-for-it/blob/master/wait-for-it.sh>.
- Порядок запуска служб в Compose: <https://docs.docker.com/compose/start-up-order/>.
- Официальный образ Redis платформы Docker: [https://hub.docker.com/\\_/redis](https://hub.docker.com/_/redis).
- Документация WSGI: <https://wsgi.readthedocs.io/en/latest/>.
- Список параметров сервера приложений uWSGI: <https://uwsgi-docs.readthedocs.io/en/latest/Options.html>.
- Официальный образ NGINX платформы Docker: [https://hub.docker.com/\\_/nginx](https://hub.docker.com/_/nginx).
- Документация веб-сервера NGINX: <https://nginx.org/en/docs/>.
- Настроочный параметр ALLOWED\_HOSTS: <https://docs.djangoproject.com/en/4.1/ref/settings/#allowed-hosts>.
- Встроенный в Django фреймворк проверки системы: <https://docs.djangoproject.com/en/4.1/topics/checks/>.
- Политика обеспечения строгой безопасности передачи информации по протоколу HTTP с Django: <https://docs.djangoproject.com/en/4.1/ref/middleware/#http-strict-transport-security>.
- Контрольный список развертывания Django: <https://docs.djangoproject.com/en/4.1/howto/deployment/checklist/>.
- Каталог самогенерированных SSL/TLS-сертификатов: <https://github.com/PacktPublishing/Django-4-by-example/blob/main/Chapter17/educa/ssl/>.
- Центр сертификации Let's Encrypt: <https://letsencrypt.org/>.
- Исходный код веб-сервера Daphne: <https://github.com/django/daphne>.
- Использование инструмента Docker Compose в производственной среде: <https://docs.docker.com/compose/production/>.
- Режим Docker Swarm: <https://docs.docker.com/engine/swarm/>.
- Kubernetes: <https://kubernetes.io/docs/home/>.
- Промежуточные программные компоненты Django: <https://docs.djangoproject.com/en/4.1/topics/http/middleware/>.
- Создание конкретно-прикладных команд управления: <https://docs.djangoproject.com/en/4.1/howto/custom-management-commands/>.

## Резюме

В этой главе вы сформировали производственную среду с помощью инструмента Docker Compose. Вы сконфигурировали веб-сервер NGINX, сервер приложений uWSGI и асинхронный веб-сервер Daphne под раздачу своего приложения в производственной среде. Вы защитили свою среду с помощью SSL/TLS. Вы также реализовали конкретно-прикладной промежуточный программный компонент и научились создавать конкретно-прикладные команды управления.

Вы добрались до конца этой книги. Поздравляем! Вы приобрели навыки, необходимые для создания успешных веб-приложений с помощью веб-фреймворка Django. Эта книга провела вас через процесс разработки практических проектов и интеграции Django с другими технологиями. Теперь вы готовы создать свой собственный проект Django, будь то простой прототип или масштабное веб-приложение.

Удачи в вашем следующем приключении с Django!

# Предметный указатель

## Символы

{% blocktrans %}, шаблонный тег, 546  
  \_\_iter\_\_(), метод, 409  
  \_\_str\_\_(), метод, 43  
{% trans %}, шаблонный тег, 546  
  \_\_unicode\_\_(), метод, 43

## А

API. См. *Интерфейс прикладного программирования*  
API кеша низкоуровневый, 665  
  применение, 665  
  работа, 665  
Asynchronous Server Gateway Interface (ASGI), 36

## С

Сервер разработки, обеспечение работы по HTTPS, 245  
Celery  
  добавление в проект Django, 436  
  использование с Django, 433  
  отслеживание с помощью Flower, 440  
  работа, 436  
  ссылка на справочные материалы, 434  
  установка, 434  
Channels  
  использование в цикле запроса/ответа, 710  
  реально-временной Django, 709  
  установка приложения, 712  
Compose, спецификация  
  ссылка на справочные материалы, 747  
connect(), 715  
CSRF. См. *Подделка межсайтовых запросов*  
CSV-файл. См. *Файл разделенных запятыми значений (CSV)*  
curl, ссылка на справочные материалы, 686

## Д

Daphne, 771  
  включение в конфигурацию веб-сервера NGINX, 773  
  использование для приложения Django Channels, 771  
  ссылка на справочные материалы, 772  
disconnect(), 715  
Django  
  архитектура, 34  
  доставка через веб-сервер NGINX, 754  
  доставка через сервер приложений WSGI, 754  
  интернационализация (i18n), 535  
  использование с Celery, 433  
  использование с RabbitMQ, 433  
  компоненты веб-фреймворка, 33  
  контрольный список развертывания, 766  
  ссылка на справочные материалы, 766  
  настроочный параметр ALLOWED\_HOSTS, ссылка на справочные материалы, 761  
  отправка электронных писем, 101  
  поддержка асинхронных представлений на основе классов, 710  
  ссылка на справочные материалы, 710  
  поддержка написания асинхронных представлений, ссылка на справочные материалы, 710  
  разработка форм, 98  
  рекомендация постов по электронной почте, 97  
  установка, 31  
  установка с помощью pip, 31  
  философия дизайна, ссылка на справочные материалы, 33  
  фреймворк проверки системы, ссылка на справочные материалы, 764  
  цикл запроса/ответа, 75  
Django 4  
  общий обзор, 33

