

Джанго 2 по примерам

Создание мощных и надежных веб-приложений Python с нуля



Packt

www.packt.com

By Antonio Melé

Django 2 в примерах

Создавайте мощные и надежные веб-приложения Python с нуля



BIRMINGHAM - MUMBAI

Django 2 в примерах

Copyright © 2018 Packt Publishing

Все права защищены. Никакая часть этой книги не может быть воспроизведена, сохранена в поисковой системе или передана в любой форме или любыми средствами без предварительного письменного разрешения издателя, за исключением случаев коротких цитат, включенных в статьи или обзоры.

При подготовке этой книги были предприняты все усилия для обеспечения точности представленной информации. Однако информация, содержащаяся в этой книге, подается без гарантий, явных или подразумеваемых. Ни автор, ни Packt Publishing, ни его дилеры и дистрибуторы не будут нести ответственность за любые убытки, вызванные или предположительно вызванные прямо или косвенно этой книгой.

Content Development Editor: Arun Nadar

Technical Editor: Prajakta Mhatre

Copy Editor: Dhanya Baburaj and Safis Editing

Project Coordinator: Sheejal Shah

Proofreader: Safis Editing

Indexer: Rekha Nair

Production Coordinator: Nilesh Mohite

First published: May 2018

Production reference: 1250518

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78847-248-7

www.packtpub.com

Моей сестре посвящается



mapt.io

Mapt - это онлайн-библиотека, которая дает вам полный доступ к более 5000 книгам и видеороликам, а также ведущие в отрасли инструменты, которые помогут вам спланировать свое личное развитие и продвинуть свою карьеру. Для получения более подробной информации, пожалуйста, посетите наш веб-сайт.

Зачем подписываться?

- Потратите меньше времени на обучение и больше времени на кодинг с помощью электронных книг и видео от более чем 4000 профессионалов отрасли
- Вы улучшите свое обучение с помощью Плана навыков, созданного специально для вас
- Получайте бесплатную электронную книгу или видео каждый месяц
- Mapt полностью доступен для поиска
- Копирование и вставка, печать и закладка

PacktPub.com

Знаете ли вы, что Packt предлагает электронные версии каждой опубликованной книги в форматах PDF и ePub? Вы можете перейти на версию электронной книги на www.PacktPub.com И в качестве пользователя печатной книги, вы имеете право на скидку на электронную книгу. Свяжитесь с нами по адресу service@packtpub.com для получения более подробной информации.

На www.PacktPub.com вы также можете прочитать сборник бесплатных технических статей, подписаться на широкий спектр бесплатных информационных бюллетеней и получать эксклюзивные скидки и предложения по бумажным и электронным книгам Packt.

Авторы

Об авторе

Antonio Melé является техническим директором Exo Investing и основателем Zenx IT. Антонио разрабатывает проекты Django с 2006 года для клиентов в нескольких отраслях. Он работал в качестве технического директора и в качестве консультанта по технологиям для множества технологических стартапов, и он руководил командами разработчиков, которые строят проекты для крупных цифровых предприятий. Антонио имеет степень магистра в области компьютерных технологий от Universidad Pontificia Comillas. Его отец вдохновил его страсть к компьютерам и программированию.

О чём эта книга

Глава 1, *Создание приложения для блога*, вводит вас в основы фреймворка, создавая приложение для блога. Вы создадите основные модели блога, представления, шаблоны и URL-адреса для отображения сообщений в блоге. Вы узнаете, как создавать QuerySets с помощью ORM Django, и вы настроите сайт администрирования Django.

Глава 2, *Улучшение вашего блога с помощью расширения функционала*, учит, как обрабатывать формы, в том числе формы созданные из моделей, отправлять электронные письма с помощью Django и интегрировать сторонние приложения. Вы будете внедрять систему комментариев для своих сообщений в блоге и позволите своим пользователям делиться сообщениями по электронной почте. В главе также рассказывается о процессе создания системы тегов.

Глава 3, *Расширение вашего приложения блог*, изучим, как создавать собственные теги и фильтры шаблонов. В главе также показано, как использовать фреймворк sitemaps и создать канал RSS для своих сообщений. Вы завершите свое приложение для блога, создав поисковую систему с возможностями полнотекстового поиска PostgreSQL.

Глава 4, *Создание сайта социальной сети*, объясняет, как создать социальный веб-сайт. Для создания представлений учетных записей пользователей вы будете использовать фреймворк проверки подлинности Django. Вы узнаете, как создать клиентскую модель профиля пользователя и построить социальную аутентификацию в своем проекте с использованием основных социальных сетей.

Глава 5, *Совместное использование контента на вашем сайте*, научит вас, как превратить ваше социальное приложение в веб-сайт закладок изображений. Вы определите отношения «многие ко многим» для моделей, и вы создадите букмарклет AJAX в JavaScript и интегрируете его в свой проект. В главе показано, как создавать эскизы изображений и как создать собственные декораторы для ваших представлений.

Глава 6, *Отслеживание действий пользователя*, показывает, как создать систему отслеживающую пользователей. Вы завершите создание своего сайта с закладками изображений, создав приложение потока активности пользователя. Вы узнаете, как оптимизировать QuerySets, и вы будете работать с сигналами. Вы интегрируете Redis в свой проект, чтобы подсчитывать просмотры изображения.

Глава 7, *Создание интернет-магазина*, покажет, как создать интернет-магазин. Вы будете создавать модели каталога, и вы создадите корзину покупок, используя сеансы Django. Вы создадите процессор контекста для корзины покупок, и вы узнаете, как реализовать отправку асинхронных уведомлений пользователям, используя Celery.

Глава 8, *Управление платежами и заказами*, объясняет, как интегрировать платежный шлюз в ваш магазин. Вы также настроите сайт администрирования для экспорта заказов в CSV-файлы, и вы будете генерировать PDF-счета динамически.

Глава 9, *Расширение магазина*, учит, как создавать купонные системы для применения скидок к заказам. В главе показано, как добавить интернационализацию к вашему проекту и как перевести модели. Вы также создадите механизм рекомендаций по продуктам, используя Redis.

Глава 10, *Создание платформы электронного обучения (E-Learning)*, поможет вам создать платформу электронного обучения. Вы добавите fixtures в свой проект, используете

наследование модели, создадите пользовательские поля моделей, будите использовать представления на основе классов и научитесь управлять группами и разрешениями. Вы создадите систему управления контентом и научитесь работать с formsets.

[Глава 11](#), *Отображение и кеширование контента*, покажет, как создать систему регистрации учащихся и управлять зачислением учащихся на курсы. Вы будете предоставлять разнообразный контент курса, и вы узнаете, как использовать фреймворк кэша.

[Глава 12](#), *Создание API*, научит создавать RESTful API для вашего проекта с использованием фреймворка Django REST.

[Глава 13](#), *Продвигаемся в жизнь*, показывает, как настроить производственную среду с помощью uWSGI и NGINX, и как обезопасить ее с помощью SSL. В главе объясняется, как создать собственное промежуточное программное обеспечение и пользовательские команды управления.

О рецензистах

Норберт Мате является веб-разработчиком. Он начал свою карьеру еще в 2008 году. Его первым языком программирования в качестве профессионального веб-разработчика был PHP, затем он перешел к JavaScript/Node.js и Python/Django/Django REST. Он увлечен архитектурой программного обеспечения, шаблонами проектирования и чистым кодом. Норберт был рецензентом другой книги о Джанго *Django RESTful Web Services* Packt Publishing.

Я хотел бы поблагодарить мою жену за ее поддержку.

Packt ищет авторов, таких же как ты

Если ты заинтересован стать автором для Packt, посети authors.packtpub.com и зарегистрируйся сегодня. Мы работали с тысячами разработчиками и техническими специалистами, таких же как и вы, чтобы помочь им поделиться своими знаниями с мировым технологическим сообществом. Вы можете сделать общее приложение, подать заявку на конкретную тему, на которую мы набираем автора, или представить вашу собственную идею.

Содержание

[Титульная страница](#)

[Django 2 в примерах](#)

[Посвящается](#)

[Packt Upsell](#)

[Зачем подписываться?](#)

[PacktPub.com](#)

[Авторы](#)

[Об авторе](#)

[О рецензистах](#)

[Packt ищет авторов, таких же как ты](#)

[Предисловие](#)

[Для кого эта книга](#)

[О чем эта книга](#)

[Как получить максимальную отдачу от этой книги](#)

[Загрузка файлов примеров кода](#)

[Пользовательское соглашение](#)

Отзывы

1. Глава 1 Создание приложения для блога

Установка Django

Создание изолированной среды Python

Установка Django с помощью pip

Создание первого проекта

Запуск сервера разработки

Настройки проекта

Проекты и приложения

Создание приложения

Проектирование схемы базы данных блога

Активация приложения

Создание и применение миграции

Создание сайта администрирования для ваших моделей

Создание суперпользователя

Сайт администрирования Django

Добавление ваших моделей на сайт администрирования

Настраиваем способ отображения моделей

Работа с QuerySet и менеджерами

[Создание объектов](#)

[Обновление объектов](#)

[Извлечение объектов](#)

[Использование метода filter\(\)](#)

[Использование exclude\(\)](#)

[Использование order_by\(\)](#)

[Удаление объектов](#)

[Когда выполняется QuerySets](#)

[Создание менеджеров моделей](#)

[Создание списка и детального представлений](#)

[Создание представлений для списка и детального отображения сообщений](#)

[Добавление шаблонов URL для ваших представлений](#)

[Канонические URL для моделей](#)

[Создание шаблонов для ваших представлений](#)

[Добавление пагинации](#)

[Использование базовых классов представлений](#)

[Резюме](#)

[2. Глава 2 Улучшение вашего блога с помощью расширения функционала](#)

[Обмен сообщениями по электронной почте](#)

[Создание форм в Django](#)

[Обработка форм в представлениях](#)

[Отправка электронной почты с помощью Django](#)

[Отображение форм в шаблонах](#)

[Создание системы комментариев](#)

[Создание форм из моделей](#)

[Работа с ModelForms в представлениях](#)

[Добавление комментариев к шаблону детального отображения](#)

[сообщения](#)

[Добавление функциональности тегов](#)

[Получение похожих сообщений](#)

[Резюме](#)

[3. Глава 3 Расширение вашего приложения блог](#)

[Создание собственных шаблонных тегов и фильтров](#)

[Создание тегов шаблонов](#)

[Создание фильтров шаблонов](#)

[Добавление sitemap на ваш сайт](#)

[Создание фидов для сообщений в блоге](#)

[Добавление полнотекстового поиска в ваш блог](#)

[Установка PostgreSQL](#)

[Простые поисковые запросы](#)

[Поиск по нескольким полям](#)

Создание представления поиска

Анализ и ранжирование результатов

Взвешивание запросов

Поиск с помощью сходства триграмм

Другие полнотекстовые поисковые системы

Резюме

4. Глава 4 Создание сайта социальной сети

Создание проекта социального веб-сайта

Запуск проекта сайта социальной сети

Использование фреймворка аутентификации Django

Создание представления входа в систему

Использование представления аутентификации Django

Представления входа и выхода

Представление изменения пароля

Представление сброса пароля

Регистрация пользователей и профили пользователей

Регистрация пользователя

Расширение модели user

Использование собственной модели для пользовате

ля

Использование фреймворка сообщений

Создание собственного бэкэнда для аутентификации
Добавление социальной аутентификации на ваш сайт

Аутентификация с использованием Facebook

Аутентификация с помощью Twitter

Аутентификация с помощью Google

Резюме

5. Глава 5 Совместное использование контента на вашем сайте

Создание веб-сайта закладок (bookmarking) изображений

Построение модели изображения

Создание отношений «многие ко многим»

Регистрация модели изображения на сайте администрирования

Я

Публикация контента с других сайтов

Очистка полей формы

Переопределение метода save() в ModelForm

Создание bookmarklet с помощью jQuery

Создание детального представления для изображений

Создание эскизов изображений с использованием sorl-thumbnail
Добавление AJAX с помощью jQuery

[Загрузка jQuery](#)

[Подделка межсайтовых запросов в AJAX](#)

[Выполнение запросов AJAX с помощью jQuery](#)

[Создание пользовательских декораторов для ваших представлений](#)

[Добавление AJAX пагинации для вашего представления списка](#)

[Резюме](#)

[6. Глава 6 Отслеживание действий пользователя](#)

[Построение системы отслеживания](#)

[Создание отношений «многие ко многим» с применением промежуточной модели](#)

[Создание списков и подробных представлений для профилей пользователей](#)

[Создание представления AJAX для отслеживания пользовательской](#)

[Построение приложения для отображения общей активности](#)

[Использование фреймворка contenttypes](#)

[Добавление общих отношений к вашим моделям](#)

[Избегаем дублирования действий в потоке активности](#)

[Добавление действий пользователя в поток активности](#)

[Отображение потока активности](#)

[Оптимизация QuerySet, которые взаимодействуют со связанными объектами](#)

[Использование select_related\(\)](#)

Использование prefetch_related()

Создание шаблона для actions

Использование сигналов для денормализации счетчиков

Работа с сигналами

Классы конфигурации приложений

Использование Redis для хранения запросов к представлениям

Установка Redis

Использование Redis в Python

Сохранение запросов к представлениям в Redis

Сохранение рейтинга в Redis

Следующие шаги с Redis

Резюме

7. Глава 7 Создание интернет-магазина

Создание проекта интернет-магазина

Создание моделей каталога товаров

Регистрация моделей на сайте администратора

Создание представлений каталога

Создание шаблонов каталога

Создание корзины покупок

[Использование сессий в Django](#)

[Настройки сессий](#)

[Завершение сессий](#)

[Хранение корзины покупок в сессиях](#)

[Создание представления для корзины покупок](#)

[Добавление товаров в корзину](#)

[Создание шаблона для отображения корзины](#)

[Добавление продуктов в корзину](#)

[Обновление количества товара в корзине](#)

[Создание контекстного процессора для текущей корзины](#)

[Контекстные процессоры](#)

[Настройка контекста запроса для корзины](#)

[Регистрация заказов клиентов](#)

[Создание моделей заказов](#)

[Добавление модели заказов на сайт администрирования](#)

[Создание заказов клиентов](#)

[Запуск асинхронных задач с Celery](#)

[Установка Celery](#)

[Установка RabbitMQ](#)

[Добавление Celery в ваш проект](#)

[Добавление асинхронных задач в приложение](#)

[Мониторинг Celery](#)

Резюме

8. Глава 8 Управление платежами и заказами

Интеграция платежного шлюза

Создание учетной записи в "песочнице" от Braintree

Установка модуля Python Braintree

Интеграция платежного шлюза

Интеграция Braintree с использованием Hosted Fields

elds

Тестирование платежей

Внедряем в жизнь

Экспорт заказов в CSV-файлы

Добавление пользовательских действий на сайт администрирования

Расширение сайта администратора с помощью пользовательских предложений

Динамическое создание счетов-фактур PDF

Установка WeasyPrint

Создание шаблона PDF

Отрисовывание PDF файлов

Отправка файлов PDF по электронной почте

Резюме

9. Глава 9 Расширение магазина

Создание системы купонов

Построение модели купонов

Применение купона к корзине покупок

Применение купонов к заказам

Добавление интернационализации и локализации

Интернационализация с помощью Django

Настройки интернационализации и локализации

Команды управления интернационализацией

Как добавить переводы в проект Django

Как Django определяет текущий язык

Подготовка нашего проекта для интернационализации

Перевод кода Python

Стандартные переводы

Ленивый (Lazy) перевод

Переводы включаемых переменных

Множественные формы в переводах

Перевод вашего кода

Перевод шаблонов

[Тег шаблона `{% trans %}`](#)

[Тег шаблона `{% blocktrans %}`](#)

[Перевод шаблонов магазина](#)

[Использование интерфейса перевода Rosetta](#)

[Неточные \(Fuzzy\) переводы](#)

[Шаблоны URL для интернационализации](#)

[Добавление префикса языка к шаблонам URL](#)

[Перевод шаблонов URL](#)

[Разрешение пользователям переключать язык](#)

[Перевод моделей с помощью django-parler](#)

[Установка django-parler](#)

[Перевод полей модели](#)

[Интеграция переводов в сайт администрирования](#)

[Создание миграции для переведенных моделей](#)

[Адаптация представлений для переводов](#)

[Локализация форматов](#)

[Использование django-localflavor для проверки полей форм](#)

ы

[Создание механизма рекомендаций](#)

[Рекомендация товаров на основе предыдущих покупок](#)

[Резюме](#)

10. Глава 10 Создание платформы электронного обучения (E-Learning)

Настройка проекта электронного обучения

Создание моделей курса

Регистрация моделей на сайте администрирования

Использование fixtures для предоставления исходных данных для моделей

Создание моделей для разнообразного контента

Использование наследования модели

Абстрактные модели

Много-табличное наследование модели

Прокси-модели

Создание моделей контента

Создание настраиваемых полей модели

Добавление порядкового номера к модулю и объекту контента
а

Создание CMS

Добавление системы аутентификации

Создание шаблонов аутентификации

Создание представлений на основе базовых классов

Использование миксин для представлений на основе базовых классов
Работа с группами и разрешениями

Ограничение доступа к представлениям на основе

базовых классов

Управление модулями курса и содержимым

Использование formset для модулей курса

Добавление контента в модули курса

Управление модулями и контентом

Переупорядочивание модулей и контента

Использование mixins из django-braces

Резюме

11. Глава 11 Отображение и кеширование контента

Отображение курсов

Добавление регистрации учащихся

Создание представления регистрации учащихся

Зачисление на курсы

Доступ к содержимому курса

Отображение различных типов контента

Использование фреймворка кэшак

Доступные бэкэнды кэша

Установка Memcached

[Настройки кеша](#)
[Добавление Memcached в ваш проект](#)

[Мониторинг Memcached](#)

[Уровни кеширования](#)
[Использование низко-уровневого API кэша](#)

[Кэширование на основе динамических данных](#)

[Кэширование фрагментов шаблона](#)
[Кэширование представлений](#)

[Использование кеша для всего сайта](#)

[Резюме](#)

12. Глава 12 Создание API

[Создание RESTful API](#)

[Установка Django REST фреймворка](#)
[Определение сериализаторов](#)
[Понимание парсеров и рендеринга](#)
[Создание списка и детального представлений](#)
[Создание вложенных сериализаторов](#)
[Создание пользовательских представлений](#)
[Обработка аутентификации](#)
[Добавление разрешений в представления](#)
[Создание наборов представлений и маршрутизаторов](#)

Добавление дополнительных действий к наборам представле

ний

Создание пользовательских разрешений

Сериализация содержимого курса

Резюме

13. Глава 13 Продвигаем в жизнь

Создание производственной среды

Управление настройками для множества сред

Использование PostgreSQL

Проверка вашего проекта

Обслуживание Django через WSGI

Установка uwsgi

Конфигурация uwsgi

Установка NGINX

Производственная среда

Конфигурация NGINX

Обслуживание статических и медиа файлов

Защита соединения при помощи SSL

Создание сертификата SSL

Настройка NGINX для использования SSL

Настройка нашего проекта для SSL

Создание настраиваемого промежуточного программного обеспечения

Создание промежуточного ПО субдомена

Обслуживание нескольких субдоменов с помощью NGINX

Внедрение пользовательских команд управления

Резюме

Другие книги, которыми вы можете наслаждаться

Оставте отзыв - пусть другие читатели узнают, что вы думаете

Предисловие

Django - мощный веб-фреймворк Python, который поощряет быстрое развитие и чистый, прагматичный дизайн, предлагая относительно неглубокую кривую обучения. Это делает его привлекательным как для начинающих, так и для опытных программистов.

Эта книга проведет вас через весь процесс разработки профессиональных веб-приложений с Django. Книга не только охватывает наиболее важные аспекты фреймворка, но и научит вас, как интегрировать другие популярные технологии в ваши проекты Django.

Книга проведет вас через создание реальных приложений, решение общих проблем и внедрение передовых методов с пошаговым подходом, который легко перенять.

После прочтения этой книги у вас будет хорошее представление о том, как работает Django и как создавать практические, продвинутые веб-приложения.

Для кого эта книга

Эта книга предназначена для разработчиков Python, которые хотят изучать Django путем создания реальных приложений. Возможно, вы совершенно новичок в Django, или вы уже знаете немного, но вы хотите получить максимальную отдачу. Эта книга поможет вам овладеть наиболее важными областями фреймворка, создав реальные проекты с нуля. Чтобы прочитать эту книгу, вам необходимо ознакомиться с концепциями программирования. Предполагаются некоторые знания HTML и JavaScript.

Как получить максимальную отдачу от этой книги

Чтобы получить максимальную отдачу от этой книги, рекомендуется иметь хорошие знания Python. Вы также должны ориентироваться в HTML и JavaScript. Перед чтением этой книги рекомендуется прочитать 1 - 3 части официальной документации Django в <https://docs.djangoproject.com/en/2.0/intro/tutorial01/>.

Загрузка файлов примеров кода

Вы можете загрузить файлы примеров кода для этой книги из своей учетной записи по адресу www.packtpub.com. Если вы приобрели эту книгу в другом месте, вы можете посетить www.packtpub.com/support и зарегистрироваться для получения файлов по электронной почте.

Вы можете загрузить файлы кода, выполнив следующие действия:

1. Войдите или зарегистрируйтесь www.packtpub.com.
2. Выберите вкладку SUPPORT tab.
3. Нажмите на Code Downloads & Errata.
4. Введите имя книги в поле Search и следуйте инструкциям на экране.

После загрузки файла убедитесь, что вы распаковываете или извлекаете папку, используя последнюю версию:

- WinRAR/7-Zip для Windows
- Zipeg/iZip/UnRarX для Mac
- 7-Zip/PeaZip для Linux

Набор кода для книги также размещен на GitHub по адресу <https://github.com/PacktPublishing/Django-2-by-Example>. В случае обновления

кода он будет обновлен в существующем репозитории GitHub.

У нас также есть другие комплекты кода из нашего богатого каталога книг и видеороликов, доступных на <https://github.com/PacktPublishing/>. Ознакомтесь с ними!

Пользовательское соглашение

В тексте этой книге используется ряд условных обозначений.

CodeInText: Указывает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, имена путей, URL-адреса, ввод данных пользователя и твиттер-дескрипторы. Вот пример: "Вы можете отключить свою среду в любое время с помощью команды `deactivate`."

Блок кода задается следующим образом:

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

admin.site.register(Post)
```

Когда мы хотим обратить ваше внимание на определенную часть блока кода, соответствующие строки или элементы 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',
]
```

Любой ввод или вывод в командной строке записывается следующим образом:

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

Bold: Указывает новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговые окна появляются в тексте следующим образом. Вот пример: "Заполните форму и нажмите кнопку SAVE ."

Предупреждения или важные примечания выглядят следующим образом.

Так выглядят советы и трюки.

Отзывы

Пожалуйста, оставьте отзыв. После того, как вы прочитали и проработали эту книгу, почему бы не оставить отзыв на сайте, на котором вы ее купили? Потенциальные читатели смогут увидеть и использовать ваше непредвзятое мнение для принятия решения о покупке, мы в Packt сможем понять, что вы думаете о наших продуктах, а наши авторы смогут увидеть ваши отзывы о своей книге. Спасибо!

Для получения дополнительной информации о Packt, пожалуйста, посетите packtpub.com.

Глава 1

Создание приложения для блога

В этой книге вы узнаете, как создавать проекты Django, готовые к использованию в производстве. Если вы еще не установили Django, вы узнаете, как это сделать в первой части этой главы. В этой главе описывается, как создать простое приложение для блога с помощью Django. Цель этой главы - получить общее представление о том, как работает система, понять, как разные компоненты взаимодействуют друг с другом, чтобы научить вас навыкам, позволяющие легко создавать проекты Django с базовыми функциями. Вы создадите готовый проект без подробного описания всех его деталей. В этой книге будут рассмотрены различные компоненты инфраструктуры.

В этой главе будут рассмотрены следующие темы:

- Установка Django и создание первого проекта
- Проектирование и миграции моделей
- Создание сайта администрирования для ваших моделей
- Работа с QuerySet и менеджерами
- Создание представлений, шаблонов и URL-адресов
- Добавление разбивки на страницы (пагинации) для

представления списка

- Использование представлений на основе базовых классов Django

Установка Django

Если вы уже установили Django, вы можете пропустить этот раздел и перейти непосредственно к *Создание первого проекта*. Django поставляется в виде пакета Python и поэтому может быть установлен в любой среде Python. Если вы еще не установили Django, следующее краткое руководство поможет в установке для локальной разработки.

Django 2.0 требует Python версии 3.4 или выше. В примерах для этой книги мы будем использовать Python 3.6.5. Если вы используете Linux или macOS X, возможно, у вас установлен Python. Если вы используете Windows, вы можете загрузить установщик Python по адресу <https://www.python.org/downloads/windows/>.

Если вы не уверены, установлен ли Python на вашем компьютере, вы можете проверить это, введя `python` в терминале. Если вы видите что-то вроде следующего, то на вашем компьютере установлен Python:

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

Если установленная версия Python ниже 3.4 или если Python не установлен на вашем компьютере, загрузите Python 3.6.5 из <https://www.python.org/downloads/> и установите его.

Поскольку вы будете использовать Python 3, вам не нужно устанавливать базу данных. Эта версия Python поставляется со встроенным SQLite базой данных. SQLite - это легкая база данных, которую вы можете использовать в Django для

разработки. Если вы планируете развернуть свое приложение в производственной среде, вам следует использовать расширенную базу данных, такую как PostgreSQL, MySQL или Oracle. Вы можете получить дополнительную информацию о том, как запустить вашу базу данных с помощью Django <https://docs.djangoproject.com/en/2.0/topics/install/#database-installation>.

Создание изолированной среды Python

Рекомендуется использовать `virtualenv` для создания изолированной среды Python, чтобы вы могли использовать разные версии пакетов для разных проектов, это гораздо практичнее, чем установка пакетов Python глобально в систему. Еще одно преимущество использования `virtualenv` в том что вам не понадобятся какие-либо административные привилегии для установки пакетов Python. Выполните следующую команду в своей консоли для установки `virtualenv`:

```
| pip install virtualenv
```

Примечание переводчика: Если у вас в системе по умолчанию указана 2 версия Python то вам возможно придется выполнить установку через `pip3`:

```
| sudo pip3 install virtualenv
```

Примечание переводчика: правильнее будет вместо `pip` и `python` явно указывать версию `pip3` и `python3`, это убережет вас от возможных ошибок.

После установки `virtualenv`, создайте изолированную среду следующей командой:

```
| virtualenv my_env
```

Это создаст `my_env/` папку, включающую вашу среду Python. Любые библиотеки Python, которые вы устанавливаете во время активной виртуальной среды, будут находиться в папке `my_env/lib/python3.6/site-packages`.

Если ваша система поставляется с Python 2.X, и вы установили Python 3.X, вы должны указать `virtualenv` использовать последнюю.

Вы можете найти путь, где установлен Python 3, и использовать его для создания виртуальной среды следующими командами:

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

Выполните следующую команду для активации вашей виртуальной среды:

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

В командной строке будет указано имя активной виртуальной среды, заключенной в круглые скобки, следующим образом:

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

Вы можете отключить свою среду в любое время с помощью команды `deactivate`.

Вы можете найти дополнительную информацию о `virtualenv` по адресу <https://virtualenv.pypa.io/en/latest/>.

Вместо `virtualenv`, вы можете использовать `virtualenvwrapper`. Этот инструмент обеспечивает оболочку, которая упрощает создание и управление вашей виртуальной средой. Вы можете скачать его с <https://virtualenvwrapper.readthedocs.io/en/latest/>.

Установка Django с помощью pip

Система управления пакетами `pip` является предпочтительным способом установки Django. Python 3.6 поставляется с предустановленным `pip`, но вы можете найти инструкцию установки `pip` по адресу <https://pip.pypa.io/en/stable/installing/>.

Выполните следующую команду в терминале, чтобы установить Django с помощью `pip`:

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

Django будет установлен в каталог Python `site-packages/` вашей виртуальной среды.

Теперь проверьте, был ли успешно установлен Django. Запустите `python` в терминале, импортируйте Django, и проверьте его версию следующим образом:

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

Если вы получите предыдущий вывод, Django был успешно установлен на вашем компьютере.

Django можно установить несколькими способами. Вы можете найти полное руководство по установке <https://docs.djangoproject.com/en/2.0/topics/install/>.

Создание первого проекта

Нашим первым проектом Django будет блог. Django предоставляет команду, которая позволяет вам создать начальную структуру файлов проекта. Выполните следующую команду из вашего терминала:

```
django-admin startproject mysite
```

Это создаст проект Django с именем `mysite`.

Избегайте присвоения таких же имен как у встроенных модулей Python или Django, для предотвращения конфликтов.

Давайте посмотрим на структуру сгенерированного проекта:

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

Мы получили следующие файлы:

- `manage.py`: Это утилита командной строки, используемая для взаимодействия с вашим проектом. Это тонкая обертка вокруг инструмента `django-admin.py`. Вам не нужно редактировать этот файл.
- `mysite/`: Это ваш каталог проекта, который состоит из следующих файлов:

- `__init__.py`: Пустой файл, который сообщает Python об обработке каталога `mysite` как модуля Python.
- `settings.py`: Это файл настроек и конфигурации вашего проекта и содержит начальные настройки по умолчанию.
- `urls.py`: Это место, где находятся ваши шаблоны URL. Каждый указанный здесь URL сопоставляется с соответствующим представлением.
- `wsgi.py`: Это файл конфигурации для запуска вашего проекта как приложения **Web Server Gateway Interface (WSGI)**.

Сгенерированный файл `settings.py` содержит параметры проекта, включая базовую конфигурацию для использования базы данных SQLite 3 и список с именем `INSTALLED_APPS`, который содержит обычные приложения Django, по умолчанию добавленные в ваш проект. Мы рассмотрим эти приложения позже в разделе *Настройки проекта*.

Чтобы завершить настройку проекта, нам нужно будет создать таблицы в базе данных, необходимые для приложений, перечисленных в `INSTALLED_APPS`. Откройте консоль и выполните следующие команды:

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

Вы увидите вывод, который заканчивается следующими 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 contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying sessions.0001_initial... OK
```

Предыдущие строки - это миграции баз данных, созданные Django. Применяя миграцию мы создаем таблицы для исходных приложений в базе данных. Вы узнаете о команде управления `migrate` в разделе *Создание и применение миграции* этой главы.

Запуск сервера разработки

Django поставляется с легким веб-сервером для быстрого запуска кода без необходимости тратить время на настройку производственного сервера. Когда вы запускаете сервер разработки Django, он проверяет изменения в вашем коде. Он перезагружается автоматически, освобождая вас от ручной перезагрузки после изменения кода. Однако он может не заметить некоторые действия, такие как добавление новых файлов в ваш проект, в таких случаях вам придется перезагрузить сервер вручную.

Запустите сервер разработки, введя следующую команду из корневой папки вашего проекта:

```
python manage.py runserver
```

Вы должны увидеть что-то вроде этого:

```
Performing system checks...

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

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



Django Documentation

Topics, references, & how-to's



Tutorial: A Polling App

Get started with Django



Django Community

Connect, get help, or contribute

Предыдущий скриншот показывает, что Django запущен. Если вы гляните на свою консоль, вы увидите GET запрос, выполненный вашим браузером:

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

Каждый HTTP-запрос регистрируется в консоли сервером разработки. Любая ошибка, возникающая при запуске сервера разработки, также появится в консоли.

Вы можете указать Django запускать сервер разработки на пользовательском хосте и порту или сообщить ему, чтобы запустил проект, загружая другой файл настроек, следующим образом:

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

Когда вам приходится иметь дело с несколькими средами, требующими разных конфигураций, вы можете создать другой файл настроек для каждой среды.

Помните, что этот сервер предназначен только для разработки и не подходит для использования в производстве. Чтобы развернуть Django в производственной среде, вы должны запустить его как приложение WSGI с использованием реального веб-сервера, такого как Apache, Gunicorn или uWSGI. Вы можете найти дополнительную информацию о том, как развернуть Django с разными веб-серверами на

<https://docs.djangoproject.com/en/2.0/howto/deployment/wsgi/>.

[Глава 13](#) объясняет, как настроить производственную среду для ваших проектов Django.

Настройки проекта

Давайте откроем файл `settings.py` и посмотрим на конфигурацию нашего проекта. Есть несколько настроек, которые Django включает в этот файл, но они являются лишь частью всех доступных параметров Django. Вы можете увидеть все настройки и их значения по умолчанию в

<https://docs.djangoproject.com/en/2.0/ref/settings/>.

На следующие параметры стоит обратить особое внимание:

- `DEBUG` – это логическое значение, которое устанавливает режим отладки в значение включить или выключить. Если он установлен в `True`, Django будет отображать подробные страницы ошибок. Когда вы переходите в производственную среду, помните, что вы должны установить его в `False`. Никогда не устанавливайте в производственной среде сайт в `DEBUG` включен, потому что это предоставит доступ к конфиденциальным данным потенциальным злоумышленникам.
- `ALLOWED_HOSTS` не применяется во включенном режиме отладки или при выполнении тестов. Как только вы переместите свой сайт на производство и установите `DEBUG` в `False`, вам придется добавить свой домен/хост к этому параметру, чтобы он мог обслуживать ваш сайт Django.
- `INSTALLED_APPS` это настройка, которую вам придется

редактировать для всех проектов. Этот параметр указывает Django, какие приложения активны для этого сайта. По умолчанию Django включает следующие приложения:

- `django.contrib.admin`: Сайт администрации
- `django.contrib.auth`: Фреймворк аутентификации
- `django.contrib.contenttypes`: Фреймворк для обработки типов контента
- `django.contrib.sessions`: Фреймворк сессий
- `django.contrib.messages`: Фреймворк обмена сообщениями
- `django.contrib.staticfiles`: Фреймворк для управления статическими файлами
- `MIDDLEWARE` представляет собой список, для выполнения промежуточного программного обеспечения.
- `ROOT_URLCONF` указывает модуль Python, в котором определены корневые шаблоны URL вашего приложения.
- `DATABASES` ЭТО словарь, содержащий настройки для всех баз данных, которые будут использоваться в проекте. Конфигурация по умолчанию использует базу данных SQLite3.
- `LANGUAGE_CODE` Определяет код языка по умолчанию для этого сайта.

Примечание переводчика: для русского языка установите LANGUAGE_CODE = 'ru-ru'

- use_tz сообщает Django активировать/деактивировать поддержку часового пояса. По умолчанию включено. Этот параметр установлен в `True` когда вы создаете новый проект используя `startproject`.

Не беспокойтесь, если вы чего то не понимаете. Вы узнаете более подробно о настройках Django в следующих главах.

Проекты и приложения

В этой книге вы снова и снова будете сталкиваться с проектами и приложениями. В Django проектом считается установка Django с некоторыми настройками. Приложение представляет собой группу моделей, представлений, шаблонов и URL-адресов. Приложения взаимодействуют с каркасом, чтобы обеспечить некоторые конкретные функции и могут быть повторно использованы в различных проектах. Вы можете думать о проекте как о своем веб-сайте, который содержит несколько приложений, таких как блог, вики или форум, которые также могут использоваться другими проектами.

Создание приложения

Теперь давайте создадим наше первое приложение Django. Мы создадим приложение для блога с нуля. В корневом каталоге проекта выполните следующую команду:

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

Это создаст базовую структуру приложения, которая выглядит так:

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

Мы получили следующие файлы:

- `admin.py`: Здесь вы регистрируете модели для их включения в сайт администрирования Django. Использование админ-сайта Django является необязательным.
- `apps.py`: Этот файл включает в себя основную конфигурацию приложения `blog`.
- `migrations`: Этот каталог будет содержать миграции базы данных вашего приложения. Миграции позволяют

Django отслеживать изменения модели и соответственно синхронизировать базу данных.

- `models.py`: Модели данных вашего приложения. Все приложения Django должны иметь файл `models.py`, но этот файл можно оставить пустым.
- `tests.py`: Сюда вы можете добавить тесты для своего приложения.
- `views.py`: Логика вашего приложения находится здесь; каждое представление получает HTTP-запрос, обрабатывает его и возвращает ответ.

Проектирование схемы базы данных блога

Мы начнем разработку нашей схемы базы данных блога, определив модели данных для нашего блога. Модель представляет собой класс Python, который является подклассом `django.db.models.Model`, в котором каждый атрибут представляет собой поле базы данных. Django создаст таблицу для каждой модели, определенной в файле `models.py`. Когда вы создаете модель, Django предоставляет вам простой API для запроса объектов из базы данных.

Во-первых, мы определим модель `Post`. Добавте следующие строки в файл `models.py` приложения `blog`:

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

class Post(models.Model):
    STATUS_CHOICES = (
        ('draft', 'Draft'),
        ('published', 'Published'),
    )
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250,
                           unique_for_date='publish')
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10,
                            choices=STATUS_CHOICES,
                            default='draft')
```

```
class Meta:  
    ordering = ('-publish',)  
  
    def __str__(self):  
        return self.title
```

Это наша модель данных для сообщений в блоге. Давайте посмотрим на поля, которые мы только что определили для этой модели:

- `title`: Это поле для заголовка сообщения. Это поле `CharField`, которое переводится в `VARCHAR` в базе данных SQL.
- `slug`: Это поле, предназначенное для использования в URL-адресах. `Slug` - это короткая метка, содержащая только буквы, цифры, символы подчеркивания или дефисы. Мы будем использовать поле `slug` для создания красивых, SEO-дружественных URL-адресов для наших сообщений в блоге. Мы добавили параметр `unique_for_date` для этого поля, чтобы создавать URL-адреса для сообщений, используя их дату публикации и `slug`. Django предотвратит множество сообщений с тем же `slug` на определенную дату.
- `author`: Это поле является внешним ключом. Он определяет отношения «много-к-одному». Мы сообщаем Django, что каждый пост написан пользователем, и пользователь может писать любое количество сообщений. Для этого поля Django создаст внешний ключ в базе данных, используя первичный ключ соответствующей модели. В этом случае мы полагаемся на модель `user` системы аутентификации Django. Параметр `on_delete` указывает поведение, которое

необходимо принять, когда связанный объект ссылки удаляется. Это не относится к Django; это стандарт SQL. Используя `CASCADE`, мы укажем, что когда связанный пользователь удаляется, база данных также удалит связанные записи в блоге. Вы можете посмотреть все возможные варианты https://docs.djangoproject.com/en/2.0/ref/models/fields/#django.db.models.ForeignKey.on_delete. Мы указываем имя обратной связи, из `User` к `Post`, через атрибут `related_name`. Это позволит нам легко получить доступ к связанным объектам. Мы узнаем об этом позже.

- `body`: Содержание поста. Это поле является текстовым полем, которое преобразуется в столбец `TEXT` в базе данных SQL.
- `publish`: Это поле времени и даты указывает, когда сообщение было опубликовано. Мы используем метод Django `now` как значение по умолчанию. Он возвращает текущее время и дату в формате, соответствующем часовому поясу. Вы можете думать об этом как о версии стандартного метода Python `datetime.now`.
- `created`: Это поле дата-время указывает, когда сообщение было создано. Поскольку мы используем `auto_now_add` то дата будет автоматически сохранена при создании объекта.
- `updated` : Это поле дата-время указывает на последний раз, когда сообщение было обновлено. Поскольку мы используем `auto_now` то дата будет автоматически обновляться при сохранении объекта.

- `status` : в этом поле отображается статус сообщения. Мы используем параметр `choices`, поэтому значение этого поля может быть установлено только в один из указанных вариантов.

Django поставляется с различными типами полей, которые вы можете использовать при создании ваших моделей. Вы можете найти все типы полей

В <https://docs.djangoproject.com/en/2.0/ref/models/fields/>.

Класс `Meta` внутри модели содержит метаданные. Мы указываем Django сортировать результаты по полю `publish` в порядке убывания при запросе к базе данных. Мы указываем убывающий порядок, используя отрицательный префикс. Таким образом, первыми публикуются недавно добавленный сообщения.

Метод `__str__()` является человеко-читаемым представлением объекта. Django будет использовать его во многих местах, таких как сайт администрирования.

Если вы используете Python 2.X, обратите внимание, что в Python 3 все строки изначально считаются Unicode, поэтому мы используем только метод `__str__()`. Метод `__unicode__()` устарел.

Активация приложения

Чтобы Django отслеживал наше приложение и имел возможность создавать таблицы базы данных для ваших моделей, мы должны активировать его. Для этого отредактируйте файл `settings.py` и добавьте `blog.apps.BlogConfig` в настройку `INSTALLED_APPS`. Она должна выглядеть так:

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

Класс `BlogConfig` - это конфигурация вашего приложения. Теперь Django знает, что наше приложение активно для этого проекта и сможет загружать его модели.

Создание и применение миграции

Теперь, когда у нас есть модель данных для наших сообщений в блоге, нам понадобится таблица базы данных. Django поставляется с системой миграции, которая отслеживает изменения, внесенные в модели, и позволяет переносить их в базу данных. Команда `migrate` применяет миграции для всех приложений, перечисленных в `INSTALLED_APPS`, она синхронизирует базу данных с текущими моделями и существующими миграциями.

Сначала вам нужно создать начальную миграцию для нашей модели `Post`. В корневом каталоге вашего проекта выполните следующую команду:

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

Вы должны получить следующий результат:

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

Django только что создал файл `0001_initial.py` в каталоге `migrations` приложения `blog`. Вы можете открыть этот файл, чтобы увидеть, как происходит миграция. Миграция определяет зависимости от других миграций и операций работы в базе данных для синхронизации с изменениями модели.

Давайте рассмотрим код SQL, который Django будет выполнять в базе данных, чтобы создать таблицу для нашей модели.

Команда `sqlmigrate` принимает имена миграции и возвращает их SQL без выполнения. Выполните следующую команду для проверки вывода SQL нашей первой миграции:

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

Результат должен выглядеть следующим образом:

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

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

Точный результат зависит от используемой вами базы данных. Предыдущий вывод создается для SQLite. Как мы видим, Django генерирует имена таблиц, комбинируя имя приложения и имя модели в нижнем регистре (`blog_post`), но вы также можете указать имя настраиваемой базы данных в классе модели `Meta`, используя атрибут `db_table`. Django автоматически создает первичный ключ для каждой модели, но вы также можете переопределить это, указав `primary_key=True` в одном из полей модели. Первичный ключ по умолчанию - это столбец `id`, состоящий из целого числа, которое автоматически увеличивается. Этот столбец соответствует полю `id`, которое автоматически добавляется к вашим моделям.

Давайте синхронизируем нашу базу данных с новой моделью. Выполните следующую команду для применения существующих миграций:

```
python manage.py migrate
```

Вы получите результат, который заканчивается следующей строкой:

```
Applying blog.0001_initial... ok
```

Мы просто применили миграцию для приложений, перечисленных в `INSTALLED_APPS`, включая наше приложение `blog`. После применения миграции база данных отражает текущий статус наших моделей.

Если вы редактируете файл `models.py`, чтобы добавить, удалить или изменить поля существующих моделей или добавить новые модели, вам нужно будет создать новую миграцию с помощью `makemigrations`. Миграция позволит Django отслеживать изменения модели. Затем вам придется применить ее с помощью команды `migrate`, чтобы синхронизировать базу данных с вашими моделями.

Создание сайта администрирования для ваших моделей

Теперь, когда у нас есть модель `Post`, мы создадим простой административный сайт для управления вашими сообщениями в блоге. Django поставляется со встроенным административным интерфейсом, который очень полезен для редактирования контента. Административный сайт Django создается динамически, читая метаданные модели и предоставляя готовый для производства интерфейс редактирования контента. Вы можете использовать его из коробки, настраивая его таким образом, как бы вы хотели, чтобы ваши модели отображались на нем.

Приложение `django.contrib.admin` уже включено в настройку `INSTALLED_APPS`, поэтому нам не нужно его добавлять.

Создание суперпользователя

Во-первых, нам нужно будет создать пользователя для управления сайтом администрирования. Выполните следующую команду:

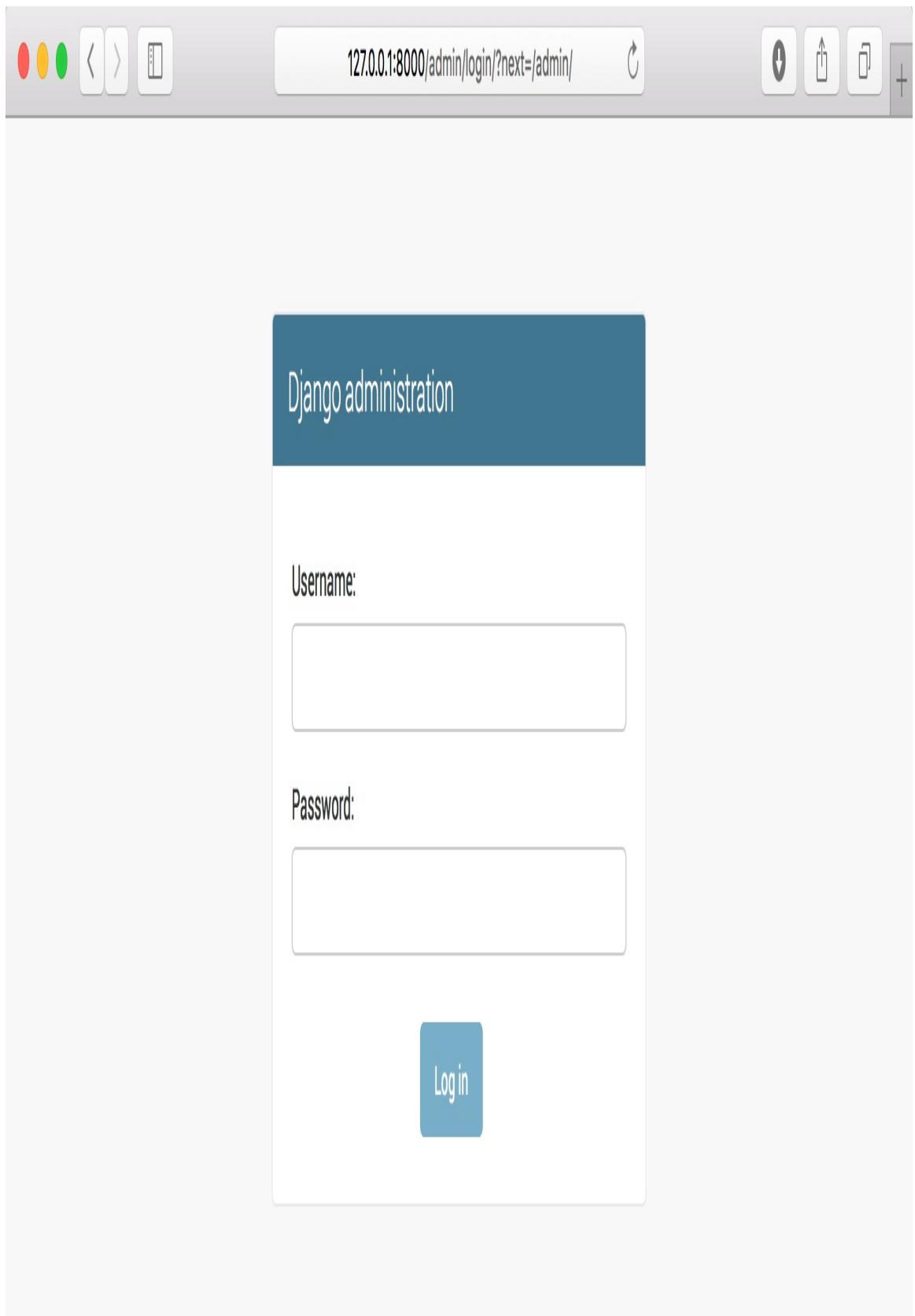
```
python manage.py createsuperuser
```

Вы увидите следующие строки (введите желаемое имя пользователя, адрес электронной почты и пароль, как показано ниже):

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

Сайт администрации Django

Теперь запустите сервер разработки с помощью команды `python manage.py runserver` и откройте `http://127.0.0.1:8000/admin/` в вашем браузере. Вы должны увидеть страницу входа в систему администратора, как показано на следующем снимке экрана:



Войдите в систему, используя учетные данные пользователя, созданного на предыдущем шаге. Вы увидите главную страницу сайта администрирования, как показано на следующем скриншоте:

The screenshot shows the Django administration interface. At the top, it says "Django administration" and "WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below this, the title "Site administration" is displayed. On the left, there is a sidebar titled "AUTHENTICATION AND AUTHORIZATION" containing links for "Groups" and "Users", each with "Add" and "Change" buttons. To the right, there are two sections: "Recent actions" (which is currently empty) and "My actions" (which also says "None available").

Модели `Group` и `User`, которые вы видели на предыдущем снимке экрана, являются частью фреймворка аутентификации Django, расположенного в `django.contrib.auth`. Если вы кликните по `Users`, вы увидите пользователя, которого вы создали ранее. Модель `Post` вашего приложения `blog` имеет связь с моделью `User`. Помните, что это отношение, определяемое полем `author`.

Добавление ваших моделей на сайт администрирования

Давайте добавим ваши модели блога на сайт администрирования. Отредактируйте файл `admin.py` вашего приложения `blog` следующим образом:

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

admin.site.register(Post)
```

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

The screenshot shows the Django administration interface. At the top, there's a header bar with "Django administration" on the left and "WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT" on the right. Below this is a "Site administration" heading. On the left, there's a sidebar with "AUTHENTICATION AND AUTHORIZATION" sections for "Groups" and "Users", each with "Add" and "Change" buttons. On the right, there's a "Recent actions" section which is currently empty. Below this is a "My actions" section with the message "None available". The main content area has a blue header "BLOG" containing a "Posts" section. The "Posts" section lists one item with "Add" and "Change" buttons.

BLOG	
Posts	+ Add Change

Это было легко, не так ли? Когда вы регистрируете модель на админ-сайте Django, вы получаете удобный интерфейс, сгенерированный путем интроспекции ваших моделей, который позволяет вам перечислить, отредактировать, создать и удалить объекты простым способом.

Нажмите ссылку Add (Добавить) рядом с Posts чтобы добавить новое сообщение. Вы получите форму для добавления, которую Django создала динамически из вашей модели, как показано на следующем скриншоте:

Django administration

WELCOME, ADMIN. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Blog > Posts > Add post

Add post

Title:

Slug:

Author:

-----  

Body:

Publish:

Date: 2017-12-14 

Time: 08:54:24 

Note: You are 2 hours ahead of server time.

Status:

Draft 

[Save and add another](#)

[Save and continue editing](#)

SAVE

Django использует различные виджеты форм для каждого типа поля. Даже сложные поля, такие как `DateTimeField`, отображаются с помощью простого интерфейса, например, для выбора даты в JavaScript.

Заполните форму и нажмите кнопку SAVE (Сохранить). Вы должны быть перенаправлены на страницу с только что созданным сообщением, как показано на следующем скриншоте:

Django administration

WELCOME, ADMIN. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home › Blog › Posts

✓ The post "Who was Django Reinhardt?" was added successfully.

Select post to change

[ADD POST](#) +

Action: 0 of 1 selected

POST

Who was Django Reinhardt?