- ссылка на справочные материалы, 32  
функциональные возможности, 32
- Django реально-временной  
на основе приложения-обертки  
Channels, 709
- django-braces  
использование примесных классов, 633  
ссылка на документацию, 641
- Django Channels, ссылка на справочные  
материалы, 706
- django.db.models, функции  
агрегирования, 142
- Django Debug Toolbar. См. *Меню  
отладочных инструментов Django*
- django-localflavor, использование для  
валидации полей форм, 573
- django-parler  
интеграция переводов моделей на сайт  
администрирования, 565  
перевод полей моделей, 563  
применение для перевода моделей, 562  
установка, 562
- Django Redisboard  
отслеживание сервера Redis, 676  
ссылка на справочные материалы, 676
- Django REST framework  
ссылка на справочные материалы, 700  
установка, 681
- django-taggit, ссылка на справочные  
материалы, 132
- django.urls, функции-утилиты, ссылка на  
справочные материалы, 81
- Docker, установка, 375
- Docker-образ Python, ссылка на  
справочные материалы, 744
- Docker-образ Redis, ссылка на справочные  
материалы, 753
- Docker Compose, 743  
добавление требующихся для Python  
пакетов, 745  
использование, 743  
конфигурирование службы  
PostgreSQL, 749  
конфигурирование службы Redis, 753  
применение миграций базы  
данных, 752  
создание суперпользователя, 752  
создание файла, 746  
создание файла Dockerfile, 744  
ссылка на справочные материалы, 743  
установка, 743
- Docker Desktop, ссылка на инструкцию по  
установке, 743
- Dockerfile, 744  
создание, 744  
ссылка на справочные материалы, 745
- Docker Swarm, режим, ссылка на  
справочные материалы, 776
- ## E
- easy-thumbnails  
создание миниатюр изображений, 309  
ссылка на справочные материалы, 312
- enum, ссылка на справочные  
материалы, 48
- exclude(), метод, применение для  
извлечения объектов, 64
- ## F
- Facebook  
использование для добавления  
социальной авторизации, 248  
ссылка на справочные материалы по  
порталу для разработчика, 280
- Fetch API на JavaScript, интерфейс, 312  
ссылка на справочные материалы, 313
- filter(), метод, применение для  
извлечения объектов, 63
- Flower  
использование для отслеживания  
Celery, 440  
ссылка на справочные материалы, 442
- Fluent Reader, ссылка на скачивание, 166
- ## G
- GET-запрос, 426
- get\_object\_or\_404, функция сокращенного  
доступа, применение, 68
- gettext, инструментарий, установка, 536
- Google, использование для добавления  
социальной авторизации, 268
- ## H
- HTML5 API перетаскивания, ссылка на  
справочные материалы, 632
- HTML5 Sortable, библиотека, ссылка на  
документацию, 633
- HTTP-аутентификация базовая, ссылка  
на справочные материалы, 703

## HTTP-запрос

- выполнение с помощью JavaScript, 317
- защита от CSRF, 315
- режимы, 319

## HTTP-трафик, перенаправление на

### HTTPS, 770

## HTTPS, выполнение сервера разработки по HTTPS, 245

## J

- JavaScript, добавление действий пользователей по подписке/отписке, 342
- JavaScript асинхронный и XML (AJAX), 312
- выполнение HTTP-запросов с помощью JavaScript, 317
  - добавление асинхронных действий с помощью JavaScript, 312
  - загрузка JavaScript в DOM, 314
  - защита от CSRF для HTTP-запросов, 315
- json\_script шаблонный фильтр, ссылка на справочные материалы, 718

## K

Kubernetes, ссылка на справочные материалы, 776

## L

- Let's Encrypt, URL-адрес центра сертификации, 769
- Let's Encrypt, сервис, 302
- ссылка на справочные материалы, 302
- login(), функция, 404

## M

- Memcached, 663
- добавление в проект, 664
  - установка, 663
  - установка привязки к Python, 663
  - установка Docker-образа, 663
  - URL-адрес, 663
- minHeight, константа, 297
- minWidth, константа, 297
- ModelForm
- метод save(), 288
  - обработка в представлениях, 116

## N

### NGINX

- доставка Django, 754, 757

## использование, 757

- использование для раздачи нескольких поддоменов, 780
- использование для раздачи статических файлов, 762
- конфигурация вместе с сервером Daphne, 773
- конфигурирование, 758
- конфигурирование под использование SSL/TLS, 767
- раздача медиафайлов, 761
- раздача статических файлов, 761
- ссылка на справочные материалы, 757, 759

## O

- OAuth 2.0, стандарт, 242
- order\_by(), метод, применение для извлечения объектов, 64