1 post

Настраиваем способ отображения моделей

Теперь мы рассмотрим, как настроить сайт администратора. Отредактируйте файл `admin.py` Вашего приложения блог и измените его следующим образом:

```
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 Django, что наша модель зарегистрирована на сайте администратора, используя специальный класс, который наследует от `ModelAdmin`. В этом классе мы можем включить информацию о том, как отображать модель на сайте администратора и как взаимодействовать с ней. Атрибут `list_display` позволяет вам установить поля модели, которые вы хотите отобразить на странице списка объектов admin. Декоратор `@admin.register()` выполняет то же что и функция `admin.site.register()`, которую мы заменили.

Давайте настроим модель администратора с дополнительными параметрами, используя следующий код:

```
@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 сообщений. Теперь она будет выглядеть так:

Django administration

WELCOME, ADMIN. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Blog > Posts

Select post to change

[ADD POST](#) +

Q

« 2017 December 14 »

Action: 0 of 1 selected

<input type="checkbox"/>	TITLE	SLUG	AUTHOR	PUBLISH	2 ▲ STATUS	1 ▲
<input type="checkbox"/>	Who was Django Reinhardt?	who-was-django-reinhardt	admin	Dec. 14, 2017, 8:54 a.m.	Draft	

1 post

FILTER

By status

All

Draft

Published

By created

Any date

Today

Past 7 days

This month

This year

By publish

Any date

Today

Past 7 days

This month

This year

Вы можете видеть, что поля, отображаемые на странице списка сообщений, указаны в атрибуте `list_display`. Страница списка теперь включает правую боковую панель, которая позволяет фильтровать результаты по полям, включенными в атрибут `list_filter`. Поле Search появилось на странице. Это связано с тем, что мы определили список полей для поиска, используя `search_fields` атрибут. Ниже Search есть навигационные ссылки для навигации по иерархии дат: это было определено атрибутом `date_hierarchy`. Вы также можете видеть, что сообщения упорядочены по Status и Publish колонкам. Мы указали порядок по умолчанию, используя атрибут `ordering`.

Сейчас кликните на ссылку Add Post. Здесь вы также заметите некоторые изменения. Когда вы вводите заголовок нового сообщения, поле `slug` заполняется автоматически. Мы указали Django, чтобы поле `slug` с вводом поля `title` использовало `prepopulated_fields` атрибут. Кроме того, теперь поле `author` отображается с помощью виджета поиска, который может масштабироваться намного лучше, чем выпадающий список выбора, когда у вас тысячи пользователей, как показано на следующем снимке экрана:



В несколько строк кода мы настроили способ отображения нашей модели на сайте администратора. Существует множество способов настройки и расширения сайта администрирования Django. Вы узнаете об этом позже из этой книги.

Работа с QuerySet и менеджерами

Теперь, когда у вас есть полностью функциональный административный сайт для управления содержимым вашего блога, пришло время узнать, как извлекать информацию из базы данных и взаимодействовать с ней. Django поставляется с мощным API абстракции базы данных, который позволяет легко создавать, извлекать, обновлять и удалять объекты. Джанго **Object-relational mapper** совместим с MySQL, PostgreSQL, SQLite и Oracle. Помните, что вы можете определить базу данных вашего проекта в настройке `DATABASES` файла `settings.py`. Django может работать с несколькими базами данных одновременно, и вы можете программировать маршрутизаторы баз данных для создания настраиваемых схем маршрутизации.

После создания ваших моделей базы данных Django даст вам API для взаимодействия с ними. Вы можете найти ссылку на модель данных в официальной документации на

<https://docs.djangoproject.com/en/2.0/ref/models/>.

Создание объектов

Откройте терминал и запустите следующую команду:

```
python manage.py shell
```

Затем введите следующие строки:

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

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

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

Метод `get()` позволяет вам получить один объект из базы данных. Обратите внимание, что этот метод ожидает результат, соответствующий запросу. Если результат не будет возвращен базой данных, этот метод вызовет исключение `DoesNotExist`, а если база данных вернет более одного результата, будет возбуждено исключение `MultipleObjectsReturned`. Оба исключения - это атрибуты класса модели, к которым выполняется запрос.

Затем мы создаем экземпляр `Post` с пользовательским `title`, `slug` и `body`, а также мы устанавливаем пользователя, которого мы ранее получили в качестве автора сообщения:

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

Этот объект находится в памяти и не сохраняется в базе данных.

Наконец, мы сохраняем объект `Post` в базе данных с помощью метода `save()`:

```
post.save()
```

Предыдущее действие за кулисами выполняет оператор SQL `INSERT`. Мы видели, что сначала был создан объект в памяти, а затем он был сохранен в базе данных, но мы также можем создать объект и сохранить его в базе данных за одну операцию с помощью метода `create()` следующим образом:

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

Обновление объектов

Измените заголовок сообщения на другой и снова сохраните объект:

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

На этот раз метод `save()` выполняет оператор SQL `UPDATE`.

Изменения, внесенные вами в объект, не сохраняются в базе данных до тех пор, пока вы не вызовете метод `save()`.

Извлечение объектов

Django **object-relational mapping (ORM)** основан на `QuerySet`. `QuerySet` представляет собой набор объектов из вашей базы данных, который может иметь несколько фильтров для ограничения результатов. Вы уже знаете, как получить один объект из базы данных с помощью метода `get()`. Мы получили доступ к этому методу, используя `Post.objects.get()`. Каждая модель Django имеет как минимум один менеджер, который по умолчанию называется **objects**. Вы получаете объект `QuerySet`, используя диспетчер модели. Чтобы получить все объекты из таблицы, вы просто используете метод `all()` в диспетчере объектов по умолчанию, например:

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

Таким образом мы создаем `QuerySet`, который возвращает все объекты из базы данных. Обратите внимание, что этот `QuerySet` еще не выполнен. Django `QuerySet` - *ленивые (lazy)*; они выполняются только тогда, когда вынуждены. Такое поведение делает `QuerySet` очень эффективными. Если мы не установим `QuerySet` в переменную, а вместо этого напишем его непосредственно в консоли Python, выполнится оператор SQL `QuerySet`, потому что мы вынуждаем его выводить результаты:

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

Использование метода filter()

Чтобы отфильтровать QuerySet, вы можете использовать метод менеджера `filter()`. Например, мы можем получить все сообщения, опубликованные в 2017 году, используя следующий QuerySet:

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

Вы также можете фильтровать по нескольким полям. Например, мы можем получить все сообщения, опубликованные в 2017 году автором с именем пользователя `admin`:

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

Это эквивалентно построению одного и того же набора QuerySet с несколькими фильтрами:

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

Запросы с методами поиска в поле построены с использованием двух символов подчеркивания, например `publish__year`, но эти же обозначения используются также для доступа к полям связанных моделей, например `author__username`.

Использование exclude()

Вы можете исключить определенные результаты из своего QuerySet с помощью метода `exclude()`. Например, мы можем получить все сообщения, опубликованные в 2017 году, названия которых не начинаются с `why`:

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

Использование `order_by()`

Вы можете сортировать результаты в разных полях с помощью метода `order_by()`. Например, вы можете получить все объекты, упорядоченные по их `title`, следующим образом:

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

Подразумевается возрастающий порядок. Вы можете указать нисходящий порядок с префиксом в виде знака минус, например:

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

Удаление объектов

Если вы хотите удалить объект, вы можете сделать это из экземпляра объекта с помощью метода `delete()`:

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

Обратите внимание, что удаление объектов также удалит любые зависимые отношения для `ForeignKey`, определенные с помощью `on_delete`, установленного в `CASCADE`.

Когда выполняется QuerySets

Вы можете объединить столько фильтров в QuerySet, сколько захотите, и вы не удалите из базы данных до тех пор, пока не будет выполнен QuerySet. Он выполняется только в следующих случаях:

- Когда вы перебираете (итерируете) их
- Когда вы делаете их срез, например `Post.objects.all()[3:]`
- Когда вы сохраняете или кешируете их
- Когда вы вызываете `repr()` ИЛИ `len()` К НИМ
- Когда вы явно вызываете `list()` К НИМ
- Когда вы сравниваете их, например `bool()`, `or` , `and`, ИЛИ `if`

Создание менеджеров моделей

Как мы уже упоминали, `objects` является менеджером по умолчанию для каждой модели, которая извлекает все объекты из базы данных. Однако мы можем также определить пользовательские менеджеры для наших моделей. Мы создадим пользовательский менеджер для получения всех сообщений со статусом `published`.

Есть два способа добавить менеджер в свои модели: вы можете добавить дополнительные методы менеджера или изменить начальный менеджер `QuerySet`. Первый метод предоставляет вам `QuerySet API`, такой как `Post.objects.my_manager()`, а последний предоставляет вам `Post.my_manager.all()`. Менеджер позволит нам получать сообщения, используя `Post.published.all()`.

Отредактируйте файл `models.py` вашего приложения `blog` и добавте пользовательский менеджер:

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

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

Метод менеджера `get_queryset()` возвращает `QuerySet`, который будет выполнен. Мы переопределяем этот метод, чтобы включить наш настраиваемый фильтр в последний `QuerySet`.

Мы определили наш пользовательский менеджер и добавили его в модель `Post`; теперь мы можем использовать его для выполнения запросов. Давайте проверим это.

Выполните следующую команду:

```
python manage.py shell
```

Теперь вы можете получить все опубликованные сообщения, название которых начинается с `Who` используя следующую команду:

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

Создание представлений для списка и детального отображения

Теперь, когда у вас есть знания о том, как использовать ORM, вы готовы строить представления приложения для блога.

Представление Django - это просто функция Python, которая получает веб-запрос и возвращает веб-ответ. Вся логика для возвращаемого ответа содержится в представлении.

Во-первых, мы создадим наши представления, затем мы определим шаблон URL для каждого представления, и, наконец, мы создадим шаблоны HTML для визуализации данных, сгенерированных представлениями.

Создание представлений для списка и детального отображения сообщений

Начнем с создания представления для отображения списка сообщений. Измените файл `views.py` вашего приложения `blog` и добавьте в него следующее:

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

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

Вы только что создали свое первое представление Django. Представление `post_list` принимает объект `request` как единственный параметр. Помните, что этот параметр требуется для всех представлений. В этом представлении мы извлекаем все сообщения со статусом `published` с помощью менеджера `published`, который мы создали ранее.

Наконец, мы используем функцию `render()`, предоставленную Django, чтобы отобразить список сообщений с заданным шаблоном. Эта функция принимает объект `request`, путь к шаблону и контекстные переменные для визуализации данного шаблона. Она возвращает объект `HttpResponse` с визуализированным текстом (обычно, HTML-кодом). Функция `render()` учитывает контекст запроса, поэтому любая переменная, заданная шаблонами контекстных процессоров, доступна для данного шаблона. Контекстные процессоры шаблонов - это

просто вызывающие элементы, которые задают переменные в контексте. Вы узнаете, как их использовать в [главе 3, *Расширение приложения blog*](#).

Давайте создадим второе представление для отображения одного сообщения. Добавьте следующую функцию в файл `views.py`:

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

Это представление отображает подробную информацию о сообщении. Оно принимает параметры `year`, `month`, `day` и `post` для извлечения опубликованного сообщения с данным `slug` и датой. Обратите внимание, что когда мы создали модель `Post`, мы добавили параметр `unique_for_date` в поле `slug`. Таким образом, мы гарантируем, что будет только одна запись с `slug` на определенную дату, и, таким образом, мы сможем получить отдельные сообщения, используя дату и `slug`. В детальном представлении мы используем функцию `get_object_or_404()`, чтобы получить нужную запись. Эта функция возвращает объект, который соответствует указанным параметрам, или запускает исключение HTTP 404 (not found), если объект не найден. Наконец, мы используем функцию `render()` для отображения полученного сообщения с использованием шаблона.

Добавление шаблонов URL для ваших представлений

Шаблоны URL позволяют отображать URL-адреса в представлениях. Шаблон URL состоит из строкового шаблона, представления и, при необходимости, имени, которое позволяет вам указывать URL-адрес по всему проекту. Django просматривает каждый шаблон URL и останавливается на первом, который соответствует запрашиваемому URL. Затем Django импортирует представление соответствующего шаблона URL-адреса и выполняет его, передавая экземпляр класса `HttpRequest` и ключевое слово или позиционные аргументы.

Создайте файл `urls.py` в каталоге приложения `blog` и добавьте к нему следующие строки:

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

app_name = 'blog'

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

В предыдущем коде мы определяем пространство имен приложения с переменной `app_name`. Это позволяет нам упорядочивать URL-адреса приложения и использовать их при обращении к ним. Мы определяем два разных шаблона с помощью функции `path()`. Первый шаблон URL не принимает

никаких аргументов и сопоставляется с представлением `post_list`. Второй шаблон принимает следующие четыре аргумента и сопоставляется с представлением `post_detail`:

- `year`: Требуется целое число
- `month`: Требуется целое число
- `day`: Требуется целое число
- `post`: Может состоять из слова и дефиса

Мы используем угловые скобки для захвата значений из URL. Любое значение, указанное в шаблоне URL как `<parameter>`, фиксируется как строка. Мы используем преобразователи пути, такие как `<int:year>`, чтобы конкретно сопоставлять и возвращать целое число и `<slug:post>` который соответствует `slug` (строке, состоящей из букв или цифр ASCII, плюс символы дефиса и подчеркивания). Вы можете увидеть все преобразователи пути, предоставленные Django, на <https://docs.djangoproject.com/en/2.0/topics/http/urls/#path-converters>.

Если использование `path()` и конвертеров недостаточно для вас, вы можете использовать `re_path()` для определения сложных шаблонов URL с регулярным выражениями Python. Вы можете узнать больше об определении шаблонов URL с регулярными выражениями в https://docs.djangoproject.com/en/2.0/ref/urls/#django.urls.re_path. Если раньше вы не работали с регулярными выражениями, вы можете взглянуть на *Regular Expression HOWTO* находящееся по адресу <https://docs.python.org/3/howto/regex.html>

Создание файла `urls.py` для каждого приложения - лучший способ сделать ваши приложения повторно используемыми другими проектами.

Теперь вы должны включить шаблоны URL-адресов приложения `blog` в основной шаблон URL-адреса проекта.

Отредактируйте файл `urls.py` расположенный в каталоге `mysite` вашего проекта:

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

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

Новый шаблон URL, указанный в `include`, относится к шаблонам URL, определенным в приложении блога, так что они включены в путь `blog/`. Мы включаем эти шаблоны в пространство имен `blog`. Пространства имен должны быть уникальными для всего проекта. Позже мы будем легко ссылаться на наши URL-адреса блога, включая пространство имен, создавая их, например как, `blog:post_list` и `blog:post_detail`. Вы можете узнать больше о пространствах имен URL в <https://docs.djangoproject.com/en/2.0/topics/http/urls/#url-namespaces>.

Канонические URL-адреса для моделей

Вы можете использовать URL `post_detail`, который вы определили в предыдущем разделе, для создания канонического URL-адреса для объектов `Post`. Соглашение в Django заключается в добавлении метода `get_absolute_url()` к модели, которая возвращает канонический URL-адрес объекта. Для этого метода мы будем использовать метод `reverse()`, который позволяет создавать URL-адреса по их имени и передавать необязательные параметры. Измените файл `models.py` и добавьте следующее:

```
from django.urls import reverse

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

Мы используем метод `get_absolute_url()` в наших шаблонах для связи с конкретными сообщениями.

Создание шаблонов для ваших представлений

Мы создали представления и паттерны URL для приложения `blog`. Теперь пришло время добавить шаблоны для отображения сообщений удобным для пользователя способом.

Создайте в каталоге `blog` следующие каталоги и файлы:

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

Предыдущая структура будет файловой структурой для наших шаблонов. Файл `base.html` будет содержать основную структуру HTML-сайта и разделит контент на основную область содержимого и боковую панель. Файлы `list.html` и `detail.html` наследуют файл `base.html`, чтобы отобразить список сообщений блога и детальную информацию о сообщении соответственно.

Django имеет мощный язык шаблонов, который позволяет вам указать, как отображаются данные. Он основан на *template tags*, *template variables*, и *template filters*:

- Теги шаблона управляют отображением шаблона и выглядят как `{% tag %}`.
- Переменные шаблона заменяются значениями, когда шаблон визуализируется `{{ variable }}`.

- Фильтры шаблонов позволяют изменять переменные для отображения: `{{ variable|filter }}`.

Вы можете увидеть все встроенные теги и фильтры шаблонов в <https://docs.djangoproject.com/en/2.0/ref/templates/builtins/>.

Давайте отредактируем файл `base.html` и добавим следующий 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>
```

`{% load static %}` сообщает Django загружать теги шаблона `static`, предоставляемые приложением `django.contrib.staticfiles`, которое содержится в настройке `INSTALLED_APPS`. После его загрузки вы можете использовать `{% static %}` в этом шаблоне. С его помощью вы можете включить статические файлы, такие как файл `blog.css`, который вы найдете в коде этого примера в каталоге `static/` директории `blog`. Скопируйте каталог `static/` из кода, который поставляется вместе с этой главой, в такое же место вашего проекта, чтобы применить таблицы стилей CSS.

Вы можете видеть, что есть два тега `{% block %}`. Они говорят Django, что мы хотим определить блок в этой области.

Шаблоны, наследующие этот, могут заполнять блоки содержимым. Мы определили блок под названием `title` и блок, называемый `content`.

Давайте отредактируем файл `post/list.html` и сделаем его похожим на следующее:

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

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

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

С тегом шаблона `{% extends %}` мы указываем, что Django наследуется от шаблона `blog/base.html`. Затем мы заполняем блоки `title` и `content` базового шаблона своим контентом. Мы перебираем сообщения и показываем их название, дату, автора и тело, включая ссылку в заголовке на канонический URL сообщения. В теле сообщения мы применяем два шаблонных фильтра: `truncatewords` усекает значение до указанного количества слов и `linebreaks` преобразует вывод в разрывы строк HTML. Вы можете объединить столько фильтров шаблонов, сколько пожелаете; каждый из них будет применяться к выходу, сгенерированному предыдущим.

Откройте консоль и выполните команду `python manage.py runserver` для запуска сервера разработки. Откройте `http://127.0.0.1:8000/blog/` в вашем браузере, и вы увидите, что все работает. Обратите

внимание, что вам нужно иметь несколько сообщений со статусом Published чтобы показать их здесь. Вы должны увидеть что-то вроде этого:



Затем отредактируйте файл `post/detail.html`:

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

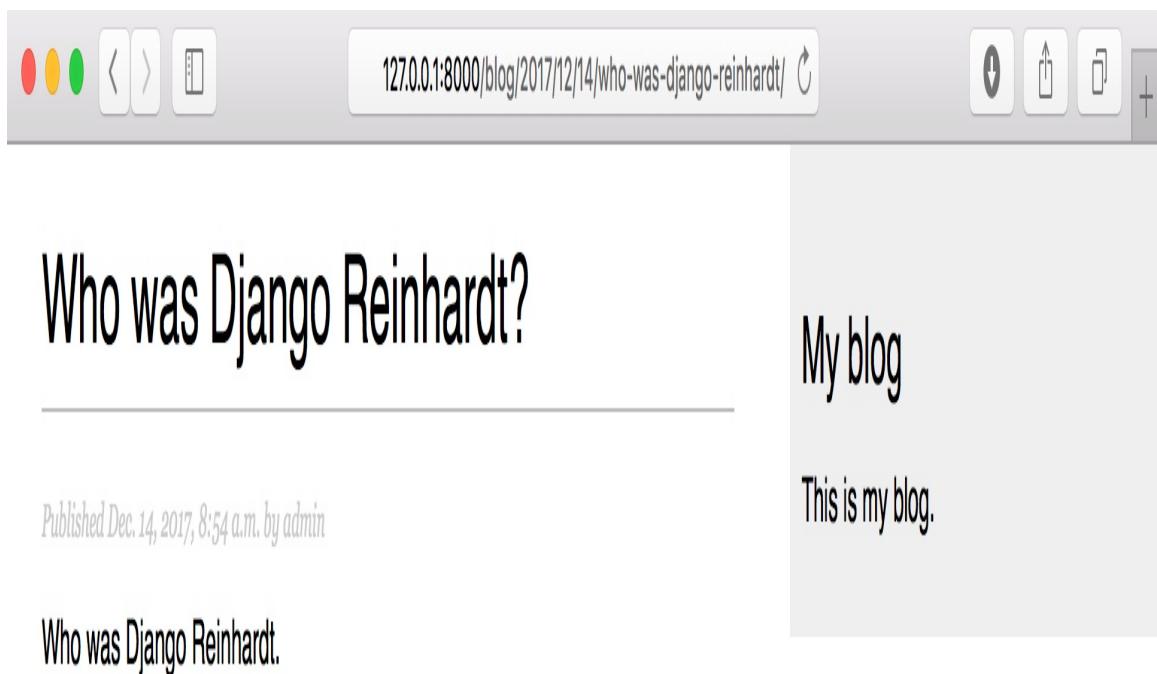
{% block title %}{{ post.title }}{% endblock %}

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


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


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

Сейчас вы можете вернуться к вашему браузеру и кликнуть по заголовку одного из сообщений чтобы отобразить детальную информацию об этом сообщении. Вы должны увидеть что-то похожее на:



Обратите внимание на URL - он должен быть следующего вида `/blog/2017/12/14/who-was-django-reinhardt/`. Мы создали SEO-дружественный URL для сообщений нашего приложения `blog`.

Добавление пагинации (разбиение на страницы)

Когда вы начнете добавлять контент в свой блог, вы скоро поймете, что вам нужно разбить список сообщений на несколько страниц. Django имеет встроенный класс разбиения на страницы, который позволяет легко управлять разбитыми на страницы данными.

Измените файл `views.py` приложения `blog`, чтобы импортировать классы paginator Django и измените представление `post_list`, как показано ниже:

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

def post_list(request):
    object_list = Post.published.all()
    paginator = Paginator(object_list, 3) # по 3 поста на каждой странице
    page = request.GET.get('page')
    try:
        posts = paginator.page(page)
    except PageNotAnInteger:
        # Если страница не является целым числом, отправляем первую страницу
        posts = paginator.page(1)
    except EmptyPage:
        # Если страница вне диапазона отправляем последнюю страницу
        # результатов
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                  {'page': page,
                   'posts': posts})
```

Вот как работает разбиение на страницы:

1. Мы создаем экземпляр класса `Paginator` с количеством объектов, которые мы хотим отобразить на каждой странице.
2. Мы получаем параметр `get`, который указывает текущий номер страницы.
3. Мы получаем объекты для желаемой страницы, вызывающие метод `page()` класса `Paginator`.
4. Если параметр `page` не является целым числом, мы получаем первую страницу результатов. Если этот параметр является числом больше чем последняя страница, мы будем извлекать последнюю страницу.
5. Мы передаем номер страницы и отправляем объекты в шаблон.

Теперь нам нужно создать шаблон для отображения страницы, чтобы он мог быть включен в любой шаблон, который использует разбиение на страницы. В папке `templates/` приложения `blog` создайте новый файл и назовите его `pagination.html`. Добавьте следующий HTML код в файл:

```
<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>
```

Шаблон разбиения на страницы ожидает объект `Page`, чтобы отображать предыдущие и следующие ссылки и отображать

текущую и конечную страницы результатов. Вернемся к шаблону `blog/post/list.html` и добавим шаблон `pagination.html` в нижнюю часть блока `{% content %}` следующим образом:

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

Поскольку объект `Page`, который мы передаем в шаблон, называется `posts`, мы включаем шаблон страницы в шаблон списка сообщений, передавая параметры для правильного отображения. Вы можете использовать этот метод для повторного использования шаблона пагинации с разбивкой на страницы разных моделей.

Теперь откройте `http://127.0.0.1:8000/blog/` в вашем браузере. Вы должны увидеть пагинацию внизу списка сообщений и иметь возможность перемещаться по страницам:

A screenshot of a web browser window displaying a blog. The address bar shows the URL `127.0.0.1:8000/blog/`. The main content area has a header "My Blog" and a post titled "Miles Davis favourite songs". A sidebar on the right contains the text "My blog" and "This is my blog.". Below the post are links to other posts and a page navigation indicator.

My Blog

Miles Davis favourite songs

Published Dec. 14, 2017, 10:01 p.m. by admin

Miles Dewey Davis III was an American jazz trumpeter, bandleader, and composer.

Notes on Duke Ellington

Published Dec. 14, 2017, 9:58 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.

Another post

Published Dec. 14, 2017, 11:45 a.m. by admin

Post body.

Page 1 of 2. [Next](#)

Использование базовых классов представлений

Представления на основе классов - это альтернативный способ реализации представлений как объектов Python вместо функций. Поскольку представление является вызываемым, которое принимает веб-запрос и возвращает веб-ответ, вы также можете определить свои представления как методы класса. Для этого Django предоставляет базовые классы представлений. Все они наследуются от класса `view`, который обрабатывает диспетчер HTTP-метода и другие общие функции.

Представления, основанные на классах, в некоторых случаях имеют преимущества перед функциональными представлениями. Они имеют следующие функции:

- Организация кода, связанного с HTTP-методами, например `get`, `post`, или `put`, в отдельных методах вместо использования условного разветвления
- Использование множественного наследования для создания многоразовых классов представлений (также известных как *mixins*)

Вы можете взглянуть на введение в представления на основе классов в <https://docs.djangoproject.com/en/2.0/topics/class-based-views/intro/>.

Мы заменим представление `post_list` на представление класса `ListView`, предлагаемого Django. Это базовое представление

позволяет вам перечислять объекты любого типа.

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

```
from django.views.generic import ListView

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

Это представление на основе класса аналогично предыдущему представлению `post_list`. В этом коде мы указываем `ListView` делать следующие вещи:

- Использовать определенный `QuerySet` вместо того, чтобы извлекать все объекты. Вместо определения атрибута `queryset` мы могли бы указать `model = Post` и Django построил бы общий `Post.objects.all()` `QuerySet` для нас.
- Использовать контекстную переменную `posts` для результатов запроса. По умолчанию используется переменная `object_list`, если мы не укажем `context_object_name`.
- Разбить результат, отображающий три объекта на страницу.
- Использовать наш собственный шаблон для отображения страницы. Если мы не укажем шаблон, `Listview` будет использовать шаблон по умолчанию `blog/post_list.html`.

Сейчас откройте файл `urls.py` вашего приложения `blog` и

закомментируйте строку относящуюся к `post_list` URL паттерну, вместо нее добавте новый URL паттерн использующий класс

`PostListView`:

```
urlpatterns = [
    # post views
    # 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 сохранения корректной работы пагинации, мы должны использовать правильный объект страницы передаваемый в шаблон. Класс Django `ListView` нашего представления передает пагинацию через переменную `page_obj`, поэтому мы должны отредактировать шаблон `post/list.html` используя правильную переменную:

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

Откройте `http://127.0.0.1:8000/blog/` в вашем браузере и проверте что все работает должным образом. Это простой пример представления на основе класса, которое использует базовый класс, предоставляемый Django. Вы узнаете больше о представлениях на основе классов в [главе 10, Создание платформы электронного обучения](#), и последующих главах.

Резюме

В этой главе вы изучили базовые основы веб-фреймворка Django, создавая основу приложения blog. Вы спроектировали модель данных и провели миграции для вашего проекта. Вы создали представления, шаблоны, пагинацию и URL для вашего болога.

В следующей главе вы изучите как расширить ваш блог с помощью системы комментариев и добавить функциональности в виде тегов, а также ваши пользователи смогут рассылать сообщения по email.

Глава 2

Улучшение вашего блога с помощью расширенного функционала

В предыдущей главе вы создали базовое приложение для блогов. Теперь вы превратите свое приложение в полнофункциональный блог с расширенными функциями, такими как совместное использование сообщений по электронной почте, добавление комментариев, пометка сообщений и получение похожих сообщений. В этой главе вы узнаете следующие темы:

- Отправка электронной почты с помощью Django
- Создание форм и обработка их в представлениях
- Создание форм из моделей
- Интеграция сторонних приложений
- Создание составных QuerySet

Обмен сообщениями по электронной почте

Во-первых, мы разрешим пользователям делиться сообщениями, отправляя электронные письма. Потратите немного времени, чтобы подумать, как вы будете использовать *представления, URLs, и шаблоны* чтобы создать эту функциональность, на основе тех знаний, что вы получили в предыдущей главе. Теперь узнайте, что вам нужно, чтобы пользователи могли отправлять сообщения по электронной почте. Вам нужно будет сделать следующее:

- Создать форму для пользователей, в которую вводиться имя и адрес электронной почты получателя и необязательные комментарии
- Создать представление в файле `views.py`, которое обрабатывает введенные данные и отправляет электронную почту
- Добавить URL паттерн для нового представления в файл `urls.py` приложения `blog`
- Создать шаблон для отображения формы

Создание форм в Django

Начнем с создания формы для обмена сообщениями. Django имеет встроенный фреймворк форм, который позволяет легко создавать формы. Фреймворк форм позволяет определить поля вашей формы, указать, как они должны отображаться, и указать, как они должны проверять входные данные. Фреймворк форм Django предлагает гибкий способ визуализации форм и обработки данных.

Django поставляется с двумя базовыми классами для создания форм:

- `Form`: Позволяет создавать стандартные формы
- `ModelForm`: Позволяет создавать формы, на основе экземпляра модели.

Для начала создайте файл `forms.py` внутри директории вашего приложения `blog` и добавьте в него следующее:

```
from django import forms

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

Это ваша первая Django форма. Взгляните на код. Мы создали форму, наследуя базовый класс `Form`. Мы используем разные типы полей Django и их проверку соответственно.

*Формы могут находиться где угодно в вашем проекте Django.
Соглашение предлагает размещать их внутри файла `forms.py` вашего
приложения.*

Поле `name` типа `CharField`. Этот тип поля отображается как HTML элемент `<input type="text">`. Каждый тип поля имеет виджет по умолчанию, который определяет, как поле отображается в HTML. Виджет по умолчанию может быть переопределен в атрибуте `widget`. В поле `comments`, мы используем виджет `Textarea` для отображения как HTML `<textarea>` вместо значения по умолчанию `<input>`.

Проверка поля также зависит от его типа. Например, `email` и `to` принадлежат к типу `EmailField`. Оба поля требуют действительного адреса электронной почты, в противном случае проверка поля вызовет исключение `forms.ValidationError`, и форма не будет проверяться. Другие параметры также учитываются при проверке формы: мы определяем максимальную длину 25 символов для поля `name` и создаем поле `comments` с опцией `required=False`. Все это также учитывается при проверке поля. Типы полей, используемые в этой форме, представляют собой только небольшую часть полей формы Django. Чтобы получить список всех доступных полей формы, вы можете посетить <https://docs.djangoproject.com/en/2.0/ref/forms/fields/>.

Обработка форм в представлении

Вы должны создать новое представление, которое обрабатывает форму и в случае успеха отправляет электронное письмо. Откройте файл `views.py` вашего приложения `blog` и добавьте следующий код:

```
from .forms import EmailPostForm

def post_share(request, post_id):
    # Получаем сообщение по его id
    post = get_object_or_404(Post, id=post_id, status='published')

    if request.method == 'POST':
        # Форма была отправлена
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Поля формы прошли проверку
            cd = form.cleaned_data
            # ... отправляем email
    else:
        form = EmailPostForm()
    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form': form})
```

Это представление работает следующим образом:

- Мы определяем представление `post_share` которое принимает в качестве параметров объект `request` и переменную `post_id`.
- Мы используем функцию `get_object_or_404()` для получения сообщения по его идентификатору и проверяем имеет

ли получаемый пост статус `published` (опубликовано).

- Мы используем одно и то же представление для отображения начальной формы и обработки введенных данных. Мы различаем, была ли форма уже отображена или нет на основе `request` метода и отправляем форму, используя `POST`. Если мы получаем запрос методом `GET`, должна быть отображена пустая форма, а если мы получим запрос методом `POST`, значит форма была отправлена и соответственно должна быть обработана. Поэтому мы используем `request.method == 'POST'` чтобы провести различие между двумя сценариями.

Ниже представлен процесс отображения и обработки формы:

1. Когда сначала загружается представление с `GET` запросом, мы создаём экземпляр новой формы `form`, который будет использоваться для отображения пустой формы в шаблоне:

```
form = EmailPostForm()
```

2. Пользователь заполняет форму и отправляет ее методом `POST`. Затем мы создаем экземпляр формы, используя предоставленные данные, которые содержатся в `request.POST`:

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

3. После этого мы проверяем предоставленные данные, используя `is_valid()` метод. Этот метод проверяет данные, введенные в форму и возвращает `True` если все поля содержат достоверные данные. Если какое-либо поле содержит недопустимые данные, тогда `is_valid()` возвращает `False`. Вы можете увидеть список ошибок проверки, обратившись к `form.errors`.
4. Если форма недействительна, мы снова визуализируем форму в шаблоне с предоставленными данными. Мы будем отображать ошибки проверки в шаблоне.
5. Если форма действительна, мы получаем подтверждение доступа к данным `form.cleaned_data`. Этот атрибут представляет собой словарь полей формы и их значений.

Если данные формы не проверяются, `cleaned_data` будет содержать только действительные поля.

Чтобы собрать все вместе, давайте узнаем, как отправлять электронные письма с помощью Django.

Отправка электронной почты с помощью Django

Отправка электронной почты с помощью Django довольно проста. Во-первых, вам нужно будет иметь локальный SMTP-сервер или определить конфигурацию внешнего SMTP-сервера, добавив следующие параметры в файл `settings.py` вашего проекта:

- `EMAIL_HOST`: Хост SMTP сервера; по умолчанию используется `localhost`
- `EMAIL_PORT`: SMTP порт; по умолчанию 25
- `EMAIL_HOST_USER`: Username для SMTP сервера
- `EMAIL_HOST_PASSWORD`: Password для SMTP сервера
- `EMAIL_USE_TLS`: Использовать ли безопасное соединение TLS
- `EMAIL_USE_SSL`: Использовать скрытое безопасное соединение TLS

Если у вас нет возможности использовать SMTP сервер, вы можете указать Django выводить письма в консоль, добавив следующий параметр в `settings.py` файл:

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

Используя этот параметр, Django выведет все письма в консоль. Это будет полезно для тестирования вашего приложения без

SMTP-сервера.

Если вы хотите отправлять электронные письма, но у вас нет локального SMTP-сервера, вы, можете использовать SMTP-сервер своего поставщика услуг электронной почты.

Следующая примерная конфигурация действительна для отправки электронной почты через серверы Gmail с помощью учетной записи Google:

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

Запускаем `python manage.py shell` комманду и в открытой консоли Python отправляем электронное письмо следующим образом:

```
>>> 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)
```

Эта `send_mail()` функция принимает тему, сообщение, отправителя и список получателей в качестве необходимых аргументов. Установив необязательный аргумент `fail_silently=False`, мы говорим, что должно быть возбуждено исключение, если письмо не может быть отправлено правильно. Если вывод, который вы видите, 1, то ваше письмо было успешно отправлено.

Если вы отправляете письма Gmail с предыдущей конфигурацией, вам может потребоваться разрешить доступ для менее защищенных приложений на <https://myaccount.google.com/lesssecureapps>, следующим образом:

Some apps and devices use less secure sign-in technology, which makes your account more vulnerable. You can **turn off** access for these apps, which we recommend, or **turn on** access if you want to use them despite the risks. [Learn more](#)

Allow less secure apps: ON



Теперь мы добавим эту функциональность в наше представление.

Отредактируем представление `post_share` в файле `views.py` приложения `blog` следующим образом:

```
from django.core.mail import send_mail

def post_share(request, post_id):
    # Получаем сообщение по его id
    post = get_object_or_404(Post, id=post_id, status='published')
    sent = False

    if request.method == 'POST':
        # Форма для отправки
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Поля формы прошли проверку
            cd = form.cleaned_data
            post_url = request.build_absolute_uri(
                post.get_absolute_url())
            subject = '{0} ({1}) recommends you reading "'
            subject += '{2}"'.format(cd['name'], cd['email'], post.title)
            message = 'Read "{0}" at {1}\n\n{2}'\
                .format(post.title, post_url, cd['name'], cd['comments'])
            send_mail(subject, message, 'admin@myblog.com',
                      [cd['to']])
            sent = True
    else:
        form = EmailPostForm()
```

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

Мы объявляем переменную `sent` и устанавливаем ее значение `True` когда сообщение отправлено. Мы будем использовать эту переменную позже, в шаблоне для отображения успешного сообщения, когда форма будет успешно отправлена. Поскольку мы должны включить ссылку на сообщение в электронное письмо, мы получаем абсолютный путь сообщения, используя его `get_absolute_url()` метод. Мы используем этот путь как вход для `request.build_absolute_uri()` для создания полного URL-адреса, включая HTTP-схему и имя хоста. Мы создаем тему и тело сообщения электронной почты, используя очищенные данные подтвержденной формы и, наконец, отправляем электронное письмо на адрес электронной почты, содержащийся в поле формы `to`.

Теперь, когда ваше представление создано, не забудьте добавить для него новый паттерн URL. Откройте файл `urls.py` вашего приложения `blog` и добавьте URL паттерн `post_share`, следующим образом:

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

Отображение форм в шаблонах

После создания формы, программирования представления и добавления шаблона URL-адреса, мы должны создать шаблон для этого представления. Создайте новый файл в директории `blog/templates/blog/post/` с именем `share.html` и добавте в него следующий код:

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

{% block title %}Share a post{% endblock %}

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

Это шаблон для отображения формы или сообщения об успешном завершении при её отправке. Как вы могли заметить, мы создаем элемент HTML формы, указывая, что она должна быть отправлена методом `POST`:

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

Затем мы включаем фактический экземпляр формы. Мы сообщаем Django, отобразить её поля в HTML-абзаце `<p>` с помощью `as_p` метода. Мы также можем отобразить форму как неупорядоченный список с `as_ul` или как HTML таблицу с `as_table`. Если мы хотим отобразить каждое поле, мы также можем перебирать поля, как в следующем примере:

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

Тег шаблона `{% csrf_token %}` вводит скрытое поле с автогенерированным токеном, чтобы избежать **атаку межсайтового поддельного запроса (CSRF)**. Эти атаки состоят из вредоносного веб-сайта или программы, выполняющих нежелательные действия для пользователя на вашем сайте. Вы можете найти дополнительную информацию об этом на [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)).

Предыдущий тег генерирует скрытое поле, которое выглядит так:

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

По умолчанию Django проверяет токен CSRF во всех POST запросах. Помните, что вы должны включать `csrf_token` во всех формах, которые отправляются через POST.

Откройте шаблон `blog/post/detail.html` и добавьте следующую ссылку на URL-адрес после переменной `{{ post.body|linebreaks }}`:

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

Помните, что мы динамически создаем URL-адрес, используя шаблонный тег `{% url %}`, предоставленный Django. Мы используем пространство имен, называемое `blog` и URL названный `post_share`, мы также передаем идентификатор сообщения в качестве параметра для создания абсолютного URL-адреса.

Теперь запустите сервер разработки с помощью `python manage.py runserver` и откройте `http://127.0.0.1:8000/blog/` в вашем браузере. Нажмите на заголовок любого сообщения, чтобы перейти на страницу с его детальной информацией. Под телом сообщения вы должны увидеть ссылку, которую мы только что добавили, как показано на следующем скриншоте:

Notes on Duke Ellington

Published Dec. 14, 2017, 9:58 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.

[Share this post](#)

My blog

This is my blog.

Нажмите на `Share this post`, и вы должны увидеть страницу, которая включает форму, чтобы отправить это сообщение по электронной почте:

Share "Notes on Duke Ellington" by e-mail

Name:

Email:

To:

Comments:

SEND E-MAIL

My blog

This is my blog.

Стили CSS для формы включены в код примера в файле static/css/blog.css. Когда вы нажмете на кнопку SEND E-MAIL , форма отправиться и будет проверенна. Если все поля содержат достоверные данные, вы получите сообщение об успешном завершении:

E-mail successfully sent

"Notes on Duke Ellington" was successfully sent to account@gmail.com.

My blog

This is my blog.

Если вы введете неверные данные, вы увидите, что форма снова отобразиться, выводя все ошибки проверки:

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

My blog

This is my blog.

Обратите внимание, что некоторые современные браузеры не дадут вам отправить форму с пустыми или ошибочными полями. Это происходит из-за проверки формы, выполняемой браузером на основе типов полей и ограничений для каждого поля. В этом случае форма не будет отправлена, и браузер отобразит сообщение об ошибке для неправильных полей.

Наша форма для обмена сообщениями по электронной почте теперь завершена. Давайте создадим систему комментариев для нашего блога.

Создание системы комментариев

Сейчас мы создадим систему комментариев для блога, в которой пользователи смогут комментировать посты. Чтобы создать систему комментариев, вам необходимо выполнить следующие действия:

1. Создать модель для хранения комментариев
2. Создать формы для отправки комментариев и проверки введенных данных
3. Добавить представление, которое обрабатывает форму и сохраняет новый комментарий в базе данных
4. Изменить шаблон деталей поста (detail.html), чтобы отобразить список комментариев и форму, чтобы добавить новый комментарий

Сначала давайте создадим модель для хранения комментариев. Откройте файл `models.py` вашего приложения `blog` и добавьте следующий код:

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

```
class Meta:  
    ordering = ('created',)  
  
    def __str__(self):  
        return 'Comment by {} on {}'.format(self.name, self.post)
```

Это наша модель `Comment`. Она содержит `ForeignKey` для связи комментария с одним сообщением. Это отношение «много-к-одному» определено в модели `Comment` потому что каждый комментарий будет оставлен к одном посту, а каждый пост может иметь несколько комментариев. Переменная `related_name` позволяет нам назвать атрибут, который мы используем для обращения от связанного объекта к этому. Указав это, мы можем получить сообщение для объекта комментария, используя `comment.post` и получить все комментарии к сообщению, используя `post.comments.all()`. Если вы не определите атрибут `related_name`, Django будет использовать имя модели в нижнем регистре, с последующим `_set` (то есть, `comment_set`) для того чтобы вызвать менеджер обратно связанного объекта.

Вы можете больше узнать о взаимоотношениях «много-к-одному» в

https://docs.djangoproject.com/en/2.0/topics/db/examples/many_to_one/.

Мы добавили логическое (`boolean`) поле `active`, которое будем использовать для ручной деактивации неуместных комментариев. Мы используем поле `created` для сортировки комментариев по умолчанию в хронологическом порядке.

Новая, только что созданная модель, `Comment` еще не синхронизирована с базой данных. Выполните следующую команду для создания новой миграции:

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

Вы должны увидеть следующий вывод:

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

Django создаст `0002_comment.py` файл внутри директории `migrations/` вашего приложения `blog`. Для применения существующих миграций выполните следующую команду:

```
python manage.py migrate
```

Вы получите вывод, который включает следующую строку:

```
Applying blog.0002_comment... ok
```

Мы только что создали миграцию, и теперь таблица `blog_comment` существует в базе данных.

Сейчас мы можем добавить нашу новую модель на сайт администрирования, чтобы управлять комментариями через простой интерфейс. Откройте файл `admin.py` приложения `blog`, импортируйте модель `Comment`, и добавьте класс `ModelAdmin`:

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

Запустите сервер разработки с помощью команды `python manage.py runserver` И откроите `http://127.0.0.1:8000/admin/` в вашем браузере. Вы должны увидеть новую модель включенной в секцию `BLOG`, как показано на следующем скриншоте:

BLOG

[Comments](#)

 Add  Change

[Posts](#)

 Add  Change

Модель теперь зарегистрирована на сайте администратора, и мы можем управлять `Comment` с использованием простого интерфейса.

Создание форм из моделей

Мы создадим форму, чтобы наши пользователи могли комментировать записи в блогах. Помните, что Django имеет два базовых класса для создания форм, `Form` и `ModelForm`. Вы использовали первый ранее, чтобы ваши пользователи могли отправлять сообщения по электронной почте. В данном случае мы будем использовать `ModelForm` потому что нам нужна динамически созданная форма из модели `Comment`. Откройте файл `forms.py` вашего приложения `blog` и добавьте следующие строки:

```
from .models import Comment

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

Чтобы создать форму из модели, нам просто нужно указать, какую модель использовать для построения формы в `Meta` классе формы. Django исследует модель и динамически формирует форму для нас. Каждый тип поля модели имеет по умолчанию соответствующий тип поля формы. Способ определения полей модели принимается во внимание при проверке формы. По умолчанию Django создает поле формы для каждого поля, содержащегося в модели. Однако вы можете явно указать какие поля вы хотите включить в свою форму, используя список `fields` или определить, какие поля вы хотите исключить, используя список полей `exclude`. Для нашей формы `CommentForm`, мы будем использовать `name`, `email`, и `body` потому что это единственныe поля, которые смогут заполнить наши пользователи.

Работа с ModelForms в представлениях

Мы будем использовать представление подробных сведений о публикации (`post_detail`), чтобы создать и обработать форму. Отредактируйте файл `views.py` и добавьте импорт для модели `Comment` и для формы `CommentForm`, а также измените представление `post_detail` чтобы он выглядел следующим образом:

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

def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post, slug=post,
                           status='published',
                           publish__year=year,
                           publish__month=month,
                           publish__day=day)

    # Список активных комментариев к этой записи
    comments = post.comments.filter(active=True)

    new_comment = None

    if request.method == 'POST':
        # Комментарий отправлен
        comment_form = CommentForm(data=request.POST)
        if comment_form.is_valid():
            # Создать объект комментариев, но еще не сохранять его в базе
            # данных
            new_comment = comment_form.save(commit=False)
            # Связать текущий пост с комментарием
            new_comment.post = post
            # Сохранить комментарий в базе данных
            new_comment.save()
    else:
        comment_form = CommentForm()
    return render(request,
                  'blog/post/detail.html',
```

```
{'post': post,
 'comments': comments,
 'new_comment': new_comment,
 'comment_form': comment_form})
```

Давайте рассмотрим, что мы добавили к нашему представлению. Мы используем `post_detail` чтобы просмотреть сообщение и его комментарии. Мы создали `QuerySet` для получения всех активных комментариев для этого сообщения, как показано ниже.:

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

Мы создаем этот `QuerySet`, начиная с объекта `post`. Мы используем менеджер для связанных объектов, которые мы определили как `comments` используя атрибут `related_name` для связи с моделью `Comment`.

Мы также используем это же представление, чтобы наши пользователи могли добавить новый комментарий. Поэтому мы инициализируем переменную `new_comment`, установив ее в `None`. Мы будем использовать эту переменную при создании нового комментария. Мы создаем экземпляр формы с `comment_form = CommentForm()` если представление вызывается `GET` запросом. Если запрос выполняется через `POST`, мы создаем форму с использованием предоставленных данных и проверяем ее, используя метод `is_valid()`. Если форма недействительна, мы визуализируем шаблон с ошибками проверки. Если форма действительна, мы делаем следующие:

1. Мы содаем новый объект `Comment`, вызваной формы методом `save()` и присваеваем его переменной `new_comment` следующим образом:

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

Метод `save()` создает экземпляр модели, с которой связана форма, и сохраняет ее в базе данных. Если вы его вызовете, с параметром `commit=False`, вы создадите экземпляр модели, но вы еще не сохраните его в базе данных. Это происходит когда вы изменяете объект, прежде чем сохранить его, что мы и сделаем дальше.

Метод `save()` доступен для `ModelForm` но не для `Form` экземпляров, поскольку те не связаны ни с одной моделью.

2. Мы присвоим текущую запись комментарию, который мы только что создали:

```
new_comment.post = post
```

Делая это, мы указываем, что новый комментарий принадлежит этому сообщению.

3. Наконец, мы сохраняем новый комментарий в базе данных, вызывая его метод `save()`:

```
new_comment.save()
```

Теперь наше представление готово отображать и обрабатывать новые комментарии.

Добавление комментариев к шаблону детального отображения сообщения

Мы создали функционал для управления комментариями поста. Теперь нам нужно будет адаптировать наш шаблон `post/detail.html` для выполнения следующих действий:

- Отобразить общее количество комментариев для сообщения
- Отобразить список комментариев
- Отобразить форму для добавления нового комментария

Во-первых, мы добавим общее количество комментариев. Откройте шаблон `post/detail.html` и добавьте следующий код в блок `content`:

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

Мы используем Django ORM в шаблоне, выполняя `QuerySet comments.count()`. Обратите внимание, что язык шаблонов Django не использует круглые скобки для вызова методов. Тег `{% with %}` позволяет присвоить значение новой переменной, которая будет доступна для использования до тега `{% endwith %}`.

Тег шаблона `{% with %}` полезен, чтобы избежать доступа к базе данных или вызова дорогостоящего метода несколько раз.

Мы используем шаблонный фильтр `pluralize` для отображения множественного суффикса для слова `comment`, в зависимости от значения `total_comments`. Фильтры шаблонов принимают значение переменной, к которой они применяются, и возвращают вычисленное значение. Мы обсудим шаблонные фильтры в [главе 3, "Расширение вашего приложения blog"](#).

Шаблонный фильтр `pluralize` возвращает строку с литерой "s" если значение отличается от 1. Предыдущий текст будет отображаться как *0 comments*, *1 comment*, или *N comments*. Django включает множество шаблонных тегов и фильтров, которые помогают отображать информацию так, как вы хотите.

Теперь давайте добавим список комментариев. Добавьте следующие строки в шаблон `post/detail.html` ниже предыдущего кода:

```
{% 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 %}
```

Мы используем шаблонный тег `{% for %}` чтобы просмотреть комментарии. Мы выводим сообщение по умолчанию, если список `comments` пустой, информировав наших пользователей, что комментариев к этому сообщению пока нет. Мы перечисляем комментарии в переменной `{{ forloop.counter }}`, которая содержит счетчик циклов на каждой итерации. Затем мы показываем имя пользователя, разместившего комментарий, дату и тело комментария.

Наконец, вам нужно отобразить форму или отобразить успешное сообщение, если оно будет успешно отправлено. Добавьте следующие строки чуть ниже предыдущего кода:

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

Код довольно прост: если `new_comment` объект существует, мы показываем успешное сообщение, потому что комментарий был успешно создан. В противном случае мы предоставляем форму с элементом абзаца `<p>` для каждого поля и включаем токен CSRF, необходимый для `POST` запроса. Откройте <http://127.0.0.1:8000/blog/> в вашем браузере и нажмите на заголовок сообщения, чтобы просмотреть страницу с его подробностями. Вы увидите что-то вроде следующего снимка экрана:

Notes on Duke Ellington

Published Dec. 14, 2017, 9:58 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.

[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.

Добавьте пару комментариев, используя форму. Они должны появиться под вашим постом в хронологическом порядке, а именно:

2 comments

Comment 1 by Antonio Dec. 14, 2017, 10:08 p.m.

It's very interesting.

Comment 2 by Bienvenida Dec. 14, 2017, 10:09 p.m.

I didn't know that.

Откройте <http://127.0.0.1:8000/admin/blog/comment/> в вашем браузере. Вы увидите страницу администратора со списком созданных вами комментариев. Нажмите на один из них, чтобы отредактировать его, снимите флажок Active, и кликните кнопку Save. Вы снова будете перенаправлены к списку комментариев, и Active столбец отобразит неактивную иконку для комментария. Он должен выглядеть как первый комментарий на следующем скриншоте:

Select comment to change

Action: 0 of 2 selected

<input type="checkbox"/>	NAME	EMAIL	POST	CREATED	▲	ACTIVE
<input type="checkbox"/>	Antonio	user1@gmail.com	Notes on Duke Ellington	Aug. 25, 2017, 5:08 p.m.		
<input type="checkbox"/>	Bienvenida	user2@gmail.com	Notes on Duke Ellington	Aug. 25, 2017, 5:08 p.m.		

2 comments

Если вы вернетесь к детальному отображению сообщения, вы заметите, что удаленный комментарий больше не выводиться; и он не учитывается в общем количестве комментариев. Благодаря `active` полю, вы можете деактивировать неуместные комментарии и не показывать их в своих сообщениях.

Добавление функциональности тегов

После внедрения системы комментариев вы создадите способ пометить наши сообщения. Вы сделаете это, интегрировав стороннее приложение для тегов в наш проект. Модуль `django-taggit` является многоразовым приложением, которое в первую очередь предлагает вам модель и менеджер `Tag`, чтобы легко добавлять теги в любую модель. Вы можете посмотреть исходный код на <https://github.com/alex/django-taggit>.