## P

- Paginator, класс, ссылка на справочные материалы, 93
- Pillow, библиотека, установка, 224
- Poedit, ссылка на скачивание, 544
- POST-запрос, 426
- POST-параметр
- action, 313
  - image\_id, 313
- Postgres.app, ссылка на справочные материалы, 172
- PostgreSQL
- конфигурирование службы, 749
  - создание базы данных, 173
  - ссылка на скачивание, 172
  - ссылка на справочные материалы по Docker-образу, 749
  - установка, 172
- Postman, ссылка на справочные материалы, 687
- Postman, API-платформа, ссылка на справочные материалы, 703
- prefetch\_related(), метод, применение, 357
- Python
- использование Redis, 377
  - установка, 29

## Q

- QuerySet. См. *Набор запросов*
- вычисление, 65

оптимизация наборов запросов, предусматривающих связанные объекты, 355  
применение метода prefetch\_related(), 357  
применение метода select\_related(), 356  
работа с набором запросов, 60

## R

### RabbitMQ

доступ к интерфейсу управления, 435  
использование с Django, 433  
ссылка на справочные материалы, 434  
установка, 434

### receive(), 715

### Redis

использование вместе с Python, 377  
использование для установления канального слоя, 724  
отслеживание с помощью приложения Django Redisboard, 676  
применение для подсчета просмотров изображений, 374  
следующие шаги, 384  
сценарии, 384  
установка, 375  
хранение просмотров изображений, 379  
хранение рейтинга, 381  
URL-адрес, 375

### Redis, кеш-бэкенд, применение, 675

### ReportLab, 488

ссылка на справочные материалы, 488  
requests, библиотека, 288  
ссылка на документацию, 703  
ссылка на справочные материалы, 288  
установка, 288

### Response, объект, ссылка на справочные материалы, 320

### REST framework, ссылка на справочные материалы, 681, 703

### RESTful API

добавление разрешений в представления, 692  
обработка аутентификации, 691  
определение сериализаторов, 682  
парсер, 683  
потребление, 686, 700  
разработка, 681  
разработка конкретно-прикладных представлений API, 690

разработка представления списка и детальной информации, 684  
рендерер, 683  
сериализация содержимого курса, 697  
создание вложенных сериализаторов, 688  
создание конкретно-прикладных разрешений, 697  
создание наборов представлений ViewSet и маршрутизаторов, 694  
установка фреймворка Django REST framework, 681  
Rosetta, ссылка на документацию, 553  
Rosetta, интерфейс перевода, применение, 551

## S

save(), метод, 62, 408  
переопределение метода класса ModelForm, 288  
scrypt, алгоритм хеширования, 219  
select\_related(), метод, применение, 356  
SendGrid, URL-адрес, 102  
siteUrl, константа, 297  
SQLite, 172  
SSL/TLS  
использование для обеспечения безопасности веб-сайтов, 764  
конфигурирование веб-сервера NGINX, 767  
конфигурирование проекта Django, 765  
creation сертификата, 767  
static(), вспомогательная функция, 580  
staticUrl, константа, 297  
Stripe, 445  
добавление в проект, 449  
интеграция оформления платежа, 452  
интерфейс командной строки (CLI), 467  
отсылка к платежам в заказах, 475  
проверка информации о платеже в информационной панели, 463  
создание купона для Checkout  
Checkout, 517  
создание учетной записи, 445  
установка библиотеки Python, 448  
URL-адрес, 447

## T

Twitter, использование для добавления социальной авторизации, 256

**У**

- URL-адрес дружественный для поисковой оптимизации, создание для постов, 82  
 URL-адрес канонический  
     видеоизменение для постов, 86  
     применение для моделей, 80  
 uWSGI, 755  
     использование, 755  
     конфигурирование, 756  
     опции, 756  
     ссылка на справочные материалы, 757

**В**

- venv, ссылка на справочные материалы, 31  
 ViewSet  
     добавление действий, 696  
     создание, 694  
     ссылка на справочные материалы, 695

**W**

- WeasyPrint, установка, 489  
 WebSocket, использование безопасных соединений, 773  
 WebSocket-клиент  
     определение событий, 719  
     реализация, 717

**Y**

- YAML, URL-адрес примера, 746

**А**

- Авторизация открытая (OAuth), 242  
 Авторизация социальная  
     добавление на веб-сайт, 242  
     добавление с использованием Facebook, 248  
     добавление с использованием Google, 268  
     добавление с использованием Twitter, 256  
     использование профиля для добавления социальной авторизации, 277  
 Агрегирование, ссылка на справочные материалы, 142  
 Атака с подделкой межсайтовых запросов (CSRF), 419

**Аутентификация**

- добавление представлений аутентификации, 598  
 добавление системы аутентификации в CMS, 598  
 обработка, 691  
 создание шаблона, 599  
 ссылка на справочные материалы, 235

**Б**

- База данных  
     добавление индекса, 46  
     переключение в проекте Django, 174  
     применение миграций, 752  
     сеансы, 404  
     служба, 743  
     создание миграции для модели профиля, 225  
 Блог, добавление полнотекстового поиска в блог, 171  
 Брокер сообщений, 432  
 Букмаклет, 293  
     разработка с помощью JavaScript, 293  
 Букмаклет Pinterest, 293  
 Бэкенд аутентификации  
     конкретно-прикладной  
         запрет на использование пользователями существующего адреса электронной почты, 238  
         разработка, 235  
 Бэкенды поддерживаемые, ссылка на справочные материалы, 243