Во-первых, вам нужно будет установить `django-taggit` через `pip` выполнив следующую команду:

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

Затем откройте файл `settings.py` проекта `mysite` и добавьте `taggit` к вашему `INSTALLED_APPS`, следующим образом:

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

Откройте файл `models.py` вашего приложения `blog` и добавьте менеджер `TaggableManager` предоставленный `django-taggit` к модели `Post` используя следующий код:

```
from taggit.managers import TaggableManager

class Post(models.Model):
    # ...
```

```
tags = TaggableManager()
```

Этот менеджер `tags` позволит вам добавлять, извлекать и удалять теги из объектов `Post`.

Выполните следующую команду для создания миграции для изменённой модели:

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

Вы должны получить следующий результат:

```
Migrations for 'blog':  
  blog/migrations/0003_post_tags.py  
    - Add field tags to post
```

Теперь запустите следующую команду для создания необходимых таблиц базы данных для модели `django-taggit` и чтобы синхронизировать изменения модели:

```
python manage.py migrate
```

Вы увидите результат, указывающий, что миграция была применена, как показано ниже:

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

Теперь ваша база данных готова к использованию модели `django-taggit`. Давайте изучим как использовать `tags` менеджер. Откройте оболочку командой `python manage.py shell` и введите следующий код; во-первых, мы найдем одно из наших сообщений (оно с 1 ID):

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

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

Примечание переводчика:

Если у вас появилось сообщение об ошибке, то укажите другой идентификатор, например id=2

Затем добавьте к нему теги и извлеките их, чтобы проверить, были ли они успешно добавлены:

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

Наконец, удалите тег и снова проверьте список тегов:

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

Это было легко, не так ли? Запустите команду `python manage.py runserver` для запуска сервера разработки и откройте `http://127.0.0.1:8000/admin/taggit/tag/` в вашем браузере. Вы должны увидеть административную страницу со списком объектов `Tag` приложения `taggit`:

Django administration

WELCOME, ADMIN. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Taggit > Tags

Select Tag to change

[ADD TAG](#) +

<input type="text"/> <input type="button" value="Search"/>			
Action:	NAME	SLUG	
<input type="checkbox"/>	django	django	
<input type="checkbox"/>	jazz	jazz	
<input type="checkbox"/>	music	music	

3 Tags

Перейдите к `http://127.0.0.1:8000/admin/blog/post/` и кликните по сообщению для его редактирования. Вы увидите, что в сообщение теперь входит новое поле Tags, в котором вы можете легко редактировать теги:

Tags:

jazz, music

A comma-separated list of tags.

Теперь мы отредактируем сообщения, чтобы показывать теги. Откройте шаблон `blog/post/list.html` и добавьте HTML код ниже заголовка (title) сообщения:

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

Этот шаблонный фильтр `join` работает как метод Python для строк `join()`, объединяя элементы с заданной строкой. Откройте `http://127.0.0.1:8000/blog/` в вашем браузере. Вы должны увидеть список тегов под каждым заголовком сообщения:

Who was Django Reinhardt?

Tags: jazz, music

Published Dec. 14, 2017, 8:54 a.m. by admin

Теперь мы будем редактировать наше представление `post_list` чтобы перечислить все сообщения, помеченные определенным тегом. Откройте файл `views.py` нашего приложения `blog` и импортируйте модель `Tag` из `django-taggit`, а также измените представление `post_list` для выборочной фильтрации сообщений по тегу, следующим образом:

```
from taggit.models import Tag

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

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

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

Наше `post_list` представление теперь работает следующим образом:

1. Оно принимает необязательный параметр `tag_slug`, который имеет значение `None` по умолчанию. Этот параметр появится в URL-адресе.
2. Внутри представления мы создаем исходный `QuerySet`, извлекаем все опубликованные сообщения и, если есть определенный тег, мы получаем объект `Tag` с данным `slug`, используя функцию `get_object_or_404()`.
3. Затем мы отфильтруем из списка сообщений те, которые содержат данный тег. Поскольку это отношение «многие ко многим», мы должны фильтровать теги, содержащиеся в данном списке, который в нашем случае содержит только один элемент.

Помните что `QuerySets` ленивый. Запросы `QuerySets` для извлечения сообщений будут выполняться только тогда, когда мы возвращаем список сообщений при визуализации шаблона.

Наконец, измените функцию `render()` в нижней части представления, чтобы передать переменную `tag` в шаблон.

Представление должно выглядеть следующим образом:

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

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

    paginator = Paginator(object_list, 3) # 3 сообщения на каждой странице
    page = request.GET.get('page')
    try:
        posts = paginator.page(page)
    except PageNotAnInteger:
        # Если страница не является целым числом, перемещаемся на первую
        страницу
        posts = paginator.page(1)
    except EmptyPage:
        # Если страница за пределами допустимого диапазона перемещаемся на
        последнюю страницу
        posts = paginator.page(paginator.num_pages)
    return render(request, 'blog/post/list.html', {'page': page,
                                                    'posts': posts,
                                                    'tag': tag})
```

Откройте файл `urls.py` вашего приложения `blog` и закомментируйте URL паттерн базового класса `PostListView`, а представления `post_list` раскомментируйте:

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

Добавьте следующий дополнительный URL шаблон для перечисления сообщений по тегам:

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

Как вы можете видеть, оба шаблона указывают на одно и то же представление, но мы вызываем их по-разному. Первый шаблон вызывает представление `post_list` без каких-либо

дополнительных параметров, тогда как второй шаблон вызывает представление с помощью `tag_slug` параметра. Мы используем конвертер пути `slug` для сопоставления параметра в виде строки с буквами ASCII или цифрами плюс символы дефиса и подчеркивания.

Поскольку мы используем представление `post_list`, отреактируйте шаблон `blog/post/list.html` и измените пагинацию на `posts` объект:

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

Добавьте следующие строки выше цикла `{% for %}`:

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

Если пользователь обращается к блогу, он увидит список всех сообщений. Если он фильтрует сообщения, помеченные определенным тегом, он будет видеть тег, который он фильтрует. Теперь измените способ отображения тегов следующим образом:

```
<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>
```

Теперь мы можем просматривать все теги сообщения, отображающего настраиваемую ссылку на URL-адрес для фильтрации сообщений по этому тегу. Мы создаем URL-адрес с помощью `{% url "blog:post_list_by_tag" tag.slug %}`, используя имя URL-

адреса и `slug` тэга как его параметр. Мы разделяем теги запятыми.

Откройте `http://127.0.0.1:8000/blog/` в вашем браузере и кликните на ссылку какого либо тега. Вы увидите список сообщений, отфильтрованных по этому тегу, например:

My Blog

Posts tagged with "jazz"

[Who was Django Reinhardt?](#)

Tags: [jazz](#) , [music](#)

Published Dec. 14, 2017, 8:54 a.m. by admin

Who was Django Reinhardt.

Page 1 of 1.

Получение похожих сообщений

Теперь, когда мы внедрили тегирование для наших сообщений, мы можем сделать с ними много интересного. Используя теги, мы можем очень хорошо классифицировать наши сообщения в блоге. Сообщения о похожих темах будут иметь несколько общих тегов. Мы создадим функциональность для отображения похожих записей по количеству тегов, которыми они владеют. Таким образом, когда пользователь читает сообщение, мы можем предложить ему прочитать другие связанные записи.

Чтобы получить похожие сообщения для определенной записи, нам необходимо выполнить следующие шаги:

1. Получить все теги для текущего сообщения
2. Получить все сообщения, помеченные любым из этих тегов
3. Исключить текущую запись из этого списка, чтобы избежать рекомендации по этому же сообщению
4. Упорядочить результаты по количеству тегов, совместно используемых с текущим сообщением
5. В случае двух или более сообщений с одинаковым количеством тегов, рекомендуем последнее сообщение
6. Ограничить запрос количеством сообщений, которые мы хотим рекомендовать.

Эти шаги переводятся в сложный QuerySet, который мы включим в наше представление `post_detail`. Откройте файл `views.py` вашего приложения блог и импортируйте в верху следующее:

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

Эта функция агрегации `Count` из Django ORM. Она позволит нам выполнять агрегированные подсчеты тегов. `django.db.models` включает следующие функции агрегации:

- `Avg`: Среднее значение
- `Max`: Максимальное значение
- `Min`: Минимальное значение
- `Count`: Количество объектов

Вы можете узнать об агрегировании на <https://docs.djangoproject.com/en/2.0/topics/db/aggregation/>.

Добавьте следующие строки внутри представления `post_detail` до функции `render()`, с тем же уровнем отступов:

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

Предыдущий код делает следующее:

1. Мы извлекаем список идентификаторов Python для тегов текущего сообщения. QuerySet этого `values_list()` возвращает кортежи со значениями для заданных

полей. Мы преобразуем его `flat=True`, чтобы получить список, как `[1, 2, 3, ...]`.

2. Мы получаем все сообщения, содержащие любые из этих тегов, за исключением самого текущего сообщения.
3. Мы используем функцию агрегации `Count` для генерации вычисленного поля—`same_tags`— которое содержит количество тегов, разделяемых всеми запрошенными тегами.
4. Мы сортируем результат по количеству общих тегов (по убыванию) и по `publish` для отображения последних сообщений сначала для постов с одинаковым количеством общих тегов. Мы отбираем только первые четыре сообщения.

Добавьте объект `similar_posts` в контекстный словарь функции `render()` следующим образом:

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

Теперь отредактируйте шаблон `blog/post/detail.html` и добавить следующий код перед списком комментариев:

```
<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 %}
```

Теперь ваша страница с подробной информацией должна выглядеть таким образом:

Who was Django Reinhardt?

Published Dec. 14, 2017, 8:54 a.m. by admin

Who was Django Reinhardt.

[Share this post](#)

Similar posts

[Miles Davis favourite songs](#)

[Notes on Duke Ellington](#)

Сейчас вы можете успешно рекомендовать похожие сообщения своим пользователям. django-taggit также включает `similar_objects()` который вы можете использовать для извлечения объектов по общим тегам. Вы можете взглянуть на все django-taggit менеджеры в <https://django-taggit.readthedocs.io/en/latest/api.html>.

Вы также можете добавить список тегов к шаблону с подробным сообщением так же, как и в `blog/post/list.html` шаблоне.

Резюме

В этой главе вы узнали, как работать с формами Django. Вы создали систему для совместной рассылки контента своего сайта по электронной почте и создали систему комментариев для своего блога. Вы добавили теги в свои сообщения в блоге и построили сложные QuerySets для извлечения объектов по подобию.

В следующей главе вы узнаете, как создавать собственные теги и фильтры шаблонов. Вы также создадите пользовательскую карту сайта и загрузите свои сообщения в блог, а также реализуете функциональность полнотекстового поиска для ваших сообщений в блоге.

Глава 3

Расширение приложения блог

В предыдущей главе были рассмотрены основы форм, и вы узнали, как интегрировать сторонние приложения в свой проект. В этой главе будут рассмотрены следующие вопросы:

- Создание собственных тегов и фильтров шаблонов
- Добавление Sitemap и post feed
- Реализация полнотекстового поиска с помощью PostgreSQL

Создание собственных тегов и фильтров шаблонов

Django предлагает множество встроенных шаблонных тегов, таких как `{% if %}` или `{% block %}`. Вы использовали некоторые из них в своих шаблонах. Вы можете найти полную ссылку на встроенные теги и фильтры шаблонов на <https://docs.djangoproject.com/en/2.0/ref/templates/builtins/>.

Однако Django также позволяет вам создать собственные теги шаблонов для выполнения пользовательских действий.

Пользовательские теги шаблонов очень удобны, когда вам нужно добавить такую функциональность к своим шаблонам, которая не покрывается основным набором шаблонных тегов Django.

Создание настраиваемых тегов шаблонов

Django предоставляет следующие вспомогательные функции, которые позволяют вам легко создавать свои собственные теги шаблонов:

- `simple_tag`: Обрабатывает данные и возвращает строку
- `inclusion_tag`: Обрабатывает данные и возвращает отображаемый шаблон

Теги шаблонов должны находиться внутри приложений Django.

Внутри директории приложения `blog`, создайте новую директорию с именем `templatetags`, и добавте в нее пустой файл `__init__.py`. Создайте другой файл в этой папке с именем `blog_tags.py`. Файловая структура приложения для блога должна выглядеть следующим образом:

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

Важно то как называется файл. Вы будете использовать имя этого модуля для загрузки тегов в шаблоны.

Мы начнем с создания простого тега, чтобы получить общее количество сообщений, публикуемых в блоге. Откройте файл `blog_tags.py`, который вы только что создали, и добавьте следующий код:

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

register = template.Library()

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

Мы создали простой тег шаблона, который возвращает количество опубликованных сообщений. Каждый модуль шаблонных тегов должен содержать переменную, называемую `register` в качестве допустимой библиотеки тегов. Эта переменная является экземпляром `template.Library`, и она используется для регистрации наших собственных тегов и фильтров шаблонов. Затем мы определяем тег, называемый `total_posts` с функцией Python и используем декоратор `@register.simple_tag` для регистрации функции как простого тега. Django будет использовать имя функции в качестве имени тега. Если вы хотите зарегистрировать его с помощью другого имени, вы можете сделать это, указав атрибут `name`, таким образом `@register.simple_tag(name='my_tag')`.

После добавления нового модуля шаблонных тегов вам нужно будет перезапустить сервер разработки Django, чтобы использовать новые теги и фильтры в шаблонах.

Прежде чем использовать пользовательские теги шаблона, вы должны сделать их доступными для шаблона, используя тег `{% load %}`. Как уже упоминалось ранее, вам нужно использовать имя модуля Python, содержащего теги и фильтры вашего шаблона. Откройте шаблон `blog/templates/base.html` и добавьте `{% load blog_tags %}` вверху, чтобы загрузить модуль шаблонных тэгов. Затем используйте тег, который вы создали, чтобы отобразить

общее количество сообщений. Просто добавьте `{% total_posts %}` к вашему шаблону. Шаблон должен выглядеть следующим образом:

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

Нам нужно будет перезапустить сервер, чтобы отслеживать новые файлы, добавленные в проект. Остановите сервер разработки с помощью `Ctrl + C` и запустите его снова, используя следующую команду:

```
python manage.py runserver
```

Откройте `http://127.0.0.1:8000/blog/` в вашем браузере. Вы должны увидеть общее количество сообщений на боковой панели сайта, как показано ниже:

My blog

This is my blog. I've written 4 posts so far.

Сила пользовательских тегов шаблонов заключается в том, что вы можете обрабатывать любые данные и добавлять их к любому шаблону независимо от выполненного представления. Вы можете выполнять QuerySet или обрабатывать любые данные для отображения результатов в ваших шаблонах.

Сейчас мы создадим еще один тег, чтобы отображать последние сообщения на боковой панели нашего блога. На этот раз мы будем использовать тег включения. Используя тег включения, вы можете отобразить шаблон с переменными контекста, возвращаемыми тегом шаблона. Отредактируйте файл `blog_tags.py` добавив в него следующий 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 предыдущем коде мы регистрируем тег шаблона, используя `@register.inclusion_tag` и указываем шаблон, который должен быть отображен с возвращаемыми значениями, используя `blog/post/latest_posts.html`. Наш шаблонный тег примет необязательный параметр по умолчанию `count` равный 5. Этот параметр позволяет указать количество сообщений, которые мы хотим отобразить. Мы используем эту переменную для ограничения результатов запроса `Post.published.order_by('-publish')[count]`. Обратите внимание, что функция возвращает словарь переменных вместо простого значения. Включенные теги должны возвращать словарь значений, который используется в качестве контекста для отображения указанного шаблона. Тег шаблона, который мы только что создали, позволяет указать дополнительное количество сообщений для отображения в виде `{% show_latest_posts 3 %}`.

Теперь создайте новый файл шаблона в разделе `blog/post/` и назовите его `latest_posts.html`. Добавте в него следующий код:

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

В предыдущем коде мы отображаем неупорядоченный список сообщений, используя переменную `latest_posts`, возвращающую нашим тегом шаблона. Теперь отредактируйте шаблон `blog/base.html` и добавте новый тег шаблона для отображения последних трех сообщений. Код боковой панели должен выглядеть следующим образом:

```
<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>
```

Вызывается шаблонный тег, передающий количество сообщений для отображения, и он визуализируется с заданным контекстом.

Сейчас вернитесь в свой браузер и обновите страницу. Теперь боковая панель должна выглядеть так:

My blog

This is my blog. I've written 4 posts so far.

Latest posts

- [Miles Davis favourite songs](#)
- [Notes on Duke Ellington](#)
- [Another post](#)

Наконец, мы создадим простой тег шаблона, который сохраняет результат в переменной, для того чтобы повторно использовать его. Мы создадим тег для отображения наиболее комментируемых сообщений. Отредактируйте файл `blog_tags.py` и добавьте в него следующий импорт и тег шаблона:

```
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 предыдущем теге шаблона мы создали `QuerySet` используя функцию `annotate()` чтобы агрегировать общее количество комментариев для каждого сообщения. Мы используем функцию агрегации `Count` для хранения количества комментариев в вычисленном поле `total_comments` для каждого `Post` объекта. Мы сортируем `QuerySet` по вычисленному полю в порядке убывания. Мы также добавляем переменную `count`, чтобы ограничить общее количество возвращаемых объектов.

В дополнении к `count`, Django предлагает функции агрегации `Avg`, `Max`, `Min`, и `Sum`. Вы можете больше узнать о функциях агрегации в <https://docs.djangoproject.com/en/2.0/topics/db/aggregation/>.

Отредактируйте шаблон `blog/base.html` и добавьте следующий код на боковую панель `<div>` элемента:

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

Мы сохраняем результат в пользовательской переменной, используя `as` аргумент, за которым следует имя переменной. Для нашего тега шаблона мы используем `{% get_most_commented_posts as most_commented_posts %}` для сохранения результата тега шаблона в новой переменной с именем `most_commented_posts`, когда мы показываем возвращенные сообщения с использованием неупорядоченного списка.

Теперь откройте свой браузер и обновите страницу, чтобы увидеть окончательный результат. Он должен выглядеть следующим образом:

A screenshot of a web browser window displaying a blog. The address bar shows the URL `127.0.0.1:8000/blog/`. The main content area features a sidebar on the right.

My Blog

Miles Davis favourite songs

Tags: [jazz](#), [music](#)

Published Dec. 14, 2017, 10:01 p.m. by admin

Miles Dewey Davis III was an American jazz trumpeter, bandleader, and composer.

Notes on Duke Ellington

Tags: [jazz](#), [music](#)

Published Dec. 14, 2017, 9:58 p.m. by admin

Edward Kennedy "Duke" Ellington was an American composer, pianist, and bandleader of a jazz orchestra.

Another post

Tags:

Published Dec. 14, 2017, 11:45 a.m. by admin

Post body.

My blog

This is my blog. I've written 4 posts so far.

Latest posts

- [Miles Davis favourite songs](#)
- [Notes on Duke Ellington](#)
- [Another post](#)

Most commented posts

- [Notes on Duke Ellington](#)
- [Who was Django Reinhardt?](#)
- [Another post](#)
- [Miles Davis favourite songs](#)

Теперь у вас есть четкое представление о том, как создавать пользовательские теги шаблонов. Вы можете узнать больше о них в <https://docs.djangoproject.com/en/2.0/howto/custom-template-tags/>.

Создание пользовательских шаблонных фильтров

Django имеет множество встроенных шаблонных фильтров, которые позволяют изменять переменные в шаблонах. Это функции Python, которые принимают один или два параметра - значение переменной, к которой оно применяется, и необязательный аргумент. Они возвращают значение, которое может отображаться или обрабатываться другим фильтром.

Фильтр выглядит как `{{ variable|my_filter }}`. Фильтры с аргументом выглядят как `{{ variable|my_filter:"foo" }}`. Вы можете применить к переменной столько фильтров, сколько хотите, например, `{{ variable|filter1|filter2 }}`, и каждый из них будет применен к выходу, сгенерированному предыдущим фильтром.

Мы создадим настраиваемый фильтр, чтобы иметь возможность использовать синтаксис markdown в наших сообщениях в блоге, а затем конвертировать содержимое сообщения в HTML внутри шаблона. Markdown - простой синтаксис форматирования текста, который очень прост в использовании и предназначен для преобразования в HTML. Вы можете изучить основы этого формата на

<https://daringfireball.net/projects/markdown/basics>.

Сначала установим Python модуль markdown через `pip` используя следующую команду:

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

Затем отредактируйте файл `blog_tags.py` и добавте в него следующий код:

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

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

Мы регистрируем фильтры шаблонов так же, как и теги шаблонов. Чтобы избежать противоречия между именем нашей функции и модулем `markdown`, назовем нашу функцию `markdown_format` а фильтр назовем `markdown` для использования в шаблонах, например `{{ variable|markdown }}`. Django избегает HTML-кода, сгенерированного фильтрами. Мы используем `mark_safe` функцию, предоставляемую Django, чтобы пометить результат как безопасный HTML, который будет отображаться в шаблоне. По умолчанию Django не будет доверять никакому HTML-коду и проигнорирует его, прежде чем помещать его в выходной файл. Единственными исключениями являются переменные, отмеченные как безопасные от экранирования. Такое поведение запрещает Django выводить потенциально опасный HTML и позволяет создавать исключения для возврата безопасного HTML.

Теперь загрузите модуль шаблонных тэгов в шаблоны списка сообщений и детального отображения. Добавьте следующую строку вверху шаблонов `blog/post/list.html` и `blog/post/detail.html` после тега `{% extends %}`:

```
{% load blog_tags %}
```

В шаблоне `post/detail.html`, найдите следующую строку:

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

Замените ее следующим:

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

Затем в файле `post/list.html`, замените строку:

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

следующим:

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

Фильтр `truncatewords_html` усекает строку после определенного количества слов, избегая закрытых тегов HTML.

Теперь откройте `http://127.0.0.1:8000/admin/blog/post/add/` в вашем браузере и добавьте сообщение со следующим содержимым:

```
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/)
```

Откройте ваш браузер и посмотрите, как отображается сообщение. Вы должны увидеть следующий вывод:

Markdown post

Published Dec. 15, 2017, 8:42 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](#)

Как вы можете видеть на предыдущем скриншоте, пользовательские фильтры шаблонов очень полезны для настройки форматирования. Дополнительную информацию о пользовательских фильтрах можно найти в <https://docs.djangoproject.com/en/2.0/howto/custom-template-tags/#writing-custom-template-filters>.

Добавление sitemap для вашего сайта

Django поставляется с картой сайта - sitemap, которая позволяет динамически создавать файлы Sitemap для вашего сайта. Sitemap - это XML-файл, который сообщает поисковым системам страницы вашего сайта, их релевантность и частоту их обновления. Используя карту сайта, вы поможете сканерам, которые индексируют контент вашего сайта.

Sitemap Django зависит от `django.contrib.sites`, который позволяет связать объекты с конкретными веб-сайтами, которые работают с вашим проектом. Это удобно, если вы хотите запускать несколько сайтов, используя один проект Django. Чтобы установить платформу sitemap, вам необходимо активировать как сайты, так и приложение sitemap в нашем проекте. Отредактируйте файл `settings.py` вашего проекта и добавьте `django.contrib.sites` а также `django.contrib.sitemaps` к настройкам `INSTALLED_APPS`. Кроме того, определите новый параметр для идентификатора сайта, как показано ниже:

```
SITE_ID = 1

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

Теперь запустите следующую команду для создания таблиц приложения сайта Django в базе данных:

```
python manage.py migrate
```

Вы должны увидеть вывод, который содержит следующие строки:

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

Теперь приложение `sites` синхронизировалось с базой данных. Создайте новый файл внутри директории приложения `blog` и дайте ему имя `sitemaps.py`. Откройте этот файл и добавьте к нему следующий 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 создаем пользовательскую карту сайта, наследуя `Sitemap` класс модуля `sitemaps`. Атрибуты `changefreq` и `priority` указывают частоту изменения ваших страниц сообщений и их актуальность на вашем сайте (максимальное значение равно 1). Метод `items()` возвращает `QuerySet` объектов для включения в файл `Sitemap`. По умолчанию Django вызывает метод `get_absolute_url()` для каждого объекта, чтобы получить его URL. Помните, что мы создали этот метод в [главе 1, Создание приложения для блога](#), для получения канонического URL-адреса сообщений. Если вы хотите указать URL для каждого объекта, вы можете добавить метод `location` в ваш `sitemap` класс. Метод `lastmod` получает каждый объект, возвращаемый `items()` и возвращает последнее время, когда объект был изменен. И

`changefreq` и `priority` также могут быть либо методами, либо атрибутами. Вы можете взглянуть на полную ссылку на карту сайта в официальной документации Django, расположенной по адресу <https://docs.djangoproject.com/en/2.0/ref/contrib/sitemaps/>.

Наконец, вам просто нужно добавить URL-адрес вашей карты. Измените основной `urls.py` файл вашего проекта и добавьте `sitemap`, следующим образом:

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

sitemaps = {
    'posts': PostSitemap,
}

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

В предыдущем коде мы включили требуемый импорт и определили словарь `sitemaps`. Мы определили шаблон URL, который соответствует `sitemap.xml` и использовали `sitemap` представление. Словарь `sitemaps` передается `sitemap` представлению. Теперь запустите сервер разработки и откройте `http://127.0.0.1:8000/sitemap.xml` в вашем браузере. Вы увидите следующий вывод XML:

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

```
<loc>  
http://example.com/blog/2017/12/14/who-was-django-reinhardt/  
</loc>  
<lastmod>2017-12-14</lastmod>  
<changefreq>weekly</changefreq>  
<priority>0.9</priority>  
</url>  
</urlset>
```

URL-адрес для каждого сообщения был создан, вызывая его `get_absolute_url()` метод.

Атрибут `lastmod` соответствует сообщению `updated` поля даты, как мы указали в нашей карте сайта, атрибуты `changefreq` и `priority` также взяты из нашего `PostSitemap` класса. Вы можете видеть, что домен, используемый для создания URL-адресов, `example.com`. Этот домен происходит из объекта `site`, хранящегося в базе данных. Этот объект по умолчанию был создан, когда мы синхронизировали структуру сайта с нашей базой данных. Откройте `http://127.0.0.1:8000/admin/sites/site/` в вашем браузере. Вы должны увидеть что-то вроде этого:

Django administration

WELCOME, ADMIN. [VIEW SITE / CHANGE PASSWORD / LOG OUT](#)

Home > Sites > Sites

Select site to change

[ADD SITE +](#)

Action: 0 of 1 selected

<input type="checkbox"/>	DOMAIN NAME	DISPLAY NAME
<input type="checkbox"/>	example.com	example.com

1 site

The screenshot shows the Django Admin interface for managing sites. At the top, it says 'Django administration' and 'WELCOME, ADMIN. [VIEW SITE / CHANGE PASSWORD / LOG OUT](#)'. Below that is a breadcrumb trail: 'Home > Sites > Sites'. The main title is 'Select site to change' with a 'ADD SITE +' button. Below this is a search bar with a magnifying glass icon and a 'Search' button. Underneath is a table with columns: 'Action', 'DOMAIN NAME', and 'DISPLAY NAME'. One row is shown: a checkbox next to 'example.com', 'example.com' under DOMAIN NAME, and 'example.com' under DISPLAY NAME. A note at the bottom says '1 site'. The table has a header row with a triangle icon for sorting.

На предыдущем скриншоте содержится представление администратора списка отображения для фреймворка сайта. Здесь вы можете настроить домен или хост, которые будут использоваться инфраструктурой сайта и приложениями, зависящими от него. Чтобы создать URL-адреса,

существующие в нашей локальной среде, измените имя домена на `localhost:8000`, как показано на следующем снимке экрана, и сохраните его:

Change site

Domain name:	<code>localhost:8000</code>
Display name:	<code>localhost:8000</code>

Теперь URL-адреса, отображаемые в вашем канале, будут построены с использованием этого имени хоста. В производственной среде вам придется использовать доменное имя для фреймворка сайта.

Создание фидов для сообщений в блоге

Django имеет встроенный фреймворк подачи синдициации, который вы можете использовать для динамического создания RSS-каналов или каналов Atom аналогичным образом как для создания файлов Sitemap с использованием фреймворка сайта. Веб-канал - это формат данных (обычно XML), который предоставляет пользователям часто обновляемый контент. Пользователи смогут подписаться на ваш канал с помощью агрегатора фидов, программного обеспечения, которое используется для чтения каналов и получения новых уведомлений о содержании.

Создайте новый файл в директории вашего приложения `blog` и назовите его `feeds.py`. Добавте в него следующие строки:

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

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

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

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

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

Во-первых, мы используем `Feed` класс фреймворка синдициации.

Атрибуты `title`, `link`, И `description` соответствуют RSS элементам `<title>`, `<link>`, И `<description>`.

Метод `items()` извлекает объекты, которые должны быть включены в фид. Мы извлекаем только последние пять опубликованных сообщений для этого фида. Методы `item_title()` И `item_description()` получают каждый объект, возвращаемый `items()` а выдают заголовок и описание для каждого элемента. Мы используем встроенный шаблонный фильтр `truncatewords` для создания описания сообщения в блоге с первыми 30 словами.

Теперь отредактируйте файл `blog/urls.py`, импортируйте `LatestPostsFeed` который вы только что сохранили и создайте экземпляр канала в новом шаблоне URL:

```
from .feeds import LatestPostsFeed

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

Перейдите к `http://127.0.0.1:8000/blog/feed/` в вашем браузере. Теперь вы должны увидеть RSS-канал, включая последние пять сообщений в блоге:

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

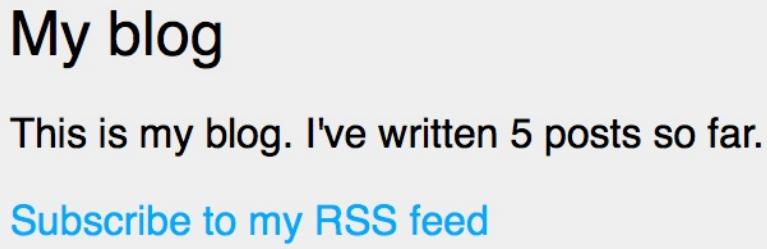
```
reinhardt/</guid>
</item>
...
</channel>
</rss>
```

Если вы откроете тот же URL-адрес в RSS-клиенте, вы сможете увидеть свой канал с помощью удобного интерфейса.

Последний шаг - добавить ссылку подписки на канал на боковую панель блога. Откройте шаблон `blog/base.html` и добавьте следующие строки под общим количеством сообщений в боковой панели `div`:

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

Сейчас откройте в вашем браузере `http://127.0.0.1:8000/blog/` и взгляните на боковую панель. Новая ссылка должна привести вас к фиду вашего блога:



Добавление полнотекстового поиска в ваш блог

Теперь вы добавите возможность поиска в свой блог. Django ORM позволяет выполнять простые операции сопоставления, используя, например, фильтр `contains` (или его нечувствительную к регистру версию, `icontains`). Вы можете использовать следующий запрос, чтобы найти сообщения, содержащие слово `framework`:

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

Однако, если вы хотите выполнять сложные поисковые запросы, вам нужно будет использовать полнотекстовую поисковую систему.

Django обеспечивает мощную функцию поиска, построенную поверх полнотекстового поиска PostgreSQL. Модуль `django.contrib.postgres` предоставляет функции PostgreSQL, которые не используются в других базах данных, поддерживаемых Django. Вы можете узнать о полнотекстовом поиске PostgreSQL в <https://www.postgresql.org/docs/10/static/textsearch.html>.

1.

Django поддерживает часть богатого набора функций, предлагаемого PostgreSQL, не разделяемого другими базами данных этого фреймворка.

Установка PostgreSQL

В настоящее время вы используете SQLite для своего проекта `blog`. Этого достаточно для разработки. Однако в рабочей среде вам понадобится более мощная база данных, такая как PostgreSQL, MySQL или Oracle. Мы заменим нашу базу данных на PostgreSQL, чтобы воспользоваться ее функциями полнотекстового поиска.

Если вы используете Linux, установите зависимости PostgreSQL для работы с Python, например:

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

Примечание переводчика: возможно вам придется указать конкретную версию питона - `python3-dev`:

`sudo apt-get install libpq-dev python3-dev`

Затем установите PostgreSQL следующей командой:

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

Если вы используете macOS X или Windows, загрузите PostgreSQL из <https://www.postgresql.org/download/> и установите ее.

Вам также необходимо установить расширение PostgreSQL Psycopg2 для Python. Запустите следующую команду в оболочке:

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

Давайте создадим пользователя для нашей базы данных PostgreSQL. Откройте терминал и выполните следующие команды:

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

Примечание переводчика: возможно вам придется использовать sudo
sudo su postgres

Вам будет предложено ввести пароль для нового пользователя. Введите желаемый пароль, а затем создайте базу данных `blog` и предоставьте право на нее пользователю, которого вы только что создали, следующей командой:

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

Затем отредактируйте файл `settings.py` вашего проекта и измените настройки `DATABASES` следующим образом:

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

Замените предыдущие данные именем базы данных и учетными данными для созданного вами пользователя. Новая база данных пуста. Выполните следующую команду, чтобы применить все миграции:

```
python manage.py migrate
```

Наконец, создайте суперпользователя следующей командой:

```
python manage.py createsuperuser
```

Теперь вы можете запустить сервер разработки и получить доступ к сайту администрирования по адресу <http://127.0.0.1:8000/admin/> с новым суперпользователем.

Поскольку мы переключили базу данных, в ней нет сообщений. Заполните новую базу данных несколькими примерами сообщений в блоге, чтобы вы могли выполнять поиск.

Простые поисковые запросы

Отредактируйте файл `settings.py` вашего проекта и добавте `django.contrib.postgres` к `INSTALLED_APPS` следующим образом:

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

Теперь вы можете выполнять поиск по одному полю, используя `search` `QuerySet`:

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

Этот запрос использует PostgreSQL для создания вектора поиска в поле `body` и поисковый запрос состоящий из термина `dango`. Результаты получены путем сопоставления запроса с вектором.

Поиск по нескольким полям

Возможно, вы захотите выполнить поиск по нескольким полям. В этом случае вам нужно будет определить `SearchVector`. Давайте построим вектор, который позволяет нам искать по полям `title` и `body` в модели `Post`:

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

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

Использование аннотации и определения `SearchVector` с обоими полями, мы предоставляем функциональность, соответствующую запросу, как по заголовку, так и по телу сообщений.

Полнотекстовый поиск - это интенсивный процесс. Если вы ищете более нескольких сотен строк, вы должны определить функциональный индекс, который соответствует используемому вами поисковому вектору. Django предоставляет поле `SearchVectorField` для ваших моделей. Вы можете узнать больше об этом на <https://docs.djangoproject.com/en/2.0/ref/contrib/postgres/search/#performance>.

Создание представления поиска

Сейчас мы создадим пользовательское представление, чтобы наши пользователи могли искать сообщения. Во-первых, нам нужна форма поиска. Отредактируйте файл `forms.py` приложения `blog` и добавьте следующую форму:

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

Мы будем использовать поле `query` чтобы пользователи вводили поисковые запросы. Отредактируйте файл `views.py` приложения `blog` и добавьте следующий код в него:

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

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

В предыдущем представлении мы создали экземпляр

формы `SearchForm`. Мы планируем представить форму, используя метод `GET` так чтобы URL-адрес включал `query` параметр. Чтобы проверить, отправлена ли форма, мы ищем `query` параметр в `request.GET` словаре. Когда форма отправляется, мы создаем ее с предоставленными `GET` данными, и мы проверяем, что данные формы действительны. Если форма действительна, мы ищем сообщения с пользовательским экземпляром `SearchVector`, построенным с помощью `title` и `body` полей.

Теперь поиск будет готов. Нам нужно создать шаблон для отображения формы и результатов, когда пользователь выполняет поиск. Создайте новый файл внутри директории шаблонов `/blog/post/` с именем `search.html`, и добавте следующий код в него:

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

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

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

Как и в режиме поиска, мы можем отличить, была ли форма представлена при наличии `query` параметра. Перед отправкой сообщения мы отобразим форму и кнопку отправки. После отправки сообщения мы показываем выполненный запрос, общее количество результатов и список возвращенных сообщений.

Наконец, отредактируйте файл `urls.py` вашего приложения `blog` и добавьте следующий URL паттерн:

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

Теперь откройте `http://127.0.0.1:8000/blog/search/` в вашем браузере. Вы должны увидеть следующую форму поиска:

Search for posts

Query:

SEARCH

Введите запрос и нажмите кнопку Search. Вы увидите результаты поискового запроса следующим образом:

Posts containing "music"

Found 2 results

[Another post more](#)

Post body.

[Who was Django Reinhardt?](#)

The Django web framework was ...

[Search again](#)

My blog

This is my blog. I've written 4 posts so far.

[Subscribe to my RSS feed](#)

Latest posts

- [Another post more](#)
- [New title](#)
- [Who was Django Reinhardt?](#)

Most commented posts

- [Who was Django Reinhardt?](#)
- [New title](#)
- [Another post more](#)
- [Old](#)

Поздравляем! Вы создали базовую поисковую систему для своего блога.

Анализ и ранжирование результатов

Django предоставляет класс `SearchQuery` для перевода терминов в объект поискового запроса. По умолчанию термины передаются с помощью алгоритмов, которые помогают получить лучшие совпадения. Вы также можете заказать результаты по релевантности. PostgreSQL предоставляет функцию ранжирования, которая упорядочивает результаты, исходя из того, как часто появляются условия запроса и насколько они близки друг к другу. Измените файл `views.py` вашего приложения `blog` и добавьте следующий импорт:

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

Затем найдите следующие строки:

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

Замените их следующими:

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

В предыдущем коде мы создали объект `SearchQuery`, отфильтровали результат и использовали `SearchRank` чтобы

упорядочить по релевантности. Вы можете открыть <http://127.0.0.1:8000/blog/search/> в вашем браузере и протестировать различные варианты поиска, чтобы проверить результат и ранжирование. Ниже приведен пример ранжирования по числу вхождений слова `django` в заголовке и тексте сообщений:

Posts containing "django"

Found 3 results

[Django, Django, Django](#)

Django is the Web Framework ...

[Django twice](#)

Django offers full text search ...

[Django once](#)

A Python web framework.

[Search again](#)

Взвешивание запросов

Вы можете увеличить определенные векторы, чтобы им приписывался больший вес при упорядочивании результатов по релевантности. Например, это можно использовать для придания большей релевантности записям, соответствующим по заголовку, а не по содержимому. Отредактируйте предыдущие строки в файле `views.py` Вашего приложения `blog` и сделать их похожими на:

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

В предыдущем коде мы применяем различные веса к векторам поиска, построенным с помощью полей `title` и `body`. Веса по умолчанию `b`, `c`, `v`, и `a` что обозначают цифры `0.1`, `0.2`, `0.4`, и `1.0` соответственно. Мы применяем вес `1.0` к `title` вектору поиска и вес `0.4` к `body` вектору: совпадения заголовков будут преобладать над совпадениями содержания тела. Мы фильтруем результаты, чтобы отображать только те, у которых ранг выше, чем `0.3`.

Поиск с помощью сходства триграмм

Другой подход к поиску - сходство триграмм. Триграмма представляет собой группу из трех последовательных символов. Вы можете измерить сходство двух строк, подсчитав количество триграмм, которыми они делятся. Этот подход оказывается очень эффективным для измерения сходства слов во многих языках.

Чтобы использовать триграммы в PostgreSQL, вам нужно будет сначала установить `pg_trgm` расширение. Выполните следующую команду из консоли, чтобы подключиться к вашей базе данных:

```
| psql blog
```

Затем выполните следующую команду, чтобы установить `pg_trgm` расширение:

```
| CREATE EXTENSION pg_trgm;
```

Давайте отредактируем наше представление и изменим его для поиска триграмм. Отредактируйте файл `views.py` вашего приложения `blog` и добавьте следующий импорт:

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

Затем замените `Post` поисковый запрос следующими строками:

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

Откройте <http://127.0.0.1:8000/blog/search/> в вашем браузере и протестируйте разные варианты поиска триграмм. В следующем примере показана гипотетическая опечатка в django с результатами поиска для yango:

Posts containing "yango"

Found 1 result

[Django Django](#)

A Python web framework.

Теперь у вас есть мощная поисковая система, встроенная в ваш проект. Вы можете найти дополнительную информацию о **ПОЛНОТЕКСТОВОМ ПОИСКЕ** в <https://docs.djangoproject.com/en/2.0/ref/contrib/postgres/search/>.

Другие полнотекстовые поисковые системы

Вы можете использовать полнотекстовую поисковую систему, отличную от PostgreSQL. Если вы хотите использовать Solr или Elasticsearch, вы можете интегрировать их в свой проект Django с помощью Haystack. Haystack - приложение Django, которое работает как слой абстракции для нескольких поисковых систем. Он предлагает простой API поиска, очень похожий на Django QuerySets. Вы можете найти более подробную информацию о Haystack на <http://haystacksearch.org/>.

Резюме

В этой главе вы узнали, как создавать пользовательские теги и фильтры шаблонов Django, чтобы предоставить шаблонам пользовательскую функциональность. Вы также создали карту сайта для сканирования вашего сайта поисковыми системами и RSS-канал для подписки на ваш блог. Вы также создали поисковую систему для своего блога, используя полнотекстовую поисковую систему PostgreSQL.

В следующей главе вы узнаете, как создать социальную сеть с использованием платформы проверки подлинности Django, как создать профили пользователей и как создать аутентификацию через соцсети.

Глава 4

Создание сайта социальной сети

В предыдущей главе вы узнали, как создавать карту сайта (Sitemap) и фиды, и создали поисковую систему для вашего приложения блог. В этой главе вы разработаете приложение социальной сети. Вы дадите пользователям возможность входа в систему, выхода из системы, редактирования и сброса пароля. Вы узнаете, как создать пользовательский профиль, и вы добавите аутентификацию через социальные сети на свой сайт.

В этой главе будут рассмотрены следующие темы:

- Использование фреймворка аутентификации Django
- Создание представлений регистрации пользователей
- Расширение модели пользователя с помощью настраиваемой модели профиля
- Добавление аутентификации через социальные сети с помощью `python-social-auth`

Начнем с создания нашего нового проекта.

Создание проекта социального веб-сайта

Мы создадим социальное приложение, которое позволит пользователям обмениваться изображениями, которые они находят в Интернете. Нам нужно будет создать следующие элементы для этого проекта:

- Система аутентификации пользователей для регистрации, входа на сайт, редактирования их профиля и изменения или сброса пароля
- Система подписчиков, позволяющая пользователям подписываться друг на друга
- Функциональность для отображения общих изображений и реализации bookmarklet для пользователей чтобы обмениваться изображениями с любого веб-сайта
- Отслеживание активности для каждого пользователя, чтобы позволить пользователям просматривать содержимое, загруженное людьми, на которых они подписаны

В этой главе рассматривается первый пункт, упомянутый в списке.

Запуск проекта сайта социальной сети

Откройте терминал и выполните следующие команды, чтобы создать виртуальную среду для своего проекта и активировать ее:

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

В командной строке появится активная виртуальная среда, как показано ниже:

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

Установите Django в свою виртуальную среду с помощью следующей команды:

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

Для создания нового проекта выполните следующую команду:

```
django-admin startproject bookmarks
```

После создания исходной структуры проекта используйте следующие команды, чтобы перейти в каталог проекта и создать новое приложение с именем account:

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

Помните, что вы должны активировать новое приложение в своем проекте, добавив его в `INSTALLED_APPS` файла `settings.py`. Поместите его в список `INSTALLED_APPS` перед любым из других установленных приложений:

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

Мы позже определим шаблоны аутентификации Django. Разместив наше приложение в настройке `INSTALLED_APPS`, мы гарантируем, что наши шаблоны аутентификации будут использоваться по умолчанию вместо любых других шаблонов аутентификации, содержащихся в других приложениях. Django ищет шаблоны по порядку появления приложения в настройках `INSTALLED_APPS`.

Выполните следующую команду для синхронизации базы данных с моделями приложений по умолчанию, включенными в `INSTALLED_APPS`:

```
python manage.py migrate
```

Вы увидите, что применяются все первоначальные миграции Django. Мы построим систему аутентификации в нашем проекте с использованием системы аутентификации Django.

Использование фреймворка аутентификации Django

Django имеет встроенную систему аутентификации, которая может обрабатывать аутентификацию пользователя, сеансы, права доступа и группы пользователей. Система аутентификации включает в себя представления для общих действий пользователя, таких как вход и выход из системы, смена пароля и сброс пароля.

Фреймворк аутентификации расположены по адресу `django.contrib.auth` и используется другими Django `contrib` пакетами. Вспомните, вы уже использовали фреймворк аутентификации в [Главе 1, Создание приложения для блога](#), при создании суперпользователя в приложении вашего блога для доступа к сайту администрирования.

Когда вы создаете новый проект Django, используя команду `startproject`, фреймворк аутентификации включается в настройки по умолчанию для вашего проекта. Он состоит из приложения `django.contrib.auth` и следующих двух классов промежуточного программного обеспечения, содержащихся в секции `MIDDLEWARE` настроек вашего проекта:

- `AuthenticationMiddleware`: Связывает пользователей с запросами, используя сессии
- `SessionMiddleware`: Обрабатывает запросы текущей сессии

Промежуточное ПО - это класс с методами, которые выполняются глобально во время фазы запроса или ответа. Вы

будете использовать классы промежуточного программного обеспечения несколько раз в этой книге, и вы научитесь создавать собственное промежуточное программное обеспечение в [Главе 13](#).

Фреймворк аутентификации также включает следующие модели:

- `User`: Модель пользователя с базовыми полями; основные поля этой модели `username`, `password`, `email`, `first_name`, `last_name`, и `is_active`
- `Group`: Модель для разделения пользователей на группы
- `Permission`: Разрешения для пользователей или групп на выполнение определенных действий

Фреймворк, по умолчанию, также включает представления и формы проверки подлинности, которые мы будем использовать позже.

Создание представления входа в систему

Мы начнем этот раздел, используя фреймворк аутентификации Django, чтобы пользователи могли войти на наш сайт. Наше представление должно выполнять следующие действия для входа пользователя:

1. Получить имя пользователя и пароль, разместив форму
2. Аутентификация пользователя по данными, хранящимися в базе данных
3. Проверить, активен ли пользователь
4. Зарегистрировать пользователя на веб-сайте и запустите сессию аутентификации

Сначала мы создадим форму входа. Создайте новый файл `forms.py` в каталоге приложения `account` и добавьте следующие строки в него:

```
from django import forms

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

Эта форма будет использоваться для аутентификации пользователей через базу данных. Обратите внимание, что мы используем виджет `PasswordInput` чтобы отображать HTML-код элемента `input`, включающего атрибут `type="password"`, так что браузер рассматривает его как поле для ввода пароля.

Откройте файл `views.py` приложения `account` и добавьте следующий код в него:

```
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:
                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})
```

Вот то, что делает наше основное представление входа в систему: при вызове `user_login` с запросом `GET` мы создаем новую форму входа в систему с помощью `form = LoginForm()` для отображения его в шаблоне. Когда пользователь отправляет форму через `POST`, мы выполняем следующие действия:

1. Создаем форму с предоставленными данными с помощью `form = LoginForm(request.POST)`.
2. Проверяем, действительна ли форма с помощью `form.is_valid()`. Если это не так, мы отображаем ошибки формы в нашем шаблоне (например, если пользователь не заполнил одно из полей).

3. Если предоставленные данные действительны, мы аутентифицируем пользователя по базе данных с помощью метода `authenticate()`. Этот метод берет из объекта `request` параметры `username`, `password` и возвращает `User`, если пользователь успешно прошел аутентификацию, или `None` в противном случае. Если пользователь не прошел проверку подлинности, мы возвращаем сообщение `HttpResponse`, `Invalid login`.
4. Если пользователь был успешно аутентифицирован, мы проверяем, активен ли пользователь, через доступ к его атрибуту `is_active`. Это атрибут пользовательской модели Django. Если пользователь не активен, мы возвращаем `HttpResponse` сообщение: `Disabled account`.
5. Если пользователь активен, мы регистрируем пользователя на веб-сайте. Мы создаем сессию пользователя, вызывая метод `login()` и возвращаем сообщение: `Authenticated successfully`.

Обратите внимание на разницу между `authenticate` и `login`: `authenticate()` проверяет учетные данные пользователя и возвращает объект `User`, если они правильны; `login()` устанавливает для пользователя сессию.

Теперь вам нужно создать шаблон URL для этого представления. Создайте новый файл `urls.py` в каталоге приложения `account` и добавьте к нему следующий код:

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

urlpatterns = [
    # post views
    path('login/', views.user_login, name='login'),
]
```

Отредактируйте основной файл `urls.py`, расположенный в каталоге проекта `bookmark`, импортировав `include` и добавив шаблоны URL-адресов из `account`, следующим образом:

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

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

Примечание переводчика: если у вас возникла ошибка то измените строку:

from django.conf.urls import path, **include**
на:
from django.urls import path, **include**

Доступ к представлению `login` теперь можно получить по URL-адресу. Настало время создать шаблон для этого представления. Поскольку у вас нет шаблонов для этого проекта, вы можете начать с создания базового шаблона, который может быть расширен шаблоном входа. Создайте следующие файлы и каталоги внутри приложения `account`:

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

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

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
    <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
```

```
<div id="header">
    <span class="logo">Bookmarks</span>
</div>
<div id="content">
    {% block content %}
    {% endblock %}
</div>
</body>
</html>
```

Это будет базовый шаблон для веб-сайта. Как и в нашем предыдущем проекте, мы включаем стили CSS в основном шаблоне. Эти статические файлы можно найти в коде, который поставляется вместе с этой главой. Скопируйте каталог `static/` приложения `account` из исходного кода главы в то же место в вашем проекте, чтобы вы могли использовать статические файлы.

Базовый шаблон определяет блок `title` и блок `content`, которые могут быть заполнены содержимым шаблонов, что включают его.

Давайте заполним шаблон для нашей формы входа. Откройте `account/login.html` и добавьте к нему следующий 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 action"." method="post">
        {{ form.as_p }}
        {% csrf_token %}
        <p><input type="submit" value="Log in"></p>
    </form>
{% endblock %}
```

Этот шаблон включает форму, созданную в представлении. Поскольку наша форма будет отправлена через `POST`, мы будем включать шаблонный тег `{% csrf_token %}` для защиты от CSRF. Вы

узнали о защите от CSRF в [главе 2](#), *Улучшение вашего блога с помощью расширения функционала*.

В вашей базе данных пока нет пользователей. Сначала вам нужно создать суперпользователя, чтобы иметь доступ к сайту администрирования для управления другими пользователями. Откройте командную строку и выполните `python manage.py createsuperuser`. Введите нужное имя пользователя, адрес электронной почты и пароль. Затем запустите сервер разработки, используя команду `python manage.py runserver` и откройте `http://127.0.0.1:8000/admin/` в вашем браузере. Войдите в сайт администрирования, используя учетные данные пользователя, которого вы только что создали. Вы увидите сайт администрирования Django, включающего модели `User` и `Group` фреймворка аутентификации Django.

Он будет выглядеть следующим образом:

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	 Add	 Change
Users	 Add	 Change

Recent actions

My actions

None available

Создайте нового пользователя, используя сайт
администрирования и откройте <http://127.0.0.1:8000/account/login/> в
вашем браузере. Вы должны увидеть созданный шаблон,
включая форму входа:



Log-in

Please, use the following form to log-in:

Username:

Password:

LOG IN

Теперь отправьте форму, оставив одно из полей пустым. В этом случае вы увидите, что форма недействительна и отображает ошибки, как показано ниже:

Username:

test

This field is required.

Password:

LOG IN

Обратите внимание, что некоторые современные браузеры не дадут вам отправить форму с пустыми или ошибочными полями. Это происходит из-за проверки формы, выполняемой браузером на основе типов полей и ограничений для каждого поля. В этом случае форма не будет отправлена, и браузер отобразит сообщение об ошибке для неправильных полей.

Если вы вводите несуществующего пользователя или неправильный пароль, вы получите сообщение Invalid login.

Если вы введете действительные учетные данные, вы получите сообщение Authenticated successfully, как здесь:



Authenticated successfully

Использование представлений аутентификации Django

Django содержит несколько форм и представлений в фреймворке аутентификации, которые вы можете использовать сразу. Созданный вами вход в систему - это хорошее упражнение для понимания процесса аутентификации пользователей в Django. Однако в большинстве случаев вы можете использовать стандартные параметры проверки подлинности Django.

Django предоставляет следующие представления на основе классов для проверки подлинности. Все они расположены в `django.contrib.auth.views`:

- `LoginView`: Обрабатывает регистрационную форму и регистрирует пользователя
- `LogoutView`: Выход пользователя

Django предоставляет следующие представления для обработки изменений пароля:

- `PasswordChangeView`: Обрабатывает форму для изменения пароля пользователя
- `PasswordChangeDoneView`: Представление перенаправляется пользователя после успешной смены пароля

Django также включает в себя следующие представления, позволяющие пользователям сбросить пароль:

- `PasswordResetView`: Позволяет пользователям сбросить свой пароль. Он генерирует одноразовую ссылку с токеном и отправляет ее на учетную запись электронной почты пользователя.
- `PasswordResetDoneView`: Сообщает пользователям, что им было отправлено электронное письмо, включающее ссылку на сброс пароля.
- `PasswordResetConfirmView`: Позволяет пользователям устанавливать новый пароль.
- `PasswordResetCompleteView`: Представление перенаправляет пользователя после успешного сброса пароля.

Представления, перечисленные в предыдущем списке, могут сэкономить вам много времени при создании веб-сайта с учетными записями пользователей. В представлениях используются значения по умолчанию, которые вы можете переопределить, например, расположение отображаемого шаблона или форму, которая будет использоваться представлением.

Вы можете получить дополнительную информацию о встроенных представлениях проверки подлинности в <https://docs.djangoproject.com/en/2.0/topics/auth/default/#all-authentication-views>.

Представления входа и выхода

Отредактируйте файл `urls.py` вашего приложения `account` следующим образом:

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

urlpatterns = [
    # previous login view
    # path('login/', views.user_login, name='login'),
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

Мы закомментировали шаблон URL для представления `user_login`, который мы создали ранее, чтобы использовать представление `LoginView` фреймворка аутентификации Django. Мы также добавляем шаблон URL для представления `LogoutView`.

Создайте новый каталог в папке `templates` вашего приложения `account` и назовите его `registration`. Это путь по умолчанию, где представления проверки подлинности Django ожидают найти ваши шаблоны аутентификации.

Модуль `django.contrib.admin` включает некоторые шаблоны аутентификации, которые используются для сайта администрирования. Мы разместили приложение `account` в верхней части `INSTALLED_APPS` так чтобы Django использовал наши шаблоны по умолчанию вместо любых шаблонов аутентификации, определенных в других приложениях.

Создайте новый файл внутри директории `templates/registration`, назовите его `login.html`, и добавьте в него следующий код:

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

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

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

Этот шаблон входа очень похож на тот, который мы создали ранее. Django использует форму `AuthenticationForm` расположенную по умолчанию в `django.contrib.auth.forms`. Эта форма пытается аутентифицировать пользователя и вызывает ошибку проверки, если логин был неудачным. В данном случае мы можем найти ошибки, используя в шаблоне `{% if form.errors %}`, чтобы проверить, являются ли предоставленные учетные данные неправильными. Обратите внимание, что мы добавили в HTML-код скрытый `<input>` элемент для предоставления значения переменной, называемой `next`. Эта переменная сначала устанавливается в окне входа в систему при передаче в запросе параметра `next` (например, `http://127.0.0.1:8000/account/login/?next=/account/`).

Параметр `next` должен быть URL-адресом. Если этот параметр

задан, представление Django login будет перенаправлять пользователя на указанный URL после успешного входа в систему.

Теперь создайте шаблон `logged_out.html` внутри папки `registration` каталога шаблонов и добавте в него следующее содержимое:

```
{% 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 Django отобразит после выхода пользователя из системы.

После добавления паттернов URL-адресов и шаблонов для входа и выхода из системы ваш сайт готов для входа в систему с использованием представлений аутентификации Django.

Теперь мы создадим новое представление для отображения панели мониторинга, когда пользователи войдут в свою учетную запись. Откройте файл `views.py` вашего приложения `account` и добавьте к нему следующий код:

```
from django.contrib.auth.decorators import login_required

@login_required
def dashboard(request):
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard'})
```

Мы декорируем наше представление с помощью декоратора `login_required` из фреймворка аутентификации. Декоратор

`login_required` проверяет, аутентифицирован ли текущий пользователь. Если пользователь аутентифицирован, он выполняет декорированное представление; если пользователь не аутентифицирован, он перенаправляет пользователя на URL-адрес входа с первоначально запрошенным URL-адресом и именем `next` в качестве `GET` параметра. Таким образом, окно входа в систему переадресовывает пользователей на URL-адрес, к которому они пытались получить доступ после успешного входа в систему. Помните, что для этой цели мы добавили скрытый ввод в форме нашего шаблона входа.

Мы также определяем переменную `section`. Мы будем использовать эту переменную для отслеживания раздела сайта, который просматривается пользователем. Несколько представлений могут соответствовать одному и тому же разделу. Это простой способ определить раздел, которому соответствует каждое представление.

Теперь вам нужно создать шаблон для представления панели мониторинга. Создайте новый файл внутри каталога `templates/account/` и назовите его `dashboard.html`. Добавте в него следующий код:

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

{% block title %}Dashboard{% endblock %}

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

Затем добавьте следующий шаблон URL для этого представления в файл `urls.py` приложения `account`:

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

Отредактируйте файл `settings.py` следующим образом:

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

Настройки, указанные в предыдущем коде, следующие:

- `LOGIN_REDIRECT_URL`: Сообщает Django, на какой URL-адрес перенаправить после успешного входа в систему, если в запросе отсутствует параметр `next`
- `LOGIN_URL`: URL-адрес для перенаправления пользователя для входа (например, представления с помощью декоратора `login_required`)
- `LOGOUT_URL`: URL-адрес для перенаправления пользователя для выхода из системы

Мы используем имена шаблонов URL, которые мы ранее определили, используя атрибут `name` функции `path()`. Для этих параметров также могут использоваться URL-адреса с жесткой привязкой вместо имен URL.

Подведем итог тому, что вы сделали:

- Вы добавили в ваш проект Django представления аутентификации для входа и выходы
- Вы создали пользовательские шаблоны для обоих представлений и определили простое представление панели управления для перенаправления пользователей после входа в систему

- Наконец, вы настроили параметры Django для использования этих URL по умолчанию

Теперь мы добавим вход и выход из системы в наш базовый шаблон, чтобы собрать все вместе. Чтобы сделать это, мы должны определить, зарегистрирован ли текущий пользователь или нет, чтобы отобразить соответствующую ссылку для каждого случая. Текущий пользователь установлен в объекте `HttpRequest` промежуточного программного обеспечения аутентификации. Вы можете получить к нему доступ через `request.user`. Вы найдете объект `user` в запросе, даже если пользователь не аутентифицирован. Пользователь, не прошедший проверку подлинности, устанавливается в запросе как экземпляр `AnonymousUser`. Лучший способ проверить подлинность текущего пользователя - получить доступ к его атрибуту только для чтения `is_authenticated`.

Отредактируйте шаблон `base.html` и измените его элемент `<div>` с ID `header` следующим образом:

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

    <span class="user">
        {% if request.user.is_authenticated %}
            Hello {{ request.user.first_name }},
            <a href="{% url "logout" %}">Logout</a>
        {% else %}
            <a href="{% url "login" %}">Login</a>
        {% endif %}
    </span>
</div>
```

```
    {% else %}  
        <a href="{% url "login" %}">Log-in</a>  
    {% endif %}  
    </span>  
</div>
```

Как вы видите в предыдущем коде, мы отображаем только меню сайта для аутентифицированных пользователей. Мы также проверяем текущий раздел, чтобы добавить атрибут класса `selected` к соответствующему элементу ``, чтобы выделить текущий раздел в меню с помощью CSS. Мы также показываем имя пользователя и ссылку для выхода из системы, если пользователь аутентифицирован, или ссылку для входа в систему в противном случае.

Теперь откройте `http://127.0.0.1:8000/account/login/` в вашем браузере. Вы должны увидеть страницу входа в систему. Введите действительное имя пользователя и пароль и нажмите Log-in кнопку. Вы должны увидеть следующий вывод:



Dashboard

Welcome to your dashboard.

Вы можете видеть что раздел `My dashboard` выделен CSS, поскольку он имеет класс `selected`. Поскольку пользователь аутентифицирован, имя пользователя отображается в правой части заголовка. Нажмите на ссылку `Logout`. Вы должны увидеть следующую страницу:

Logged out

You have been successfully logged out. You can [log-in again](#).

На странице, приведенной на предыдущем снимке экрана, вы можете видеть, что пользователь вышел из системы, и поэтому меню веб-сайта больше не отображается. Сейчас ссылка в правой стороне заголовка показывает Log-in.

Если вы видите страницу выхода административного сайта Django вместо своей собственной страницы выхода, проверьте `INSTALLED_APPS` настройки вашего проекта и убедитесь, что `django.contrib.admin` стоит после приложения `account`. Оба шаблона расположены по одному и тому же пути, а загрузчик шаблонов Django будет использовать первый найденный.

Представление изменения пароля

Нам также нужно чтобы наши пользователи имели возможность изменять свой пароль после входа на сайт. Мы будем использовать интегрированные представления аутентификации Django для смены пароля. Откройте файл `urls.py` приложения `account` и добавьте к нему следующие шаблоны URL:

```
# change password urls
path('password_change/',
      auth_views.PasswordChangeView.as_view(),
      name='password_change'),
path('password_change/done/',
      auth_views.PasswordChangeDoneView.as_view(),
      name='password_change_done'),
```

Представление `PasswordChangeView` будет обрабатывать форму для изменения пароля, а `PasswordChangeDoneView` будет отображать сообщение об успешном завершении после того, как пользователь успешно изменил свой пароль. Давайте создадим шаблон для каждого представления.

Добавте новый файл в каталог `templates/registration/` вашего приложения `account` и назовите его `password_change_form.html`. Добавте в него следующий код:

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

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

{% block content %}
<h1>Change you password</h1>
```

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

Шаблон `password_change_form.html` включает форму для изменения пароля. Теперь создайте другой файл в том же каталоге и назовите его `password_change_done.html`. Добавьте к нему следующий код:

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

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

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

Шаблон `password_change_done.html` содержит сообщение об успешном завершении, которое будет отображаться, когда пользователь изменит свой пароль.

Откройте `http://127.0.0.1:8000/account/password_change/` в вашем браузере. Если ваш пользователь не вошел в систему, браузер перенаправит вас на страницу входа. После успешной аутентификации вы увидите следующую страницу смены пароля:

Change your password

Use the form below to change your password.

Old password:

New password:

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

New password confirmation:

CHANGE

Заполните форму своим текущим паролем и новым паролем и нажмите на кнопку CHANGE . Вы увидите следующую страницу в случае успеха:

Password changed

Your password has been successfully changed.

Выйдите из системы и войдите снова, используя новый пароль, чтобы убедиться, что все работает так, как ожидалось.

Представление сброса пароля

Добавьте следующие шаблоны URL для восстановления пароля в файл urls.py приложения account:

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

Создайте новый файл в каталоге templates/registration/ вашего приложения account и назовите его password_reset_form.html. Добавьте в него следующий 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 action="." method="post">
  {{ form.as_p }}
  <p><input type="submit" value="Send e-mail"></p>
  {% csrf_token %}
</form>
{% endblock %}
```

Теперь создайте другой файл в том же каталоге и назовите его `password_reset_email.html`. Добавьте в него следующий код:

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

Шаблон `password_reset_email.html` будет использоваться для визуализации отправленного пользователем сообщения для сброса пароля.

Создайте другой файл в том же каталоге и назовите его `password_reset_done.html`. Добавьте к нему следующий код:

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

Создайте еще один шаблон в том же каталоге и назовите его `password_reset_confirm.html`. Добавьте в него следующий 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 action"." method="post">  
            {{ form.as_p }}  
            {% csrf_token %}  
            <p><input type="submit" value="Change my password" /></p>
```

```
</form>
{% else %}
    <p>The password reset link was invalid, possibly because it has
already been used. Please request a new password reset.</p>
{% endif %}
{% endblock %}
```

Мы проверяем, действительна ли предоставленная ссылка. Представление `PasswordResetConfirmView` устанавливает эту переменную и помещает ее в шаблон `password_reset_confirm.html`. Если ссылка действительна, мы показываем форму сброса пароля пользователя.

Создайте еще один шаблон и назовите его `password_reset_complete.html`. Введите в него следующий 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`, и добавьте следующий код после `<form>` элемента:

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

Теперь откройте `http://127.0.0.1:8000/account/login/` в вашем браузере и кликните на ссылку `Forgotten your password?` Вы должны увидеть следующую страницу:

Forgotten your password?

Enter your e-mail address to obtain a new password.

Email:

SEND E-MAIL

На этом этапе вам необходимо добавить конфигурацию SMTP в файл `settings.py` вашего проекта, чтобы Django смог отправлять электронные письма. Вы узнали, как добавить настройки электронной почты в свой проект в [главе 2, Улучшение вашего блога с помощью расширения функционала](#). Однако во время разработки вы можете настроить Django на отправку сообщений электронной почты на стандартный вывод вместо отправки через SMTP-сервер. Django предоставляет почтовый сервер для написания писем на консоль. Отредактируйте файл `settings.py` добавив следующие строки:

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

Настройка `EMAIL_BACKEND` указывает класс, который будет использоваться для отправки электронных писем.

Вернитесь в свой браузер, введите адрес электронной почты существующего пользователя и нажмите кнопку **SEND E-MAIL**. Вы должны увидеть следующую страницу:

Reset your password

We've emailed you instructions for setting your password.

If you don't receive an email, please make sure you've entered the address you registered with.

Посмотрите на консоль, в которой запущен 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: user@domain.com
Date: Fri, 15 Dec 2017 14:35:08 -0000
Message-ID: <20150924143508.62996.55653@zenx.local>

Someone asked for password reset for email user@domain.com. Follow the link
below:
http://127.0.0.1:8000/account/reset/MQ/45f-9c3f30caafdf523055fcc/
Your username, in case you've forgotten: zenx
```

Электронная почта отображается с помощью шаблона `password_reset_email.html`, который мы создали ранее. URL для сброса пароля включает в себя токен, который динамически генерируется Django. Скопируйте URL-адрес и откройте его в своем браузере. Вы должны увидеть следующую страницу:

Reset your password

Please enter your new password twice:

New password:

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

New password confirmation:

CHANGE MY PASSWORD

Страница для установки нового пароля соответствует шаблону `password_reset_confirm.html`. Введите новый пароль и нажмите кнопку **CHANGE MY PASSWORD**. Django зашифрует новый пароль и сохранит его в базе данных. Вы увидите следующую страницу в случае успеха:

Password set

Your password has been set. You can [log in now](#)

Теперь вы можете войти в свою учетную запись, используя новый пароль.

Каждый токен для установки нового пароля может использоваться только один раз. Если вы снова откроете ссылку, которую вы получили, вы получите сообщение о том, что токен недействителен.

В нашем проекте мы интегрировали представления фреймворка аутентификации Django. Они подходят для большинства случаев. Однако вы можете создать свои собственные представления, если вам нужно организовать другое поведение.

Django также предоставляет шаблоны URL-адресов аутентификации, которые мы только что создали. Вы можете закомментировать шаблоны URL-адресов аутентификации, которые мы добавили в файл `urls.py` приложения `account` и включить вместо них `django.contrib.auth.urls`, следующим образом:

```
from django.urls import path, include
# ...

urlpatterns = [
    # ...
    path('', include('django.contrib.auth.urls')),
]
```

Вы можете увидеть шаблоны URL аутентификации,
включенные в <https://github.com/django/django/blob/stable/2.0.x/django/contrib/auth/urls.py>.

Примечание переводчика: Если у вас появилась ошибка,
возможно вам не хватает паттерна URL 'dashboard'

```
from . import views
from django.urls import path, include
# ...

urlpatterns = [
    # ...
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
]
```

Регистрация пользователей и профили пользователей

Существующие пользователи теперь могут войти в систему, выйти из системы, изменить и сбросить свой пароль. Теперь нам нужно создать представление, чтобы посетители могли создавать учетную запись пользователя.

Регистрация пользователя

Давайте создадим простое представление, чтобы разрешить регистрацию пользователя на нашем веб-сайте. Первоначально мы должны создать форму, чтобы пользователь мог ввести имя пользователя, свое настоящее имя и пароль.

Отредактируйте файл `forms.py` находящийся внутри директории приложения `account` добавив следующий код:

```
from django.contrib.auth.models import User

class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password',
                                widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password',
                                widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ('username', 'first_name', 'email')

    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Passwords don\'t match.')
        return cd['password2']
```

Мы создали модельную форму для модели `User`. В нашу форму мы включаем только поля модели `username`, `first_name`, и `email`. Эти поля будут проверяться на основе соответствующих полей модели. Например, если пользователь выбирает имя пользователя, которое уже существует, оно получит ошибку проверки, поскольку `username` это поле, определенное с помощью `unique=True`. Мы добавили два дополнительных поля—`password` и `password2`—установить пароль и подтвердить его. Мы определили метод `clean_password2()` для сравнения второго пароля с первым чтобы не допустить подтверждения формы, если пароли не

совпадают. Эта проверка выполняется, когда мы проверяем форму, вызывая ее метод `is_valid()`. Вы можете применить метод `clean_<fieldname>()` к любому из полей формы, чтобы очистить его или проверить форму на ошибки для определенного поля. Формы также включают общий `clean()` метод для проверки всей формы, полезный для проверки полей, которые зависят друг от друга.

Django также предоставляет `UserCreationForm`. Она находится в `django.contrib.auth.forms` и очень похожа на ту, которую мы создали.

Отредактируйте файл `views.py` приложения `account` и добавьте следующий код в него:

```
from .forms import LoginForm, UserRegistrationForm

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Create a new user object but avoid saving it yet
            new_user = user_form.save(commit=False)
            # Set the chosen password
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Save the User object
            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 для создания учетных записей пользователей довольно простое. Вместо сохранения сырого пароля, введенного пользователем, мы используем метод `set_password()`. Это метод пользовательской модели, который применяет шифрование к паролю для его хранения в безопасном виде.

Теперь отредактируйте файл `urls.py` вашего приложения `account` добавив следующий URL паттерн:

```
| path('register/', views.register, name='register'),
```

Наконец, создайте новый файл в каталоге шаблонов `account/`, назовите его `register.html`, добавте в него следующее:

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

{% block title %}Create an account{% endblock %}

{% block content %}
<h1>Create an account</h1>
<p>Please, sign up using the following form:</p>
<form action="." method="post">
    {{ user_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Create my account"></p>
</form>
{% endblock %}
```

Создайте файл шаблона в том же каталоге и назовите его `register_done.html`. Добавьте к нему следующий 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 %}
```

Сейчас откройте `http://127.0.0.1:8000/account/register/` в вашем браузере. Вы увидите страницу регистрации, которую вы создали:

Create an account

Please, sign up using the following form:

Username:

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

First name:

Email address:

Password:

Repeat password:

CREATE MY ACCOUNT

Введите данные для нового пользователя и нажмите на кнопку CREATE MY ACCOUNT . Если все поля действительны, пользователь будет создан, и вы получите следующее сообщение об успешном завершении:

Welcome Paloma!

Your account has been successfully created. Now you can [log in](#).

Нажмите на ссылку log in и введите свое имя пользователя и пароль, чтобы убедиться, что вы можете получить доступ к своей учетной записи.

Теперь вы также можете добавить ссылку на регистрацию в своем шаблоне входа. Откройте файл `registration/login.html` и найдите строку:

```
<p>Please, use the following form to log-in:</p>
```

Замените ее следующим:

```
<p>Please, use the following form to log-in. If you don't have an account <a href="#">{% url "register" %}">register here</a></p>
```

Мы добавили ссылку для регистрации на страницу входа.

Расширение модели user

Когда вам придется иметь дело с учетными записями пользователей, вы обнаружите, что пользовательская модель фреймворка аутентификации Django подходит для обычных случаев. Потому что модель User имеет очень простые поля. Возможно, вы захотите расширить модель, включив дополнительные данные. Лучший способ сделать это - создать модель профиля, содержащую все дополнительные поля в отношении «один к одному» с пользовательской моделью Django.

Отредактируйте файл `models.py` вашего приложения `account` добавив в него следующий код:

```
from django.db import models
from django.conf import settings

class Profile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
                               on_delete=models.CASCADE)
    date_of_birth = models.DateField(blank=True, null=True)
    photo = models.ImageField(upload_to='users/%Y/%m/%d/',
                             blank=True)

    def __str__(self):
        return 'Profile for user {}'.format(self.user.username)
```

Для того, чтобы сохранить ваш код универсальным, используйте метод `get_user_model()` для извлечения User модели и настройки `AUTH_USER_MODEL`, чтобы ссылаться на него при определении отношения модели к User модели, а не ссылаться напрямую к User модели.

Поле `user` «один к одному» позволяет вам связывать профили с пользователями. Мы используем `CASCADE` для параметра `on_delete`, чтобы связанный с ним профиль также удалялся при удалении пользователя. Поле `photo` имеет тип `ImageField`. Вам нужно будет

установить библиотеку `Pillow` для обработки изображений.
Установите `Pillow` выполнив следующую команду в консоли:

```
pip install Pillow==5.1.0
```

Чтобы Django мог обслуживать мультимедийные файлы, загруженные пользователями на сервер разработки, добавьте следующие параметры в файл `settings.py` вашего проекта:

```
MEDIA_URL = '/media/'  
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

`MEDIA_URL` это базовый URL-адрес для загрузки медиафайлов, а `MEDIA_ROOT` – это локальный путь, по которому они находятся. Мы динамически строим путь относительно нашего проекта, чтобы сделать наш код более общим.

Теперь отредактируйте главный файл `urls.py` вашего проекта `bookmarks` модифицировав код следующим образом:

```
from django.contrib import admin  
from django.urls import path, include  
from django.conf import settings  
from django.conf.urls.static import static  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('account/', include('account.urls')),  
]  
  
if settings.DEBUG:  
    urlpatterns += static(settings.MEDIA_URL,  
                          document_root=settings.MEDIA_ROOT)
```

Таким образом, сервер разработки Django будет отвечать за обслуживание медиафайлов во время разработки (то есть, если для параметра `DEBUG` установлено значение `True`).

Функция `static()` подходит для разработки, но не для использования в

производстве. Никогда не обслуживайте ваши статические файлы Django в производственной среде.

Откройте консоль и выполните следующую команду для создания миграции базы данных новой модели:

```
python manage.py makemigrations
```

Вы получите следующий результат:

```
Migrations for 'account':  
  account/migrations/0001_initial.py  
    - Create model Profile
```

Затем выполните синхронизацию базы данных с помощью следующей команды:

```
python manage.py migrate
```

Вы увидите вывод, который включает следующую строку:

```
Applying account.0001_initial... OK
```

Отредактируйте файл `admin.py` приложения `account` и зарегистрируйте модель `Profile` на сайте администратора следующим образом:

```
from django.contrib import admin  
from .models import Profile  
  
@admin.register(Profile)  
class ProfileAdmin(admin.ModelAdmin):  
    list_display = ['user', 'date_of_birth', 'photo']
```

Запустите сервер разработки используя команду `python manage.py runserver` И ОТКРОЙТЕ `http://127.0.0.1:8000/admin/` В вашем браузере. Теперь вы должны увидеть модель Profiles на сайте

администрирования вашего проекта:



The screenshot shows a blue header bar with the word 'ACCOUNT'. Below it, there's a list of 'Profiles'. To the right of the list are two buttons: a green '+' icon labeled 'Add' and a yellow pencil icon labeled 'Change'.

Теперь мы разрешим пользователям редактировать свой профиль на веб-сайте. Добавьте следующий код в файл `forms.py` приложения `account`:

```
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 делают следующие:

- `UserEditForm`: Позволит пользователям редактировать свое имя, фамилию и адрес электронной почты, которые являются атрибутами встроенной пользовательской модели Django.
- `ProfileEditForm`: Позволит пользователям редактировать данные профиля, которые мы сохраняем в пользовательской модели `Profile`. Пользователи смогут отредактировать дату рождения и загрузить изображение для своего профиля.

Отредактируйте файл `views.py` Приложения `account` И импортируйте `Profile` модель:

```
from .models import Profile
```

Затем добавьте следующие строки в представление `register` ниже `new_user.save()`:

```
# Create the user profile
Profile.objects.create(user=new_user)
```

Когда пользователи регистрируются на нашем сайте, мы создаем пустой профиль, связанный с ними. Вы должны создать объект `Profile` вручную, используя сайт администрирования для пользователей, которых вы создали ранее.

Теперь мы разрешим пользователям редактировать их профиль. Добавьте следующий код в тот же файл:

```
from .forms import LoginForm, UserRegistrationForm, \
                  UserEditForm, ProfileEditForm

@login_required
def edit(request):
    if request.method == 'POST':
        user_form = UserEditForm(instance=request.user,
                               data=request.POST)
        profile_form = ProfileEditForm(
            instance=request.user.profile,
            data=request.POST,
            files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
    else:
        user_form = UserEditForm(instance=request.user)
        profile_form = ProfileEditForm(
            instance=request.user.profile)
    return render(request,
                 'account/edit.html',
```

```
{'user_form': user_form,
'profile_form': profile_form})
```

Мы используем декоратор `login_required` потому что пользователи должны пройти аутентификацию для редактирования своего профиля. В этом случае мы используем две формы модели: `UserEditForm` для хранения данных встроенной пользовательской модели и `ProfileEditForm` для хранения дополнительных данных профиля в пользовательской `Profile` модели. Чтобы проверить предоставленные данные, мы выполним метод `is_valid()` для обоих форм. Если обе формы содержат достоверные данные, мы сохраним обе формы, вызвав метод `save()`, чтобы обновить соответствующие объекты в базе данных.

Добавте следующий URL паттерн в файл `urls.py` приложения `account`:

```
path('edit/', views.edit, name='edit'),
```

Наконец, создайте шаблон для этого представления в `templates/account/` и назовите его `edit.html`. Добавьте в него следующий 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 action="." method="post" enctype="multipart/form-data">
{{ user_form.as_p }}
{{ profile_form.as_p }}
{% csrf_token %}
<p><input type="submit" value="Save changes"></p>
</form>
{% endblock %}
```

Мы включили `enctype="multipart/form-data"` в нашей форме, чтобы разрешить загрузку файлов. Мы используем HTML-форму для

отображения форм `user_form` и `profile_form`.

Зарегистрируйте нового пользователя и откройте `http://127.0.0.1:8000/account/edit/`. Вы должны увидеть следующую страницу:

Edit your account

You can edit your account using the following form:

First name:

Paloma

Last name:

Melé

Email address:

paloma@zenxit.com

Date of birth:

1981-04-14

Photo:

no file selected

SAVE CHANGES

Теперь вы также можете отредактировать страницу панели мониторинга и включить ссылки на страницу редактирования профиля и страницу смены пароля. Откройте `account/dashboard.html`

шаблон:

```
|<p>Welcome to your dashboard.</p>
```

Замените предыдущую строку следующей:

```
|<p>Welcome to your dashboard. You can <a href="{% url "edit" %}">edit your  
profile</a> or <a href="{% url "password_change" %}">change your  
password</a>.</p>
```

Теперь пользователи могут получить доступ к форме для редактирования своего профиля из панели управления. Откройте `http://127.0.0.1:8000/account/` в вашем браузере и протестируйте новую ссылку, чтобы отредактировать профиль пользователя:

Dashboard

Welcome to your dashboard. You can [edit your profile](#) or [change your password](#).

Использование собственной модели для пользователя

Django также предлагает способ заменить модель Users своей собственной пользовательской моделью. Ваш класс User должен наследоваться от Django класса `AbstractUser`, который обеспечивает полную реализацию User по умолчанию в качестве абстрактной модели. Вы можете больше узнать об этом

методе <https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#substituting-a-custom-user-model>.

Использование пользовательской модели даст вам больше гибкости, но это также может осложнить интеграцию с подключаемыми приложениями, которые взаимодействуют с пользовательской моделью Django.

Использование фреймворка сообщений

Если вы разрешаете пользователям взаимодействовать с вашей платформой, есть много способов, чтобы вы могли сообщить им о результатах их действий. Django имеет встроенный фреймворк сообщений, который позволяет отображать одноразовые уведомления для ваших пользователей.

Фреймворк сообщений находится в `django.contrib.messages` и включен по умолчанию в список `INSTALLED_APPS` файла `settings.py` когда вы создаете новый проект командой `python manage.py startproject`. Вы заметите, что ваш файл настроек содержит промежуточное программное обеспечение с именем `django.contrib.messages.middleware.MessageMiddleware` в настройках `MIDDLEWARE`.

Фреймворк сообщений обеспечивает простой способ отправки сообщений пользователям. Сообщения хранятся в файле cookie по умолчанию (отсылка к хранилищу сессии), и они отображаются в следующем запросе, который выполняет пользователь. Вы можете использовать фреймворк сообщений в своих представлениях, импортируя модуль `messages` и добавляя новые сообщения с помощью простой функции, как показано ниже:

```
from django.contrib import messages
messages.error(request, 'Something went wrong')
```

Вы можете создавать новые сообщения с помощью метода `add_message()` или любого из нижеследующих методов:

- `success()`: Сообщения успеха, которые будут отображаться после успешного действия
- `info()`: Информационные сообщения
- `warning()`: Неудача еще не случилась, но может наступить
- `error()`: Действие не увенчалось успехом, или что-то не получилось
- `debug()`: Отладочные сообщения, которые будут удалены или проигнорированы в производственной среде

Давайте добавим сообщения на нашу платформу. Поскольку фреймворк сообщений применяется глобально к проекту, мы можем отображать сообщения для пользователя в нашем базовом шаблоне. Откройте шаблон `base.html` приложения `account` и добавьте следующий код между `<div>` элементом с ID `header` и элементом `<div>` с ID `content`:

```
{% if messages %}  
    <ul class="messages">  
        {% for message in messages %}  
            <li class="{{ message.tags }}>  
                {{ message|safe }}  
                <a href="#" class="close">x</a>  
            </li>  
        {% endfor %}  
    </ul>  
{% endif %}
```

Фреймворк сообщений включает в себя процессор контекста `django.contrib.messages.context_processors.messages` что добавляет `messages` переменную к контексту запроса. Вы можете найти его в списке `context_processors` настроек `TEMPLATES` вашего проекта. Вы можете использовать эту переменную в своих шаблонах, чтобы отображать все существующие сообщения пользователю.

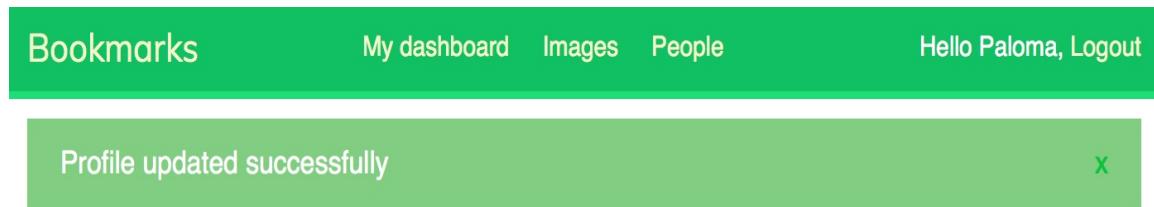
Теперь давайте изменим наше представление редактирования, чтобы использовать фреймворк сообщений. Отредактируйте файл `views.py` пориложения `account`, импортируя `messages`, и сделав представление `edit` похожим на следующее:

```
from django.contrib import messages

@login_required
def edit(request):
    if request.method == 'POST':
        # ...
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
            messages.success(request, 'Profile updated '\
                             'successfully')
        else:
            messages.error(request, 'Error updating your profile')
    else:
        user_form = UserEditForm(instance=request.user)
        # ...
```

Мы добавляем успешное сообщение, когда пользователь успешно обновляет свой профиль. Если какая-либо из форм содержит недопустимые данные, мы вместо этого добавляем сообщение об ошибке.

Откройте `http://127.0.0.1:8000/account/edit/` в вашем браузере и отредактируйте свой профиль. Когда профиль успешно обновлен, вы увидите следующее сообщение:



Если данные недействительны, например, используя неверно отформатированную дату для поля Date of birth, вы должны увидеть следующее сообщение:

Error updating your profile

X

Вы можете больше узнать о фреймворке сообщений в <https://docs.djangoproject.com/en/2.0/ref/contrib/messages/>.

Создание собственного бэкэнда для аутентификации

Django позволяет вам аутентифицироваться из разных источников. Параметр `AUTHENTICATION_BACKENDS` включает в себя список аутентификационных баз для вашего проекта. По умолчанию этот параметр устанавливается следующим образом:

```
['django.contrib.auth.backends.ModelBackend']
```

По умолчанию `ModelBackend` аутентифицирует пользователей применяя базу данных, используя пользовательскую модель `dango.contrib.auth`. Это подойдет для большинства ваших проектов. Тем не менее, вы можете создать собственный бэкэнд для аутентификации пользователя с другими источниками, такими как каталог LDAP или любая другая система.

Вы можете узнать больше о настройке аутентификации в <https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#other-authentication-sources>.

Всякий раз, когда вы используете `authenticate()` функцию `dango.contrib.auth`, Django пытается аутентифицировать пользователя по каждому из бэкендов, определенных в `AUTHENTICATION_BACKENDS`, один за другим, пока один из них не будет успешно аутентифицировать пользователя. Только если все бэкенды не смогут аутентифицироваться, пользователь не будет аутентифицирован на вашем сайте.

Django предоставляет простой способ определить свои собственные бэкэнды аутентификации. Бэкэнд

аутентификации - это класс, который предоставляет следующие два метода:

- `authenticate()`: Он принимает объект `request` и учетные данные пользователя как параметры. Он должен вернуть объект `user`, который соответствует этим учетным данным, если учетные данные действительны, или `None` в противном случае. Параметр `request` является `HttpRequest` объектом, или `None` если он не предоставляется `authenticate()`.
- `get_user()`: Принимает параметр идентификатора пользователя и должен вернуть `user` объект.

Создание собственного бэкэнда аутентификации так же просто, как запись класса Python, который реализует оба метода. Мы создадим бэкэнд аутентификации, чтобы пользователи могли аутентифицироваться на нашем сайте, используя свой адрес электронной почты вместо имени пользователя.

Создайте новый файл в каталоге приложения `account` и назовите его `authentication.py`. Добавьте к нему следующий код:

```
from django.contrib.auth.models import User

class EmailAuthBackend(object):
    """
    Authenticate using an e-mail address.
    """

    def authenticate(self, request, username=None, password=None):
        try:
            user = User.objects.get(email=username)
            if user.check_password(password):
                return user
            return None
        except User.DoesNotExist:
            return None
```

```
def get_user(self, user_id):
    try:
        return User.objects.get(pk=user_id)
    except User.DoesNotExist:
        return None
```

Предыдущий код представляет собой простой бэкэнд аутентификации. Метод `authenticate()` получает объект `request` и необязательные параметры `username` и `password`. Мы могли бы использовать разные параметры, но мы используем `username` и `password` чтобы обеспечить работу с фреймворками проверки подлинности. Предыдущий код работает следующим образом:

- `authenticate()`: Мы пытаемся получить пользователя с заданным адресом электронной почты и проверить пароль с помощью встроенного метода `check_password()` модели пользователя. Этот метод хеширует пароль для сравнения данного пароля с тем что хранится в базе данных.
- `get_user()`: Мы получаем пользователя через идентификатор, установленный в параметре `user_id`. Django использует бэкэнд, который аутентифицировал пользователя для извлечения объекта `User` в течение всей пользовательской сессии.

Отредактируйте файл `settings.py` добавив в него следующие настройки:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
]
```

В предыдущей настройке мы сохраняли значение по умолчанию `ModelBackend` которое используется для аутентификации с именем пользователя и паролем и включает наш собственный бэкэнд на основе электронной почты. Теперь откройте `http://127.0.0.1:8000/account/login/` в вашем браузере. Помните, что Django попытается выполнить аутентификацию пользователя по отношению к каждому из бэкэндов, поэтому теперь мы сможем легко входить в систему, используя ваше имя пользователя или учетную запись электронной почты. Пользовательские учетные данные будут проверяться с помощью `ModelBackend` аутентификации и если пользователь не будет возвращен, учетные данные будут проверяться с использованием нашего пользователяского `EmailAuthBackend` бэкенда.

Порядок бэкендов, перечислены в `AUTHENTICATION_BACKENDS` настройке. Если одни и те же учетные данные действительны для нескольких бэкендов, Django остановится на первом бэкэнд, который успешно аутентифицирует пользователя.

Добавление социальной аутентификации на ваш сайт

Вы также можете добавить социальную аутентификацию на свой сайт, используя Facebook, Twitter или Google. Python Social Auth - это модуль Python, который упрощает процесс добавления социальной аутентификации на наш веб-сайт.

Благодаря этому модулю, вы можете позволить своим пользователям войти на ваш сайт, используя учетную запись других служб. Вы можете найти код этого модуля в <https://github.com/python-social-auth>.

Этот модуль поставляется с аутентификационными базами для разных фреймворков Python, включая Django. Чтобы установить пакет в Django с помощью pip, откройте консоль и выполните следующую команду:

```
pip install social-auth-app-django==2.1.0
```

Затем добавьте social_django к INSTALLED_APPS настройкам в файл settings.py вашего проекта:

```
INSTALLED_APPS = [  
    #...  
    'social_django',  
]
```

Это приложение по умолчанию для добавления проектов python-social-auth в Django. Теперь запустите следующую команду, чтобы синхронизировать модели python-social-auth с вашей базой данных:

```
python manage.py migrate
```

Вы должны увидеть, что миграция для приложения по умолчанию применяется следующим образом:

```
Applying social_django.0001_initial... OK
Applying social_django.0002_add_related_name... OK
...
Applying social_django.0008_partial_timestamp... OK
```

Python-social-auth включает в себя бэкенды для нескольких сервисов. Вы можете увидеть список всех <https://python-social-auth.readthedocs.io/en/latest/backends/index.html#supported-backends>.

Мы включим аутентификационные базы для Facebook, Twitter и Google.

Вам нужно будет добавить паттерны URL для социального входа в свой проект. Откройте основной файл `urls.py` проекта `bookmarks` и включите URL-адрес `social_django` следующим образом:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('account/', include('account.urls')),
    path('social-auth/',
        include('social_django.urls', namespace='social')),
]
```

После успешной проверки подлинности несколько социальных служб не позволят перенаправлять пользователей на `127.0.0.1` или `localhost`. Чтобы сделать работу по социальной аутентификации, вам понадобится домен. Чтобы исправить это, под Linux или macOS X отредактируйте свой `/etc/hosts` добавив в него:

```
127.0.0.1 mysite.com
```

Это сообщит компьютеру о том, что `mysite.com` ИМЯ хоста для вашей собственной машины. Если вы используете Windows, ваш файл `hosts` находится по адресу `C:\Windows\System32\Drivers\etc\hosts`.

Чтобы убедиться, что перенаправление хоста работает, запустите сервер разработки с помощью `python manage.py runserver` и откройте `http://mysite.com:8000/account/login/` в Вашем браузере. Появится следующая ошибка:

DisallowedHost at /account/login/

Invalid HTTP_HOST header: 'mysite.com:8000'. You may need to add 'mysite.com' to ALLOWED_HOSTS.

Django управляет хостами, которые могут обслуживать ваше приложение, используя `ALLOWED_HOSTS` настройки. Это мера безопасности для предотвращения атак HTTP-заголовков. Django разрешает только хостам, включенными в этот список, обслуживать приложение. Вы можете узнать больше о `ALLOWED_HOSTS` настройках в <https://docs.djangoproject.com/en/2.0/ref/settings/#allowed-hosts>.

Откройте файл `settings.py` вашего проекта и добавьте в настройки `ALLOWED_HOSTS` следующее:

```
ALLOWED_HOSTS = ['mysite.com', 'localhost', '127.0.0.1']
```

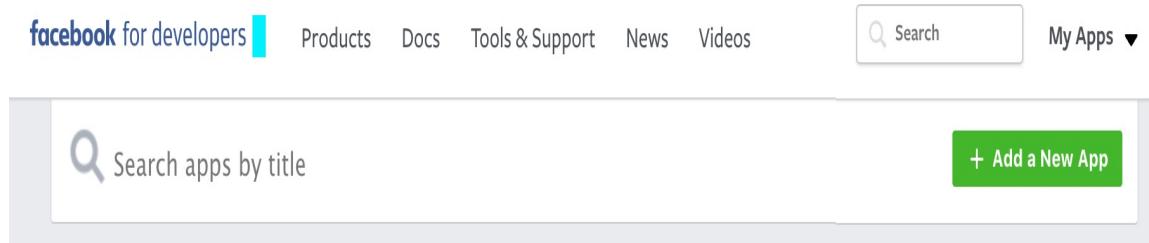
Помимо хоста `mysite.com` мы явно включаем `localhost` и `127.0.0.1`. Мы делаем это, чтобы иметь возможность доступа к сайту через `localhost`, который устанавливается в Django по умолчанию, когда `DEBUG` равно `True` и `ALLOWED_HOSTS` пустой. Теперь вы сможете открыть `http://mysite.com:8000/account/login/` в вашем браузере.

Аутентификация с использованием Facebook

Чтобы ваши пользователи вошли через учетную запись Facebook на ваш сайт, добавьте следующую строку в AUTHENTICATION_BACKENDS в файле `settings.py` Вашего проекта:

```
'social_core.backends.facebook.FacebookOAuth2',
```

Чтобы добавить социальную аутентификацию с Facebook, вам нужна учетная запись разработчика Facebook и создать новое приложение Facebook. Откройте <https://developers.facebook.com/apps/> в вашем браузере. На этом сайте вы увидите следующий заголовок:



Нажмите на кнопку Add a New App. Вы увидите следующую форму для создания нового ID приложения:

Create a New App ID

Get started integrating Facebook into your app or website

Display Name

Bookmarks

Contact Email

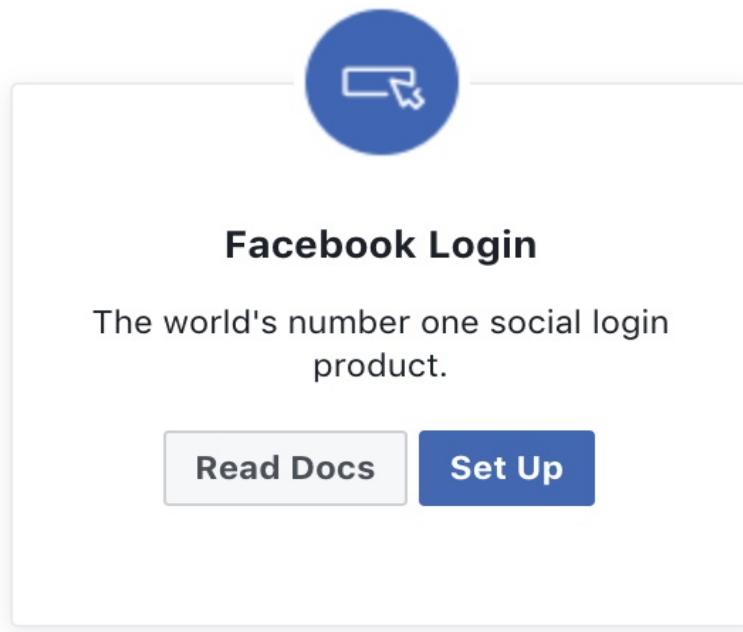
antonio.mele@zenxit.com

By proceeding, you agree to the [Facebook Platform Policies](#)

[Cancel](#)

[Create App ID](#)

Введите Bookmarks в Display Name, добавьте контактный адрес электронной почты и нажмите на Create App ID. Вы увидите панель для вашего нового приложения, в которой отображаются различные функции, которые вы можете настроить для своего приложения. Найдите следующее Facebook Login и нажмите Set Up:



Вам будет предложено выбрать платформу следующим образом:

Use the Quickstart to add Facebook Login to your app. To get started, select the platform for this app.



Выберите Web платформу. Вы увидите следующую форму:

1. Tell Us about Your Website



Tell us what the URL of your site is.

Site URL

Save

Continue

Введите `http://mysite.com:8000/` как ваш Site URL и нажмите кнопку Save. Вы можете пропустить оставшуюся часть процесса быстрого запуска. В левом меню нажмите Dashboard. Вы увидите что-то похожее на следующее:

The screenshot shows the Facebook App Dashboard for an app named 'Bookmarks'. The left sidebar contains links for Dashboard, Settings, Roles, Alerts, App Review, and a section for PRODUCTS with options like Facebook Login and '+ Add Product'. The main dashboard area displays the app's logo, name ('Bookmarks'), and status ('This app is in development mode and can only be used by app admins, developers and testers [?]'). It also shows API Version (v2.10), App ID (1865597340135974), and a redacted App Secret. A 'Show' button is available for the App Secret.

Скопируйте App ID и App Secret ключи и добавьте их в файл `settings.py`:

```
SOCIAL_AUTH_FACEBOOK_KEY = 'XXX' # Facebook App ID
SOCIAL_AUTH_FACEBOOK_SECRET = 'XXX' # Facebook App Secret
```

При желании вы можете определить `SOCIAL_AUTH_FACEBOOK_SCOPE` с дополнительными разрешениями, которые вы хотите задать для пользователей Facebook:

```
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email']
```

Теперь вернитесь в Facebook и нажмите **Settings**. Вы увидите форму с несколькими настройками для своего приложения. Добавте `mysite.com` под App Domains, следующим образом:

App Domains



mysite.com ×

Нажмите на **Save Changes**. Затем в левом меню нажмите **Facebook Login**. Убедитесь, что активны только следующие настройки:

- Client OAuth Login
- Web OAuth Login
- Embedded Browser OAuth Login

Ведите `http://mysite.com:8000/social-auth/complete/facebook/` под **Valid OAuth redirect URIs**. Выбор должен выглядеть следующим образом:

Client OAuth Settings



Client OAuth Login

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. [?]



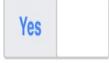
Web OAuth Login

Enables web based OAuth client login for building custom login flows. [?]



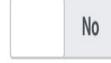
Force Web OAuth Reauthentication

When on, prompts people to enter their Facebook password in order to log in on the web. [?]



Embedded Browser OAuth Login

Enables browser control redirect uri for OAuth client login. [?]

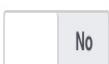


Use Strict Mode for Redirect URIs

Only allow redirects that use the Facebook SDK or that exactly match the Valid OAuth Redirect URIs. Strongly recommended. [?]

Valid OAuth redirect URIs

http://mysite.com:8000/social-auth/complete/facebook/ 



Login from Devices

Enables the OAuth client login flow for devices like a smart TV [?]

Откройте шаблон `registration/login.html` вашего приложения `account` и добавьте следующий код в нижнюю часть блока `content`:

```
<div class="social">
  <ul>
    <li class="facebook"><a href="{% url "social:begin" "facebook" %}">Sign
      in with Facebook</a></li>
  </ul>
</div>
```

Откройте `http://mysite.com:8000/account/login/` в вашем браузере.
Теперь страница входа будет выглядеть следующим образом:

Bookmarks Log-in

Log-in

Please, use the following form to log-in. If you don't have an account [register here](#)

Username:

Sign in with Facebook

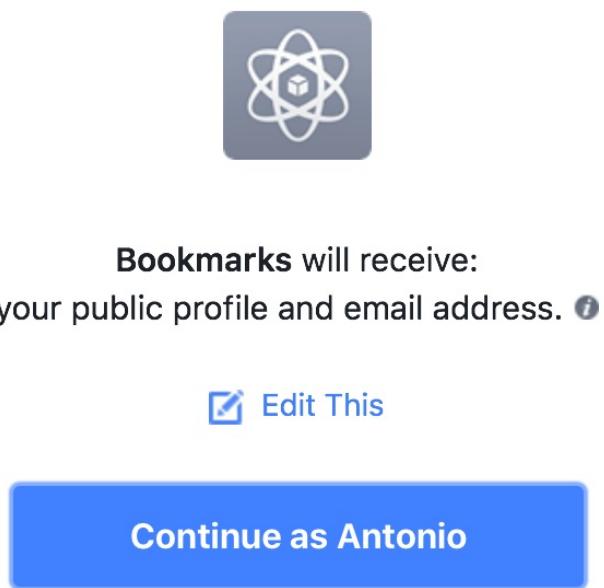
Password:

LOG-IN

[Forgotten your password?](#)

Нажмите на кнопку Sign in with Facebook. Вы будете

перенаправлены на Facebook, и вы увидите модальный диалог с просьбой разрешить *Bookmarks* приложению, доступ к вашему общедоступному профилю Facebook:



Нажмите на кнопку Continue as Вы войдете в систему и будете перенаправлены на страницу панели мониторинга вашего сайта. Помните, что мы установили этот URL в `LOGIN_REDIRECT_URL` настройке. Как вы можете видеть, добавить социальную аутентификации на ваш сайт довольно легко.

Аутентификация с помощью Twitter

Для социальной аутентификации с использованием Twitter добавьте следующую строку в `AUTHENTICATION_BACKENDS` настройки в файл `settings.py` вашего проекта:

```
'social_core.backends.twitter.TwitterOAuth',
```

Вам нужно будет создать новое приложение в своей учетной записи Twitter. Откройте <https://apps.twitter.com/app/new> в вашем браузере. Вы увидите следующую форму:

Application Details

Name *

Bookmarks

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Test Django application.

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

http://mysite.com:8000/

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.

(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

http://mysite.com:8000/social-auth/complete/twitter/

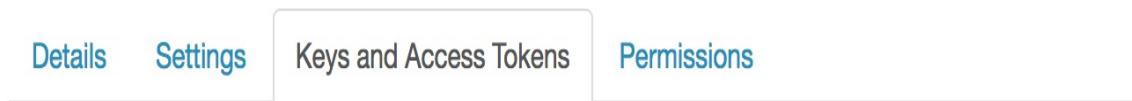
Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Ведите информацию о своем приложении, включая следующие настройки:

- Website: <http://mysite.com:8000/>
- Callback URL: <http://mysite.com:8000/social-auth/complete/twitter/>

Затем нажмите Create your Twitter application. Вы увидите детали приложения. Нажмите на Keys and Access Tokens. Вы должны увидеть следующую информацию:

Bookmarks



Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

Consumer Key (API Key) eJJU1AzzEQFJ6PAgqLjc18TH1

Consumer Secret (API Secret)

Access Level Read and write (modify app permissions)

Скопируйте Consumer Key и Consumer Secret ключи в файл настроек в `settings.py` вашего проекта:

```
SOCIAL_AUTH_TWITTER_KEY = 'XXX' # Twitter Consumer Key  
SOCIAL_AUTH_TWITTER_SECRET = 'XXX' # Twitter Consumer Secret
```

Теперь откройте шаблон `registration/login.html` и добавьте следующий код в `` элемент:

```
<li class="twitter"><a href="{% url "social:begin" "twitter" %}">Login with Twitter</a></li>
```

Откройте `http://mysite.com:8000/account/login/` в вашем браузере и нажмите ссылку Login with Twitter. Вы будете перенаправлены в Twitter, и он попросит вас разрешить заявку следующим образом:

Authorize Bookmarks Test to use your account?



Bookmarks

mysite.com:8000/

Test Django application.

Authorize app

Cancel

This application will be able to:

- Read Tweets from your timeline.
- See who you follow.

Will not be able to:

- Follow new people.
- Update your profile.
- Post Tweets for you.
- Access your direct messages.
- See your email address.
- See your Twitter password.

Нажмите на Authorize app. Вы войдете в систему и будете перенаправлены на страницу панели мониторинга вашего сайта.

Аутентификация с помощью Google

Google предлагает аутентификацию OAuth2. Вы можете прочитать о реализации OAuth2 в Google <https://developers.google.com/identity/protocols/OAuth2>.

Чтобы реализовать аутентификацию с помощью Google, добавьте следующую строку в AUTHENTICATION_BACKENDS в файл settings.py вашего проекта:

```
'social_core.backends.google.GoogleOAuth2',
```

Во-первых, вам нужно будет создать ключ API в консоли разработчика Google. Откройте <https://console.developers.google.com/apis/credentials> в вашем браузере. Нажмите на Select a project чтобы создать новый проект:



New Project

i You have 12 projects remaining in your quota. [Learn more.](#)

Project name

Bookmarks

Your project ID will be bookmarks-185117 [Edit](#)

[Create](#)

[Cancel](#)

После того, как проект будет создан, в Credentials, нажмите на Create credentials и выберите OAuth client ID:

APIs

Credentials

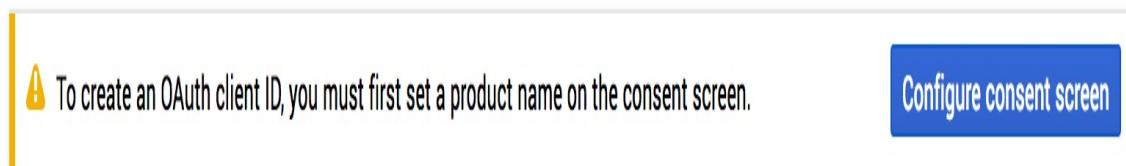
You need credentials to access APIs. [Enable the APIs that you plan to use](#) and then create the credentials that they require.

Depending on the API, you need an API key, a service account or an OAuth 2.0 client ID. [Refer to the API documentation](#) for details.

[Create credentials ▾](#)

API key	Identifies your project using a simple API key to check quota and access.
OAuth client ID	Requests user consent so your app can access the user's data.
Service account key	Enables server-to-server, app-level authentication using robot accounts.
Help me choose	Asks a few questions to help you decide which type of credential to use

Google попросит вас сначала настроить экран согласия:



На предыдущем рисунке отображается страница, которая будет показана пользователям, чтобы дать свое согласие на доступ к вашему сайту с помощью своей учетной записи Google. Нажмите на кнопку **Configure consent screen**. Выберите свой адрес электронной почты, введите Bookmarks под **Product name**, и нажмите на кнопку **Save**. Будет настроен экран согласия для вашего приложения, и вы будете перенаправлены, чтобы завершить создание своего идентификатора клиента ID.

Заполните форму со следующей информацией:

- Application type: Выберите Web application
- Name: Введите Bookmarks
- Authorized redirect URIs: Добавьте `http://mysite.com:8000/social-auth/complete/google-oauth2/`

Форма должна выглядеть так:

Application type

Web application
 Android [Learn more](#)
 Chrome App [Learn more](#)
 iOS [Learn more](#)
 PlayStation 4
 Other

Name

Bookmarks

Restrictions

Enter JavaScript origins, redirect URIs or both

Authorised JavaScript origins

For use with requests from a browser. This is the origin URI of the client application. It cannot contain a wildcard (`https://*.example.com`) or a path (`https://example.com/subdir`). If you're using a non-standard port, you must include it in the origin URI.

https://www.example.com

Authorised redirect URIs

For use with requests from a web server. This is the path in your application that users are redirected to after they have authenticated with Google. The path will be appended with the authorisation code for access. Must have a protocol. Cannot contain URL fragments or relative paths. Cannot be a public IP address.

http://mysite.com:8000/social-auth/complete/google-oauth2/ ×

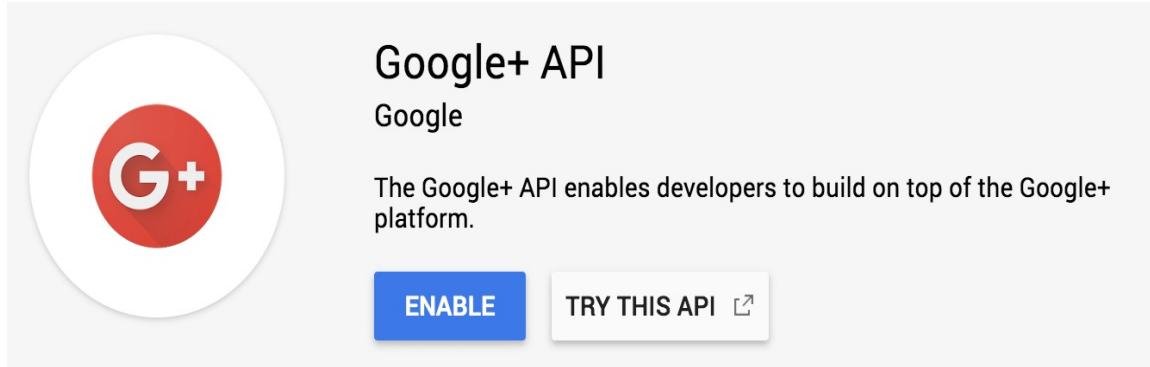
https://www.example.com/oauth2callback

Create Cancel

Нажмите на кнопку Create. Вы получите Client ID и Client Secret ключи. Добавте их в ваш файл `settings.py`:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = 'XXX' # Google Consumer Key  
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = 'XXX' # Google Consumer Secret
```

В левом меню консоли Google Developers Console APIs & Services section, кликните на ссылку Library. Вы увидите список, содержащий все API Google. Нажмите на Google+ API и кликните на кнопку ENABLE на следующей странице:



Откройте шаблон `login.html` и добавьте следующий код в `` элемент:

```
<li class="google"><a href="{% url "social:begin" "google-oauth2" %}">Login  
with Google</a></li>
```

Откройте `http://mysite.com:8000/account/login/` в вашем браузере. Страница входа в систему должна выглядеть следующим образом:

Log-in

Please, use the following form to log-in. If you don't have an account [register here](#)

Username:

[Sign in with Facebook](#)

Password:

[Login with Twitter](#)

[Login with Google](#)

LOG-IN

Нажмите на кнопку Login with Google. Вы будете зарегистрированы и перенаправлены на страницу панели управления вашего сайта.