**В**

- Веб-перехватчик  
     получение уведомлений о платежах, 467  
     создание конечной точки, 467  
     тестирование уведомлений, 472  
 Веб-сайт  
     добавление социальной авторизации, 242  
     обеспечение безопасности с помощью SSL/TLS, 764  
 Веб-сайт для управления визуальными закладками  
     добавление асинхронных действий с помощью JavaScript, 312  
     добавление бесконечной постраничной прокрутки в список изображений, 323

отправка контента из других веб-сайтов, 286  
разработка модели изображения, 282  
регистрация модели изображения на сайте администрирования, 285  
создание, 282  
создание взаимосвязей многие-ко-многим, 284  
создание миниатюр изображений с помощью easy-thumbnails, 309  
создание представления детальной информации об изображениях, 306  
справочные материалы, 330  
Веб-служба, 743  
Веб-сокет. См. *WebSocket*  
Взаимосвязь многие-к-одному добавление, 50  
ссылка на справочные материалы, 113  
Взаимосвязь многие-ко-многим создание, 284  
создание с помощью промежуточной модели, 333  
ссылка на справочные материалы, 138  
Вход в систему единый (SSO), 242  
Выделение основ слов, 182

## Д

Данные, загрузка в новую базу данных, 176  
Данные сеансовые, варианты хранения, 404  
Действие административное, 480  
Действие конкретно-прикладное, добавление на сайт администрирования, 480  
Действие пользователя по подписке/отписке, добавление с помощью JavaScript, 342  
Документация по пакету Django Extensions, ссылка на справочные материалы, 248  
Документация по *рір*, ссылка на справочные материалы, 31

## З

Задание асинхронное, видоизменение потребителя чата, 733  
Задача асинхронная, 431  
добавление в приложение, 438  
работа с асинхронными задачами, 431  
Заказ, экспорт в CSV-файлы, 480

Заказ покупателя  
включение моделей на сайт администрирования, 424  
регистрация, 421  
создание, 425  
создание моделей, 422  
Запрос методом GET, 426  
Запрос методом POST, 426  
Значение порядка, автоматическое присвоение, 592

## И

Индекс, ссылка на справочные материалы, 47  
Индустрия платежных карт (PCI), 445  
Интеграция платежного шлюза, 444  
справочные материалы, 497  
Интернационализация (i18n), 535  
добавление переводов в проект, 537  
команды управления, 536  
настройки, 535  
определение текущего языка, 537  
подготовка проекта, 538  
установка инструментария gettext, 536  
шаблоны URL-адресов, 535  
Интернет вещей (IoT), 706  
Интернет-магазин  
разработка представлений каталога товаров, 395  
регистрация моделей каталога товаров на сайте администрирования, 393  
создание, 388  
создание моделей каталога товаров, 389  
создание шаблонов каталога товаров, 397  
Интерфейс прикладного программирования (API), 445  
Интерфейс шлюза асинхронного сервера (ASGI), 32, 36, 771  
разработка чат-сервера с помощью асинхронной связи через ASGI, 710  
ссылка на справочные материалы, 710  
Интерфейс шлюза веб-сервера (WSGI), 36, 754  
доставка Django, 754  
ссылка на справочные материалы, 755  
Итерфейс шлюза веб-сервера (WSGI), 710

## К

Карта сайта, добавление на веб-сайт, 159

- Кеш сайтовый**
- деактивирование, 675
  - применение, 674
- Кеш-фреймворк**
- бэкенды, 662
  - деактивирование сайтового кеша, 675
  - добавление Memcached в проект, 664
  - кеширование на основе динамических данных, 671
  - кеширование представлений, 673
  - кеширование фрагментов шаблона, 672
  - настроечные параметры, 664
  - применение, 661
  - применение кеш-бэкенда Redis, 675
  - применение низкоуровневого API кеша, 665
  - применение сайтового кеша, 674
  - проверка запросов к кешу с помощью меню отладочных инструментов Django, 667
  - ссылка на справочные материалы, 662
  - уровни кэша, 665
  - установка кэш-сервера Memcached, 663
- Класс примесный.** См. *Примесь*
- Код исходный Python**, перевод, 539
- Команда управления**
- конкретно-прикладная, реализация, 780
- Комментарий**
- добавление в детальную информацию о посте, 122
  - добавление в представление детальной информации о посте, 121
  - добавление на сайт администрации, 114
- Компонент программный**
- промежуточный, ссылка на справочные материалы, 778
- Компонент программный**
- промежуточный конкретно-прикладной, создание, 777
- Компонент программный**
- промежуточный поддомена
  - раздача нескольких поддоменов с помощью веб-сервера NGINX, 780
  - создание, 778
- Контент**
- отправка из других веб-сайтов, 286
  - очистка полей формы, 287
  - переопределение метода save(), 288
  - разработка букмеклата с помощью JavaScript, 293
  - установки библиотеки requests, 288
- Контент полиморфный, создание моделей**, 586
- Корзина покупок**
- добавление товаров, 411, 415
  - настройки сеанса, 404
  - обновление количества товара, 417
  - представления, 410
  - разработка, 403
  - разработка с помощью сессий Django, 403
  - разработка шаблона отображения, 413
  - создание процессора контекста, 418
  - срок истечения сеанса, 405
  - установка в контекст запроса, 419
  - хранение в сессиях, 406
- Купон**
- добавление к заказам в счетах-фактурах в формате PDF, 520
  - добавление к заказам на сайте администрирования, 520
  - применение к заказам, 512
  - применение к корзине покупок, 504
  - создание для платежного инструмента Stripe Checkout, 517
- Курс, отображение**, 643
- Л**
- Лента новостная, создание для постов блога**, 164
- Локализация формата**, 572
- ссылка на справочные материалы, 573
- Локализация (l10n)**, 535
- настройки, 535
- М**
- Маршрутизатор**
- создание, 694
  - ссылка на справочные материалы, 695, 703
- Маршрутизация**, 716
- Медиафайл**
- раздача, 224, 761
  - раздача в проекте электронного обучения, 579
- Менеджер моделей**
- работа с ним, 63
  - создание, 65
- Меню отладочных инструментов Django**
- добавление в проект, 667
  - использование, 366