Вы добавили в свой проект социальную аутентификацию. Вы можете легко реализовать социальную аутентификацию с помощью других популярных онлайн-сервисов, используя Python Social Auth.

Резюме

В этой главе вы узнали, как создать систему аутентификации на своем сайте и создать профили пользователей. Вы также добавили социальную аутентификацию на свой сайт.

В следующей главе вы узнаете, как создать систему закладок изображений, создавать эскизы изображений и создавать представления AJAX.

Глава 5

Совместное использование контента на вашем сайте

В предыдущей главе вы создали учетную запись пользователя и аутентификацию на своем веб-сайте. Вы научились создавать модель пользовательского профиля и добавили социальную аутентификацию на свой сайт с помощью крупных социальных сетей.

В этой главе вы узнаете, как создать bookmarklet JavaScript для совместного использования контента с других сайтов на вашем веб-сайте, и вы будете использовать функции AJAX в своем проекте, используя jQuery и Django.

В этой главе будут рассмотрены следующие вопросы:

- Создание отношений «многие ко многим»
- Настройка поведения для форм
- Использование jQuery с Django
- Создание jQuery bookmarklet
- Создание эскизов изображений с использованием sorl-thumbnail
- Внедрение представлений AJAX и их интеграция с

jQuery

- Создание пользовательских декораторов для представлений
- Построение AJAX пагинации

Создание веб-сайта закладок (bookmarking) изображений

Мы позволим пользователям, на нашем сайте, добавлять и обмениваться ссылками с других сайтов. Для этого нам нужно будет выполнить следующие задачи:

1. Спроектировать модель для хранения изображений и информацию о них
2. Создать форму и представление для обработки изображений
3. Создать систему, в которой пользователи смогут добавлять изображения, которые они находят на внешних сайтах

Сначала создайте новое приложение в каталоге проекта `bookmarks` с помощью следующей команды:

```
django-admin startapp images
```

Добавте приложение в `INSTALLED_APPS` файла `settings.py` следующим образом:

```
INSTALLED_APPS = [  
    # ...  
    'images.apps.ImagesConfig',  
]
```

Мы активировали приложение `images` в проекте.

Построение модели изображения

Откройте файл `models.py` приложения `images` и добавьте следующий код в него:

```
from django.db import models
from django.conf import settings

class Image(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL,
                            related_name='images_created',
                            on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200,
                           blank=True)
    url = models.URLField()
    image = models.ImageField(upload_to='images/%Y/%m/%d/')
    description = models.TextField(blank=True)
    created = models.DateTimeField(auto_now_add=True,
                                   db_index=True)

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

Это модель, которую мы будем использовать для хранения изображений, размещенных на разных сайтах. Давайте посмотрим на поля этой модели:

- `user`: Указывает на объект `User`, который помещает изображение в закладки. Это поле внешнего ключа, поскольку оно определяет отношения «один ко многим». Пользователь может отправлять несколько изображений, но каждое изображение отправляется

одним пользователем. Мы используем `CASCADE` для параметра `on_delete` чтобы связанные изображения также удалялись, когда удаляется пользователь.

- `title`: Заголовок для изображения.
- `slug`: Короткая метка, содержащая только буквы, цифры, символы подчеркивания или дефисы, которые будут использоваться для создания красивых URL-адресов, ориентированных на SEO.
- `url`: Исходный URL для этого изображения.
- `image`: Файл изображения.
- `description`: Необязательное описание для изображения.
- `created`: Дата и время, указывающие, когда объект был создан в базе данных. Поскольку мы используем `auto_now_add`, этот `datetime` автоматически устанавливается при создании объекта. Мы используем `db_index=True` так чтобы Django создавал индекс в базе данных для этого поля.

Индексы базы данных улучшают производительность запросов. Установите настройку `db_index=True` для полей, которые вы часто запрашиваете с помощью `filter()`, `exclude()`, или `order_by()`. Поля `ForeignKey` или поля с `unique=True` подразумевают создание индекса. Вы также можете использовать `Meta.index_together` для создания индексов для нескольких полей.

Мы переопределим метод `save()` модели `Image`, чтобы автоматически генерировать поле `slug` на основе значения поля `title`. Импортируйте функцию `slugify()` и добавьте метод `save()` в модель `Image` следующим образом:

```
| from django.utils.text import slugify
```

```
class Image(models.Model):
    # ...
    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.title)
        super(Image, self).save(*args, **kwargs)
```

В предыдущем коде мы используем функцию `slugify()`, предоставленную Django, чтобы автоматически генерировать `slug` изображений для данного `title`. Затем мы сохраним объект. Мы будем генерировать `slug` для изображений автоматически, чтобы пользователям не приходилось вручную вводить их для каждого изображения.

Создание отношений «многие ко многим»

Мы добавим другое поле в модель `Image`, чтобы сохранить пользователей, которым нравится изображение. В этом случае нам понадобятся отношения «многие-ко-многим», потому что пользователю может нравиться несколько изображений, и каждое изображение может понравиться нескольким пользователям.

Добавьте следующее поле в модель `Image`:

```
users_like = models.ManyToManyField(settings.AUTH_USER_MODEL,
                                    related_name='images_liked',
                                    blank=True)
```

Когда вы определяете `ManyToManyField`, Django создает промежуточную связывающую таблицу, используя первичные ключи обеих моделей. `ManyToManyField` может быть определен в любой из двух связанных моделей.

Как и в полях `ForeignKey`, атрибут `related_name` отношения `ManyToManyField` позволяет нам именовать отношение из связанного объекта обратно к этому. В полях `ManyToManyField` предоставляется менеджер «многие-ко-многим», который позволяет нам получать связанные объекты, такие как `image.users_like.all()`, или от `user`, например `user.images_liked.all()`.

Откройте командную строку и выполните следующую команду для создания начальной миграции:

```
python manage.py makemigrations images
```

Вы должны увидеть следующий результат:

```
Migrations for 'images':  
  images/migrations/0001_initial.py  
    - Create model Image
```

Теперь для выполнения миграции выполните следующую команду:

```
python manage.py migrate images
```

Вы получите вывод, который включает следующую строку:

```
Applying images.0001_initial... ok
```

Теперь модель `Image` синхронизировалась с базой данных.

Регистрация модели изображения на сайте администрирования

Откройте файл `admin.py` приложения `images` и зарегистрируйте модель `Image` на сайте администратора:

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

@admin.register(Image)
class ImageAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'image', 'created']
    list_filter = ['created']
```

Запустите сервер разработки с помощью команды `python manage.py runserver`. Откройте `http://127.0.0.1:8000/admin/` в вашем браузере, и вы увидите `Image` модель на сайте администрирования:

IMAGES

Images

 Add  Change

Публикация контента с других сайтов

Мы разрешим пользователям добавлять закладки с внешних веб-сайтов. Пользователь предоставит URL-адрес изображения, заголовок и дополнительное описание. Наше приложение загрузит изображение и создаст новый объект `Image` в базе данных.

Начнем с создания формы для отправки новых изображений. Создайте новый файл `forms.py` внутри каталога приложений `Images` и добавьте к нему следующий код:

```
from django import forms
from .models import Image

class ImageCreateForm(forms.ModelForm):
    class Meta:
        model = Image
        fields = ('title', 'url', 'description')
        widgets = {
            'url': forms.HiddenInput,
        }
```

Как вы заметили в предыдущем коде, эта форма представляет собой форму `ModelForm`, построенную из модели `Image`, включая только поля `title`, `url`, и `description`. Пользователи не будут вводить URL-адрес изображения непосредственно в форме. Вместо этого мы предоставим им инструмент JavaScript для выбора изображения с внешнего сайта, и наша форма получит его URL как параметр. Мы переопределяем виджет по умолчанию для поля `url`, чтобы использовать виджет `HiddenInput`. Этот виджет отображается как элемент ввода HTML с атрибутом `type="hidden"`. Мы используем его, потому что мы не хотим, чтобы это поле было

видимым для пользователей.

Очистка полей формы

Чтобы убедиться, что URL-адрес предоставленного изображения действителен, мы проверим, что имя файла заканчивается расширением `.jpg` или `.jpeg`, чтобы разрешать только файлы JPEG. Как вы видели в предыдущей главе, Django позволяет вам определять методы формы для очистки определенных полей с помощью нотации `clean_<fieldname>()`. Этот метод выполняется для каждого поля, когда вы вызываете `is_valid()` в экземпляре формы. В этом методе вы можете изменить значение поля или при необходимости получить какие-либо ошибки проверки для этого конкретного поля. Добавьте следующий метод для `ImageCreateForm`:

```
def clean_url(self):
    url = self.cleaned_data['url']
    valid_extensions = ['jpg', 'jpeg']
    extension = url.rsplit('.', 1)[1].lower()
    if extension not in valid_extensions:
        raise forms.ValidationError('The given URL does not ' \
                                    'match valid image extensions.')
    return url
```

В предыдущем коде мы определяем метод `clean_url()` для очистки поля `url`. Код работает следующим образом:

1. Мы получаем значение поля `url`, обращаясь к словарю `cleaned_data` экземпляра формы.
2. Мы разделили URL-адрес, чтобы получить расширение файла, и проверить, является ли оно одним из допустимых расширений. Если расширение недействительно, мы вызываем `ValidationError`, и

экземпляр формы не будет проверен. Здесь мы выполняем очень простую проверку. Вы можете использовать более сложные методы, чтобы проверить, содержит ли данный URL действительный файл изображения.

В дополнение к проверке данного URL, нам также необходимо загрузить файл изображения и сохранить его. Мы могли бы, например, использовать представление, которое обрабатывает форму для загрузки файла изображения. Вместо этого мы возьмем более общий подход, переопределив метод `save()` нашей типовой формы для выполнения этой задачи каждый раз, когда форма сохраняется.

Переопределение метода save() в ModelForm

Как вы знаете `ModelForm` предоставляет метод `save()`, чтобы сохранить текущий экземпляр модели в базу данных и вернуть объект. Этот метод получает `boolean` параметр `commit`, который позволяет указать, должен ли объект сохраняться в базе данных. Если `commit` равен `False`, метод `save()` вернет экземпляр модели, но не сохранит его в базе данных. Мы переопределим метод `save()` нашей формы, чтобы получить данное изображение и сохранить его.

Добавьте следующий импорт в начало файла `forms.py`:

```
from urllib import request
from django.core.files.base import ContentFile
from django.utils.text import slugify
```

Затем добавьте следующий метод `save()` к форме `ImageCreateForm`:

```
def save(self, force_insert=False,
         force_update=False,
         commit=True):
    image = super(ImageCreateForm, self).save(commit=False)
    image_url = self.cleaned_data['url']
    image_name = '{}.{}'.format(slugify(image.title),
                                image_url.rsplit('.', 1)[1].lower())

    # скачать изображение с данного URL-адреса
    response = request.urlopen(image_url)
    image.image.save(image_name,
                     ContentFile(response.read()),
                     save=False)
    if commit:
        image.save()
    return image
```

Мы переопределяем метод `save()`, сохраняя параметры, необходимые `ModelForm`. Предыдущий код работает следующим образом:

1. Мы создаем новый экземпляр `image`, вызывая метод формы `save()` с параметром `commit=False`.
2. Мы получаем URL-адрес из словаря формы `cleaned_data`.
3. Мы генерируем имя изображения, комбинируя `slug` заголовка `image` с исходным расширением файла.
4. Мы используем модуль Python `urllib` для загрузки изображения, а затем вызываем метод `save()` поля `image`, передавая ему объект `contentFile` который создается с загруженным содержимым файла. Таким образом, мы сохраняем файл в медиа-каталоге нашего проекта. Мы также передаем параметр `save=False`, чтобы избежать сохранения объекта в базе данных.
5. Чтобы поддерживать то же поведение, что и метод `save()`, который мы переопределяем, мы сохраняем форму в базе данных только тогда, когда параметр `commit` равен `True`.

Теперь нам понадобится представление для обработки формы. Отредактируйте файл `views.py` приложения изображений и добавьте к нему следующий код:

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
from django.contrib import messages
from .forms import ImageCreateForm

@login_required
def image_create(request):
```

```
if request.method == 'POST':
    # форма отправлена
    form = ImageCreateForm(data=request.POST)
    if form.is_valid():
        # данные формы действительны
        cd = form.cleaned_data
        new_item = form.save(commit=False)

        # назначить текущего пользователя элементу
        new_item.user = request.user
        new_item.save()
        messages.success(request, 'Image added successfully')

        # перенаправить на внов созданны объект детального представления
        return redirect(new_item.get_absolute_url())
else:
    # создать форму с данными, предоставленными букмарклетом через GET
    form = ImageCreateForm(data=request.GET)

return render(request,
              'images/image/create.html',
              {'section': 'images',
               'form': form})
```

Мы добавляем декоратор `login_required` в представление `image_create`, чтобы предотвратить доступ для пользователей, не прошедших аутентификацию. Вот как выглядит это представление:

1. Мы ожидаем получить исходные данные через `GET`, чтобы создать экземпляр формы. Эти данные будут состоять из атрибутов изображения `url` и `title` с внешнего веб-сайта и будут предоставлены через метод `GET` с помощью инструмента `JavaScript`, который мы создадим позже. Пока мы просто предполагаем, что эти данные будут там изначально.
2. Если форма отправлена, мы проверяем, действительна ли она. Если данные формы действительны, мы создаем новый экземпляр `Image`, но препятствуем сохранению объекта в базе данных путем передачи параметра

`commit=False` методу `save()`.

3. Мы назначаем текущего пользователя новому объекту `image`. Так мы можем узнать, кто загрузил каждое изображение.
4. Мы сохраняем объект `image` в базе данных.
5. Наконец, мы создаем сообщение об успешной работе с использованием платформы обмена сообщениями Django и перенаправляем пользователя на канонический URL нового изображения. Мы еще не реализовали метод `get_absolute_url()` модели `Image`; мы это сделаем позже.

Создайте новый файл `urls.py` в приложении `images` и добавте в него следующий 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` включив в него шаблон приложения `images`:

```
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, вам нужно будет создать шаблон для визуализации

формы. Создайте следующую структуру каталогов внутри каталога приложений `images`:

```
templates/
  images/
    image/
      create.html
```

Откройте новый шаблон `create.html` и добавте в него следующий код:

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

{% block title %}Bookmark an image{% endblock %}

{% block content %}
  <h1>Bookmark an image</h1>
  
  <form action=". " method="post">
    {{ form.as_p }}
    {% csrf_token %}
    <input type="submit" value="Bookmark it!">
  </form>
{% endblock %}
```

Теперь откройте `http://127.0.0.1:8000/images/create/?title=...&url=...` в вашем браузере вставив GET параметры `title` и `url` содержащие существующий URL изображения с расширением JPEG в конце.

Например, вы можете использовать следующий URL:

`http://127.0.0.1:8000/images/create/?title=%20Django%20and%20Duke&url=http://upload.wikimedia.org/wikipedia/commons/8/85/Django_Reinhardt_and_Duke_Ellington_%28Gottlieb%29.jpg`.

Вы увидите форму предварительного просмотра изображения, как показано ниже:

Bookmark an image



Title:

Django and Duke

Description:

BOOKMARK IT!

Добавьте описание и нажмите на кнопку **BOOKMARK IT!**.
Новый `Image` объект будет сохранен в базе данных. Однако вы получите сообщение об ошибке, указывающее, что модель `Image` не имеет метода `get_absolute_url()`, как показано ниже:

AttributeError at /images/create/
`'Image' object has no attribute 'get_absolute_url'`

Не беспокойтесь об этом сейчас; мы добавим этот метод позже.
Откройте `http://127.0.0.1:8000/admin/images/image/` в вашем браузере и убедитесь, что новый `image` объект был сохранен, например:

Action: 0 of 1 selected

<input type="checkbox"/>	TITLE	SLUG	IMAGE	CREATED
<input type="checkbox"/>	Django and Duke	django-and-duke	images/2017/11/05/django-and-duke.jpg	Dec. 16, 2017

Создание bookmarklet с помощью jQuery

bookmarklet - это закладка, хранящаяся в веб-браузере, которая содержит код JavaScript для расширения функциональности браузера. Когда вы нажимаете на закладку, код JavaScript выполняется на веб-странице, отображаемой в браузере. Это очень полезно для создания инструментов, которые взаимодействуют с другими веб-сайтами.

Некоторые онлайн-сервисы, такие как Pinterest, реализуют свои собственные букмарклеты, позволяя пользователям совместно использовать контент с других сайтов на своей платформе. Аналогичным образом мы создадим букмарклет, чтобы пользователи могли обмениваться изображениями с других сайтов на нашем сайте.

Мы будем использовать jQuery для создания нашего букмарклета. jQuery - популярный фреймворк JavaScript, который позволяет быстрее развивать клиентскую функциональность. Вы можете узнать больше о jQuery на официальном сайте, <https://jquery.com/>.

Вот как ваши пользователи будут добавлять букмарклет в свой браузер и использовать его:

1. Пользователь перетаскивает ссылку со своего сайта на закладки своего браузера. Ссылка содержит код JavaScript в атрибуте `href`. Этот код будет сохранен в закладке.
2. Пользователь переходит на любой сайт и щелкает по

закладке. Выполняется JavaScript-код закладки.

Поскольку код JavaScript будет сохранен как закладка, вы не сможете его обновить позже. Это важный недостаток, который вы можете решить, выполнив сценарий запуска, чтобы загрузить актуальный JavaScript из URL-адреса. Ваши пользователи сохранят этот сценарий запуска в качестве закладки, и вы сможете в любое время обновить код блюмрклета. Это подход, который мы будем использовать для создания нашего блюмрклета. Давайте начнем!

Создайте новый шаблон `images/templates/` и назовите его `bookmarklet_launcher.js`. Это будет сценарий запуска. Добавьте следующий код JavaScript в этот файл:

```
(function(){
    if (window.myBookmarklet !== undefined){
        myBookmarklet();
    }
    else {

        document.body.appendChild(document.createElement('script')).src='http://127.0
        .0.1:8000/static/js/bookmarklet.js?
        r='+Math.floor(Math.random()*9999999999999999);
    }
})();
```

Предыдущий скрипт обнаруживает, была ли уже загружена закладка, проверяя, определена ли переменная `myBookmarklet`. Поступая таким образом, мы не загружаем его повторно, если пользователь нажимает на блюмрклет повторно. Если `myBookmarklet` не определен, мы загружаем другой файл JavaScript, добавляя в документ элемент `<script>`. Скрипт загружает сценарий `bookmarklet.js`, используя случайное число в качестве параметра, чтобы предотвратить загрузку файла из кеша браузера.

Фактический код блюмрклета будет находиться в статическом

файле `bookmarklet.js`. Это позволит нам обновить наш код бокмаклета, не требуя от наших пользователей обновлять закладки, которые они ранее добавили в свой браузер. Давайте добавим панель запуска бокмаклета на страницы панели мониторинга, чтобы наши пользователи могли скопировать ее в закладки.

Откройте шаблон `account/dashboard.html` приложения `account` и сделайте его прохожим на следующее:

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

{% block title %}Dashboard{% endblock %}

{% block content %}
<h1>Dashboard</h1>

{% with total_images_created=request.user.images_created.count %}
<p>Welcome to your dashboard. You have bookmarked {{ total_images_created }} image{{ total_images_created|pluralize }}.</p>
{% endwith %}

<p>Drag the following button to your bookmarks toolbar to bookmark images from other websites → <a href="javascript:{% include "bookmarklet_launcher.js" %}" class="button">Bookmark it</a><p>

<p>You can also <a href="{% url "edit" %}">edit your profile</a> or <a href="{% url "password_change" %}">change your password</a>.<p>
{% endblock %}
```

На панели теперь отображается общее количество изображений, помеченных пользователем. Мы используем шаблонный тег `{% with %}`, чтобы установить переменную с общим количеством изображений, помеченных клиентом текущего пользователя. Мы также включаем ссылку с атрибутом `href`, который содержит сценарий запуска бокмаклета. Мы включим этот код JavaScript из `bookmarklet_launcher.js` шаблона.

Откройте `http://127.0.0.1:8000/account/` в вашем браузере. Вы должны увидеть следующую страницу:

Dashboard

Welcome to your dashboard. You have bookmarked 1 image.

Drag the following button to your bookmarks toolbar to bookmark images from other websites →

BOOKMARK IT

You can also [edit your profile](#) or [change your password](#).

Теперь создайте следующие каталоги и файлы внутри каталога приложения `images`:

```
static/
  js/
    bookmarklet.js
```

Вы найдете каталог `static/css/` в каталоге приложения `images` в коде, который поставляется вместе с этой главой. Скопируйте каталог `css/` в `static/` вашего приложения. Файл `css/bookmarklet.css` предоставляет стили для нашего бокмаклера JavaScript.

Отредактируйте статический файл `bookmarklet.js` и добавьте к нему следующий код JavaScript:

```
(function(){
  var jquery_version = '3.3.1';
  var site_url = 'http://127.0.0.1:8000/';
  var static_url = site_url + 'static/';
  var min_width = 100;
  var min_height = 100;

  function bookmarklet(msg) {
```

```

    // Here goes our bookmarklet code
};

// Check if jQuery is loaded
if(typeof window.jQuery != 'undefined') {
    bookmarklet();
} else {
    // Check for conflicts
    var conflict = typeof window.$ != 'undefined';
    // Create the script and point to Google API
    var script = document.createElement('script');
    script.src = '//ajax.googleapis.com/ajax/libs/jquery/' +
        jquery_version + '/jquery.min.js';
    // Add the script to the 'head' for processing
    document.head.appendChild(script);
    // Create a way to wait until script loading
    var attempts = 15;
    (function(){
        // Check again if jQuery is undefined
        if(typeof window.jQuery == 'undefined') {
            if(--attempts > 0) {
                // Calls himself in a few milliseconds
                window.setTimeout(arguments.callee, 250)
            } else {
                // Too much attempts to load, send error
                alert('An error occurred while loading jQuery')
            }
        } else {
            bookmarklet();
        }
    })();
}
})()

```

Это основной скрипт загрузчика jQuery. Он заботится об использовании jQuery, если он уже загружен на текущий веб-сайт. Если jQuery не загружен, скрипт загружает jQuery из сети доставки контента Google, в которой размещаются популярные JavaScript-фреймворки. Когда jQuery загружается, он выполняет функцию `bookmarklet()`, которая будет содержать наш код бокмарклета. Мы также устанавливаем некоторые переменные в верхней части файла:

- `jquery_version`: Версия jQuery для загрузки

- `site_url` И `static_url`: Основной URL для нашего сайте и URL базовых статических файлов
- `min_width` И `min_height`: Минимальная ширина и высота в пикселях для изображений, которые наш бокмаклет попытается найти на сайте

Теперь давайте реализуем функцию `bookmarklet`. Отредактируйте функцию `bookmarklet()`, чтобы она выглядела следующим образом:

```
function bookmarklet(msg) {  
    // load CSS  
    var css = jQuery('<link>');  
    css.attr({  
        rel: 'stylesheet',  
        type: 'text/css',  
        href: static_url + 'css/bookmarklet.css?r=' +  
Math.floor(Math.random()*9999999999999999)  
    });  
    jQuery('head').append(css);  
  
    // load HTML  
    box_html = '<div id="bookmarklet"><a href="#" id="close">&times;</a>  
<h1>Select an image to bookmark:</h1><div class="images"></div></div>';  
    jQuery('body').append(box_html);  
  
    // close event  
    jQuery('#bookmarklet #close').click(function(){  
        jQuery('#bookmarklet').remove();  
    });  
};
```

Предыдущий код работает следующим образом:

1. Мы загружаем таблицу стилей `bookmarklet.css`, используя случайное число в качестве параметра, чтобы предотвратить возврат из браузера кэшированного файла.

2. Мы добавляем пользовательский HTML к элементу `<body>` текущего веб-сайта. Он состоит из элемента `<div>`, который будет содержать изображения, найденные на текущем веб-сайте.
3. Мы добавляем событие, которое удаляет наш HTML-код из документа, когда пользователь нажимает ссылку закрытия нашего блока HTML. Мы используем селектор `#bookmarklet #close`, чтобы найти элемент HTML с идентификатором с именем `close`, который имеет родительский элемент с идентификатором с именем `bookmarklet`. Селекторы jQuery позволяют находить HTML-элементы. Селектор jQuery возвращает все элементы, найденные данным селектором CSS. Вы можете найти список селекторов jQuery в <https://api.jquery.com/category/selectors/>.

После загрузки стилей CSS и кода HTML для блюмаклета нам нужно будет найти изображения на веб-сайте. Добавьте следующий код JavaScript в нижней части функции `bookmarklet()`:

```
// find images and display them
jQuery.each(jQuery('img[src$=".jpg"]'), function(index, image) {
  if (jQuery(image).width() >= min_width && jQuery(image).height()
    >= min_height)
  {
    image_url = jQuery(image).attr('src');
    jQuery('#bookmarklet .images').append('<a href="#"></a>');
  }
});
```

В предыдущем коде используется селектор `img[src$=".jpg"]` чтобы найти все `` HTML-элементы, чей атрибут `src` заканчивается строкой `.jpg`. Это означает, что мы будем искать все

изображения JPEG, отображаемые на текущем веб-сайте. Мы перебираем результаты с помощью метода `each()` для jQuery. Мы добавляем изображения с размером, большим, чем размер, указанный с помощью переменных `min_width` и `min_height` для нашего `<div class="images">` HTML контейнера.

Вам нужно будет загрузить бокмаклеть на любом сайте, включая сайты, обслуживаемые через HTTPS. SSL стал широко использоваться на большинстве веб-сайтов которые сегодня обслуживают контент через HTTPS. По соображениям безопасности ваш браузер не позволит вам запускать бокмаклеть через HTTP на сайте, обслуживаемом HTTPS.

Сервер разработки Django предназначен только для разработки и не поддерживает HTTPS. Чтобы проверить бокмаклеть через HTTPS, мы будем использовать Ngrok. Ngrok - это инструмент, который создает туннель, чтобы выставить свой локальный хост в Интернет через HTTP и HTTPS.

Загрузите Ngrok для своей операционной системы из <https://ngrok.com/download> и запустите его из консоли, используя следующую команду:

```
./ngrok http 8000
```

С помощью предыдущей команды вы указываете Ngrok создать туннель на localhost на порту 8000 и назначить ему имя для доступа к Интернету. Вы должны увидеть результат, похожий на этот:

Session	Status	online				
Version		2.2.8				
Region		United States (us)				
Web Interface		http://127.0.0.1:4040				
Forwarding		http://3f6ad53c.ngrok.io -> localhost:8000				
Forwarding		https://3f6ad53c.ngrok.io -> localhost:8000				
Connections	ttl	open	rt1	rt5	p50	p90

	0	0	0.00	0.00	0.00	0.00
--	---	---	------	------	------	------

Ngrok сообщает нам, что наш сайт, работающий локально на localhost 8000 порте с использованием сервера разработки Django, предоставляется в Интернет через <http://3f6ad53c.ngrok.io> И <https://3f6ad53c.ngrok.io> Используя протоколы HTTP и HTTPS, соответственно. Ngrok также предоставляет URL-адрес для доступа к веб-интерфейсу, который отображает информацию о запросах, отправленных на сервер на localhost порте 4040.

Отредактируйте файл `settings.py` вашего проекта и добавьте хост, предоставленный Ngrok, в настройки `ALLOWED_HOSTS`:

```
ALLOWED_HOSTS = [
    'mysite.com',
    'localhost',
    '127.0.0.1',
    '3f6ad53c.ngrok.io'
]
```

Это позволит вам обслуживать приложение через новое имя хоста. Затем откройте URL <https://3f6ad53c.ngrok.io/account/login/> В вашем браузере, заменив хост на тот, который предоставлен Ngrok. Вы сможете увидеть сайт входа в систему.

Отредактируйте шаблон `bookmarklet_launcher.js` и замените URL <http://127.0.0.1:8000/> URL-адресом HTTPS, предоставленным Ngrok, следующим образом:

```
(function(){
    if (window.myBookmarklet !== undefined){
        myBookmarklet();
    }
    else {

        document.body.appendChild(document.createElement('script')).src='https://3f6ad53c.ngrok.io/static/js/bookmarklet.js?
r='+Math.floor(Math.random()*9999999999999999);
    }
})
```

```
|});();
```

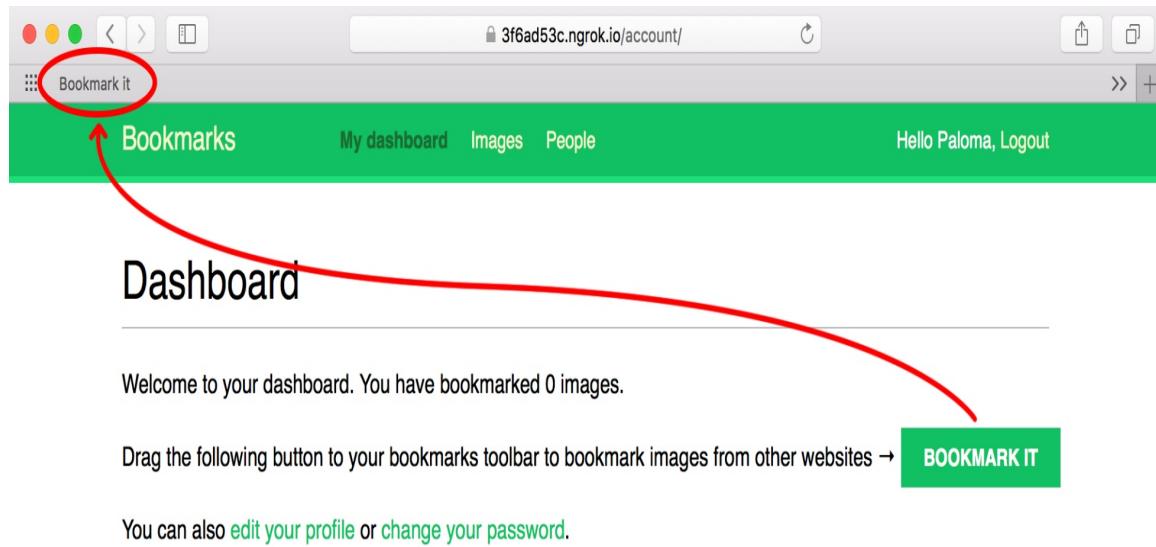
Откройте статический файл `js/bookmarklet.js` и найдите следующую строку:

```
| var site_url = 'http://127.0.0.1:8000/';
```

Замените предыдущую строку следующей, добавив URL HTTPS, предоставленный Ngrok:

```
| var site_url = 'https://3f6ad53c.ngrok.io/';
```

Откройте `https://3f6ad53c.ngrok.io/account/` в вашем браузере, заменив хост на тот, который предоставлен Ngrok. Войдите в систему с существующим пользователем, а затем перетягните BOOKMARK IT кнопку на панель инструментов закладок вашего браузера:



Откройте любой веб-сайт в своем браузере и нажмите на свой букмарклет. Вы увидите, что на веб-сайте появляется новое белое окно, отображающее все изображения в формате JPEG с размерами выше 100 x 100 пикселей. Оно должно выглядеть следующим образом:

NEW & INTERESTING FINDS ON AMAZON [EXPLORE](#)

amazon Try Prime

Departments ▾ Browsing History ▾ Antonio's Amazon.com Today's Deals Gift Cards Registry EN

1-16 of 11,256 results for "django reinhardt"

Show results for

CDs & Vinyl
Jazz
Pop
European Jazz
Swing Jazz
Gypsy Music
[See more](#)

Books
Jazz Music
[See more](#)
[See All 21 Departments](#)

Refine by

International Shipping (What's this?)
 Ship to Spain

Amazon Prime
 prime

Eligible for Free Shipping

Introducing Amazon Music Unlimited. Listen to any song, anywhere.
[Learn More about Amazon Music Unlimited](#)

Showing most relevant results. See all results for django reinhardt.

Solo Flight: The Music of Django Reinhardt
by J.P. McShane and Django Reinhardt

\$15.00
FREE Shipping on eligible orders
Only 2 left in stock - order soon.

[The Essential DJANGO REINHARDT](#)

Select an image to bookmark:

Контейнер HTML содержит изображения, которые можно занести в закладки. Мы хотим, чтобы пользователь нажал на нужное изображение и добавил его в закладки. Измените статический файл `js/bookmarklet.js` и добавьте следующий код в нижней части функции `bookmarklet()`:

```
// when an image is selected open URL with it
jQuery('#bookmarklet .images a').click(function(e){
    selected_image = jQuery(this).children('img').attr('src');
    // hide bookmarklet
    jQuery('#bookmarklet').hide();
    // open new window to submit the image
    window.open(site_url + 'images/create/?url='
        + encodeURIComponent(selected_image)
        + '&title='
        + encodeURIComponent(jQuery('title').text()),
        '_blank');
```

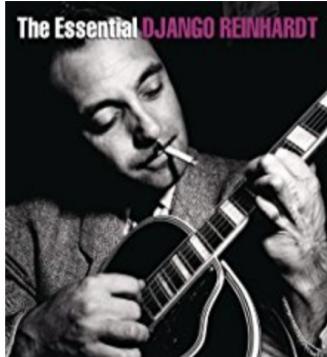
```
|});
```

Предыдущий код работает следующим образом:

1. Мы добавляем событие `click()` к элементам ссылок изображений.
2. Когда пользователь нажимает на изображение, мы устанавливаем новую переменную с именем `selected_image`, которая содержит URL выбранного изображения.
3. Мы закрываем блюмаклет и открываем новое окно браузера с URL-адресом для закладки нового изображения на нашем сайте. Мы передаем элемент `<title>` веб-сайта и URL выбранного изображения в качестве параметров `GET`.

Откройте новый URL-адрес в своем браузере и снова нажмите на свой блюмаклет, чтобы отобразить окно выбора изображения. Если вы нажмете на изображение, вы будете перенаправлены на страницу создания изображения, передав название веб-сайта и URL-адрес выбранного изображения в качестве параметров `GET`:

Bookmark an image



Title:

Django Reinhardt

Description:

BOOKMARK IT!

Поздравляем! Это ваш первый букмарклет для JavaScript, и он полностью интегрирован в ваш проект Django.

Создание детального представления для изображений

Теперь мы создадим простое детальное представление для отображения изображения, которое было сохранено на нашем сайте. Откройте файл `views.py` приложения `images` и добавьте к нему следующий 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 для отображения изображения. Отредактируйте файл `urls.py` приложения `images` добавив следующий паттерн URL:

```
path('detail/<int:id>/<slug:slug>/',
      views.image_detail, name='detail'),
```

Измените файл `models.py` приложения `images` и добавьте метод `get_absolute_url()` в модель `Image` следующим образом:

```
from django.urls import reverse

class Image(models.Model):
    # ...
```

```
def get_absolute_url(self):
    return reverse('images:detail', args=[self.id, self.slug])
```

Помните, что простой способ предоставить канонические URL-адреса для объектов заключается в определении метода `get_absolute_url()` в МОДЕЛИ.

Наконец, создайте шаблон внутри каталога шаблонов `/images/image/` приложения `images` и назовите его `detail.html`. Добавьте к нему следующий 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>
            {{ image.description|linebreaks }}
        </div>
        <div class="image-likes">
            {% for user in image.users_like.all %}
                <div>
                    
                    <p>{{ user.first_name }}</p>
                </div>
            {% empty %}
                Nobody likes this image yet.
            {% endfor %}
        </div>
    {% endwith %}
    {% endblock %}
```

Это шаблон для отображения деталей закладки. Мы используем тег `{% with %}`, чтобы сохранить результат `QuerySet`, подсчитывая всех пользователей в новой переменной, называемой `total_likes`. Поступая таким образом, мы избегаем

вызыва одного и того же QuerySet дважды. Мы также включаем описание изображения и итерацию по `image.users_like.all`, чтобы отобразить всех пользователей, которым нравится эта картинка.

Использовать шаблонный тэг `{% with %}` полезно для предотвращения множества раз вызова QuerySets в Django.

Теперь добавьте новое изображение с помощью букмаклета. После публикации изображения вы будете перенаправлены на страницу с подробными сведениями о изображении. Страница будет содержать сообщение об успешном завершении:

The screenshot shows a web application interface with a green header bar. From left to right, the header contains: 'Bookmarks' (highlighted in white), 'My dashboard', 'Images', 'People', and 'Hello Paloma, Logout'. Below the header is a green success message box containing the text 'Image added successfully' with a close button ('X') on the right. The main content area has a title 'Django Reinhardt' followed by a black and white photo of Django Reinhardt playing a guitar. To the right of the photo is a circular button with '0 likes'. Below the photo is the caption 'The Essential Django Reinhardt.' and the text 'Nobody likes this image yet.'

Создание эскизов изображений с использованием sorl-thumbnail

Мы показываем исходное изображение на странице отображения детальной информации, но размеры для разных изображений могут сильно различаться. Кроме того, исходные файлы для некоторых изображений могут быть огромными, и загрузка их может занять слишком много времени. Лучшим способом отображения оптимизированных изображений в едином виде является создание эскизов. Для этой цели мы будем использовать приложение Django с именем `sorl-thumbnail`.

Откройте терминал и установите `sorl-thumbnail`, используя следующую команду:

```
pip install sorl-thumbnail==12.4.1
```

Откройте файл `settings.py` проекта `bookmarks` и добавьте `sorl.thumbnail` к настройкам `INSTALLED_APPS`:

```
INSTALLED_APPS = [  
    # ...  
    'sorl.thumbnail',  
]
```

Затем запустите следующую команду для синхронизации приложения с вашей базой данных:

```
python manage.py migrate
```

Вы должны увидеть вывод, который включает следующую строку:

```
Applying thumbnail.0001_initial... OK
```

Приложение `sorl-thumbnail` предлагает вам различные способы создания эскизов изображений. Приложение предоставляет тег шаблона `{% thumbnail %}` для создания эскизов в шаблонах и пользовательского `ImageField`, если вы хотите определить эскизы в своих моделях. Мы будем использовать подход тега шаблона. Отредактируйте шаблон `images/image/detail.html` и замените строку:

```

```

Следующие строки должны заменить предыдущую:

```
{% load thumbnail %}
{% thumbnail image.image "300" as im %}
<a href="{{ image.image.url }}>
    
</a>
{% endthumbnail %}
```

Здесь мы определяем миниатюру с фиксированной шириной 300 пикселей. В первый раз, когда пользователь загрузит эту страницу, будет создано уменьшенное изображение.

Сгенерированный эскиз будет обслуживаться в следующих запросах. Запустите сервер разработки с помощью команды `python manage.py runserver` и обратитесь к странице подробных сведений о изображении для существующего изображения. Эскиз будет создан и отображен на сайте.

Приложение `sorl-thumbnail` предлагает несколько вариантов настройки миниатюр, включая алгоритмы обрезки и

различные эффекты, которые могут быть применены. Если у вас возникли трудности с созданием эскизов, вы можете добавить `THUMBNAIL_DEBUG = True` в вашем файле `settings.py` для получения отладочной информации. Вы можете ознакомиться с полной документацией по `sorl-thumbnail` на <https://sorl-thumbnail.readthedocs.io/>.

Добавление AJAX с помощью jQuery

Теперь мы добавим AJAX в наше приложение. AJAX происходит от **Asynchronous JavaScript and XML**. Этот термин включает в себя группу методов для создания асинхронных HTTP-запросов. Он состоит из отправки и получения данных с сервера асинхронно, без перезагрузки всей страницы. Несмотря на название, XML не требуется. Вы можете отправлять или извлекать данные в других форматах, таких как JSON, HTML или обычный текст.

Мы добавим ссылку на страницу с детальным отображением изображения, чтобы пользователи могли щелкнуть по ней, если изображение понравилось. Мы выполним это действие с помощью вызова AJAX, чтобы избежать перезагрузки всей страницы. Во-первых, мы создадим представление для пользователей чтобы они могли отметить изображение (лайк/дизлайк). Откройте файл `views.py` приложения `images` и добавте следующий код в него:

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST

@login_required
@require_POST
def image_like(request):
    image_id = request.POST.get('id')
    action = request.POST.get('action')
    if image_id and action:
        try:
            image = Image.objects.get(id=image_id)
            if action == 'like':
                image.users_like.add(request.user)
            else:
```

```
        image.users_like.remove(request.user)
    return JsonResponse({'status':'ok'})
except:
    pass
return JsonResponse({'status':'ko'})
```

Мы будем использовать два декоратора для нашего представления. Декоратор `login_required` не позволяет пользователям, которые не вошли в систему, получить доступ к этому представлению. Декоратор `require_POST` возвращает объект `HttpResponseNotAllowed` (код состояния `405`) если HTTP-запрос не выполняется через `POST`. Таким образом, мы разрешаем `POST` запросы для этого представления. Django также предоставляет декоратор `require_GET` чтобы разрешить `GET` запросы и декоратор `require_http_methods` которому вы можете передать список разрешенных методов в качестве аргумента.

В этом представлении мы используем два параметра:

- `image_id`: Идентификатор объекта изображения, над которым пользователь выполняет действие
- `action`: Действие, которое пользователь хочет выполнить, которое мы считаем строкой со значением `like` ИЛИ `unlike`

Мы используем диспетчер, предоставленный Django для поля `users_like` для отношения многих-ко-многим модели `Image`, чтобы добавлять или удалять объекты из отношения с помощью `add()` или `remove()`. Вызов `add()`, при передаче объекта, который уже присутствует в соответствующем наборе объектов, не дублирует его и соответственно, вызыв `remove()`, к объекту который не находится в отношении со связанным объектом, ничего не делает. Другим полезным методом для менеджера «многие ко многим» является `clear()`, который удаляет все объекты из соответствующего набора объектов.

Наконец, мы используем класс `JsonResponse`, предоставленный

Django, который возвращает ответ HTTP с типом контента `application/json`, преобразовывая данный объект в вывод JSON.