- команды, 373  
панели, 370  
проверка запросов к кешу, 667  
установка, 367, 667
- Механизм рекомендательный  
разработка, 523  
товары, рекомендуемые на основе  
предыдущих покупок, 524
- Миграция  
применение, 51  
создание, 51  
создание для переводов моделей, 566
- Миграция данных, ссылка на справочные  
материалы, 586
- Модели данных для блога  
добавление взаимосвязи  
многие-к-одному, 50  
добавление индекса базы данных, 46  
добавление полей даты/времени, 44  
добавление поля статуса, 47  
определение предустановленного  
порядка сортировки, 45  
применение миграций, 51  
создание, 42  
создание миграций, 51  
создание модели Post, 42
- Модель  
добавление обобщенных  
отношений, 347  
перевод с помощью модуля  
django-parler, 562  
применение канонических  
URL-адресов, 80  
создание для полиморфного  
контента, 586  
создание для хранения комментариев  
пользователей, 112  
создание для хранения комментариев  
пользователей к постам, 112  
создание сайта администрирования  
для модели, 54  
создание форм, 116
- Модель абстрактная, 587, 588
- Модель блога, добавление на сайт  
администрирования, 56
- Модель документа объектная (DOM), 297,  
315, 600, 717
- Модель купонная, разработка, 500
- Модель курса  
разработка, 580  
регистрация на сайте  
администрирования, 582
- Модель пользователя  
расширение, 223
- ссылка на справочные материалы, 223  
Модель пользователя  
конкретно-прикладная  
использование, 231  
ссылка на справочные материалы, 231
- Модель промежуточная, создание  
взаимосвязи многие-ко-многим, 333
- Модель содержимого, создание, 589
- Модель Post, создание, 42
- Модуль, добавление поля ordering, 594
- Модуль курса  
добавление содержимого, 621  
использование набора форм, 616  
переупорядочивание, 632  
переупорядочивание содержимого, 632  
управление, 616, 627  
управление содержимым, 616, 627
- ## Н
- Набор запросов, 60
- Набор форм, использование для модулей  
курса, 616
- Набор форм модельный Django, ссылка на  
справочные материалы, 641
- Набор форм Django, ссылка на  
справочные материалы, 641
- Намерение платежа, 464
- Наследование многотабличных  
моделей, 587, 588
- Наследование модельное,  
использование, 587
- Настройка кеша в Django, 664
- Несколько сред, управление  
настроочными параметрами Django, 739
- ## О
- Объект  
извлечение, 63  
извлечение с помощью метода  
exclude(), 64  
извлечение с помощью метода  
filter(), 63  
извлечение с помощью метода  
order\_by(), 64  
обновление, 62  
создание, 61  
удаление, 64
- Объект content, добавление поля  
ordering, 594
- Объект order, разработка по отношению  
к другим полям, 592

Очередь сообщений, 432  
 Ошибка постраничной разбивки, обработка, 91

## П

Пакеты, требующиеся для Python, добавление, 745  
 Панель информационная портала Twitter для разработчика, ссылка на справочные материалы, 257  
 Параметр настроечный Django, управление несколькими средами, 739  
 Параметр сеанса настроечный, 404  
     ссылка на справочные материалы, 405  
 Парсер, 683  
     ссылка на справочные материалы, 684, 703  
 Первым вошел – первым вышел (FIFO), 432  
 Перевод  
     собственный исходный код, 541  
     с переменными, 540  
     ссылка на справочные материалы, 539  
     формы множественного числа, 541  
 Перевод ленивый, 540  
 Перевод моделей  
     адаптация представлений, 570  
     использование  
     с ORM-преобразователем, 569  
 Перевод модели, интеграция на сайт администрирования, 565  
 Перевод нечеткий, 554  
 Перевод стандартный, 540  
 Передача репрезентативного состояния (REST), 681  
 Подделка межсайтовых запросов (CSRF), 108, 195, 315, 766  
     защита HTTP-запросов на JavaScript, 315  
     ссылка на справочные материалы, 109, 316  
 Поиск полнотекстовый, 177  
     взвешивание запросов, 184  
     выгрузка существующих данных, 174  
     выделение основ слов стоп-слов в разных языках, 183  
     добавление в блог, 171  
     загрузка данных в новую базу данных, 176  
     переключение базы данных в проекте Django, 174