Отредактируйте файл `urls.py` приложения `images` и добавьте к нему следующий паттерн URL:

```
| path('like/', views.image_like, name='like'),
```

Загрузка jQuery

Нам нужно будет добавить функциональность AJAX к нашему шаблону деталей изображения. Чтобы использовать jQuery в наших шаблонах, мы сначала включим его в шаблон `base.html` нашего проекта. Откройте шаблон `base.html` приложения `account` и добавьте следующий код перед закрытием HTML тэга `</body>`:

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js">
</script>
<script>
$(document).ready(function(){
    {% block domready %}
    {% endblock %}
});
</script>
```

Мы загружаем структуру jQuery из CDN от Google. Вы также можете скачать jQuery из <https://jquery.com/> и добавить его папку `static` вашего приложения.

Мы добавили тэг `<script>` включающий JavaScript код. `$(document).ready()` является функцией jQuery, которая принимает обработчик, который выполняется, когда иерархия DOM полностью построена. **DOM** расшифровывается как **Document Object Model**. DOM создается браузером при загрузке веб-страницы и строится как дерево объектов. Включив наш код внутри этой функции, мы будем следить за тем, чтобы все элементы HTML, с которыми мы собираемся взаимодействовать, загрузились в DOM. Наш код будет выполнен только после того, как DOM будет готов.

Внутри обработанной документами функции обработчика мы

включаем блок шаблона Django с именем `domready`, в котором шаблоны, расширяющие базовый шаблон, смогут включать в себя конкретный JavaScript.

Не путайте JavaScript-код с тегами шаблона Django. Язык шаблона Django отображается на стороне сервера, выводя окончательный документ HTML, а JavaScript выполняется на стороне клиента. В некоторых случаях полезно генерировать JavaScript-код динамически с помощью Django.

В примерах этой главы мы включаем код JavaScript в шаблонах Django. Предпочтительным способом включения кода JavaScript является загрузка `.js` файлов, как статических файлов, особенно когда они являются большими сценариями.

Подделка межсайтовых запросов в AJAX

Вы узнали о *Cross-Site Request Forgery* в [главе 2](#), *Улучшение вашего блога с помощью расширения функционала*. При активной защите CSRF Django проверяет токен CSRF во всех запросах `POST`. Когда вы отправляете формы, вы можете использовать тег шаблона `{% csrf_token %}`, чтобы отправить токен вместе с формой. Тем не менее, для AJAX-запросов недопустимо передавать токен CSRF в виде данных с каждым `POST` запросом. Поэтому Django позволяет вам настроить пользовательский заголовок `x-CSRFToken` в ваших запросах AJAX со значением токена CSRF. Это позволяет вам настроить jQuery или любую другую библиотеку JavaScript, чтобы автоматически устанавливать заголовок `x-CSRFToken` в каждом запросе.

Чтобы включить токен во все запросы, вам необходимо выполнить следующие шаги:

1. Получить токен CSRF из файла cookie `csrftoken`, который установлен, если активна защита CSRF
2. Отправить токен в запросе AJAX, используя заголовок `x-CSRFToken`

Вы можете найти дополнительную информацию о защите CSRF и AJAX на <https://docs.djangoproject.com/en/2.0/ref/csrf/#ajax>.

Отредактируйте последний код, который вы включили в свой шаблон `base.html`, следующим образом:

```
<script  
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js">  
</script>  
<script src="https://cdn.jsdelivr.net/npm/js-cookie@2/src/js.cookie.min.js">  
</script>  
<script>  
    var csrftoken = Cookies.get('csrftoken');  
    function csrfSafeMethod(method) {  
        // these HTTP methods do not require CSRF protection  
        return /^(GET|HEAD|OPTIONS|TRACE)$/.test(method);  
    }  
    $.ajaxSetup({  
        beforeSend: function(xhr, settings) {  
            if (!csrfSafeMethod(settings.type) && !this.crossDomain) {  
                xhr.setRequestHeader("X-CSRFToken", csrftoken);  
            }  
        }  
    });  
    $(document).ready(function(){  
        {% block domready %}  
        {% endblock %}  
    });  
</script>
```

Предыдущий код делает следующее:

1. Мы загружаем плагин JS Cookie из общедоступного CDN, чтобы мы могли легко взаимодействовать с файлами cookie. JS Cookie - это легкий JavaScript для обработки файлов cookie. Вы можете узнать больше об [ЭТОМ НА https://github.com/js-cookie/js-cookie.](https://github.com/js-cookie/js-cookie)
2. Мы читаем значение cookie `csrftoken` ИЗ `Cookies.get()`.
3. Мы создаем функцию `csrfSafeMethod()` чтобы проверить, является ли метод HTTP безопасным. Безопасные методы не требующие защиты CSRF - ЭТО `GET`, `HEAD`, `OPTIONS`, И `TRACE`.
4. Мы создали запросы jQuery AJAX, используя `$.ajaxSetup()`.

Перед выполнением каждого запроса AJAX мы проверяем, является ли метод запроса безопасным, а также что текущий запрос не является междоменным. Если запрос небезопасен, мы устанавливаем заголовок `X-CSRFToken` со значением, полученным из файла cookie. Эта настройка будет применяться ко всем запросам AJAX, выполняемым с помощью jQuery.

Маркер CSRF будет включен во все запросы AJAX, которые используют небезопасные HTTP-методы, такие как `POST` или `PUT`.

Выполнение запросов AJAX с помощью jQuery

Откройте шаблон `images/image/detail.html` приложения `images` и найдите следующую строку:

```
{% with total_likes=image.users_like.count %}
```

Замените ее следующим:

```
{% with total_likes=image.users_like.count users_like=image.users_like.all %}
```

Затем измените элемент `<div>` класса `image-info` следующим образом:

```
<div class="image-info">
  <div>
    <span class="count">
      <span class="total">{{ total_likes }}</span>
      like{{ total_likes|pluralize }}
    </span>
    <a href="#" data-id="{{ image.id }}" data-action="{% if
      request.user in users_like %}un{% endif %}like"
      class="like button">
      {% if request.user not in users_like %}
        Like
      {% else %}
        Unlike
      {% endif %}
    </a>
  </div>
  {{ image.description|linebreaks }}
</div>
```

Во-первых, мы добавили еще одну переменную в тег шаблона

{% with %}, чтобы сохранить результат запроса `image.users_like.all` и не выполнять его дважды. Мы показываем общее количество пользователей, которым нравится это изображение, и включаем ссылку на `like/unlike` для изображения: мы проверяем, находится ли пользователь в соответствующем наборе объекта `users_like` для отображения `like` или `unlike`, исходя из текущих отношений между пользователем и этим изображением. Мы добавляем следующие атрибуты в элемент HTML `<a>`:

- `data-id`: ID отображаемого изображения
- `data-action`: Действие которое выполняется, когда пользователь нажимает на ссылку. Это может быть `like` ИЛИ `unlike`

Мы отправим значение обоих атрибутов в запросе AJAX в представление `image_like`. Когда пользователь нажимает на `like/unlike` ссылку, нам нужно будет выполнить следующие действия на стороне клиента:

1. Вызвать представление AJAX, передав ему идентификатор изображения и параметры действия.
2. Если запрос AJAX успешный, обновить атрибут `data-action` HTML элемента `<a>` с помощью противоположного действия (`like / unlike`) и соответствующим образом изменить отображаемый текст.
3. Обновить общее количество отображаемых `likes`.

Добавте блок `domready` в нижней части `images/image/detail.html` шаблона со следующим кодом JavaScript:

```
|  {% block domready %}
```

```

$('a.like').click(function(e){
    e.preventDefault();
    $.post('{% url "images:like" %}',
    {
        id: $(this).data('id'),
        action: $(this).data('action')
    },
    function(data){
        if (data['status'] == 'ok')
        {
            var previous_action = $('a.like').data('action');

            // toggle data-action
            $('a.like').data('action', previous_action == 'like' ?
            'unlike' : 'like');
            // toggle link text
            $('a.like').text(previous_action == 'like' ? 'Unlike' :
            'Like');

            // update total likes
            var previous_likes = parseInt($('span.count .total').text());
            $('span.count .total').text(previous_action == 'like' ?
            previous_likes + 1 : previous_likes - 1);
        }
    }
);
});
{%
endblock %}

```

Предыдущий код работает следующим образом:

1. Мы используем `$('a.like')` jQuery селектор чтобы найти все `<a>` элементы в HTML документе с классом `like`.
2. Мы определяем функцию обработчика события `click`. Эта функция будет выполняться каждый раз, когда пользователь нажимает на `like/unlike` ссылку.
3. Внутри функции обработчика мы используем `e.preventDefault()` чтобы избежать поведения по умолчанию элемента `<a>`. Это будет препятствовать тому, чтобы ссылка перенаправила нас куда либо.
4. Мы используем `$.post()` для выполнения асинхронного

запроса `POST` на сервер. jQuery также предоставляет `$.get()` метод для `GET` запросов и низкоуровневых `$.ajax()` методов.

5. Мы используем шаблонный тег Django `{% url %}` чтобы создать URL для AJAX запроса.
6. Мы создаем `POST` параметры для отправки запроса. Это `id` и `action` параметры, ожидаемые нашим представлением Django. Мы извлекаем эти значения из элемента `<a>` атрибута `data-id` и `data-action`.
7. Мы определяем функцию обратного вызова, которая выполняется при получении ответа HTTP; он принимает атрибут `data`, который содержит ответ.
8. Мы получаем доступ к атрибуту `status` полученных данных и проверяем, равен ли он `ok`. Если возвращаемые данные соответствуют ожиданиям, мы переключим атрибут `data-action` ссылки и его текст. Это позволяет пользователю отменить действие.
9. Мы увеличиваем или уменьшаем общее количество просмотров на единицу, в зависимости от выполненного действия.

Откройте страницу с детальным отображением изображения в вашем браузере для изображения, которое вы загрузили. Вы должны видеть следующие начальные числа, и кнопку LIKE :



Нажмите на кнопку LIKE. Вы заметите, что общее количество просмотров увеличивается на единицу, а текст кнопки

изменяется на UNLIKE, следующим образом:



Когда вы нажмете на кнопку UNLIKE, текст кнопки изменяется на LIKE, и соответственно меняется общее значение счетчика.

При программировании JavaScript, особенно при выполнении запросов AJAX, рекомендуется использовать инструмент для отладки JavaScript и HTTP-запросов. Большинство современных браузеров включают инструменты разработчика для отладки JavaScript. Обычно вы можете щелкнуть правой кнопкой мыши в любом месте веб-сайта и нажать Inspect element для доступа к инструментам веб-разработчика.

Создание пользовательских декораторов для ваших представлений

Мы ограничим наши представления AJAX только запросами созданными через AJAX. Объект запроса Django предоставляет метод `is_ajax()`, который проверяет, выполняется ли запрос с помощью `XMLHttpRequest`, что будет означать, что это запрос AJAX. Это значение указано в заголовке `HTTP_X_REQUESTED_WITH` который включен в запросы AJAX большинством библиотек JavaScript.

Мы создадим декоратор для проверки заголовка `HTTP_X_REQUESTED_WITH` в наших представлениях. Декоратор - это функция, которая принимает другую функцию и расширяет поведение последний без явной модификации. Если концепция декораторов вам не чужда, вы можете взглянуть на <https://www.python.org/dev/peps/pep-0318/> прежде чем продолжить чтение.

Поскольку наш декоратор будет общим и может быть применен к любому представлению, мы создадим в нашем проекте пакет Python `common`. Создайте следующий каталог и файлы в каталоге проекта `bookmarks`:

```
common/
    __init__.py
    decorators.py
```

Отредактируйте файл `decorators.py` и добавьте к нему следующий код:

```
from django.http import HttpResponseRedirect

def ajax_required(f):
    def wrap(request, *args, **kwargs):
        if not request.is_ajax():
            return HttpResponseRedirect()
        return f(request, *args, **kwargs)
    wrap.__doc__=f.__doc__
    wrap.__name__=f.__name__
    return wrap
```

Предыдущий код - это наш пользовательский `ajax_required` декоратор. Он определяет функцию, которая возвращает объект `HttpResponseBadRequest` (код HTTP 400), если запрос не является AJAX. В противном случае она возвращает декорированную функцию.

Теперь вы можете отредактировать файл `views.py` приложения `images` и добавить этот декоратор к представлению AJAX `image_like`, как показано ниже:

```
from common.decorators import ajax_required

@ajax_required
@login_required
@require_POST
def image_like(request):
    # ...
```

Если вы попробуете открыть `http://127.0.0.1:8000/images/like/` напрямую в вашем браузере вы получите ответ HTTP 400.

Создавайте пользовательские декораторы для своих представлений, если обнаружите, что повторяете одни и те же проверки в нескольких представлениях.

Добавление AJAX пагинации для вашего представления списка

Нам нужно будет перечислить все закладки с изображениями на нашем веб-сайте. Мы будем использовать разбиение на страницы AJAX для создания функциональности бесконечной прокрутки. Бесконечная прокрутка достигается путем автоматической загрузки следующих результатов, когда пользователь достигает нижней части страницы.

Мы реализуем представление списка изображений, которое будет обрабатывать как стандартные запросы браузера, так и запросы AJAX, включая разбиение на страницы. Когда пользователь изначально загружает страницу списка изображений, мы отобразим первую страницу изображений. Когда они прокручиваются до нижней части страницы, мы загружаем следующую страницу элементов через AJAX и добавляем ее к нижней части главной страницы.

То же представление будет обрабатывать как стандартные, так и AJAX-страницы. Откройте файл `views.py` приложения `images` и добавьте к нему следующий код:

```
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')

    try:
        images = paginator.page(page)
    except PageNotAnInteger:
        images = paginator.page(1)
    except EmptyPage:
        images = paginator.page(paginator.num_pages)

    return render(request, 'image_list.html', {'images': images})
```

```
try:
    images = paginator.page(page)
except PageNotAnInteger:
    # If page is not an integer deliver the first page
    images = paginator.page(1)
except EmptyPage:
    if request.is_ajax():
        # If the request is AJAX and the page is out of range
        # return an empty page
        return HttpResponse('')
    # If page is out of range deliver last page of results
    images = paginator.page(paginator.num_pages)
if request.is_ajax():
    return render(request,
                  'images/image/list_ajax.html',
                  {'section': 'images', 'images': images})
return render(request,
              'images/image/list.html',
              {'section': 'images', 'images': images})
```

В этом представлении мы создаем `QuerySet` для возврата всех изображений из базы данных. Затем мы создаем объект `Paginator` для разбивки на страницы, с извлечением восьми изображений на страницу. Мы получаем исключение `EmptyPage`, если запрашиваемая страница находится за пределами допустимого диапазона. Если это так, и запрос выполняется через AJAX, мы возвращаем пустой `HttpResponse`, который поможет нам остановить разбиение на страницы AJAX на стороне клиента. Мы отрисовываем результаты в двух разных шаблонах:

- Для запросов AJAX мы создаем шаблон `list_ajax.html`. Этот шаблон будет содержать только изображения запрошенной страницы.
- Для стандартных запросов мы создаем шаблон `list.html`. Этот шаблон расширяет шаблон `base.html`, чтобы отобразить всю страницу и будет включать шаблон `list_ajax.html`, чтобы включить список изображений.

Откройте файл `urls.py` приложения `images` и добавте в него следующий URL паттерн:

```
| path('', views.image_list, name='list'),|
```

Наконец, нам нужно будет создать шаблоны, упомянутые выше. Внутри каталога шаблонов `images/image/` создайте новый шаблон и назовите его `list_ajax.html`. Добавьте к нему следующий код:

```
{% load thumbnail %}

{% for image in images %}
    <div class="image">
        <a href="{{ image.get_absolute_url }}">
            {% thumbnail image.image "300x300" crop="100%" as im %}
                <a href="{{ image.get_absolute_url }}">
                    
                </a>
            {% endthumbnail %}
        </a>
        <div class="info">
            <a href="{{ image.get_absolute_url }}" class="title">
                {{ image.title }}
            </a>
        </div>
    </div>
{% endfor %}
```

Предыдущий шаблон отображает список изображений. Мы будем использовать его для возврата результатов для запросов AJAX. Создайте еще один шаблон в том же каталоге и назовите его `list.html`. Добавьте к нему следующий код:

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

{% block title %}Images bookmarked{% endblock %}

{% block content %}
    <h1>Images bookmarked</h1>
    <div id="image-list">
        {% include "images/image/list_ajax.html" %}
    </div>
{% endblock %}
```

```
</div>
{% endblock %}
```

Шаблон списка расширяет шаблон `base.html`. Чтобы избежать повторения кода, мы включили шаблон `list_ajax.html` для отображения изображений. Шаблон `list.html` будет содержать код JavaScript для загрузки дополнительных страниц при прокрутке в нижней части страницы.

Добавьте следующий код в шаблон `list.html`:

```
{% block domready %}
    var page = 1;
    var empty_page = false;
    var block_request = false;

    $(window).scroll(function() {
        var margin = $(document).height() - $(window).height() - 200;
        if ($window.scrollTop() > margin && empty_page == false &&
            block_request == false) {
            block_request = true;
            page += 1;
            $.get('?page=' + page, function(data) {
                if(data == '') {
                    empty_page = true;
                }
                else {
                    block_request = false;
                    $('#image-list').append(data);
                }
            });
        }
    });
{% endblock %}
```

Предыдущий код обеспечивает функциональность бесконечной прокрутки. Мы включаем код JavaScript в блоке `domready`, который мы определили в шаблоне `base.html`. Код выглядит следующим образом:

1. Определяем следующие переменные:

1. `page`: Сохраняет номер текущей страницы.
 2. `empty_page`: Позволяет нам узнать, находится ли пользователь на последней странице и извлекает пустую страницу. Как только мы получим пустую страницу, мы прекратим отправку дополнительных запросов AJAX, потому что мы будем предполагать, что результатов больше нет.
 3. `block_request`: Предотвращает отправку дополнительных запросов во время выполнения запроса AJAX.
-
2. Мы используем `$(window).scroll()` для захвата события прокрутки, а также для определения функции обработчика для него.
 3. Мы вычисляем переменную `margin`, чтобы получить разницу между общей высотой документа и высотой окна, потому что это высота оставшегося содержимого для прокрутки пользователя. Мы вычитаем значение 200 из результата, чтобы загрузить следующую страницу, когда пользователь ближе 200 пикселей к нижней части страницы.
 4. Мы отправляем только запрос AJAX, если никакой другой запрос AJAX не выполняется (`block_request` должен быть `false`), и пользователь не попал на последнюю страницу результатов (`empty_page` также `false`).
 5. Мы устанавливаем `block_request` равным `true`, чтобы избежать ситуации, когда событие прокрутки запускает дополнительные запросы AJAX и увеличивает счетчик

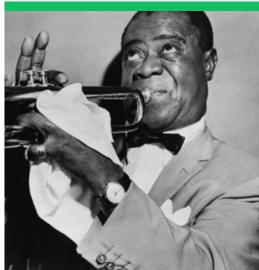
`page` на единицу чтобы получить следующую страницу.

6. Мы выполняем запрос AJAX `get`, используя `$.get()` и получаем ответ HTML в переменной `data`. Ниже перечислены два сценария:

1. **Ответ не имеет содержания:** Мы добрались до конца, и больше загружать страницы не нужно. Мы устанавливаем `empty_page` в `true` для предотвращения дополнительных запросов AJAX.
2. **Ответ содержит данные:** Мы добавляем данные в элемент HTML с ID `image-list`. Содержимое страницы расширяется за счет вертикального добавления результатов, когда пользователь приближается к нижней части страницы.

Откройте `http://127.0.0.1:8000/images/` в вашем браузере. Вы увидите список изображений, которые вы добавили на данный момент. Он должен выглядеть примерно так:

Images bookmarked



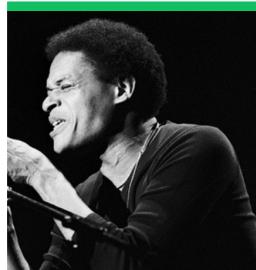
Louis Armstrong



Chick Corea



Al Jarreau



Al Jarreau



Ella Fitzgerald



Glenn Miller



Charlie Parker



Nina Simone

Прокрутите страницу вниз, чтобы загрузить дополнительные страницы. Убедитесь, что вы отметили более восьми изображений, используя букмаклэт, потому что это количество изображений, которые мы показываем на

странице. Помните, что вы можете использовать Firebug или аналогичный инструмент для отслеживания запросов AJAX и отладки вашего кода JavaScript.

Наконец откройте шаблон `base.html` приложения `account` и добавьте URL-адрес для элемента изображения в главном меню, а именно:

```
<li {% if section == "images" %}class="selected"{% endif %}>
  <a href="{% url "images:list" %}">Images</a>
</li>
```

Теперь вы можете получить доступ к списку изображений из главного меню.

Резюме

В этой главе вы создали бокмаклеть JavaScript для обмена изображениями с других сайтов на вашем сайте. Вы реализовали представления AJAX с помощью jQuery и добавили AJAX разбивку на страницы.

В следующей главе мы научим вас, как построить отслеживающую систему и поток активности. Вы будете работать с родовыми отношениями, сигналами и денормализацией. Вы также узнаете, как использовать Redis с Django.

Глава 6

Отслеживание действий пользователя

В предыдущей главе вы реализовали представление AJAX в своем проекте с помощью jQuery и создали букаркет JavaScript для совместного использования контента с других сайтов на вашей платформе.

В этой главе вы узнаете, как построить систему отслеживания действий пользователя и создать поток его активности. Вы узнаете, как работают сигналы Django и интегрируете в свой проект быстрое хранилище ввода-вывода Redis для хранения элементов представлений.

В этой главе будут рассмотрены следующие вопросы:

- Создание отношений «многие ко многим» к промежуточной модели
- Создание системы отслеживания
- Создание приложения потока активности
- Добавление общих отношений к моделям
- Оптимизация QuerySet для связанных объектов
- Использование сигналов для денормализации

счетчиков

- Сохранение сообщений представления в Redis

Построение системы отслеживания

В нашем проекте мы построим отслеживающую систему. Наши пользователи смогут отслеживать, что другие пользователи делают на платформе. Отношения между пользователями - это отношения «многие ко многим». Пользователь может следить за несколькими пользователями, и в свою очередь, за ними могут следить несколько пользователей.

Создание отношений «многие ко многим» с применением промежуточной модели

В предыдущих главах вы создали отношения «многие ко многим», добавив `ManyToManyField` к одной из связанных моделей и позволили Django создать таблицу базы данных для этой связи. Это подходит для большинства случаев, но иногда вам может понадобиться создать промежуточную модель для отношения. Создание промежуточной модели необходимо, если вы хотите сохранить дополнительную информацию об отношении, например, дату создания отношения или поле, описывающее характер отношения.

Мы создадим промежуточную модель для построения отношений между пользователями. Есть две причины, по которым мы будим использовать промежуточную модель:

- Мы используем модель `User`, предоставленную Django, и мы хотим избежать ее изменения
- Мы хотим сохранить время создания отношения

Откройте файл `models.py` вашего приложения `account` и добавте в него следующий 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,
                             db_index=True)

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

def __str__(self):
    return '{} follows {}'.format(self.user_from,
                                  self.user_to)
```

В предыдущем коде показана модель `Contact`, которую мы будем использовать для пользовательских отношений. Она содержит следующие поля:

- `user_from`: `ForeignKey` для пользователя, который создает связь
- `user_to`: `ForeignKey` для пользователя являющегося отслеживаемым
- `created`: `DateTimeField` поле с `auto_now_add=True`, чтобы сохранить время, когда была создана связь

Индекс базы данных автоматически создается в полях `ForeignKey`. Мы используем `db_index=True` для создания индекса базы данных для поля `created`. Это улучшит производительность запроса при обращении `QuerySet` к этому полю.

Используя ORM, мы могли бы создать связь для пользователя `user1` - следящего за другим пользователем, `user2`, например:

```
user1 = User.objects.get(id=1)
user2 = User.objects.get(id=2)
Contact.objects.create(user_from=user1, user_to=user2)
```

Связанные менеджеры `rel_from_set` и `rel_to_set` вернут `QuerySet` для

модели `Contact`. Чтобы получить доступ к конечной стороне отношения из модели `User`, было бы желательно, чтобы `User` содержал `ManyToManyField`, как показано ниже:

```
following = models.ManyToManyField('self',
                                   through=Contact,
                                   related_name='followers',
                                   symmetrical=False)
```

В предыдущем примере мы говорим, что Django использует нашу настраиваемую промежуточную модель для отношения, добавляя `through=Contact` к `ManyToManyField`. Это отношение «многие ко многим» модели `User` к самой себе: мы ссылаемся на `'self'` в поле `ManyToManyField`, чтобы создать отношение к той же модели.

Когда вам нужны дополнительные поля в отношениях «многие ко многим», создайте пользовательскую модель с `ForeignKey` для каждой стороны отношения. Добавьте `ManyToManyField` в одну из связанных моделей и укажите Django, что ваша промежуточная модель должна использоваться, включая ее в `through` параметр.

Если бы модель `User` была частью нашего приложения, мы могли бы добавить предыдущее поле в модель. Однако мы не можем изменить класс `User` напрямую, потому что он принадлежит к приложению `django.contrib.auth`. Мы будем использовать несколько иной подход, добавив это поле динамически в модель `User`. Измените файл `models.py` приложения `account` и добавьте следующие строки:

```
from django.contrib.auth.models import User

# Добавьте динамически следующее поле к User
User.add_to_class('following',
                  models.ManyToManyField('self',
                                        through=Contact,
                                        related_name='followers',
                                        symmetrical=False))
```

В предыдущем коде мы используем метод `add_to_class()` для моделей Django, чтобы обезопасить модель `User`. Имейте в виду,

Что использование `add_to_class()` не является рекомендуемым способом добавления полей в модели. Однако мы используем его в данном случае по следующим причинам:

- Мы упрощаем способ получения связанных объектов через ORM Django с помощью `user.followers.all()` и `user.following.all()`. Мы используем промежуточную модель `Contact` и избегаем сложных запросов, которые связаны с дополнительными объединениями баз данных, как это было бы, если бы мы определили отношения в нашей пользовательской модели `Profile`.
- Таблица для этого отношения «многие ко многим» будет создана с использованием модели `Contact`. Таким образом, добавленный динамически `ManyToManyField` не будет означать изменения базы данных для модели Django `User`.
- Мы не создаем пользовательскую модель, сохраняя все преимущества встроенной модели Django `User`.

Имейте в виду, что в большинстве случаев предпочтительно добавлять поля в модель `Profile`, которую мы создали ранее, вместо того, чтобы обезопасить модель `User`. Django также позволяет использовать пользовательские модели. Если вы хотите использовать свою пользовательскую модель, взгляните на документацию на <https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#specifying-a-custom-user-model>.

Вы можете заметить, что отношение включает `symmetrical=False`. Когда вы определяете `ManyToManyField` для самой модели, Django заставляет отношение быть симметричным. В этом случае мы устанавливаем `symmetrical=False` для определения

несимметричного отношения. Таким образом, если я слежу за вами, это не значит, что вы автоматически следите за мной.

Когда вы используете промежуточную модель для отношений «многие ко многим», некоторые из связанных методов менеджера отключены, например `add()`, `create()`, или `remove()`. Вместо этого вам нужно создать или удалить экземпляры промежуточной модели.

Выполните следующую команду для генерации начальных миграций для приложения `account`:

```
python manage.py makemigrations account
```

Вы получите следующий результат:

```
Migrations for 'account':  
  account/migrations/0002_contact.py  
    - Create model Contact
```

Теперь запустите следующую команду для синхронизации приложения с базой данных:

```
python manage.py migrate account
```

Вы должны увидеть вывод, который включает следующую строку:

```
Applying account.0002_contact... ok
```

Модель `Contact` теперь синхронизировалась с базой данных, и мы можем создавать отношения между пользователями. Однако наш сайт не предлагает способа просмотра пользователей или просмотра определенного профиля пользователя. Давайте создадим список и подробные представления для модели `User`.

Создание списков и подробных представлений для профилей пользователей

Откройте файл `views.py` приложения `account` и добавьте к нему следующий код:

```
from django.shortcuts import get_object_or_404
from django.contrib.auth.models import User

@login_required
def user_list(request):
    users = User.objects.filter(is_active=True)
    return render(request,
                  'account/user/list.html',
                  {'section': 'people',
                   'users': users})

@login_required
def user_detail(request, username):
    user = get_object_or_404(User,
                            username=username,
                            is_active=True)
    return render(request,
                  'account/user/detail.html',
                  {'section': 'people',
                   'user': user})
```

Это простые представления списка и детальное представление для объекта `user`. В представлении `user_list` отображаются все активные пользователи. Модель Django `User` содержит флаг `is_active`, чтобы указать, считается ли учетная запись пользователя активной. Мы фильтруем запрос `is_active=True`, чтобы возвращать только активных пользователей. Это

представление возвращает все результаты, но вы можете улучшить его, добавив разбивку на страницы так же, как мы сделали для представления `image_list`.

В представлении `user_detail` используется функция `get_object_or_404()` для извлечения активного пользователя с заданным именем пользователя. Представление возвращает ответ HTTP 404, если активный пользователь с указанным именем не найден.

Отредактируйте файл `urls.py` приложения `account` и добавьте паттерн URL для каждого представления, как показано ниже:

```
urlpatterns = [
    # ...
    path('users/', views.user_list, name='user_list'),
    path('users/<username>', views.user_detail, name='user_detail'),
]
```

Мы будем использовать шаблон URL `user_detail` для создания канонического URL-адреса для пользователей. Вы уже определили метод `get_absolute_url()` в модели, чтобы вернуть канонический URL-адрес для каждого объекта. Другой способ указать URL-адрес модели - это добавить в проект `ABSOLUTE_URL_OVERRIDES`.

Измените файл `settings.py` вашего проекта и добавьте к нему следующий код:

```
from django.urls import reverse_lazy

ABSOLUTE_URL_OVERRIDES = {
    'auth.user': lambda u: reverse_lazy('user_detail',
                                        args=[u.username])
}
```

Django динамически добавляет метод `get_absolute_url()` к любым моделям, которые отображаются в настройке `ABSOLUTE_URL_OVERRIDES`.

Этот метод возвращает соответствующий URL для данной модели, указанной в настройке. Для данного пользователя мы возвращаем URL-адрес `user_detail`. Теперь вы можете использовать `get_absolute_url()` с экземпляром `User` для получения соответствующего URL-адреса.

Откройте консоль Python с помощью команды `python manage.py shell` и запустите следующий код для проверки:

```
>>> 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. Нам нужно будет создать шаблоны для только что построенных представлений. Добавьте следующий каталог и файлы в каталог `templates/account/` приложения `account`:

```
/user/
    detail.html
    list.html
```

Отредактируйте шаблон `account/user/list.html` и добавьте к нему следующий 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 }}">
                {% thumbnail user.profile.photo "180x180" crop="100%" as im %}
                      
    <a href="{{ user.get_absolute_url }}" class="title">  
      {{ user.get_full_name }}  
    </a>  
  </div>  
  </div>  
  {% endfor %}  
</div>  
{% endblock %}
```

Предыдущий шаблон позволяет нам перечислить всех активных пользователей на сайте. Мы перебираем этих пользователей и используем тег шаблона sorl-thumbnail `{% thumbnail %}` для создания эскизов изображений профиля.

Откройте шаблон `base.html` вашего проекта и включите URL `user_list` в атрибут `href` следующего пункта меню:

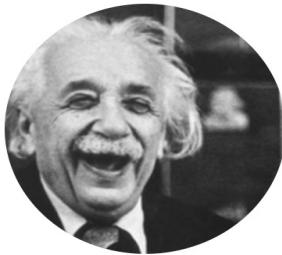
```
<li {% if section == "people" %}class="selected"{% endif %}>  
  <a href="{% url "user_list" %}">People</a>  
</li>
```

Запустите сервер разработки с помощью команды `python manage.py runserver` и откройте `http://127.0.0.1:8000/account/users/` в вашем браузере. Вы должны увидеть список пользователей, например, следующий:

People



Tesla



Einstein



Turing

Помните, что если у вас возникли трудности с созданием эскизов, вы можете добавить `THUMBNAIL_DEBUG = True` в свой файл `settings.py`, чтобы получить отладочную информацию в консоль.

Отредактируйте шаблон `account/user/detail.html` приложения `account` и добавьте к нему следующий код:

```
{% extends "base.html" %}  
{% load thumbnail %}  
  
{% block title %}{{ user.get_full_name }}{% endblock %}  
  
{% block content %}  
    <h1>{{ user.get_full_name }}</h1>  
    <div class="profile-info">
```

```
{% thumbnail user.profile.photo "180x180" crop="100%" as im %}
    
{% endthumbnail %}
</div>
{% with total_followers=user.followers.count %}
    <span class="count">
        <span class="total">{{ total_followers }}</span>
        follower{{ total_followers|pluralize }}
    </span>
    <a href="#" data-id="{{ user.id }}" data-action="{% if request.user
        in user.followers.all %}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_ajax.html" with
            images=user.images_created.all %}
    </div>
{% endwith %}
{% endblock %}
```

В подробном шаблоне мы отобразим профиль пользователя и с помощью тега шаблона `{% thumbnail %}` отобразим изображение профиля. Мы показываем общее количество подписчиков и ссылку, чтобы подписаться или отменить подписку на пользователя. Мы выполним запрос AJAX, чтобы подписаться/отменить подписку на конкретного пользователя. Мы добавляем атрибуты `data-id` и `data-action` в HTML-элемент `<a>`, включая идентификатор пользователя и начальное действие, которое будет выполняться когда он кликнет, подписаться или отписаться, это зависит от пользователя, запрашивающего страницу, является ли он подписчиком данного пользователя, или нет. Мы показываем изображения, помеченные пользователем, включая шаблон `images/image/list_ajax.html`.

Откройте браузер снова и нажмите на пользователя, у которого есть закладки для каких либо изображений. Вы увидите подробные сведения о профиле:

Tesla



0 followers

FOLLOW



Django and Duke



Louis Armstrong



Chick Corea

Создание представления AJAX для отслеживания пользователей

Мы создадим простое представление для подписки/отписки используя AJAX. Отредактируйте файл `views.py` приложения `account` добавив в него следующий код:

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST
from common.decorators import ajax_required
from .models import Contact

@ajax_required
@require_POST
@login_required
def user_follow(request):
    user_id = request.POST.get('id')
    action = request.POST.get('action')
    if user_id and action:
        try:
            user = User.objects.get(id=user_id)
            if action == 'follow':
                Contact.objects.get_or_create(
                    user_from=request.user,
                    user_to=user)
            else:
                Contact.objects.filter(user_from=request.user,
                                      user_to=user).delete()
            return JsonResponse({'status': 'ok'})
        except User.DoesNotExist:
            return JsonResponse({'status': 'ko'})
    return JsonResponse({'status': 'ko'})
```

Представление `user_follow` очень похоже на представление `image_like`, которое мы создали ранее. Поскольку мы используем пользовательскую посерединическую модель для отношения

«многие-ко-многим» для пользователей, по умолчанию методы `add()` и `remove()` автоматического менеджера `ManyToManyField` недоступны. Мы используем посредническую модель `Contact` для создания или удаления пользовательских отношений.

Отредактируйте файл `urls.py` приложения `account` и добавьте к нему следующий паттерн URL:

```
| path('users/follow/', views.user_follow, name='user_follow'),
```

Убедитесь, что вы поместили предыдущий паттерн перед паттерном URL `user_detail`. В противном случае любые запросы `/users/follow/` будут соответствовать регулярному выражению шаблона `user_detail`, и это представление будет выполнено вместо него. Помните, что в каждом HTTP-запросе Django проверяет запрашиваемый URL-адрес на соответствие каждому паттерну в порядке появления и останавливается при первом соответствии.

Отредактируйте шаблон `user/detail.html` приложения `account` и добавьте к нему следующий код:

```
{% block domready %}
$('a.follow').click(function(e){
  e.preventDefault();
  $.post('{% url "user_follow" %}', {
    id: $(this).data('id'),
    action: $(this).data('action')
  },
  function(data){
    if (data['status'] == 'ok') {
      var previous_action = $('a.follow').data('action');

      // toggle data-action
      $('a.follow').data('action',
        previous_action == 'follow' ? 'unfollow' : 'follow');
      // toggle link text
      $('a.follow').text(
        previous_action == 'follow' ? 'Unfollow' : 'Follow');
    }
  });
});
```

```
// update total followers
var previous_followers = parseInt(
    $('span.count .total').text());
$('span.count .total').text(previous_action == 'follow' ?
    previous_followers + 1 : previous_followers - 1);
}
);
});
{%- endblock %}
```

Предыдущий код - это код JavaScript который выполняет запрос AJAX для отслеживания или отмены доступа к определенному пользователю, а также для переключения ссылки подписаться/отписаться. Мы используем jQuery для выполнения запроса AJAX и устанавливаем атрибут `data-action` и текст элемента HTML `<a>` на основе его предыдущего значения. Когда выполняется действие AJAX, мы также обновляем общий счетчик, отображаемый на странице. Откройте страницу сведений о пользователе для существующего пользователя и нажмите ссылку FOLLOW чтобы проверить функциональность, которую мы только что создали. Вы увидите, что счет отслеживаемого увеличивается:

1 followers

UNFOLLOW

Построение приложения для отображения общей активности

Многие социальные сайты отображают поток активности для своих пользователей, чтобы они могли отслеживать, что другие пользователи делают на платформе. Поток активности представляет собой список последних действий, выполняемых пользователем или группой пользователей. Например, лента новостей Facebook является потоком активности. Примерами могут быть *пользователь X создал закладку изображения Y* или *пользователь X теперь отслеживает пользователя Y*. Мы создадим приложение потока активности, чтобы каждый пользователь мог видеть последние действия пользователей, которых они отслеживают. Для этого нам понадобится модель для сохранения действий, выполняемых пользователями на веб-сайте, и простой способ добавления действий в фид.

Создайте новое приложение с именем `actions` внутри вашего проекта с помощью следующей команды:

```
python manage.py startapp actions
```

Добавьте новое приложение в `INSTALLED_APPS` файла `settings.py`, чтобы активировать его в вашем проекте:

```
INSTALLED_APPS = [  
    # ...  
    'actions.apps.ActionsConfig',  
]
```

Откройте файл `models.py` приложения `actions` и добавьте в него следующий код:

```
from django.db import models

class Action(models.Model):
    user = models.ForeignKey('auth.User',
                            related_name='actions',
                            db_index=True,
                            on_delete=models.CASCADE)
    verb = models.CharField(max_length=255)
    created = models.DateTimeField(auto_now_add=True,
                                   db_index=True)

    class Meta:
        ordering = ('-created',)
```

В предыдущем коде показана модель `Action`, которая будет использоваться для хранения пользовательских действий. Поля этой модели следующие:

- `user`: Пользователь, выполнивший действие; это `ForeignKey` для модели Django `User`.
- `verb`: Глагол, описывающий действие, которое выполнил пользователь.
- `created`: Дата и время создания этого действия. Мы используем `auto_now_add=True`, чтобы автоматически установить на текущее время, когда объект впервые сохраняется в базе данных.

С помощью этой базовой модели мы можем хранить только действия, такие как *пользователь X сделал что-то*. Нам нужно дополнительное поле `ForeignKey`, чтобы сохранить действия, которые включают объект `target`, например *пользователь X добавил в закладки изображение Y* или *пользователь X теперь следит за пользователем Y*. Как вы

уже знаете, обычный `ForeignKey` может указывать только на одну модель. Вместо этого нам нужно, чтобы объект цели (target) действия был экземпляром существующей модели. Именно здесь на сцене появляется Django фреймворк типов контента.

Использование фреймворка contenttypes

Django включает фреймворк contenttypes, расположенный в `django.contrib.contenttypes`. Это приложение может отслеживать все модели, установленные в вашем проекте, и предоставляет общий интерфейс для взаимодействия с вашими моделями.

Приложение `django.contrib.contenttypes` содержится в настройках `INSTALLED_APPS` по умолчанию при создании нового проекта с помощью команды `startproject`. Оно используется другими пакетами `contrib`, такими как фреймворк проверки подлинности и приложение администратора.

Приложение `contenttypes` содержит модель `ContentType`. Экземпляры этой модели представляют собой реальные модели вашего приложения, а новые экземпляры `ContentType` автоматически создаются при установке новых моделей в ваш проект. Модель `ContentType` имеет следующие поля:

- `app_label`: Указывает имя приложения, к которому принадлежит модель. Оно автоматически берется из атрибута `app_label` параметров модели `Meta`. Например, наша модель `Image` относится к приложению `images`.
- `model`: Имя класса модели.
- `name`: Указывает на человекочитаемое имя модели. Оно автоматически берется из атрибута `verbose_name` параметров модели `Meta`.

Давайте посмотрим, как мы можем взаимодействовать с объектами `contentType`. Откройте консоль с помощью команды `python manage.py shell`. Вы можете получить объект `ContentType`, соответствующий конкретной модели, выполнив запрос с атрибутами `app_label` и `model` следующим образом:

```
>>> from django.contrib.contenttypes.models import ContentType  
>>> image_type = ContentType.objects.get(app_label='images', model='image')  
>>> image_type  
<ContentType: image>
```

Вы также можете получить класс модели из объекта `ContentType`, вызывав метод `model_class()`:

```
>>> image_type.model_class()  
<class 'images.models.Image'>
```

Таким же образом обычно получают объект `ContentType` для определенного класса модели, а именно:

```
>>> from images.models import Image  
>>> ContentType.objects.get_for_model(Image)  
<ContentType: image>
```

Это лишь некоторые примеры использования типов контента. Django предлагает больше способов для работы с ними. Вы можете найти официальную документацию о фреймворке типов контента на <https://docs.djangoproject.com/en/2.0/ref/contrib/contenttypes/>.

Добавление общих отношений к вашим моделям

В общих (родовых) отношениях объекты `ContentType` играют роль указания на модель, используемую для отношений. Для создания родового (общего) отношения в модели вам понадобятся три поля:

- **Поле ForeignKey к ContentType:** Это подскажет нам модель для отношений
- **Поле для хранения первичного ключа связанного объекта:** Обычно это будет `PositiveIntegerField` для соответствия полям автоматического первичного ключа Django
- **Поле для определения и управления общим отношением с использованием двух предыдущих полей:** Фреймворк типа контента предлагает поле `GenericForeignKey` для этой цели

Отредактируйте файл `models.py` приложения `actions` следующим образом:

```
from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey

class Action(models.Model):
    user = models.ForeignKey('auth.User',
                           related_name='actions',
```

```
        db_index=True,
        on_delete=models.CASCADE)
verb = models.CharField(max_length=255)
target_ct = models.ForeignKey(ContentType,
    blank=True,
    null=True,
    related_name='target_obj',
    on_delete=models.CASCADE)
target_id = models.PositiveIntegerField(null=True,
    blank=True,
    db_index=True)
target = GenericForeignKey('target_ct', 'target_id')
created = models.DateTimeField(auto_now_add=True,
    db_index=True)

class Meta:
    ordering = ('-created',)
```

Нам необходимо добавить следующие поля к модели `Action`:

- `target_ct`: ПОЛЕ `ForeignKey`, указывающее на модель `ContentType`
- `target_id`: `PositiveIntegerField` для хранения первичного ключа связанного объекта
- `target`: `GenericForeignKey` ПОЛЕ К СВЯЗАННОМУ ОБЪЕКТУ НА ОСНОВЕ КОМБИНАЦИИ ДВУХ ПРЕДЫДУЩИХ ПОЛЕЙ

Django не создает никаких полей в базе данных для полей `GenericForeignKey`. Единственными полями, которые сопоставляются с полями базы данных, являются `target_ct` и `target_id`. Оба поля имеют атрибуты `blank=True` И `null=True`, ПОЭТОМУ объект `target` не требуется при сохранении `Action` объекта.

Вы можете сделать свои приложения более гибкими, используя общие отношения вместо внешних ключей, когда есть смысл иметь общее отношение.

Для создания начальных миграций для этого приложения выполните следующую команду:

```
python manage.py makemigrations actions
```

Вы должны увидеть следующий вывод:

```
Migrations for 'actions':  
  actions/migrations/0001_initial.py  
    - Create model Action
```

Затем запустите следующую команду для синхронизации приложения с базой данных:

```
python manage.py migrate
```

Вывод команды должен указывать, что новые миграции были применены следующим образом:

```
Applying actions.0001_initial... ok
```

Давайте добавим модель `Action` на сайт администрирования. Измените файл `admin.py` приложения `actions` и добавьте к нему следующий 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. Выполните команду `python manage.py runserver` для инициализации сервера разработки и откройте <http://127.0.0.1:8000/admin/actions/action/add/> в вашем браузере. Вы должны увидеть страницу для создания нового объекта `Action`, как показано ниже:

Django administration

WELCOME, ANTONIO. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Actions > Actions > Add action

Add action

User: edit +

Target ct:

Target id:

Verb:

Save and add another Save and continue editing SAVE

Как вы могли заметить на предыдущем снимке экрана, отображаются только поля `target_ct` и `target_id`, которые отображаются в фактические поля базы данных. Поле `GenericForeignKey` не отображается в форме. Поле `target_ct` позволяет вам выбрать любую зарегистрированную модель вашего проекта Django. Вы можете ограничить типы контента выбором ограниченного набора моделей с помощью атрибута `limit_choices_to` в поле `target_ct`: атрибут `limit_choices_to` позволяет вам ограничить содержимое полей `ForeignKey` определенным набором значений.

Создайте новый файл в каталоге приложений `actions` и назовите его `utils.py`. Мы определим функцию быстрого доступа, которая позволит нам создавать новые объекты `Action` простым способом. Отредактируйте новый файл `utils.py` и добавьте к нему следующий 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 нам создавать действия, которые необязательно включают объект `target`. Мы можем использовать эту функцию в любом месте нашего кода для добавления новых действий в поток активности.

Избегаем дублирования действий в потоке активности

Иногда ваши пользователи могут выполнять действие несколько раз. Они могут несколько раз щелкнуть по кнопкам LIKE или UNLIKE или выполняют одно и то же действие несколько раз за короткий промежуток времени. Это приведет к хранению и отображению повторяющихся действий. Чтобы избежать этого, мы улучшим функцию `create_action()`, чтобы пропустить очевидные дублированные действия.

Отредактируйте файл `utils.py` Приложения `actions` следующим образом:

```
import datetime
from django.utils import timezone
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    # проверяем на какое-либо подобное действие 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 подобных действий не найдено
        action = Action(user=user, verb=verb, target=target)
        action.save()
```

```
    return True  
    return False
```

Мы изменили функцию `create_action()`, чтобы избежать сохранения повторяющихся действий и возвратить Boolean, чтобы определить, было ли действие сохранено или нет. Так мы избегаем дубликатов:

- Во-первых, мы получаем текущее время, используя метод `timezone.now()`, предоставленный Django. Этот метод делает то же самое, что и `datetime.datetime.now()`, но возвращает объект `timezone-aware`. Django предоставляет параметр под названием `use_tz`, чтобы включить или отключить поддержку часового пояса. Файл `settings.py` по умолчанию, созданный с помощью команды `startproject`, включает `use_tz=True`.
- Мы используем переменную `last_minute` для хранения даты и времени одну минуту назад и извлекаем любые идентичные действия, выполненные пользователем с тех пор.
- Мы создаем объект `Action`, если в последнюю минуту не было совершено подобного действия. Мы возвращаем `True`, если был создан объект `Action`, иначе `False`.

Добавление действий пользователя в поток активности

Пришло время добавить некоторые действия к нашим представлениям, чтобы создать поток активности для наших пользователей. Мы будем хранить действие для каждого из следующих взаимодействий:

- Пользователь добавляет в закладки изображение
- Пользователь лайкнул изображение
- Пользователь создает учетную запись
- Пользователь начинает отслеживать другого пользователя

Отредактируйте файл `views.py` приложения `images` и добавьте следующий импорт:

```
from actions.utils import create_action
```

В представлении `image_create` добавте `create_action()` после сохранения изображения, таким образом:

```
new_item.save()
create_action(request.user, 'bookmarked image', new_item)
```

В представлении `image_like` добавьте `create_action()` после

добавления пользователя в отношение `users_like`, как показано ниже:

```
image.users_like.add(request.user)
create_action(request.user, 'likes', image)
```

Теперь отредактируйте файл `views.py` приложения `account` и добавьте следующий импорт:

```
from actions.utils import create_action
```

В представлении `register` добавьте `create_action()` после создания объекта `Profile`, как показано ниже:

```
Profile.objects.create(user=new_user)
create_action(new_user, 'has created an account')
```

В представление `user_follow` добавте `create_action()`:

```
Contact.objects.get_or_create(user_from=request.user,
                               user_to=user)
create_action(request.user, 'is following', user)
```

Как вы можете видеть в предыдущем коде, благодаря нашей модели `Action` и нашей вспомогательной функции, очень легко сохранить новые действия в поток активности.

Отображение потока активности

Наконец, нам понадобится способ отображения потока активности для каждого пользователя. Мы включим поток активности в панель управления пользователя.

Отредактируйте файл `views.py` приложения `account`. Импортируйте модель `Action` и измените представление панели мониторинга следующим образом:

```
from actions.models import Action

@login_required
def dashboard(request):
    # Отображать все действия по умолчанию
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',
                                                       flat=True)
    if following_ids:
        # Если пользователь следит за другими, извлекать 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 предыдущем представлении мы извлекаем все действия из базы данных, за исключением тех, которые выполняются текущим пользователем. По умолчанию мы будем получать последние действия, выполняемые всеми пользователями на платформе. Если пользователь следит за другими пользователями, мы ограничиваем запрос на получение только действий, выполняемых пользователями, которых он отслеживает. Наконец, мы ограничиваем результат возвратом

первых 10 действий. Мы не используем `order_by()` в `QuerySet`, потому что мы полагаемся на порядок по умолчанию, указанный в параметрах `Meta` модели `Action`. Последние действия выводятся первыми, так как мы установили `ordering = ('-created',)` в модель `Action`.

Оптимизация QuerySet, которые взаимодействуют со связанными объектами

Каждый раз, когда вы извлекаете объект `Action`, вы обычно обращаетесь к связанному с ним объекту `User` и связанному с ним объекту `Profile`. Django ORM предлагает простой способ одновременного извлечения связанных объектов, тем самым избегая дополнительных запросов к базе данных.

Использование select_related()

Django предлагает метод `QuerySet` с именем `select_related()`, который позволяет извлекать связанные объекты для отношений «один ко многим». Это приводит к одному, более сложному `QuerySet`, но вы избегаете дополнительных запросов при доступе к связанным объектам. Метод `select_related` предназначен для полей `ForeignKey` и `OneToOne`. Он работает путем выполнения SQL `JOIN` и включает в себя поля связанного объекта в инструкции `SELECT`.

Чтобы воспользоваться `select_related()`, отредактируйте следующую строку предыдущего кода:

```
actions = actions[:10]
```

Кроме того, добавьте `select_related` в поля, которые вы будете использовать, например:

```
actions = actions.select_related('user', 'user_profile')[:10]
```

Мы используем `user_profile` для присоединения к таблице `Profile` в одном SQL запросе. Если вы вызываете `select_related()` без передачи каких-либо аргументов, он будет извлекать объекты из всех отношений `ForeignKey`. Всегда ограничивайте `select_related()` отношениями, которые после этого будут доступны.

Используйте `select_related()` - это может значительно улучшить время выполнения.

Использование prefetch_related()

`select_related()` поможет вам повысить производительность для извлечения связанных объектов в отношениях «один ко многим». Однако `select_related()` не может работать для отношений «многие-ко-многим» или «многие-к-одному» (`ManyToMany` или обратный `ForeignKey`). Django предлагает другой метод `QuerySet`, называемый `prefetch_related`, который работает для отношений «многие-ко-многим» и «многие-к-одному» в дополнение к отношениям, поддерживаемым `select_related()`. Метод `prefetch_related()` выполняет отдельный поиск для каждого отношения и объединяет результаты с использованием Python. Этот метод также поддерживает предварительную выборку `GenericRelation` и `GenericForeignKey`.

Измените файл `views.py` приложения `account` и выполните запрос, добавив к нему `prefetch_related()` для целевого поля `GenericForeignKey`, как показано ниже:

```
actions = actions.select_related('user', 'user__profile')\
    .prefetch_related('target')[:10]
```

Этот запрос теперь оптимизирован для извлечения действий пользователя, включая связанные объекты.

Создание шаблона для actions

Теперь мы создадим шаблон для отображения конкретного объекта `Action`. Создайте новый каталог внутри каталога приложения `actions` и назовите его `templates`. Добавьте к нему следующую структуру файлов:

```
actions/
    action/
        detail.html
```

Отредактируйте шаблон `actions/action/detail.html` добавив в него следующий код:

```
{% load thumbnail %}

{% with user=action.user profile=action.user.profile %}
<div class="action">
    <div class="images">
        {% if profile.photo %}
            {% thumbnail user.profile.photo "80x80" crop="100%" as im %}
                <a href="{{ user.get_absolute_url }}>
                    
                </a>
            {% endthumbnail %}
        {% endif %}

        {% if action.target %}
            {% with target=action.target %}
                {% if target.image %}
                    {% thumbnail target.image "80x80" crop="100%" as im %}
                        <a href="{{ target.get_absolute_url }}>
                            
                        </a>
                    {% endthumbnail %}
                {% endif %}
            {% endwith %}
        {% endif %}
    </div>
</div>
```

```

        {% 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, используемый для отображения объекта `Action`. Во-первых, мы используем тег шаблона `{% with %}`, чтобы получить пользователя, выполняющего действие, и связанный с ним объект `Profile`. Затем мы показываем изображение объекта `target`, если объект `Action` имеет связанный объект `target`. Наконец, мы показываем ссылку на пользователя, выполнившего действие, само действие (глагол) и объект `target`, если таковые имеются.

Теперь отредактируйте шаблон `account/dashboard.html` приложения `account` и добавьте следующий код в конец блока `content`:

```

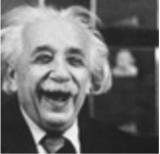
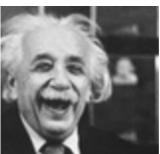
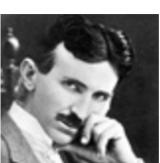
<h2>What's happening</h2>
<div id="action-list">
    {% for action in actions %}
        {% include "actions/action/detail.html" %}
    {% endfor %}
</div>

```

Откройте `http://127.0.0.1:8000/account/` в своем браузере. Войдите в систему под существующим пользователем и выполните несколько действий, чтобы они были сохранены в базе данных.