поиск по нескольким полям, 177  
 по триграммному сходству, 185  
 простые поисковые запросы, 177  
 разработка представления поиска, 178  
 ранжирование результатов, 182  
 результаты выделения основ слов, 182  
 создание базы данных PostgreSQL, 173  
 ссылка на справочные материалы, 186  
 удаление стоп-слов в разных языках, 183  
 установка PostgreSQL, 172  
 Поле даты/времени, добавление, 44  
 Поле модели, ссылка на справочные материалы, 48  
 Поле модельное конкретно-прикладное создание, 592  
     ссылка на справочные материалы, 594  
 Поле статуса, добавление, 47  
 Поле формы  
     валидация с помощью модуля django-localflavor, 573  
     очистка, 287  
 Полиморфизм, 586  
 Политика обеспечения строгой безопасности передачи информации по протоколу HTTP (HSTS), 766  
 Порядок сортировки предустановленный, определение, 45  
 Пост  
     видоизменение канонического URL-адреса, 86  
     извлечение по схожести, 141  
     создание дружественных для поисковой оптимизации URL-адресов, 82  
 Пост блога, создание новостных лент, 164  
 Потребитель  
     написание, 714  
     обновление с целью широковещательной рассылки сообщений, 725  
 Почта электронная, отправка в представлениях, 106  
 Представление  
     адаптация под переводы моделей, 570  
     видоизменение, 85  
     добавление шаблонов URL-адресов, 69  
     кэширование, 673  
     обработка форм, 99  
     обработка форм ModelForm, 116  
     отправка электронной почты, 106

- 
- Представление детальной информации
    - применение функции сокращенного доступа `get_object_or_404`, 68
    - создание, 67
    - формирование, 67, 684
  - Представление детальной информации об изображениях, создание, 306
  - Представление детальной информации о посте, добавление комментариев, 121
  - Представление конкретно-прикладное, расширение сайта
    - администрирования, 483
  - Представление на основе класса и CMS
    - использование примесных классов, 605
    - ограничение доступа, 610
    - создание, 604
  - Представление на основе классов
    - потребность в нем, 94
    - преимущества, 95
    - применение для отображения списка постов, 95
    - разработка, 94
    - ссылка на справочные материалы, 97
  - Представление поиска, разработка, 178
  - Представление списка
    - создание, 67
    - формирование, 67
  - Представление списка и детальной информации, создание для профилей пользователей, 336
  - Представление списка постов, добавление постраничной разбивки, 87
  - Представление API
    - конкретно-прикладное, разработка, 690
  - Преобразователь объектно-реляционный (ORM), 60
    - использование переводов моделей, 569
  - Префикс языка, добавление в шаблоны URL-адресов, 555
  - Приложение, конфигурационные классы, 364
  - Приложение для ведения блога, активирование, 47
  - Приложение для ведения чата
    - активация канального слоя, 714
    - видоизменение под полную асинхронность, 733
    - интеграция с существующими представлениями, 735
    - конфигурирование маршрутизации, 714
    - конфигурирование потребителя, 714
  - реализация, 706
  - реализация WebSocket-клиента, 714
  - создание, 705
  - Приложение для потока активности типовое
    - добавление действий пользователей, 352
    - добавление обобщенных отношений в модели, 347
    - игнорирование повторных действий, 351
    - оптимизация наборов запросов, предусматривающих связанные объекты, 355
    - отображение, 355
    - применение фреймворка `contenttypes`, 346
    - разработка, 345
    - создание шаблона действий, 357
  - Приложение Django, создание, 41
  - Примесь
    - использование для представлений на основе классов, 605
    - использование примесных классов из `django-braces`, 633
    - ссылка на справочные материалы, 606
  - Примесь Django, ссылка на документацию, 641
  - Программа быстрого запуска Python, ссылка на справочные материалы, 29
  - Проект, подготовка к интернационализации (`i18n`), 538
  - Проект социального веб-сайта
    - инициирование, 189
    - создание, 189
  - Проект электронного обучения
    - настройка, 578
    - подготовка к раздаче медиафайлов, 579
    - разработка моделей курса, 580
  - Проект Django
    - выполнение сервера разработки, 37
    - конфигурирование под SSL/TLS, 765
    - настроечные параметры, 39
      - ссылка на справочные материалы, 39
    - применение первоначальных миграций базы данных, 36
    - проверка готовности к эксплуатации в производственных условиях, 764
    - проект и приложение, 40
    - ресурсы, 76
    - создание, 35
    - структурная, 41

Прокрутка бесконечная постраничная  
  добавление в список изображений, 323

Прокси-модель, 587, 589

Просмотр изображений  
  подсчет числа с помощью Redis, 374  
  хранение в Redis, 379