Затем войдите в систему с использованием другого пользователя, который отслеживает отслеживаемого пользователя и взгляните на генерированный поток действий на странице панели мониторинга. Он должен 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

Мы просто создали полный поток активности для наших пользователей, и мы можем легко добавить к нему новые действия пользователя. Вы также можете добавить функциональность бесконечной прокрутки в поток активности, внедряя один и тот же файл-указатель AJAX, который вы использовали для представления `image_list`.

Использование сигналов для денормализации счетчиков

Есть случаи, когда вы хотите денормализовать свои данные. Денормализация делает избыточность данных таким образом, что это оптимизирует производительность чтения. Вы должны быть осторожны с денормализацией и использовать ее, только когда вам это действительно нужно. Самая большая проблема, которую вы обнаружите с денормализацией, заключается в том, что сложно сохранить ваши денормализованные данные обновленными.

Мы рассмотрим пример того, как улучшить наши запросы, денормализируя подсчеты. Недостатком является то, что мы должны обновлять избыточные данные. Мы будем денормализовать данные из нашей модели `Image` и использовать сигналы Django для обновления данных.

Работа с сигналами

Django поставляется с диспетчером сигналов, который позволяет функциям `receiver` получать уведомления, когда происходят определенные действия. Сигналы очень полезны, когда вам нужно, чтобы ваш код делал что-то каждый раз, когда что-то происходит. Вы также можете создавать свои собственные сигналы, чтобы другие могли получать уведомления, когда происходит событие.

Django предоставляет несколько сигналов для моделей, расположенных в `django.db.models.signals`. Некоторые из этих сигналов следующие:

- `pre_save` И `post_save` отправляются до или после вызова метода модели `save()`
- `pre_delete` И `post_delete` отправляются до или после вызова метода модели `delete()` или `QuerySet`
- `m2m_changed` отправляется при изменении `ManyToManyField` в модели

Это всего лишь часть сигналов, предоставляемых Django. Вы можете найти список всех встроенных сигналов в <https://docs.djangoproject.com/en/2.0/ref/signals/>.

Предположим, вы хотите получить изображения отсортированные по популярности. Вы можете использовать функции агрегации Django для получения изображений, упорядоченных по количеству пользователей, которым они

нравятся. Вспомните, вы использовали функции агрегации Django в [главе 3](#), *Расширение вашего приложения блог*. Следующий код будет извлекать изображения в зависимости от количества 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 упорядочивание изображений путем подсчета общего количества лайков более дорогое с точки зрения производительности, чем упорядочение их по полю, в котором хранится общее количество лайков. Вы можете добавить поле в модель `Image` для денормализации общего количества лайкнувших пользователей, чтобы повысить производительность в запросах, связанных с этим полем. Теперь проблема в том, как обновить это поле?

Измените файл `models.py` приложения `images` и добавьте следующее поле `total_likes` в модель `Image`:

```
class Image(models.Model):
    # ...
    total_likes = models.PositiveIntegerField(db_index=True,
                                              default=0)
```

Поле `total_likes` позволит нам хранить общее количество пользователей, которым нравится изображение. Денормализационные подсчеты полезны, когда вы хотите фильтровать или выполнять `QuerySet` по ним.

Существует несколько способов повысить производительность, которые вы должны учитывать, прежде чем денормализовать поля. Обратите внимание на индексы базы данных, оптимизацию запросов и кеширование, прежде чем начинать денормализовывать свои данные.

Выполните следующую команду для создания миграций чтобы

добавить новые поля в таблицу базы данных:

```
python manage.py makemigrations images
```

Вы должны увидеть следующий результат:

```
Migrations for 'images':  
  images/migrations/0002_image_total_likes.py  
    - Add field total_likes to image
```

Затем выполните следующую команду, чтобы применить миграцию:

```
python manage.py migrate images
```

Вывод должен включать следующую строку:

```
Applying images.0002_image_total_likes... OK
```

Мы присоединим функцию `receiver` к сигналу `m2m_changed`. Создайте новый файл в каталоге приложений `images` и назовите его `singals.py`. Добавьте к нему следующий 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()
```

Во-первых, мы регистрируем функцию `users_like_changed` в качестве функции `receiver`, используя декоратор `receiver()`, и присоединяем ее к `m2m_changed`. Мы подключаем функцию к `Image.users_like.through`, чтобы функция вызывалась только, если этим отправителем был отправлен сигнал `m2m_changed`. Существует

альтернативный метод регистрации функции `receiver`, который состоит из метода `connect()` объекта `Signal`.

Сигналы Django синхронны и блокируют. Не путайте сигналы с асинхронными задачами. Однако вы можете комбинировать их для запуска асинхронных задач, когда ваш код получает уведомление по сигналу.

Вы должны подключить вашу функцию приемник к сигналу, чтобы он вызывался каждый раз, когда посыпался сигнал.

Рекомендуемый способ регистрации ваших сигналов - импортировать их в метод `ready()` вашего класса конфигурации приложения. Django предоставляет реестр приложений, который позволяет настраивать и анализировать ваши приложения.

Классы конфигурации приложений

Django позволяет указать классы конфигурации для ваших приложений. Когда вы создаете приложение с помощью команды `startapp`, Django добавляет файл `apps.py` в каталог приложения, включая базовую конфигурацию приложения, которая наследуется от `AppConfig` .

Класс конфигурации приложения позволяет хранить метаданные и конфигурацию для приложения и обеспечивает интроспекцию для приложения. Дополнительную информацию о конфигурациях приложений можно найти на странице <https://docs.djangoproject.com/en/2.0/ref/applications/>.

Чтобы зарегистрировать функции вашего сигнала `receiver` , когда вы используете декоратор `receiver()` , вам просто нужно импортировать модуль сигналов вашего приложения внутри `ready()` класса конфигурации приложения. Этот метод вызывается, как только полностью заполняется реестр приложений. Любые другие инициализации для вашего приложения также должны быть включены в этот метод.

Отредактируйте файл `apps.py` приложения `images` добавив следующий код:

```
from django.apps import AppConfig

class ImagesConfig(AppConfig):
    name = 'images'

    def ready(self):
        # import signal handlers
```

```
| import images.signals
```

Мы импортируем сигналы для этого приложения в метод `ready()`, чтобы они импортировались при загрузке приложения `images`.

Запустите сервер разработки с помощью следующей команды:

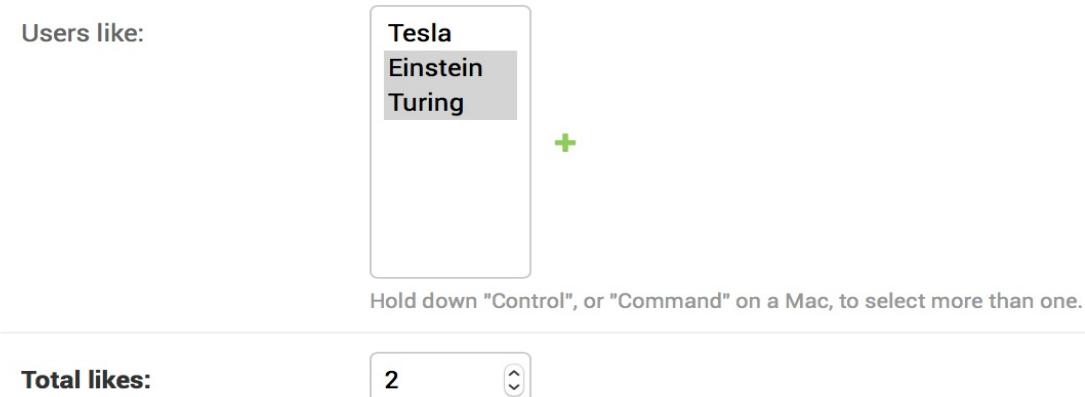
```
| python manage.py runserver
```

Откройте браузер, чтобы просмотреть страницу с подробной информацией об изображении, и нажмите кнопку LIKE.

Вернитесь на сайт администрирования, перейдите к URL-адресу редактируемого изображения, например

`http://127.0.0.1:8000/admin/images/image/1/change/`, и взгляните на `total_likes`.

Вы должны увидеть, что атрибут `total_likes` обновляется общим количеством пользователей, которым нравится изображение, следующим образом:



Теперь вы можете использовать атрибут `total_likes` для

упорядочивания изображений по популярности или отображения значения в любом месте, избегая сложных запросов для его расчета. Рассмотрим следующий запрос, чтобы получить картинки упорядоченные в соответствии с их количеством лайков:

```
from django.db.models import Count  
  
images_by_popularity = Image.objects.annotate(  
    likes=Count('users_like')).order_by('-likes')
```

Этот запрос теперь можно записать следующим образом:

```
images_by_popularity = Image.objects.order_by('-total_likes')
```

Что приводит к менее дорогостоящему SQL-запросу. Это всего лишь пример использования сигналов Django.

Используйте сигналы с осторожностью, так как они затрудняют контроль потока управления. Во многих случаях вы можете избежать использования сигналов, если знаете, какие приемники должны быть уведомлены.

Вам нужно будет установить начальные подсчеты в соответствии с текущим статусом базы данных. Откройте консоль с помощью команды `python manage.py shell` и запустите следующий код:

```
from images.models import Image  
for image in Image.objects.all():  
    image.total_likes = image.users_like.count()  
    image.save()
```

Количество просмотров для каждого изображения теперь обновляется.

Использование Redis для хранения запросов к представлениям

Redis - это расширенная база данных ключ/значение, которая позволяет вам сохранять разные типы данных и очень быстро работает в операциях ввода-вывода. Redis хранит все в памяти, но данные могут сохраняться путем временного сброса набора данных на диск или добавления каждой команды в журнал.

Redis очень универсален по сравнению с другими хранилищами "ключ/значение". Он предоставляет набор мощных команд и поддерживает различные структуры данных, такие как строки, хэши, списки, наборы, упорядоченные наборы и даже растровые изображения или HyperLogLogs.

Несмотря на то, что SQL лучше всего подходит для постоянного хранения данных с помощью схемы, Redis предлагает множество преимуществ при работе с быстро изменяющимися данными, энергозависимым хранилищем или когда необходим быстрый кеш. Давайте посмотрим, как Redis можно использовать для создания новой функциональности в нашем проекте.

Установка Redis

Загрузите последнюю версию Redis из <https://redis.io/download>.

Разархивируйте файл `tar.gz`, войдите в каталог `redis` и скомпилируйте Redis с помощью команды `make`, как показано ниже:

```
cd redis-4.0.9  
make
```

После его установки используйте следующую команду консоли для инициализации сервера Redis:

```
src/redis-server
```

Вы должны увидеть вывод, который заканчивается следующими строками:

```
# Server initialized  
* Ready to accept connections
```

По умолчанию Redis работает на порту 6379. Вы можете указать пользовательский порт, используя флаг `--port`, например, `redis-server --port 6655`.

Сохраните сервер Redis и откройте другую консоль. Запустите клиент Redis с помощью следующей команды:

```
src/redis-cli
```

Вы увидите приглашение консоли клиента Redis следующим

образом:

```
127.0.0.1:6379>
```

Клиент Redis позволяет выполнять команды Redis непосредственно из консоли. Попробуем несколько команд. Введите команду `SET` в оболочке Redis для сохранения значения в ключе:

```
127.0.0.1:6379> SET name "Peter"
OK
```

Предыдущая команда создает ключ `name` со строковым значением `"Peter"` в базе данных Redis. Вывод `OK` указывает, что ключ был успешно сохранен. Затем извлеките значение с помощью команды `GET`, как показано ниже:

```
127.0.0.1:6379> GET name
"Peter"
```

Вы также можете проверить, существует ли ключ с помощью команды `EXISTS`. Эта команда возвращает `1`, если данный ключ существует, `0` в противном случае:

```
127.0.0.1:6379> EXISTS name
(integer) 1
```

Вы можете установить время истечения срока действия ключа, используя команду `EXPIRE`, которая позволяет вам установить время для жизни в секундах. Другой вариант - использовать команду `EXPIREAT`, которая ожидает отметку времени Unix. Выделение ключа полезно использовать в Redis в качестве кеша или чтобы хранить изменяемые данные:

```
127.0.0.1:6379> GET name
"Peter"
```

```
127.0.0.1:6379> EXPIRE name 2
(integer) 1
```

Подождите две секунды и попробуйте снова получить тот же ключ:

```
127.0.0.1:6379> GET name
(nil)
```

Ответ `(nil)` является нулевым ответом и означает, что ключ не найден. Вы также можете удалить любой ключ 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 основные команды для ключевых операций. Redis включает в себя большой набор команд для других типов данных, таких как строки, хэши, наборы и упорядоченные наборы. Вы можете просмотреть все команды Redis в <https://redis.io/commands> и все типы данных Redis в <https://redis.io/topics/data-types>.

Использование Redis в Python

Нам понадобится программа для взаимодействия Python с Redis. Установите `redis-py` через `pip`, используя следующую команду:

```
pip install redis==2.10.6
```

Вы можете найти документацию по `redis-py` на <https://redis-py.readthedocs.io/>.

Пакет `redis-py` предлагает два класса для взаимодействия с Redis: `StrictRedis` и `Redis`. Оба имеют одинаковую функциональность.

Класс `StrictRedis` пытается придерживаться официального синтаксиса команд Redis. Класс `Redis` расширяет `StrictRedis`, переопределяя некоторые методы для обеспечения обратной совместимости. Мы будем использовать `StrictRedis`, поскольку он следует синтаксису команд Redis. Откройте консоль Python и выполните следующий код:

```
>>> import redis  
>>> r = redis.StrictRedis(host='localhost', port=6379, db=0)
```

Предыдущий код создает соединение с базой данных Redis. В Redis базы данных идентифицируются индексом целого числа вместо имени базы данных. По умолчанию клиент подключается к базе данных 0. Количество доступных баз данных Redis установлено в 16, но вы можете изменить это в конфигурационном файле `redis.conf`.

Теперь установите ключ с помощью консоли Python:

```
>>> r.set('foo', 'bar')
True
```

Команда возвращает `True`, сообщая, что ключ был успешно создан. Теперь вы можете получить ключ с помощью команды `get()`:

```
>>> r.get('foo')
b'bar'
```

Как вы можете заметить из предыдущего кода, методы `StrictRedis` следуют синтаксису команд Redis.

Давайте интегрируем Redis в наш проект. Измените файл `settings.py` проекта `bookmark` и добавьте в него следующие настройки:

```
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 0
```

Это настройки для сервера Redis и базы данных, которые мы будем использовать для нашего проекта.

Сохранение запросов к представлениям в Redis

Давайте найдем способ сохранить общее количество просмотров изображения. Если мы реализуем это с помощью Django ORM, он будет включать запрос SQL UPDATE каждый раз при отображении изображения. Если вместо этого мы будем использовать Redis, нам просто нужно увеличить счетчик, хранящийся в памяти, что приведет к значительно лучшей производительности и меньшим накладным расходам.

Измените файл `views.py` приложения `images` и добавьте следующий код к нему после существующих операторов `import`:

```
import redis
from django.conf import settings

# connect to redis
r = redis.StrictRedis(host=settings.REDIS_HOST,
                      port=settings.REDIS_PORT,
                      db=settings.REDIS_DB)
```

С помощью предыдущего кода мы устанавливаем соединение с Redis, чтобы использовать его в наших представлениях.

Отредактируйте представление `image_detail` следующим образом:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # увеличить общее количество просмотров на 1
    total_views = r.incr('image:{}:views'.format(image.id))
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

В этом представлении мы используем команду `incr`, которая увеличивает значение заданного ключа на 1. Если ключ не существует, команда `incr` создаст его заранее. Метод `incr()` возвращает окончательное значение ключа после выполнения операции. Мы сохраняем значение в переменной `total_views` и передаем его в контексте шаблона. Мы создаем ключ Redis с использованием обозначений, таких как `object-type:id:field` (для примера, `image:33:id`).

Соглашение для наименования ключей Redis состоит в использование двоеточия в качестве разделителя для создания ключей с именами. Таким образом, имена ключей являются особенно подробными, а связанные ключи совместно используют одну и ту же схему в своих именах.

Отредактируйте шаблон `images/image/detail.html` приложения `images` и добавить к нему следующий код, после элемента ``:

```
<span class="count">
  {{ total_views }} view{{ total_views|pluralize }}
</span>
```

Теперь откройте в браузере страницу детальной информации об изображении и перезагрузите ее несколько раз. Вы увидите, что каждый раз, когда просмотр обрабатывается, общее количество отображаемых просмотров увеличивается на 1. Взгляните на следующий пример:

Django and Duke



0 likes

16 views

LIKE

Django and Duke image.

Nobody likes this image yet.

Прекрасно! Вы успешно интегрировали Redis в свой проект, чтобы хранить счетчик элементов.

Сохранение рейтинга в Redis

Давайте создадим что-то более сложное с Redis. Мы создадим рейтинг наиболее просматриваемых изображений на нашей платформе. Для построения этого рейтинга мы будем использовать отсортированные наборы Redis.

Отсортированный набор представляет собой неповторяющийся набор строк, в котором каждый член связан со счетчиком. Элементы сортируются по их оценке.

Отредактируйте файл `views.py` приложения `images` и измените представление `image_detail` следующим образом:

```
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    # increment total image views by 1
    total_views = r.incr('image:{}:views'.format(image.id))
    # увеличиваем рейтинг изображения на 1
    r.zincrby('image_ranking', image.id, 1)
    return render(request,
                  'images/image/detail.html',
                  {'section': 'images',
                   'image': image,
                   'total_views': total_views})
```

Мы используем команду `zincrby()` для хранения изображений в отсортированном наборе с помощью `image:ranking` ключа. Мы сохраним `id` изображения и соответствующий балл `1`, который будет добавлен к общей оценке этого элемента в отсортированном наборе. Это позволит нам отслеживать все изображения в глобальном масштабе и иметь отсортированный набор, упорядоченный по общему количеству просмотров.

Теперь создайте новое представление для отображения

ранжирования наиболее просматриваемых изображений.
Добавьте следующий код в файл `views.py` приложения `images`:

```
@login_required
def image_ranking(request):
    # получить словарь ранжирования изображений
    image_ranking = r.zrange('image_ranking', 0, -1,
                             desc=True)[:10]
    image_ranking_ids = [int(id) for id in image_ranking]
    # получать 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:

1. Мы используем команду `zrange()` для получения элементов в отсортированном наборе. Эта команда ожидает пользовательский диапазон в соответствии с самым низким и самым высоким значением счетчика. Используя `0` как самый низкий и `-1` как самый высокий балл, мы сообщаем Redis возвращать все элементы в отсортированном наборе. Мы также указываем `desc=True` для извлечения элементов, упорядоченных по убыванию. Наконец, мы ограничиваем результаты, используя `[:10]`, чтобы получить первые 10 элементов с наивысшим результатом.
2. Мы создаем список возвращаемых идентификаторов изображений и сохраняем его в переменной `image_ranking_ids` в виде списка целых чисел. Мы извлекаем объекты `Image` для этих идентификаторов и принудительно выполняем запрос с помощью функции

`list()`. Важно принудительно выполнить `QuerySet`, потому что теперь мы будем использовать метод списка `sort()` (на данный момент нам нужен список объектов вместо `QuerySet`).

3. Мы сортируем объекты `Image` по их индексу в ранжировании изображений. Теперь мы можем использовать список `most_viewed` в нашем шаблоне для отображения 10 наиболее просматриваемых изображений.

Создайте новый шаблон `ranking.html` в каталоге шаблона `images/image/` приложения `images` и добавьте к нему следующий 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 %}
```

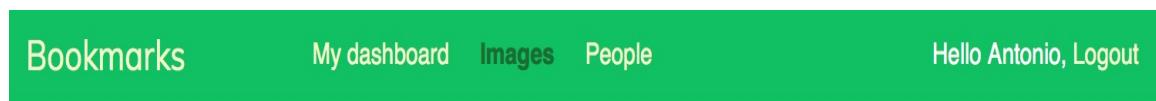
Шаблон довольно прост. Мы перебираем объекты `Image`, содержащиеся в списке `most_viewed`, и отображаем их имена, включая ссылку на страницу подробных сведений о изображении.

Наконец, вам нужно будет создать паттерн URL для нового представления. Откройте файл `urls.py` приложения `images` и

добавьте к нему следующий шаблон:

```
| path('ranking/', views.image_ranking, name='create'),
```

Запустите сервер разработки, зайдите на свой сайт в веб-браузере и перезагрузите несколько раз страницы с разными изображениями. Затем зайдите на <http://127.0.0.1:8000/images/ranking/> из своего браузера. Вы должны иметь возможность видеть рейтинг изображений следующим образом:



Images ranking

-
1. [Chick Corea](#)
 2. [Louis Armstrong](#)
 3. [Al Jarreau](#)
 4. [Django Reinhardt](#)
 5. [Django and Duke](#)

Великолепно! Вы только что создали рейтинг с Redis.

Следующие шаги с Redis

Redis не заменит вашу базу данных SQL, а обеспечит быстрое хранение в памяти, которое лучше подходит для определенного круга задач. Добавьте его в ваш стек технологий и используйте, когда вы действительно чувствуете, что это необходимо. Ниже приведены некоторые сценарии, в которых Redis подходит довольно хорошо:

- **Подсчет:** Как вы видели, очень легко управлять счетчиками с Redis. Вы можете использовать `incr()` и `incrby()` для подсчета.
- **Хранение последних значений:** Вы можете добавлять элементы в начало/конец списка, используя `lpush()` и `rpush()`. Удалять и вставлять первый/последний элемент, используя `lpop()`/`rpop()`. Вы можете обрезать длину списка, используя `ltrim()`, чтобы сохранить нужную длину.
- **Очереди:** В дополнение к командам `push` и `pop`, Redis предлагает команды блокировки очереди.
- **Кэширование:** Используя `expire()` и `expireat()`, вы можете использовать Redis в качестве кеша. Вы также можете использовать резервные копии Redis в Django.
- **Pub/sub:** Redis предоставляет команды для подписки/отмены подписки и отправки сообщений на каналы.
- **Ранжирование и рейтинги:** Redis отсортировал

наборы с множествами, что очень упростило создание лидеров.

- **Отслеживание в реальном времени:** Быстрый ввод-вывод Redis делает его идеальным для сценариев реального времени.

Резюме

В этой главе вы создали систему отслеживающего устройства и потока активности пользователя. Вы узнали, как работают сигналы Django и интегрировали Redis в ваш проект.

В следующей главе вы узнаете, как создать интернет-магазин. Вы создадите каталог продуктов и создадите корзину покупок, используя сессии. Вы также узнаете, как запускать асинхронные задачи с помощью Celery.

Глава 7

Создание интернет-магазина

В предыдущей главе вы создали отслеживающую систему и поток активности пользователя. Вы также узнали, как работают сигналы Django и интегрировали Redis в свой проект, чтобы подсчитывать лайки изображения. В этой главе вы узнаете, как создать базовый интернет-магазин. Вы создадите каталог продуктов и реализуете корзину покупок с помощью сессий Django. Вы также узнаете, как создавать пользовательские процессоры контекста и запускать асинхронные задачи с помощью Celery.

В этой главе вы научитесь:

- Создавать каталог товаров
- Создавать корзину покупок с помощью сеансов Django
- Управлять заказами клиентов
- Отправлять асинхронные уведомления клиентам, используя Celery

Создание проекта интернет-магазина

Мы собираемся начать с нового проекта Django по созданию интернет-магазина. Наши пользователи смогут просматривать каталог продуктов и добавлять товары в корзину покупок. Наконец, они смогут проверить корзину и разместить заказ. В этой главе будут рассмотрены следующие функции интернет-магазина:

- Создание моделей каталога продуктов, их добавление на сайт администрирования и создание основных представлений для отображения каталога
- Создание системы корзины покупок с использованием сессий Django, чтобы пользователи могли сохранять выбранные продукты во время просмотра сайта
- Создание формы и функциональности для размещения заказов на сайте
- Отправка асинхронного подтверждения по электронной почте пользователям при размещении заказа

Откройте консоль, создайте виртуальную среду для нового проекта и активируйте ее следующими командами:

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

Установите Django в свою виртуальную среду с помощью следующей команды:

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

Запустите новый проект под названием `myshop`, открыв консоль и выполнив следующие команды:

```
django-admin startproject myshop
cd myshop/
django-admin startapp shop
```

Измените файл `settings.py` вашего проекта и добавьте приложение `shop` в настройки `INSTALLED_APPS` следующим образом:

```
INSTALLED_APPS = [
    # ...
    'shop.apps.ShopConfig',
]
```

Теперь ваше приложение активно для этого проекта. Давайте создадим модели для каталога продуктов.

Создание моделей каталога товаров

Каталог нашего магазина будет состоять из продуктов, которые организованы в разные категории. Каждый продукт будет иметь имя, необязательное описание, необязательное изображение, цену и доступность. Откройте файл `models.py` приложения `shop`, которое вы только что создали, и добавьте следующий код:

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200,
                           db_index=True)
    slug = models.SlugField(max_length=200,
                           unique=True)

    class Meta:
        ordering = ('name',)
        verbose_name = 'category'
        verbose_name_plural = 'categories'

    def __str__(self):
        return self.name

class Product(models.Model):
    category = models.ForeignKey(Category,
                                 related_name='products',
                                 on_delete=models.CASCADE)
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True)
    image = models.ImageField(upload_to='products/%Y/%m/%d',
                             blank=True)
    description = models.TextField(blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
```

```
class Meta:  
    ordering = ('name',)  
    index_together = (('id', 'slug'),)  
  
def __str__(self):  
    return self.name
```

Это модели `Category` и `Product`. Модель `Category` состоит из поля `name` и уникального поля `slug` (`unique` подразумевает создание индекса). Поле модели `Product` выглядит следующим образом:

- `category`: `ForeignKey` к модели `Category`. Это отношение «много-к-одному»: продукт относится к одной категории, а категория содержит несколько продуктов.
- `name`: Название продукта.
- `slug`: `slug` этого продукта для создания красивых URL-адресов.
- `image`: Дополнительное изображение продукта.
- `description`: Дополнительное описание продукта.
- `price`: Это поле использует тип Python `decimal.Decimal` для хранения десятичного числа с фиксированной точностью. Максимальное количество цифр (включая десятичные разряды) устанавливается с помощью атрибута `max_digits` и десятичных знаков с атрибутом `decimal_places`.
- `available`: Логическое значение, указывающее, доступен ли продукт или нет. Оно будет использоваться для включения/выключения продукта в каталоге.
- `created`: Это поле хранит дату создания объекта.

- `updated`: Это поле хранит последнее обновление объекта.

Для поля `price` мы используем `DecimalField` вместо `FloatField`, чтобы избежать проблем округления.

Всегда используйте `DecimalField` для хранения денежных сумм. `FloatField` использует внутренний тип Python `float`, тогда как `DecimalField` использует тип Python `Decimal`. Используя `Decimal`, вы избежите проблем с округлением `float`.

В классе `Meta` модели `Product` мы используем мета-параметр `index_together`, чтобы указать индекс для `id` и `slug` вместе. Мы определяем этот индекс, потому что мы планируем запрашивать продукты с помощью `id` и `slug`. Оба поля индексируются вместе для улучшения характеристик запросов, в которых используются два поля.

Поскольку мы собираемся иметь дело с изображениями в наших моделях, откройте консоль и установите `Pillow` с помощью следующей команды:

```
pip install Pillow==5.1.0
```

Теперь запустите следующую команду для создания начальных миграций для вашего проекта:

```
python manage.py makemigrations
```

Вы увидите следующий результат:

```
Migrations for 'shop':  
  shop/migrations/0001_initial.py  
    - Create model Category  
    - Create model Product  
    - Alter index_together for product (1 constraint(s))
```

Выполните следующую команду для синхронизации базы

данных:

```
| python manage.py migrate
```

Вы увидите вывод, который включает следующую строку:

```
| Applying shop.0001_initial... OK
```

Теперь база данных синхронизировалась с вашими моделями.

Регистрация моделей на сайте администратора

Давайте добавим наши модели на сайт администрирования, чтобы мы могли легко управлять категориями и продуктами. Откройте файл `admin.py` приложения `shop` и добавьте к нему следующий код:

```
from django.contrib import admin
from .models import Category, Product

@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug']
    prepopulated_fields = {'slug': ('name',)}

@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price',
                   'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'available']
    prepopulated_fields = {'slug': ('name',)}
```

Помните, что мы используем атрибут `prepopulated_fields` для указания полей, где значение автоматически устанавливается с использованием значения других полей. Как вы видели раньше, это удобно для создания `slug`. Мы используем атрибут `list_editable` в классе `ProductAdmin` для установки полей, которые можно редактировать со страницы отображения списка на сайте администрирования. Это позволит вам редактировать сразу несколько строк. Любое поле в `list_editable` также должно быть указано в атрибуте `list_display`, так как редактировать можно только отображаемые поля.

Теперь создайте суперпользователя для своего сайта, используя следующую команду:

```
python manage.py createsuperuser
```

Запустите сервер разработки с помощью команды `python manage.py runserver`. Откройте <http://127.0.0.1:8000/admin/shop/product/add/> в своем браузере и войдите в систему как пользователь, которого вы только что создали. Добавьте новую категорию и продукт, используя интерфейс администрирования. Страница списка продуктов на сайте администрирования будет выглядеть следующим образом:

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Shop > Products

 The product "Green tea" was added successfully.

Select product to change

[ADD PRODUCT !\[\]\(119cb08482036a0dad21aede908014a5_img.jpg\)](#)

Action: 0 of 1 selected

<input type="checkbox"/>	NAME	SLUG	PRICE	STOCK	AVAILABLE	CREATED	UPDATED
<input type="checkbox"/>	Green tea	green-tea	<input type="text" value="30"/>	<input type="text" value="22"/>	<input checked="" type="checkbox"/>	Dec. 5, 2017, 6:17 p.m.	Dec. 5, 2017, 6:17 p.m.

1 product

FILTER

By available

All

Yes

No

By created

Any date

Today

Past 7 days

This month

This year

By updated

Any date

Today

Past 7 days

This month

This year

Создание представлений каталога

Чтобы отобразить каталог продуктов, нам необходимо создать представление, которое будет перечислить все продукты или фильтровать их по данной категории. Откройте файл `views.py` приложения `shop` и добавьте к нему следующий код:

```
from django.shortcuts import render, get_object_or_404
from .models import Category, Product

def product_list(request, category_slug=None):
    category = None
    categories = Category.objects.all()
    products = Product.objects.filter(available=True)
    if category_slug:
        category = get_object_or_404(Category, slug=category_slug)
        products = products.filter(category=category)
    return render(request,
                  'shop/product/list.html',
                  {'category': category,
                   'categories': categories,
                   'products': products})
```

Мы будем фильтровать `QuerySet` с помощью `available=True` для извлечения только доступных продуктов. Мы используем необязательный параметр `category_slug` для необязательной фильтрации продуктов по данной категории.

Нам также нужно представление для извлечения и отображения одного продукта. Добавьте следующее 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})
```

В представлении `product_detail` ожидаются параметры `id` и `slug` для извлечения экземпляра `Product`. Мы можем получить этот экземпляр только через идентификатор, так как это уникальный атрибут. Тем не менее, мы включаем `slug` в URL-адресе для создания URL-адресов, ориентированных на дружественный SEO для продуктов.

После создания списка продуктов и детального представлений мы должны определить шаблоны URL для них. Создайте новый файл в каталоге приложений `shop` и назовите его `urls.py`. Добавьте к нему следующий 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 для нашего каталога продуктов. Мы определили два разных шаблона URL для представления `product_list`: шаблон с именем `product_list`, который вызывает представление `product_list` без каких-либо параметров; и шаблон с именем `product_list_by_category`, который предоставляет параметр `category_slug` для фильтрации продуктов в соответствии с данной категорией. Мы добавили шаблон для представления `product_detail`, который передает параметры `id` и `slug` в представление, чтобы получить конкретный продукт.

Измените файл `urls.py` проекта `myshop`, чтобы он выглядел следующим образом:

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

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

В основных шаблонах URL проекта мы будем включать URL-адреса для приложения `shop` в пользовательском пространстве имен с именем `shop`.

Теперь отредактируйте файл `models.py` приложения `shop`, импортируйте функцию `reverse()` и добавьте метод `get_absolute_url()` для моделей `Category` и `Product` следующим образом:

```
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 соглашение для извлечения URL-адреса для данного объекта. Здесь мы будем использовать шаблоны URL-адресов, которые мы только что определили в файле `urls.py`.

Создание шаблонов каталога

Теперь нам нужно создать шаблоны для списка продуктов и подробного (детального) представлений. Создайте следующие папки и файлы внутри каталога приложения `shop`:

```
templates/
    shop/
        base.html
        product/
            list.html
            detail.html
```

Нам нужно определить базовый шаблон, а затем расширить его в списке продуктов и детальном шаблонах. Измените шаблон `shop/base.html` и добавьте к нему следующий 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 мы будем использовать для нашего магазина. Чтобы включить стили и изображения CSS, которые используются шаблонами, вам необходимо скопировать статические файлы, которые сопровождают эту главу, расположенную в каталоге `static/` приложения `shop`. Скопируйте их в то же место в своем проекте.

Примечание переводчика: если у вас не отображаются правильные стили CSS то скорее всего Django нашел и использует другой файл с таким же именем `base.css` как и наш. Вы можете изменить путь к нашему файлу css поместив его в каталог с именем нашего приложения `shop`:

`static/css/shop/base.css`

Вам также надо будет изменить путь для загрузки **base.css** в файле **base.html** с:

`href="{% static "css/base.css" %}"`

на:

`href="{% static "css/shop/base.css" %}"`

Отредактируйте шаблон `shop/product/list.html` добавив в него следующий 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 %}
```

Это шаблон списка продуктов. Он расширяет шаблон `shop/base.html` и использует контекстную переменную `category` для отображения всех категорий на боковой панели и `products` для отображения продуктов текущей страницы. Этот же шаблон используется и для перечисление списка всех доступных продуктов и для продуктов, отфильтрованных по категории. Поскольку поле `image` модели `Product` может быть пустым, нам необходимо предоставить изображение по умолчанию для продуктов, у которых нет изображения. Изображение находится в нашем каталоге статических файлов с относительным путем `img/no_image.png`.

Поскольку мы используем `ImageField` для хранения изображений продуктов, нам нужен сервер разработки для облуживания загруженных файлов изображений.

Откройте файл `settings.py` приложения `myshop` и добавьте следующие настройки:

```
MEDIA_URL = '/media/'
```

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

`MEDIA_URL` это базовый URL-адрес, который служит для загрузки мультимедийных файлов пользователями. `MEDIA_ROOT` - это локальный путь, по которому находятся эти файлы, которые мы создаем, динамически добавляя переменную `BASE_DIR`.

Чтобы Django обслуживал загруженные файлы мультимедиа с помощью сервера разработки, отредактируйте основной файл `urls.py` приложения `myshop` и добавьте к нему следующий код:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ...
]
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                      document_root=settings.MEDIA_ROOT)
```

Помните, что мы обслуживаем статические файлы таким образом только во время разработки. В производственной среде вы никогда не должны обслуживать статические файлы с помощью Django.

Добавьте несколько продуктов в свой магазин, используя сайт администрирования, и откройте в своем браузере `http://127.0.0.1:8000/`. Вы увидите страницу списка продуктов, которая выглядит следующим образом:

My shop

Your cart is empty.

Categories

Products

All

Tea

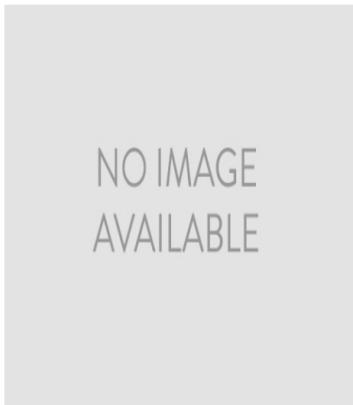


Green tea
\$30

Red tea
\$45.5

Tea powder
\$21.2

Если вы создаете продукт на сайте администрирования и не загружаете его изображение, вместо него будет отображаться изображение no_image.png по умолчанию:



NO IMAGE
AVAILABLE



Green tea
\$30



Red tea
\$45.5

Tea powder
\$21.2

Давайте отредактируем шаблон деталей продукта. Откройте шаблон `shop/product/detail.html` и добавьте к нему следующий 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 %}
```

Мы вызываем метод `get_absolute_url()` для связанного объекта категории для отображения доступных продуктов, относящихся к одной и той же категории. Теперь откройте `http://127.0.0.1:8000/` в своем браузере и нажмите на любой

продукт, чтобы увидеть страницу с подробными сведениями о продукте. Она будет выглядеть следующим образом:

My shop

Your cart is empty.



Red tea

Tea

\$45.5

Lore 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.

Теперь вы создали базовый каталог продуктов.

Создание корзины покупок

После создания каталога продуктов следующим шагом является создание корзины покупок, чтобы пользователи могли выбирать продукты, которые они хотят приобрести. Корзина позволяет пользователям выбирать товары и установить сумму, на которую они хотят сделать заказ, а затем временно хранить эту информацию, пока они просматривают сайт до тех пор, пока они в конечном итоге не сделают заказ. Корзина должна сохраняться в сессии, чтобы элементы корзины сохранялись на время посещения сайта пользователем.

Мы будем использовать фреймворк сессий Django для сохранения корзины. Корзина будет храниться в сессии до тех пор, пока сессия не закончится, или пользователь не выйдет из корзины. Нам также нужно будет создать дополнительные модели Django для корзины и ее товаров.

Использование сессий в Django

Django предоставляет фреймворк сессий, который поддерживает анонимные и пользовательские сессии. Фреймворк сессий позволяет хранить произвольные данные для каждого посетителя. Данные сессий хранятся на стороне сервера, а файлы cookie содержат идентификатор сессии, если вы не используете механизм сессий на основе файлов cookie. Промежуточное программное обеспечение сессий управляет отправкой и получением файлов cookie. Механизм сессий по умолчанию хранит данные сессий в базе данных, но вы можете выбирать между различными механизмами сессий.

Чтобы использовать сессии, вы должны убедиться, что параметр `MIDDLEWARE` вашего проекта содержит '`'django.contrib.sessions.middleware.SessionMiddleware'`'. Это промежуточное программное обеспечение управляет сессиями. Оно добавляется по умолчанию к настройке `MIDDLEWARE` при создании нового проекта с помощью команды `startproject`.

Промежуточное программное обеспечение сессий делает текущую сессию доступной в объекте `request`. Вы можете получить доступ к текущей сессии с помощью `request.session`, рассматривая его как словарь Python для хранения и получения данных сеанса. Словарь сессий принимает по умолчанию любой объект Python, который может быть сериализован в JSON. Вы можете установить переменную в сессии следующим образом:

```
request.session['foo'] = 'bar'
```

Получить ключ сессии следующим образом:

```
request.session.get('foo')
```

Удалить ключ, который вы ранее сохранили в сессии, следующим образом:

```
del request.session['foo']
```

Вы можете рассматривать `request.session` как стандартный словарь Python.

Когда пользователи регистрируются на сайте, их анонимная сессия теряется и создается новая сессия для аутентифицированных пользователей. Если вы храните элементы в анонимной сессии, которые необходимо сохранить после входа пользователя в систему, вам придется скопировать старые данные сессии в новую сессию.

Настройки сессий

Существует несколько настроек, которые вы можете использовать для настройки сессий в своем проекте. Наиболее важной является `SESSION_ENGINE`. Этот параметр позволяет указать место хранения сессий. По умолчанию Django хранит сессии в базе данных с помощью модели `Session` приложения `django.contrib.sessions`.

Django предлагает следующие варианты хранения данных сессий:

- **База данных сессий:** Данные сессий хранятся в базе данных. Это механизм сессий по умолчанию.
- **Базовые файлы сессий:** Данные сессий хранятся в файловой системе.
- **Кеширование сессий:** Данные сессий хранятся в кеш-памяти. Вы можете указать кеш-серверы, используя параметр `CACHES`. Хранение данных сессий в кеш-системе обеспечивает лучшую производительность.
- **Кеширование базы данных сессий:** Данные сессий хранятся в кэше и базе данных. База данных читается только если данные еще не находятся в кеше.
- **Сессии на основе Cookie:** Данные сессий хранятся в файлах cookie, которые отправляются в браузер.

Для повышения производительности используйте механизм сессий на

основе кеша. Django поддерживает Memcached из коробки, и вы можете найти сторонние кэш-серверы для Redis и других систем кэширования.

Вы можете создать сессии с определенными настройками. Вот некоторые из важных настроек, связанных с сессиями:

- `SESSION_COOKIE_AGE`: Продолжительность сессионных файлов cookie в секундах. Значение по умолчанию: `1209600` (две недели).
- `SESSION_COOKIE_DOMAIN`: Домен, используемый для файлов cookie сессии. Установите для этого `mydomain.com`, чтобы включить файлы cookie для нескольких доменов или используйте `None` для стандартного cookie домена.
- `SESSION_COOKIE_SECURE`: Логическое значение, указывающее, что cookie следует отправлять только в том случае, если соединение является HTTPS-соединением.
- `SESSION_EXPIRE_AT_BROWSER_CLOSE`: Логическое значение, указывающее, что сессия должна заканчиваться, когда браузер закрыт.
- `SESSION_SAVE_EVERY_REQUEST`: Логическое значение, которое, если `True`, сохранит сессию в базе данных для каждого запроса.

Вы можете просмотреть все настройки сессий и их значения по умолчанию <https://docs.djangoproject.com/en/2.0/ref/settings/#sessions>.

Завершение сессий

Вы можете выбрать использовать сессии продолжительностью работы браузера или постоянные сессии, используя параметр `SESSION_EXPIRE_AT_BROWSER_CLOSE`. По умолчанию установлено значение `False`, устанавливая продолжительность сессий равным значению, хранящемуся в настройке `SESSION_COOKIE_AGE`. Если вы установите `SESSION_EXPIRE_AT_BROWSER_CLOSE` в `True`, сессия истечет, когда пользователь закроет браузер, а параметр `SESSION_COOKIE_AGE` не будет иметь никакого эффекта.

Вы можете использовать метод `set_expiry()` из `request.session`, чтобы перезаписать продолжительность текущей сессии.

Хранение корзины покупок в сессиях

Нам нужно создать простую структуру, которая может быть сериализована в JSON чтобы хранить элементы корзины в сессии. Корзина должна включать следующие данные для каждого содержащегося в ней элемента:

- ID экземпляра товара
- Количество выбранного товара
- Цена за единицу товара

Поскольку цены на продукцию могут отличаться, мы сохраняем цену продукта вместе с самим продуктом, когда он добавляется в корзину. Поступая таким образом, мы используем текущую цену продукта, когда пользователи добавляют его в свою корзину, независимо от того, будет ли впоследствии изменена цена продукта.

Теперь вам нужно создать функциональность для корзины и связать ее с сессией. Корзина должна работать следующим образом:

- Когда нужна корзина, мы проверяем, установлен ли пользовательский ключ сессии. Если в сессии нет корзины, мы создаем новую корзину и сохраняем ее ключ в сессии корзины.

- Для последующих запросов мы выполняем ту же проверку и получаем элементы корзины из ключа сессии корзины. Мы извлекаем элементы корзины из сессии и связанные с ними объекты `Product` из базы данных.

Измените файл `settings.py` вашего проекта и добавьте в него следующие настройки:

```
CART_SESSION_ID = 'cart'
```

Это ключ, который мы собираемся использовать для хранения корзины в пользовательской сессии. Поскольку сессии Django устанавливаются для каждого посетителя, мы можем использовать один и тот же ключ сессии корзины для всех сессий.

Давайте создадим приложение для управления сессиями. Откройте терминал и создайте новое приложение, выполнив следующую команду из каталога проекта:

```
python manage.py startapp cart
```

Затем отредактируйте файл `settings.py` вашего проекта и добавьте новое приложение в настройку `INSTALLED_APPS` следующим образом:

```
INSTALLED_APPS = [  
    # ...  
    'shop.apps.ShopConfig',  
    'cart.apps.CartConfig',  
]
```

Создайте новый файл в каталоге приложения `cart` и назовите его `cart.py`. Добавьте к нему следующий код:

```
from decimal import Decimal
from django.conf import settings
from shop.models import Product

class Cart(object):

    def __init__(self, request):
        """
        Initialize the cart.
        """
        self.session = request.session
        cart = self.session.get(settings.CART_SESSION_ID)
        if not cart:
            # сохранить пустую корзину в сессии
            cart = self.session[settings.CART_SESSION_ID] = {}
        self.cart = cart
```

Это класс `Cart`, который позволит нам управлять корзиной покупок. Мы указываем, чтобы корзина была инициализирована с помощью объекта `request`. Мы сохраняем текущий сеанс, используя `self.session=request.session`, чтобы сделать его доступным для других методов класса `cart`. Сначала мы пытаемся получить корзину из текущего сеанса, используя `self.session = request.session`. Если в сеансе нет корзины, мы создаем пустую корзину, устанавливая пустой словарь в сеансе. Мы ожидаем, что наш словарь в корзине будет использовать идентификаторы продуктов в качестве ключей и словарь с количеством и ценой в качестве значения для каждого ключа. Поступая таким образом, мы можем гарантировать, что продукт не добавляется более одного раза в корзину; таким образом, мы также упрощаем способ извлечения элементов корзины.

Давайте создадим метод добавления продуктов в корзину или обновления их количества. Добавьте следующие методы `add()` и `save()` в класс `Cart`:

```
class Cart(object):
    # ...
    def add(self, product, quantity=1, update_quantity=False):
```

```

"""
Add a product to the cart or update its quantity.
"""

product_id = str(product.id)
if product_id not in self.cart:
    self.cart[product_id] = {'quantity': 0,
                           'price': str(product.price)}
if update_quantity:
    self.cart[product_id]['quantity'] = quantity
else:
    self.cart[product_id]['quantity'] += quantity
self.save()

def save(self):
    # отмечаем сессию как "измененную" ("modified"), чтобы сохранить её
    self.session.modified = True

```

Метод `add()` принимает в качестве входных данных следующие параметры:

- `product`: Экземпляр `product` для добавления или обновления в корзине.
- `quantity`: Необязательное целое число с количеством продукта. По умолчанию это 1.
- `update_quantity`: Это логическое значение, указывающее, нужно ли обновлять количество до заданной величины (`True`), или нужно добавить новое количество к существующему количеству (`False`).

Мы используем идентификатор продукта в качестве ключа в словаре контента корзины. Мы преобразуем идентификатор продукта в строку, потому что Django использует JSON в сериализации данных сеанса, а JSON допускает только имена строковых ключей. Идентификатор продукта - это ключ, а значение, которое мы сохраняем, - это словарь с количественными и ценовыми показателями для продукта. Цена продукта конвертируется из десятичной строки в строку

для ее сериализации. Наконец, мы вызываем метод `save()`, чтобы сохранить корзину в сессии.

Метод `save()` отмечает сессию как измененную с помощью `session.modified = True`. Это говорит Django, что сессия изменилась и должна быть сохранен.

Нам также нужен способ удаления продуктов из корзины. Добавьте следующий класс в `cart`:

```
class Cart(object):
    # ...
    def remove(self, product):
        """
        Remove a product from the cart.
        """
        product_id = str(product.id)
        if product_id in self.cart:
            del self.cart[product_id]
            self.save()
```

Метод `remove()` удаляет данный продукт из словаря корзины и вызывает метод `save()` для обновления корзины в сессии.

Нам придется перебирать элементы, содержащиеся в корзине, и обращаться к соответствующим экземплярам `Product`. Для этого вы можете определить метод `__iter__()` в своем классе. Добавьте следующий метод в класс `Cart`:

```
class Cart(object):
    # ...
    def __iter__(self):
        """
        Iterate over the items in the cart and get the products
        from the database.
        """
        product_ids = self.cart.keys()
        # получить объект продукта и добавить его 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__()` мы извлекаем экземпляры `Product`, которые присутствуют в корзине, чтобы включить их в элементы корзины. Мы скопируем текущую корзину в переменную `cart` и добавим к ней экземпляры `Product`. Наконец, мы перебираем элементы корзины, преобразуя стоимость товара обратно в десятичную и добавляем атрибут `total_price` для каждого элемента. Теперь мы можем легко перебирать предметы в корзине.

Нам также нужен способ вернуть количество предметов в корзине. Когда функция `len()` выполняется для объекта, Python вызывает метод `__len__()` для получения его длины. Мы собираемся определить собственный метод `__len__()`, чтобы вернуть общее количество элементов, хранящихся в корзине. Добавьте следующий метод `__len__()` в класс `Cart`:

```
class Cart(object):
    # ...
    def __len__(self):
        """
        Count all items in the cart.
        """
        return sum(item['quantity'] for item in self.cart.values())
```

Мы возвращаем сумму всех элементов корзины.

Добавьте следующий метод для расчета общей стоимости предметов в корзине:

```
class Cart(object):
    # ...
    def get_total_price(self):
```

```
    return sum(Decimal(item['price']) * item['quantity'] for item in
self.cart.values())
```

И, наконец, добавьте метод очистки сеанса корзины:

```
class Cart(object):
    # ...
    def clear(self):
        # удалить корзину из сессии
        del self.session[settings.CART_SESSION_ID]
        self.save()
```

Наш класс `Cart` теперь готов управлять корзиной.

Создание представления для корзины покупок

Теперь, когда у нас есть класс `cart` для управления корзиной, нам нужно создать представления для добавления, обновления или удаления элементов из нее. Нам нужно создать следующие представления:

- Представление для добавления или обновления элементов в корзине, которое может обрабатывать текущее и новое количество
- Представление для удаления товаров из корзины
- Представление для отображения позиций и общего состояния корзины

Добавление товаров в корзину

Чтобы добавить элементы в корзину, нам нужна форма, которая позволяет пользователю выбирать количество. Создайте файл `forms.py` в каталоге приложения `cart` и добавьте в него следующий 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)
    update = forms.BooleanField(required=False,
                                initial=False,
                                widget=forms.HiddenInput)
```

Мы будем использовать эту форму для добавления продуктов в корзину. Наш класс `CartAddProductForm` содержит следующие два поля:

- `quantity`: Позволяет пользователю выбирать количество между 1-20. Мы используем поле `TypedChoiceField` с `coerce=int` для преобразования ввода в целое число.
- `update`: Позволяет указать, должно ли количество быть добавлено к существующему количеству в корзине для этого продукта (`false`), или нужно обновлять существующее количество с заданной величиной (`true`).

Мы используем виджет `HiddenInput` для этого поля, так как мы не хотим отображать его для пользователя.

Давайте создадим представление для добавления элементов в корзину. Откройте файл `views.py` приложения `cart` и добавьте к нему следующий код:

```
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,
                  quantity=cd['quantity'],
                  update_quantity=cd['update'])
    return redirect('cart:cart_detail')
```

Это представление для добавления продуктов в корзину или обновления количества для существующих продуктов. Мы используем декоратор `require_POST`, чтобы разрешать только `POST` запросы, так как это представление будет изменять данные. Представление получает идентификатор продукта в качестве параметра. Мы извлекаем экземпляр `Product` с данным идентификатором и проверяем `CartAddProductForm`. Если форма действительна, мы либо добавляем, либо обновляем продукт в корзине. Представление перенаправляется на URL `cart_detail`, который отображает содержимое корзины. Мы вскоре создадим представление `cart_detail`.

Нам также нужно представление для удаления продуктов из корзины. Добавьте следующий код в файл `views.py` приложения

cart:

```
def cart_remove(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    cart.remove(product)
    return redirect('cart:cart_detail')
```

Представление `cart_remove` получает идентификатор продукта в качестве параметра. Мы извлекаем экземпляр `Product` с данным идентификатором и удаляем продукт из корзины. Затем мы перенаправляем пользователя на URL `cart_detail`.

Примечание переводчика: если вы удалите из корзины последний товар то получите сообщение об ошибке. Чтобы избежать этого измените код представления следующим образом:

```
def cart_remove(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    cart.remove(product)
    if cart:
        return redirect('cart:cart_detail')
    return redirect('/')
```

Наконец, нам нужно представление, чтобы отобразить корзину и ее элементы. Добавьте следующее 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 отобразить ее.

Мы создали представления для добавления элементов в

корзину, обновления количества, удаления элементов из корзины и отображения содержимого корзины. Давайте добавим паттерны URL для этих представлений. Создайте новый файл в каталоге приложения `cart` и назовите его `urls.py`. Добавьте к нему следующие URL:

```
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'),
]
```

Измените основной файл `urls.py` проекта `myshop` и добавьте следующий паттерн URL, чтобы включить URL-адреса корзины:

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

Убедитесь, что вы включили этот паттерн URL перед паттерном `shop.urls`, поскольку он более конкретный.

Создание шаблона для отображения корзины

Представления `cart_add` и `cart_remove` не отображают никаких шаблонов, но нам нужно создать шаблон для представления `cart_detail` для отображения элементов и окончательного состояния корзины.

Создайте следующую структуру файлов в каталоге приложения `cart`:

```
templates/
    cart/
        detail.html
```

Откройте шаблон `cart/detail.html` и добавьте к нему следующий 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><a href="{% url "cart:cart_remove" product.id %}">Remove</a></td>
                    <td class="num">${{ item.price }}</td>
                    <td class="num">${{ item.total_price }}</td>
                </tr>
            {% endwith %}
            {% endfor %}
        <tr class="total">
            <td>Total</td>
            <td colspan="4">
                <td 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 %}

```

Это шаблон, который используется для отображения содержимого корзины. Он содержит таблицу с элементами, хранящимися в текущей корзине. Мы разрешаем пользователям изменять количество выбранных продуктов, используя форму, отправленную в представлении `cart_add`. Мы также разрешаем пользователям удалять элементы из корзины, предоставляя ссылку `Remove` для каждого из них.