Пространство имен URL-адресов  
  ссылка на справочные материалы, 71

Протокол облегченного доступа  
к каталогам (LDAP), 235

Протокол очередности сообщений  
продвинутый (AMQP), 433

Протокол передачи почты простой  
(SMTP), 101, 212, 431, 495

Протокол управления передачей  
(TCP), 710

Профиль пользователя, создание  
представлений списка и детальной  
информации, 336

Профиль социальный, создание для  
пользователей, регистрирующихся  
посредством социальной  
автентификации, 277

Процессор контекста, 418  
  создание для корзины покупок, 418  
  ссылка на справочные материалы, 419

Процесс платежа  
  интеграция платежного продукта Stripe  
    Checkout, 452  
  отсылка к платежам Stripe, 475  
  получение уведомлений о платежах  
  посредством веб-перехватчиков, 467  
  проверка информации о платеже  
  в информационной панели Stripe, 463  
  процедура оформления платежа, 452  
  публикация, 479  
  тестирование использования  
    кредитных карт, 461  
  тестирование процесса оформления  
    платежа, 459  
  формирование, 450

**P**

Работник, 432

Работник Celery, 437

Разбивка постраничная  
  добавление, 87  
  добавление в представления списка  
    постов, 87

Развертывание Django с ASGI, ссылка на  
справочные материалы, 710

Разрешение  
  добавление в представления, 692  
  ссылка на справочные материалы, 703

Разрешение конкретно-прикладное  
  создание, 697  
  ссылка на справочные материалы, 608

Регистрация пользователя на  
платформе, 216

Регистрация студентов  
  добавление, 648  
  зачисление на курсы, 651  
  создание представлений, 649

Рендерер, 683  
  ссылка на справочные материалы, 684,  
    703

Ресурсы, 76, 129, 186, 239, 279, 330, 385,  
443, 497, 532, 575, 641, 678, 703, 736, 783

**C**

Сайт администрирования  
  добавление комментариев, 114  
  добавление конкретно-прикладных  
    действий, 480  
  добавление моделей блога, 56  
  интеграция перевода моделей, 565  
  отображение и адаптация под  
    конкретно-прикладную задачу, 58  
  расширение за счет конкретно-  
    прикладных представлений, 483  
  регистрация моделей курса, 582  
  создание для моделей, 54  
  создание суперпользователя, 54  
  ссылка на справочные материалы, 60

**Сеанс**

на основе базы данных, 404

на основе базы данных  
  кешированный, 404  
  на основе кеша, 404  
  на основе файла, 404  
  на основе cookie-файла, 404

Django, использование для разработки  
корзины покупок, 403

Сервис простой почтовый Amazon,  
ссылка на справочные материалы, 102

Сериализатор  
  определение, 682  
  ссылка на справочные материалы, 703

Сериализатор вложенный, создание, 688

Сеть доставки контента публичная  
(CDN), 316

- Сигнал  
применение для денормализации количественных данных, 361  
работа с сигналами, 361
- Система комментариев, создание, 112
- Система купонная, создание, 499
- Система подписки  
добавление действий пользователей по подписке/отписке с помощью JavaScript, 342  
разработка, 333  
создание взаимосвязи  
многие-ко-многим с помощью промежуточной модели, 333  
создание представлений списка и детальной информации для профилей пользователей, 336
- Система управления контентом (CMS)  
работа с группами и разрешениями, 608  
создание, 604  
создание представлений на основе классов, 605
- Скриптинг межсайтовый (XSS), 479
- Слой защищенных сокетов (SSL), 245, 764
- Слой канальный, 723  
активирование, 723  
группы, 724  
добавление контекста в сообщения, 730  
канал, 724  
обновление потребителя с целью широковещательной рассылки сообщений, 725  
установление с использованием Redis, 724
- Служба кеша, 743
- Служба Redis, конфигурирование, 753
- Совместимость с Django, ссылка на справочные материалы, 565
- Содержимое курса  
доступ, 655  
отображение, 659  
прорисовка типов содержимого, 659  
серIALIZАЦИЯ, 697
- Соединение безопасное, использование для веб-сокетов, 773
- Соединение защищенное транспортного уровня (TLS), 101, 245, 764
- Сообщение  
добавление контекста, 730  
обновление потребителя с целью широковещательной рассылки, 725
- Список, разработка представления, 684
- Среда виртуальная Python, создание, 30
- Среда локальная  
запуск и управление, 740  
конфигурирование настроек, 740
- Среда производственная  
настройки, 741  
создание, 739  
цикл запроса/ответа, 759
- Стемминг. См. *Выделение основ слов*
- Суперпользователь, создание, 752
- Схема единого входа в систему (SSO), 242
- Счет-фактура в формате PDF  
динамическое генерирование, 488  
отправка PDF-файлов по электронной почте, 494  
прорисовка PDF-файлов, 490  
создание шаблона, 489
- T**
- Тег, 132
- Тег включения шаблонный, создание, 150
- Тег и фильтр встроенный шаблонный, ссылка на справочные материалы, 146
- Тег шаблонный, перевод, 545
- Тег шаблонный конкретно-прикладной, создание, 146
- Тег шаблонный простой  
создание, 147  
создание тега, который возвращает набор запросов QuerySet, 152
- Технология единого входа в систему (SSO), 242
- Технология интерфейса шлюза асинхронного сервера (ASGI), 771
- Технология интерфейса шлюза веб-сервера (WSGI), 36, 710, 754
- Тип события Stripe, ссылка на справочные материалы, 468
- Токен подделки межсайтовых запросов (CSRF), 633
- Триграммма, 185
- У**
- Условия предварительные для рекомендации постов по электронной почте, 98  
обработка форм в визуальных представлениях, 99  
отправка писем с помощью Django, 101  
отправка почты в представлениях, 106

прорисовка форм в шаблонах, 107  
 создание форм с помощью Django, 98  
**Условия предварительные для создания системы комментариев**, 112  
 добавление комментариев в представление детальной информации о посте, 121  
 добавление комментариев в шаблон детальной информации о посте, 122  
 добавление комментариев на сайт администрации, 114  
 обработка форм ModelForm в представлениях, 116  
 создание форм из моделей, 116  
 создание формы для хранения комментариев пользователей к постам, 112  
 шаблон, создание для комментарной формы, 119  
**Установщик Python**, ссылка на скачивание, 29

## Ф

**Файл разделенных запятыми значений (CSV)**, 480  
 экспорт, 480  
**Файл сообщений**, 535  
**Файл статический**  
 раздача, 761  
 раздача с помощью веб-сервера NGINX, 762  
 сбор, 761  
**Фикстура**, 174, 583  
 применение с целью предоставления моделям первоначальных данных, 583  
 тестирование, ссылка на справочные материалы, 586  
**Фильтр шаблонный конкретно-прикладной**  
 реализация, 154  
 создание, 146  
 создание с поддержкой синтаксиса Markdown, 154  
**Форма**  
 обработка в визуальных представлениях, 99  
 прорисовка в шаблонах, 107  
 создание из моделей, 116  
 создание с помощью Django, 98  
 ссылка на справочные материалы, 116  
 типы полей, ссылка на справочные материалы, 99

**Форма комментарная, создание шаблона**, 119  
**Фрагмент шаблона, кеширование**, 672  
**Фреймворк аутентификации, встроенный в Django**  
 использование, 191  
 использование представлений, 199  
 модели, 191  
 представления входа на платформу, 199  
 представления выхода из платформы, 199  
 представления сброса пароля, 208  
 представления смены пароля, 205  
 создание представления входа на платформу, 192  
**Фреймворк сайтов, ссылка на справочные материалы**, 163  
**Фреймворк синдицированных новостных лент, ссылка на справочные материалы**, 171  
**Фреймворк сообщений**  
 использование, 231  
 ссылка на справочные материалы, 234  
**Фреймворк contenttypes, применение**, 346  
**Функциональность тегирования, добавление**, 132

## Х

**Хост-имя, использование**, 760  
**Хосты, разрешенные в Django**, ссылка на справочные материалы, 244

## Ц

**Центр сертификации (CA)**, 246, 766  
**Цикл запроса/ответа, на основе приложения-обертки Channels**, 710

## Ш

**Шаблон**  
 прорисовка форм, 107  
 создание базового шаблона, 72  
 создание для комментарной формы, 119  
 создание для представления, 71  
 шаблон представления  
 доступ к приложению, 74  
 создание базового шаблона, 72  
 создание шаблона списка постов, 73  
**Шаблон архитектурного дизайна MTV (Model-Template-View)**, 33

Шаблон архитектурного дизайна MVC (Model-View-Controller), 33  
Шаблон детальной информации о посте, добавление комментариев, 122  
Шаблон постраничной разбивки, создание, 88  
Шаблон списка постов, создание, 73  
Шаблон shop, перевод, 547  
Шаблон URL-адреса  
    видоизменение, 84  
    для интернационализации (i18n), 554

добавление для представлений, 69  
добавление префикса языка, 555  
    перевод, 556  
Шлюз платежный, 444

**Я**

Язык сайта, переключение пользователями, 560  
Язык шаблонов Django, ссылка на справочные материалы, 70

Книги издательства «ДМК ПРЕСС»  
можно купить оптом и в розницу  
в книготорговой компании «Галактика»  
(представляет интересы издательств  
«ДМК ПРЕСС», «СОЛООН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38, оф. 10;  
тел.: (499) 782-38-89, электронная почта: [books@aliens-kniga.ru](mailto:books@aliens-kniga.ru).  
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: <http://www.galaktika-dmk.com/>.

Антонио Меле

## Django 4 в примерах

Главный редактор *Мовчан Д. А.*  
*dmkpress@gmail.com*

Зам. главного редактора *Сенченкова Е. А.*  
Перевод *Логунов А. В.*  
Корректор *Синяева Г. И.*  
Верстка *Чаннова А. А.*  
Дизайн обложки *Мовчан А. Г.*

Гарнитура РТ Serif. Печать цифровая.  
Усл. печ. л. 65. Тираж 200 экз.

Веб-сайт издательства: [www.dmkpress.com](http://www.dmkpress.com)