Добавление продуктов в корзину

Теперь нам нужно добавить кнопку Add to cart на страницу сведений о продукте. Измените файл `views.py` приложения `shop` и добавьте `CartAddProductForm` в представление `product_detail` следующим образом:

```
from cart.forms import CartAddProductForm

def product_detail(request, id, slug):
    product = get_object_or_404(Product, id=id,
                                slug=slug,
                                available=True)
    cart_product_form = CartAddProductForm()
    return render(request,
                  'shop/product/detail.html',
                  {'product': product,
                   'cart_product_form': cart_product_form})
```

Отредактируйте шаблон `shop/product/detail.html` приложения `shop` и добавьте следующую форму к цене продукта:

```
<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 }}
```

Запустите сервер разработки командой `python manage.py runserver`. Теперь откройте `http://127.0.0.1:8000/` в браузере и перейдите к странице подробностей о продукте. Теперь она содержит форму для выбора количества перед добавлением продукта в

корзину. Страница будет выглядеть так:

My shop

Your cart is empty.



Red tea
Tea
\$45.5

Quantity: Add to cart

Выберите количество и нажмите кнопку Add to cart. Форма отправляется в представление `cart_add` через `POST`. Представление добавляет продукт в корзину сессии, включая текущую цену и выбранное количество. Затем оно перенаправляет пользователя на страницу подробной информации о корзине, которая будет выглядеть следующим образом:

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Red tea	2	Remove	\$45.5	\$91.0
Total					\$91.0

[Continue shopping](#)[Checkout](#)

Обновление количества товара в корзине

Когда пользователи видят корзину, они могут захотеть изменить количество товара, прежде чем оформить заказ. Мы разрешаем пользователям изменять количество на странице подробной информации о корзине.

Откройте файл `views.py` приложения `cart` и измените представление `cart_detail` следующим образом:

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
            initial={'quantity': item['quantity'],
                      'update': True})
    return render(request, 'cart/detail.html', {'cart': cart})
```

Мы создаем экземпляр `CartAddProductForm` для каждого элемента в корзине, чтобы разрешить изменять количество товаров. Мы инициализируем форму с текущим количеством элементов и устанавливаем поле `update` в `True`, чтобы при отправке формы в представление `cart_add`, текущее количество заменялось новым.

Теперь отредактируйте шаблон `cart/detail.html` приложения `cart` и найдите строку:

```
<td>{{ item.quantity }}</td>
```

Замените её следующим кодом:

```
<td>
  <form action="{% url "cart:cart_add" product.id %}" method="post">
    {{ item.update_quantity_form.quantity }}
    {{ item.update_quantity_form.update }}
    <input type="submit" value="Update">
    {% csrf_token %}
  </form>
</td>
```

Откройте <http://127.0.0.1:8000/cart/> в своем браузере. Вы увидите форму для редактирования количества для каждого элемента корзины, показанную ниже:

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Red tea	<input style="width: 20px; height: 20px; border: 1px solid #ccc; padding: 2px; margin-right: 5px;" type="button" value="2"/> <input style="background-color: #0072BD; color: white; border: none; padding: 2px 10px; border-radius: 5px; font-weight: bold;" type="button" value="Update"/>	Remove	\$45.5	\$91.0
Total					\$91.0

[Continue shopping](#)

[Checkout](#)

Измените количество элементов и нажмите кнопку Update чтобы проверить новые функции. Вы также можете удалить элемент из корзины, нажав ссылку Remove.

Создание контекстного процессора для текущей корзины

Возможно, вы заметили, что сообщение Your cart is empty отображается в заголовке сайта, даже если корзина содержит элементы. Мы должны отображать общее количество элементов в корзине и общую стоимость. Поскольку это должно отображаться на всех страницах, мы создадим контекстный процессор для включения текущей корзины в контексте запроса, независимо от представления, обрабатывающего запрос.

Контекстные процессоры

Контекстный процессор представляет собой функцию Python, которая принимает объект `request` в качестве аргумента и возвращает словарь, который добавляется в контекст запроса. Они необходимы, когда вам нужно сделать что-то доступное глобально для всех шаблонов.

По умолчанию при создании нового проекта с помощью команды `startproject` ваш проект содержит следующие шаблонные контекстные процессоры в опции `context_processors` внутри `TEMPLATES` настроек проекта:

- `django.template.context_processors.debug`: Задает переменные `boolean debug` И `sql_queries` В контекст, представляющем список SQL-запросов, выполненных в запросе.
- `django.template.context_processors.request`: Устанавливает переменную `request` В контекст.
- `django.contrib.auth.context_processors.auth`: Задает переменную `user` В запросе.
- `django.contrib.messages.context_processors.messages`: Устанавливает переменную `messages` В контекст, содержащую все сообщения, которые были сгенерированы с использованием фреймворка сообщений.

Django также позволяет `django.template.context_processors.csrf` избегать атак с подделкой запросов на межсайтовый запрос. Этот контекстный процессор отсутствует в настройках, но он всегда

включен и не может быть отключен по соображениям безопасности.

Вы можете увидеть список всех встроенных контекстных процессоров в <https://docs.djangoproject.com/en/2.0/ref/templates/api/#built-in-template-context-processors>.

Настройка контекста запроса для корзины

Давайте создадим контекстный процессор, чтобы установить текущую корзину в контекст запроса. Мы сможем получить доступ к корзине из любого шаблона.

Создайте новый файл в каталоге приложения `cart` и назовите его `context_processors.py`. Контекстные процессоры могут находиться в любом месте вашего кода, но их создание в данном месте будет способствовать хошой организации коды. Добавьте в файл следующий код:

```
from .cart import Cart

def cart(request):
    return {'cart': Cart(request)}
```

Контекстный процессор - это функция, которая получает объект `request` в качестве параметра и возвращает словарь объектов, который будет доступен для всех шаблонов, созданных с помощью `RequestContext`. В нашем контекстном процессоре мы создаем корзину с использованием объекта `request` и делаем ее доступной для шаблонов как переменную с именем `cart`.

Измените файл `settings.py` вашего проекта и добавьте `cart.context_processors.cart` в параметр `context_processors` внутри `TEMPLATES` следующим образом:

```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
```

```
'DIRS': [],
'APP_DIRS': True,
'OPTIONS': {
    'context_processors': [
        # ...
        'cart.context_processors.cart',
    ],
},
],
]
```

Контекстный процессор `cart` будет выполняться каждый раз, когда шаблон визуализируется с использованием `RequestContext` Django. Переменная `cart` будет установлена в контексте ваших шаблонов.

Контекстные процессоры выполняются во всех запросах, которые используют `RequestContext`. Возможно, вам понадобится создать собственный тег шаблона вместо обработчика контекста, если ваши функции не нужны во всех шаблонах, особенно если они связаны с запросами базы данных.

Теперь отредактируйте шаблон `shop/base.html` приложения `shop` и найдите следующие строки:

```
<div class="cart">
    Your cart is empty.
</div>
```

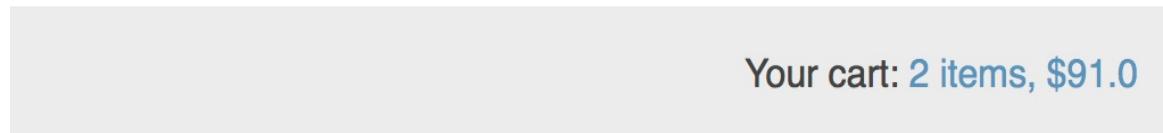
Замените их следующим кодом:

```
<div class="cart">
    {% with total_items=cart|length %}
        {% if cart|length > 0 %}
            Your cart:
            <a href="{% url "cart:cart_detail" %}">
                {{ total_items }} item{{ total_items|pluralize }},
                ${{ cart.get_total_price }}
            </a>
        {% else %}
            Your cart is empty.
        {% endif %}
    {% endwith %}
</div>
```

Перезагрузите ваш сервер используя команду `python manage.py runserver`. Откройте `http://127.0.0.1:8000/` в вашем браузере и добавьте несколько товаров в корзину.

В заголовке веб-сайта вы можете увидеть общее количество элементов в корзине и общую стоимость, а именно:

My shop



Your cart: 2 items, \$91.0

Регистрация заказов клиентов

Когда зарегистрирована корзина для покупок, вам необходимо сохранить заказ в базе данных. Заказы будут содержать информацию о клиентах и продуктах, которые они покупают.

Создайте новое приложение для управления заказами клиентов, используя следующую команду:

```
python manage.py startapp orders
```

Измените файл `settings.py` вашего проекта и добавьте новое приложение в настройку `INSTALLED_APPS` следующим образом:

```
INSTALLED_APPS = [  
    # ...  
    'orders.apps.OrdersConfig',  
]
```

Вы активировали приложение `orders`.

Создание моделей заказов

Вам понадобится одна модель для хранения деталей заказа и вторая модель для хранения купленных предметов, включая их цену и количество. Откройте файл `models.py` приложения `orders` и добавьте к нему следующий 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',)

    def __str__(self):
        return 'Order {}'.format(self.id)

    def get_total_cost(self):
        return sum(item.get_cost() for item in self.items.all())


class OrderItem(models.Model):
    order = models.ForeignKey(Order,
                             related_name='items',
                             on_delete=models.CASCADE)
    product = models.ForeignKey(Product,
                               related_name='order_items',
                               on_delete=models.CASCADE)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    quantity = models.PositiveIntegerField(default=1)
```

```
def __str__(self):
    return '{}'.format(self.id)

def get_cost(self):
    return self.price * self.quantity
```

Модель `Order` содержит несколько полей для хранения информации о клиентах и логическое поле `paid`, которое по умолчанию равно `False`. Позже мы будем использовать это поле для разграничения оплаченных и неоплаченных заказов. Мы также определяем метод `get_total_cost()`, чтобы получить общую стоимость предметов, купленных в этом заказе.

Модель `OrderItem` позволяет нам хранить продукт, количество и цену, заплаченную за каждую позицию товара. Мы включаем `get_cost()`, чтобы вернуть стоимость каждой позиции.

Выполните следующую команду для создания начальных миграций для приложения `orders`:

```
python manage.py makemigrations
```

Вы увидите следующий результат:

```
Migrations for 'orders':
  orders/migrations/0001_initial.py
    - Create model Order
    - Create model OrderItem
```

Для применения новой миграции выполните следующую команду:

```
python manage.py migrate
```

Теперь ваши модели заказов синхронизировались с базой данных.

Добавление модели заказов на сайт администрирования

Давайте добавим модели заказов на сайт администрирования. Измените файл `admin.py` приложения `orders`, чтобы он выглядел следующим образом:

```
from django.contrib import admin
from .models import Order, OrderItem

class OrderItemInline(admin.TabularInline):
    model = OrderItem
    raw_id_fields = ['product']

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                   'address', 'postal_code', 'city', 'paid',
                   'created', 'updated']
    list_filter = ['paid', 'created', 'updated']
    inlines = [OrderItemInline]
```

Мы используем класс `ModelInline` для модели `OrderItem`, чтобы включить его как *inline* в класс `OrderAdmin`. Встроенная функция позволяет включить модель на той же странице редактирования в связанную с ней модель.

Запустите сервер разработки с помощью команды `python manage.py runserver`, а затем откройте `http://127.0.0.1:8000/admin/orders/order/add/` в своем браузере. Вы увидите следующую страницу:

Add order

First name:	<input type="text"/>
Last name:	<input type="text"/>
Email:	<input type="text"/>
Address:	<input type="text"/>
Postal code:	<input type="text"/>
City:	<input type="text"/>
<input type="checkbox"/> Paid	

ORDER ITEMS			
PRODUCT	PRICE	QUANTITY	DELETE?
<input type="text"/>	Q	<input type="text"/>	<input type="text"/> 1
<input type="text"/>	Q	<input type="text"/>	<input type="text"/> 1
<input type="text"/>	Q	<input type="text"/>	<input type="text"/> 1
+ Add another Order item			

<input type="button" value="Save and add another"/>	<input type="button" value="Save and continue editing"/>	<input type="button" value="SAVE"/>
---	--	-------------------------------------

Создание заказов клиентов

Мы будем использовать модели заказов, которые мы создали, чтобы сохранить позиции, содержащиеся в корзине покупок, когда пользователь, наконец, оформит заказ. Новый заказ будет создан после следующих шагов:

1. Предоставьте пользователям форму заказа для внесения своих данных.
2. Создайте новый экземпляр `order` с введенными данными и создайте связанный экземпляр `orderItem` для каждой позиции в корзине
3. Очистите все содержимое корзины и перенаправте пользователей на страницу успешного завершения

Во-первых, нам нужна форма для ввода деталей заказа. Создайте новый файл в каталоге приложения `orders` и назовите его `forms.py`. Добавьте к нему следующий 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, которую мы собираемся использовать для создания новых объектов `order`. Теперь нам нужно представление для обработки формы и создания нового заказа. Откройте файл `views.py` приложения `orders` и добавьте к нему следующий код:

```
from django.shortcuts import render
from .models import OrderItem
from .forms import OrderCreateForm
from cart.cart import Cart

def order_create(request):
    cart = Cart(request)
    if request.method == 'POST':
        form = OrderCreateForm(request.POST)
        if form.is_valid():
            order = form.save()
            for item in cart:
                OrderItem.objects.create(order=order,
                                         product=item['product'],
                                         price=item['price'],
                                         quantity=item['quantity'])
            # 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` мы получим текущую корзину из сессии с помощью `cart = Cart(request)`. В зависимости от метода запроса мы выполним следующие задачи:

- **GET запрос:** Создает форму `OrderCreateForm` и отображает шаблон `orders/order/create.html`.
- **POST запрос:** Проверяет данные, отправленные в запросе. Если данные действительны, мы создаем новый заказ в базе данных, используя `order=form.save()`. Мы перебираем элементы корзины и создаем `OrderItem` для каждого из них. Наконец, мы очищаем содержимое корзины и вызываем шаблон `orders/order/created.html`.

Создайте новый файл в каталоге приложения `orders` и назовите его `urls.py`. Добавьте к нему следующий код:

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

app_name = 'orders'

urlpatterns = [
    path('create/', views.order_create, name='order_create'),
]
```

Это паттерн URL-адреса для представления `order_create`. Отредактируйте файл `urls.py` `myshop` и включите следующий паттерн. Не забудьте разместить его перед паттерном `shop.urls`:

```
path('orders/', include('orders.urls', namespace='orders')),
```

Откройте шаблон `cart/detail.html` приложения `cart` и найдите следующие строки:

```
<a href="#" class="button">Checkout</a>
```

Добавте `order_create` URL следующим образом:

```
<a href="{% url "orders:order_create" %}" class="button">
    Checkout
</a>
```

Теперь пользователи могут перемещаться со страницы подробной информации о корзине в форму заказа. Нам еще нужно определить шаблоны для размещения заказов. Создайте следующую структуру файлов внутри каталога приложений `orders`:

```
templates/
    orders/
```

```
order/
  create.html
  created.html
```

Откройте шаблон `orders/order/create.html` и добавьте в него следующий код:

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

{% block title %}
    Checkout
{% endblock %}

{% block content %}
    <h1>Checkout</h1>

    <div class="order-info">
        <h3>Your order</h3>
        <ul>
            {% for item in cart %}
                <li>
                    {{ item.quantity }}x {{ item.product.name }}
                    <span>${{ item.total_price }}</span>
                </li>
            {% endfor %}
        </ul>
        <p>Total: ${{ cart.get_total_price }}</p>
    </div>

    <form action"." method="post" class="order-form">
        {{ form.as_p }}
        <p><input type="submit" value="Place order"></p>
        {% csrf_token %}
    </form>
{% endblock %}
```

Этот шаблон отображает элементы корзины, включая итоговые значения, и форму для размещения заказа.

Отредактируйте шаблон `orders/order/created.html` добавив в него следующий 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 мы отображаем при успешном создании заказа.

Запустите сервер веб-разработки для отслеживания новых файлов. Откройте <http://127.0.0.1:8000/> в своем браузере, добавьте пару продуктов в корзину и перейдите на страницу проверки. Вы увидите страницу, похожую на следующую:

My shop

Your cart: 3 items, \$112.2

Checkout

First name:

Last name:

Email:

Address:

Postal code:

City:

Place order

Your order

- 1x Tea powder \$21.2
- 2x Red tea \$91.0

Total: \$112.2

Заполните форму действительными данными и нажмите кнопку Place order. Заказ будет создан, и вы увидите страницу успешного размещения заказа, например:

Thank you

Your order has been successfully completed. Your order number is **1**.

Теперь перейдите на сайт администрирования и убедитесь что заказ создан.

Запуск асинхронных задач с Celery

Все, что вы выполняете в представлении, влияет на время отклика. Во многих ситуациях вам нужно как можно быстрее вернуть ответ пользователю и позволить асинхронно выполнятся какому-либо процессу. Это особенно актуально для трудоемких процессов или процессов, закончившихся сбоем, для которых может потребоваться политика повторных попыток. Например, платформа для обмена видео позволяет пользователям загружать видео, но для перекодирования загруженных видео требуется много времени. Сайт может вернуть ответ пользователям, чтобы сообщить им, что скоро будет транскодирование, и начнет транскодирование видео асинхронно. Другой пример - отправка электронной почты пользователям. Если ваш сайт отправляет уведомления по электронной почте из представления, подключение SMTP может привести к сбою или замедлению ответа. Запуск асинхронных задач необходим, чтобы избежать блокировки выполнения кода.

Celery - это распределенная очередь задач, которая может обрабатывать огромное количество сообщений. Он выполняет обработку в реальном времени, но также поддерживает планирование задач. Используя Celery, вы можете не только легко создавать асинхронные задачи и выполнять их как можно скорее, но вы также можете планировать их запуск на определенное время.

Вы можете найти документацию по Celery на <http://docs.celeryproject.org/en/latest/index.html>.

Установка Celery

Давайте установим Celery и интегрируем его в наш проект.

Установите Celery через pip, используя следующую команду:

```
pip install celery==4.1.0
```

Celery требует брокера сообщений для обработки запросов от внешнего источника. Брокер заботится о передаче сообщений обработчикам Celery, которые обрабатывают задачи по мере их получения. Давайте установим брокер сообщений.

Примечание переводчика: Вместе с Celery устанавливаются другие программы. Если у вас возникнут проблемы с работой Celery то возможно их вызвала последняя версия программы kombu. Необходимо выполнить установку kombu стабильной версии 4.1.0

```
pip3 install kombu==4.1.0
```

Установка RabbitMQ

Существует несколько вариантов в качестве брокера сообщений для Celery, включая хранилище ключей/значений, например Redis, или настоящую систему сообщений, такую как RabbitMQ. Мы настроим Celery с RabbitMQ, так как это рекомендуемый обработчик сообщение для Celery.

Если вы используете Linux, вы можете установить RabbitMQ из консоли, используя следующую команду:

```
apt-get install rabbitmq
```

Если вам нужно установить RabbitMQ на macOS X или Windows, вы можете найти необходимую версию на <https://www.rabbitmq.com/download.html>.

После его установки запустите RabbitMQ, используя следующую команду из консоли:

```
rabbitmq-server
```

Вы увидите вывод, который заканчивается следующей строкой:

```
Starting broker... completed with 10 plugins.
```

RabbitMQ работает и готов принимать сообщения.

Добавление Celery в ваш проект

Вы должны добавить конфигурацию экземпляра Celery.

Создайте новый файл рядом с файлом `settings.py` приложения `myshop` и назовите его `celery.py`. Этот файл будет содержать конфигурацию Celery для вашего проекта. Добавьте к нему следующий код:

```
import os
from celery import Celery

# set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myshop.settings')

app = Celery('myshop')

app.config_from_object('django.conf:settings', namespace='CELERY')
app.autodiscover_tasks()
```

В этом коде мы делаем следующее:

1. Мы устанавливаем переменную `DJANGO_SETTINGS_MODULE` для программы командной строки Celery.
2. Мы создаем экземпляр приложения с помощью `app = Celery('myshop')`.
3. Мы загружаем любую настраиваемую конфигурацию из наших параметров проекта с помощью метода `config_from_object()`. Атрибут `namespace` указывает префикс, связанный с параметрами Celery, в нашем файле `settings.py`. Установив пространство имен `CELERY`, все

настройки Celery должны включать префикс `CELERY_` в своем имени (например, `CELERY_BROKER_URL`).

4. Наконец, мы сообщаем Celery автоматически обнаруживать асинхронные задачи для наших приложений. Celery будет искать файл `tasks.py` в каждом каталоге приложений, добавленных в `INSTALLED_APPS`, чтобы загрузить асинхронные задачи, определенные в нем.

Вам нужно импортировать модуль `celery` в файл `__init__.py` вашего проекта, чтобы убедиться, что он загружен при запуске Django. Отредактируйте файл `myshop/__init__.py` и добавьте к нему следующий код:

```
# import celery
from .celery import app as celery_app
```

Теперь вы можете начать программирование асинхронных задач для своих приложений.

Параметр `CELERY_ALWAYS_EAGER` позволяет выполнять задачи локально синхронно, а не отправлять их в очередь. Это полезно для запуска модульных тестов или выполнения приложения в локальной среде без использования Celery.

Добавление асинхронных задач в приложение

Мы собираемся создать асинхронную задачу для отправки уведомлений по электронной почте нашим пользователям при размещении заказа. Соглашение должно включать асинхронные задачи для вашего приложения в модуле `tasks` в вашем каталоге приложений.

Создайте новый файл в приложении `orders` и назовите его `tasks.py`. Это место, где Celery будет искать асинхронные задачи. Добавьте к нему следующий код:

```
from celery import task
from django.core.mail import send_mail
from .models import Order

@task
def order_created(order_id):
    """
    Task to send an e-mail notification when an order is
    successfully created.
    """
    order = Order.objects.get(id=order_id)
    subject = 'Order nr. {}'.format(order.id)
    message = 'Dear {},\n\nYou have successfully placed an order.\n' \
              'Your order id is {}.'.format(order.first_name,
                                             order.id)
    mail_sent = send_mail(subject,
                          message,
                          'admin@myshop.com',
                          [order.email])
    return mail_sent
```

Мы определяем задачу `order_created` с помощью декоратора `task`. Как вы можете видеть, задача Celery - это просто функция

Python, декорированная `task`. Наша функция `task` получает параметр `order_id`. При выполнении задачи всегда рекомендуется передавать задачам и объектам поиска только идентификаторы. Мы используем функцию `send_mail()`, предоставленную Django, чтобы отправить уведомление по электронной почте пользователю, разместившему заказ.

Вы узнали, как настроить Django для использования вашего SMTP-сервера в [главе 2, Улучшение вашего блога с помощью продвинутых возможностей](#). Если вы не хотите заниматься настройками электронной почты, вы можете сообщить Django о отправке писем в консоль, добавив следующий параметр в файл `settings.py`:

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

Используйте асинхронные задачи не только для трудоемких процессов, но и для других процессов, которые подвержены сбою. Которые не требуют много времени для выполнения, но которые подвержены сбоям подключения или требуют политики повтора.

Теперь нам нужно добавить задачу в наш `order_create`. Измените файл `views.py` приложения `orders`, импортируйте задачу и вызовите `order_created` – асинхронную задачу после очистки корзины:

```
from .tasks import order_created

def order_create(request):
    # ...
    if request.method == 'POST':
        # ...
        if form.is_valid():
            # ...
            cart.clear()
            # launch asynchronous task
            order_created.delay(order.id)
        # ...
```

Мы вызываем метод `delay()` задачи для его асинхронного выполнения. Задача будет добавлена в очередь и будет

выполнена обработчиком как можно скорее.

Откройте другую консоль и запустите обработчик Celery из вашего каталога проекта, используя следующую команду:

```
celery -A myshop worker -l info
```

Обработчик Celery теперь работает и готов к обработке задач. Убедитесь, что сервер разработки Django также запущен. Откройте <http://127.0.0.1:8000/> в своем браузере, добавьте некоторые товары в свою корзину и выполните заказ. В консоле вы запустили обработчик Celery, и вы увидите результат, похожий на этот:

```
[2017-12-17 17:43:11,462: INFO/MainProcess] Received task:  
orders.tasks.order_created[e990ddae-2e30-4e36-b0e4-78bbd4f2738e]  
[2017-12-17 17:43:11,685: INFO/ForkPoolWorker-4] Task  
orders.tasks.order_created[e990ddae-2e30-4e36-b0e4-78bbd4f2738e] succeeded in  
0.22019841300789267s: 1
```

Задача выполнена, и вы получите уведомление по электронной почте о вашем заказе.

Мониторинг Celery

Вы можете отслеживать выполняемые асинхронные задачи. Flower - это веб-инструмент для мониторинга Celery. Вы можете установить Flower с помощью этой команды:

```
pip install flower==0.9.2
```

После установки вы можете запустить Flower, выполнив следующую команду в каталоге проекта:

```
celery -A myshop flower
```

Откройте <http://localhost:5555/dashboard> в вашем браузере. Вы сможете увидеть активных обработчиков Celery и статистику асинхронных задач:

Active: 0

Processed: 0

Failed: 0

Succeeded: 0

Retried: 0

Search:

Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@MacBook-Air-de-Antonio.local	Online	0	0	0	0	0	2.2, 2.4, 2.36

Showing 1 to 1 of 1 entries

Вы можете найти документацию по Flower на <https://flower.readthedocs.io/>.

Резюме

В этой главе вы создали основное приложение магазина. Вы создали каталог продуктов и создали корзину покупок, используя сеансы. Вы внедрили специальный обработчик контекста, чтобы сделать корзину доступной для ваших шаблонов и создали форму для размещения заказов. Вы также узнали, как запускать асинхронные задачи с помощью Celery.

В следующей главе вы узнаете, как интегрировать платежный шлюз в свой магазин, добавлять пользовательские действия на сайт администрирования, экспортить данные в формате CSV и генерировать PDF-файлы динамически.

Глава 8

Управление платежами и заказами

В предыдущей главе вы создали базовый интернет-магазин с каталогом продуктов и корзиной покупок. Вы также узнали, как запускать асинхронные задачи с помощью Celery. В этой главе вы узнаете, как интегрировать платежный шлюз на свой сайт, чтобы позволить пользователям платить с кредитной карты. Вы также расширите сайт администрирования, чтобы экспорттировать заказы в формат CSV, и вы научитесь создавать PDF-счета-фактуры.

В этой главе вы научитесь:

- Интегрировать платежный шлюз в ваш проект
- Экспорттировать заказы в CSV-файлы
- Создавать пользовательские представления для сайта администрирования
- Динамически создавать счета-фактуры PDF

Интеграция платежного шлюза

Платежный шлюз позволяет обрабатывать платежи онлайн. Используя платежный шлюз, вы можете управлять заказами клиентов и делегировать обработку платежей надежной, безопасной третьей стороне. Вам не придется беспокоиться о обработке кредитных карт в вашей собственной системе.

На выбор есть несколько поставщиков платежных шлюзов. Мы собираемся интегрировать Braintree, который используется популярными онлайн-сервисами, такими как Uber или Airbnb. Braintree предоставляет API, который позволяет обрабатывать онлайн-платежи с помощью нескольких способов оплаты, таких как кредитная карта, PayPal, Android Pay и Apple Pay. Вы можете больше узнать о Braintree на <https://www.braintreepayments.com/>.

Braintree предоставляет различные варианты интеграции. Самой простой является интеграция *Drop-in*, которая содержит предварительно отформатированную форму оплаты. Однако, чтобы настроить поведение и проверку, мы собираемся использовать расширенную интеграцию *Hosted Fields*. Вы можете узнать больше об интеграции Hosted Fields в <https://developers.braintreepayments.com/guides/hosted-fields/overview/javascript/v3>.

Некоторые платежные поля на странице проверки, такие как номер кредитной карты, номер CVV или дата истечения срока действия, должны храниться безопасно. Интеграция Hosted Fields объединяет поля проверки в домене платежного шлюза и отображает iframe для представления полей пользователям. Это дает вам возможность настраивать внешний вид формы платежа, гарантируя, что вы соответствуете требованиям

Индустрии платежных карт (Payment Card Industry - PCI). Поскольку вы можете настроить внешний вид полей формы, пользователи не заметят iframe.

Создание учетной записи в "песочнице" от Braintree

Вам нужна учетная запись Braintree для интеграции платежного шлюза на ваш сайт. Давайте создадим учетную запись "песочницы" для тестирования API-интерфейса Braintree. Откройте в своем браузере <https://www.braintreepayments.com/sandbox>. Вы увидите форму, похожую на следующую:

Test Everything Braintree

Entering our sandbox allows you to get a feel for the Braintree experience before applying for a merchant account or going to production.

Already in the sandbox? [Sign in](#).

Sign up for the sandbox

Full name

First name Last name

Company name

Where is your business located?

Spain

Email address

Try the sandbox

Заполните детали, чтобы создать новую учетную запись для "песочницы". Вы получите письмо от Braintree со ссылкой, чтобы завершить настройку своей учетной записи. Перейдите по ссылке и выполните настройку своей учетной записи. Как только вы закончите, зайдите в <https://sandbox.braintreegateway.com/login>. Ваш идентификатор продавца и приватные/публичные ключи будут отображаться следующим образом:

Sandbox Keys & Configuration

Here are the keys to your Sandbox Account. Once you're ready to start taking payments with a production Braintree Account you'll have to update your code, replacing these with your production Braintree Account keys.

Merchant ID: 9xtfhm7sv733jznk

Public Key: q8fxx6fwkjx8dfkw

Private Key: xxxxxxxxxxxxxxxxxxxxxxxxx

Вам понадобится эта информация для аутентификации запросов к API-интерфейсу Braintree. Всегда держите приватный ключ в секрете.

Установка модуля Python Braintree

Braintree предоставляет модуль Python, который упрощает работу с его API. Исходный код находится в https://github.com/braintree/braintree_python. Мы собираемся интегрировать платежный шлюз в наш проект с помощью модуля `braintree`.

Установите модуль `braintree` из консоли, используя следующую команду:

```
pip install braintree==3.45.0
```

Добавьте следующие параметры в файл `settings.py` вашего проекта:

```
# Braintree settings
BRAINTREE_MERCHANT_ID = 'XXX' # Merchant ID
BRAINTREE_PUBLIC_KEY = 'XXX' # Public Key
BRAINTREE_PRIVATE_KEY = 'XXX' # Private key

from braintree import Configuration, Environment

Configuration.configure(
    Environment.Sandbox,
    BRAINTREE_MERCHANT_ID,
    BRAINTREE_PUBLIC_KEY,
    BRAINTREE_PRIVATE_KEY
)
```

Замените значения `BRAINTREE_MERCHANT_ID`, `BRAINTREE_PUBLIC_KEY`, и `BRAINTREE_PRIVATE_KEY` данными вашей учетной записи.

Обратите внимание, что мы используем `Environment.Sandbox` для интеграции песочницы. После того, как вы перейдете в производство

и создадите реальную учетную запись, вам нужно будет изменить ее на Environment.Production. Braintree предоставит вам новый идентификатор продавца и приватный/публичный ключи для производственной среды.

Давайте интегрируем платежный шлюз в процесс оформления заказа.

Интеграция платежного шлюза

Процесс проверки работает следующим образом:

1. Добавляем товары в корзину
2. Проверяем корзину покупок
3. Вводим данные кредитной карты и оплачиваем

Мы собираемся создать новое приложение для управления платежами. Создайте новое приложение в своем проекте, используя следующую команду:

```
python manage.py startapp payment
```

Отредактируйте файл `settings.py` вашего проекта и добавьте новое приложение в настройку `INSTALLED_APPS` следующим образом:

```
INSTALLED_APPS = [  
    # ...  
    'payment.apps.PaymentConfig',  
]
```

Приложение `payment` теперь активно.

После того как клиенты сделают заказ, нам необходимо перенаправить их на процесс оплаты. Измените файл `views.py` приложения `orders` и включите следующие импорты:

```
from django.urls import reverse
from django.shortcuts import render, redirect
```

В том же файле найдите следующие строки в представлении `order_create`:

```
# launch asynchronous task
order_created.delay(order.id)
return render(request,
              'orders/order/created.html',
              locals())
```

Замените их следующими:

```
# launch asynchronous task
order_created.delay(order.id)
# set the order in the session
request.session['order_id'] = order.id
# redirect for payment
return redirect(reverse('payment:process'))
```

С помощью этого кода, после успешного создания заказа, мы устанавливаем идентификатор заказа в текущей сессии, используя ключ сессии `order_id`. Затем мы перенаправляем пользователя на URL `payment:process`, который мы собираемся реализовать позже.

Помните, что вам нужно запустить Celery, чтобы задача `order_created` была поставлена в очередь и выполнена.

Каждый раз, когда создается заказ в Braintree, генерируется уникальный идентификатор транзакции. Мы добавим новое поле в модель `Order` приложения `orders` для хранения идентификатора транзакции. Это позволит нам связать каждый заказ с его транзакцией Braintree.

Измените файл `models.py` приложения `orders` и добавьте следующее поле в модель `Order`:

```
class Order(models.Model):
    # ...
    braintree_id = models.CharField(max_length=150, blank=True)
```

Давайте синхронизируем это поле с базой данных.

Используйте следующую команду для создания миграции:

```
python manage.py makemigrations
```

Вы увидите следующий результат:

```
Migrations for 'orders':
  orders/migrations/0002_order_braintree_id.py
    - Add field braintree_id to order
```

Примените миграцию к базе данных с помощью следующей команды:

```
python manage.py migrate
```

Вы увидите вывод, который заканчивается следующей строкой:

```
Applying orders.0002_order_braintree_id... OK
```

Изменения модели теперь синхронизируются с базой данных. Теперь вы можете хранить идентификатор транзакции Braintree для каждого заказа. Давайте интегрируем платежный шлюз.

Интеграция Braintree с использованием Hosted Fields

Интеграция *Hosted Fields* позволяет вам создать свою собственную форму платежа, используя пользовательские стили и макет. Iframe динамически добавляется на страницу с помощью SDK Braintree JavaScript. IFrame включает в себя форму оплаты Hosted Fields. Когда клиент отправляет форму, Hosted Fields безопасно принимает данные карты и пытается их токенизировать. Если токенизация завершается успешно, вы можете отправить сгенерированный для данного случая токен в свое представление, чтобы совершить транзакцию, используя модуль Python `braintree`.

Мы создадим представление для обработки платежей. Весь процесс проверки будет работать следующим образом:

1. В представлении токен клиента генерируется с использованием модуля Python `braintree`. Этот токен используется в следующем шаге, чтобы создать экземпляр JavaScript-клиента Braintree; это не токен оплаты.
2. Представление отображает шаблон оформления заказа. Шаблон загружает Braintree JavaScript SDK с использованием токена клиента и генерирует iframe с полями платежной платформы.
3. Пользователи вводят данные своей кредитной карты и

отправляют форму. Платежный токен для данного случая создается с помощью JavaScript-клиента Braintree. Мы отправляем токен нашему представлению в POST запросе.

4. В представлении оплаты принимается токен, и мы используем его для создания транзакции с использованием модуля Python `braintree`.

Начнем с просмотра платежного поручения. Откройте файл `views.py` приложения `payment` и добавьте к нему следующий код:

```
import braintree
from django.shortcuts import render, redirect, get_object_or_404
from orders.models import Order

def payment_process(request):
    order_id = request.session.get('order_id')
    order = get_object_or_404(Order, id=order_id)

    if request.method == 'POST':
        # retrieve nonce
        nonce = request.POST.get('payment_method_nonce', None)
        # create and submit transaction
        result = braintree.Transaction.sale({
            'amount': '{:.2f}'.format(order.get_total_cost()),
            'payment_method_nonce': nonce,
            'options': {
                'submit_for_settlement': True
            }
        })
        if result.is_success:
            # mark the order as paid
            order.paid = True
            # store the unique transaction id
            order.braintree_id = result.transaction.id
            order.save()
            return redirect('payment:done')
        else:
            return redirect('payment:canceled')
    else:
        # generate token
        client_token = braintree.ClientToken.generate()
```

```
    return render(request,
                  'payment/process.html',
                  {'order': order,
                   'client_token': client_token})
```

Представление `payment_process` управляет процессом оформления заказа. Сдесь выполняются следующие действия:

1. Мы получаем текущий заказ из ключа сессии `order_id`, который был установлен ранее в представлении `order_create`.
2. Мы извлекаем объект `order` для данного идентификатора или возвращаем ошибку `404 Not Found`, если он не найден.
3. Когда представление загружается с запросом `POST`, мы получаем `payment_method_nonce` для создания новой транзакции с помощью `braintree.Transaction.sale()`. Мы передаем ему следующие параметры:
 1. `amount`: Общая сумма для оплаты клиентом.
 2. `payment_method_nonce`: Токен, созданный Braintree для оплаты. Он будет сгенерирован в шаблоне с использованием JavaScript-кода Braintree JavaScript.
 3. `options`: Мы отправляем параметр `submit_for_settlement` с `True`, поэтому транзакция автоматически предъявляется к расчету.
4. Если транзакция успешно обработана, мы помечаем заказ как оплаченный, установив его `paid` атрибут в `True`, и

мы сохраняем уникальный идентификатор транзакции, возвращенный шлюзом в `braintree_id`. Мы перенаправляем пользователя на URL `payment:done`, если платеж был успешным в противном случае, на `payment:canceled`.

5. Если представление было загружено с запросом `GET`, мы создаем клиентский токен, который мы будем использовать в шаблоне, чтобы создать экземпляр JavaScript-клиента Braintree.

Давайте создадим базовые представления для перенаправления пользователей, когда платеж был успешным, или когда он был отменен по любой причине. Добавьте следующий код в файл `views.py` приложения `payment`:

```
def payment_done(request):
    return render(request, 'payment/done.html')

def payment_canceled(request):
    return render(request, 'payment/canceled.html')
```

Создайте новый файл в каталоге приложения `payment` и назовите его `urls.py`. Добавьте к нему следующий код:

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

app_name = 'payment'

urlpatterns = [
    path('process/', views.payment_process, name='process'),
    path('done/', views.payment_done, name='done'),
    path('canceled/', views.payment_canceled, name='canceled'),
]
```

Это URL-адреса для рабочего процесса оплаты. Мы включили

следующие паттерны URL:

- `process`: Представление, которое обрабатывает платеж
- `done`: Представление перенаправит пользователя, если платеж будет успешным
- `canceled`: Представление перенаправит пользователя , если платеж не увенчался успехом

Отредактируйте основной файл `urls.py` проекта `myshop` и включите паттерн URL для приложения `payment` следующим образом:

```
urlpatterns = [
    # ...
    path('payment/', include('payment.urls', namespace='payment')),
    path('', include('shop.urls', namespace='shop')),
]
```

Не забудьте разместить его перед шаблоном `shop.urls`, чтобы избежать нежелательного соответствия шаблону.

Создайте следующую структуру файлов внутри каталога приложения `payment`:

```
templates/
    payment/
        process.html
        done.html
        canceled.html
```

Откройте шаблон `payment/process.html` и добавьте к нему следующий код:

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

{% block title %}Pay by credit card{% endblock %}
```

```
{% block content %}

<h1>Pay by credit card</h1>
<form action"." id="payment" method="post">

    <label for="card-number">Card Number</label>
    <div id="card-number" class="field"></div>

    <label for="cvv">CVV</label>
    <div id="cvv" class="field"></div>

    <label for="expiration-date">Expiration Date</label>
    <div id="expiration-date" class="field"></div>

    <input type="hidden" id="nonce" name="payment_method_nonce" value="">
    {% csrf_token %}
    <input type="submit" value="Pay">
</form>
<!-- Load the required client component. -->
<script src="https://js.braintreegateway.com/web/3.29.0/js/client.min.js">
</script>
<!-- Load Hosted Fields component. -->
<script src="https://js.braintreegateway.com/web/3.29.0/js/hosted-fields.min.js"></script>
<script>
    var form = document.querySelector('#payment');
    var submit = document.querySelector('input[type="submit"]');

    braintree.client.create({
        authorization: '{{ client_token }}'
    }, function (clientErr, clientInstance) {
        if (clientErr) {
            console.error(clientErr);
            return;
        }

        braintree.hostedFields.create({
            client: clientInstance,
            styles: {
                'input': {'font-size': '13px'},
                'input.invalid': {'color': 'red'},
                'input.valid': {'color': 'green'}
            },
            fields: {
                number: {selector: '#card-number'},
                cvv: {selector: '#cvv'},
                expirationDate: {selector: '#expiration-date'}
            }
        }, function (hostedFieldsErr, hostedFieldsInstance) {
            if (hostedFieldsErr) {
```

```
        console.error(hostedFieldsErr);
        return;
    }

    submit.removeAttribute('disabled');

    form.addEventListener('submit', function (event) {
        event.preventDefault();

        hostedFieldsInstance.tokenize(function (tokenizeErr, payload) {
            if (tokenizeErr) {
                console.error(tokenizeErr);
                return;
            }
            // set nonce to send to the server
            document.getElementById('nonce').value = payload.nonce;
            // submit form
            document.getElementById('payment').submit();
        });
    }, false);
});
});

</script>
{% endblock %}
```

Это шаблон, который отображает форму платежа и обрабатывает платеж. Мы определяем контейнеры `<div>` вместо элементов `<input>` для полей ввода кредитной карты: номера кредитной карты, CVV номера и срока годности. Именно так мы указываем поля, которые клиент JavaScript Braintree будет отображать в iframe. Мы также включаем элемент `<input>` с именем `payment_method_nonce`, который мы будем использовать для отправки маркера nonce на наше представление после создания клиентом JavaScript Braintree.

В нашем шаблоне мы загружаем клиентский файл Braintree JavaScript SDK `client.min.js` и компонент Hosted Fields `hosted-fields.min.js`. Затем мы выполним следующий код JavaScript:

1. Мы создаем экземпляр JavaScript-клиента Braintree с помощью метода `braintree.client.create()`, используя `client_token`, сгенерированный в представлении

`payment_process.`

2. Мы создаем экземпляр компонента Hosted Fields с помощью метода `braintree.hostedFields.create()`.
3. Мы указываем пользовательские стили CSS для полей `input`.
4. Мы указываем селектора `id` для полей: `card-number`, `cvv`, а также `expiration-date`.
5. Мы добавляем прослушиватель событий для действия `submit` формы. Когда форма отправляется, поля маркируются с помощью SDK Braintree, а токен nonce устанавливается в поле `payment_method_nonce`. Затем форма отправляется так, что наше представление получает nonce для обработки платежа.

Откройте шаблон `payment/done.html` и добавьте к нему следующий код:

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

{% block content %}
    <h1>Your payment was successful</h1>
    <p>Your payment has been processed successfully.</p>
{% endblock %}
```

Это шаблон страницы, на которую пользователь перенаправляется после успешного платежа.

Отредактируйте шаблон `payment/canceled.html` и добавьте к нему следующий код:

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

```
{% 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 перенаправляется пользователь, когда транзакция не удалась. Давайте протестируем процесс оплаты.

Тестирование платежей

Откройте консоль и запустите RabbitMQ с помощью следующей команды:

```
rabbitmq-server
```

Откройте другую консоль и запустите обработчик Celery из вашего каталога проекта с помощью следующей команды:

```
celery -A myshop worker -l info
```

Откройте еще одну консоль и запустите сервер разработки с помощью этой команды:

```
python manage.py runserver
```

Откройте <http://127.0.0.1:8000/> в своем браузере добавьте некоторые товары в корзину покупок и заполните форму проверки. Когда вы нажимаете кнопку PLACE ORDER, заказ будет сохранен в базе данных, идентификатор заказа будет сохранен в текущем сеансе, и вы будете перенаправлены на страницу процесса оплаты.

Страница процесса платежа извлекает заказ из сеанса и отображает форму Hosted Fields в iframe следующим образом:

Pay by credit card

Card Number

CVV

Expiration Date

Pay

Вы можете посмотреть исходный код HTML, чтобы увидеть сгенерированный HTML.

Braintree предоставляет список успешных и неудачных кредитных карт, чтобы вы могли протестировать все возможные сценарии. Вы можете найти список кредитных карт для тестирования на <https://developers.braintreepayments.com/guides/credit-cards/testing-go-live/python>. Мы собираемся использовать тестовую карту VISA 4111 1111 1111 1111, которая возвращает успешную покупку. Мы будем использовать CVV 123 и любую будущую дату истечения срока действия, например 12/24. Введите данные кредитной карты следующим образом:

Pay by credit card

Card Number

4111 1111 1111 1111

CVV

123

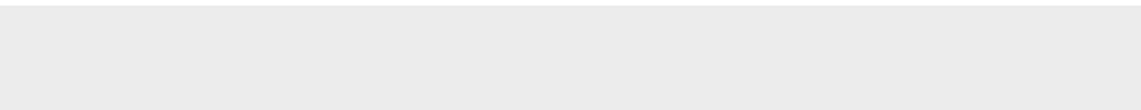
Expiration Date

12 / 20

Pay

Нажмите на кнопку Pay. Вы должны увидеть следующую страницу:

My shop



Your payment was successful

Your payment has been processed successfully.

Сделка успешно обработана. Теперь вы можете войти в свою учетную запись на <https://sandbox.braintreegateway.com/login>. В разделе Transactions вы сможете увидеть транзакцию следующим образом:

ID	Transaction Date	Type	Status	Customer	Payment Information	Amount
2bwkx5b6	02/05/2018 07:45:23 PM CST	Sale	Submitted For Settlement		 411111*****1111	21,20 € EUR

Теперь откройте <http://127.0.0.1:8000/admin/orders/order/> в вашем браузере. Сейчас заказ должен быть отмечен как оплаченный и содержать связанную транзакцию Braintree ID:

Paid

Braintree id:

2bwkx5b6

Поздравляем! Вы внедрили платежный шлюз для обработки кредитных карт.

Внедряем в жизнь

После того, как вы протестирували свою среду, вы можете создать реальную учетную запись Braintree на <https://www.braintreepayments.com>. Если вы готовы к внедрению в производство, не забудьте изменить свои учетные данные в файле `settings.py` вашего проекта и использовать `braintree.Environment.Production`, чтобы настроить среду. Все шаги, которые нужно предпринять, представлены в <https://developers.braintreepayments.com/start/go-live/python>.

.

Экспорт заказов в CSV-файлы

Иногда нужно экспортировать информацию, содержащуюся в модели, в файл, чтобы вы могли импортировать ее в любую другую систему. Одним из наиболее широко используемых форматов для экспорта/импорта данных является **значения, разделенные запятыми (CSV)**. CSV-файл представляет собой текстовый файл, состоящий из нескольких записей. Обычно для каждой строки используется одна запись, и некоторый символ разделителя, обычно запятая, разделяет поля записи. Мы собираемся настроить сайт администрирования, чтобы иметь возможность экспортировать заказы в файлы CSV.

Добавление пользовательских действий на сайт администрирования

Django предлагает вам широкий спектр возможностей для настройки сайта администрирования. Мы собираемся изменить представление списка объектов, чтобы включить настраиваемое действие администратора.

Действие администратора работает следующим образом: пользователь выбирает объекты со страницы списка объектов администратора с помощью флажков, затем выбирает действие для всех выбранных элементов и выполняет действие. На следующем снимке экрана показаны действия на сайте администрирования:

Select user to change

The screenshot shows the Django Admin interface for managing users. At the top, there's a search bar with a magnifying glass icon. Below it, a table lists users with columns for 'EMAIL ADDRESS' and 'admin'. A modal dialog box is open over the table, titled 'Action:'. It contains a dropdown menu with a checked checkbox and a blank field, followed by a 'Go' button and the text '0 of 1 selected'. Below the dropdown is a blue button labeled 'Delete selected users'. To the left of the table, there are two checkboxes: one for the first user and one for the user named 'admin'.

Создайте пользовательские действия администратора, чтобы пользователи могли применять действия сразу к нескольким элементам.

Вы можете создать настраиваемое действие, написав регулярную функцию, которая получает следующие параметры:

- Текущий отображаемый `ModelAdmin`
- Текущий объект запроса как экземпляр `HttpRequest`
- `QuerySet` для объектов, выбранных пользователем.

Эта функция будет выполняться, когда действие запускается с сайта администрирования.

Мы собираемся создать пользовательское действие для загрузки списка заказов в виде файла CSV. Отредактируйте файл `admin.py` приложения `orders` и добавьте следующий код перед классом `OrderAdmin`:

```
import csv
import datetime
from django.http import HttpResponseRedirect

def export_to_csv(modeladmin, request, queryset):
    opts = modeladmin.model._meta
    response = HttpResponseRedirect(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; \'\
        filename={}.csv'.format(opts.verbose_name)
    writer = csv.writer(response)

    fields = [field for field in opts.get_fields() if not field.many_to_many\
        and not field.one_to_many]
    # Write a first row with header information
    writer.writerow([field.verbose_name for field in fields])
    # Write data rows
    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 коде мы выполняем следующие задачи:

1. Мы создаем экземпляр `HttpResponse`, включая пользовательский тип содержимого `text/csv`, чтобы сообщить браузеру, что ответ должен рассматриваться как файл CSV. Мы также добавляем заголовок `Content-Disposition`, чтобы указать, что ответ HTTP содержит прикрепленный файл.
2. Мы создаем объект `CSV writer`, который будет записываться в объект `response`.
3. Динамически мы получаем поля `model` с помощью метода для моделей `get_fields()` опции `_meta`. Мы исключаем отношения «многие ко многим» и «один ко многим».
4. Мы пишем строку заголовка, включая имена полей.
5. Мы перебираем заданный `QuerySet` и записываем строку для каждого объекта, возвращаемого `QuerySet`. Мы заботимся о форматировании объектов `datetime`, потому что выходное значение для CSV должно быть строкой.
6. Мы настраиваем отображаемое имя для действия в шаблоне, устанавливая атрибут `short_description` для функции.

Мы создали общее действие администратора, которое можно добавить в любой класс `ModelAdmin`.

Наконец, добавьте новое действие администратора `export_to_csv` в класс `OrderAdmin` следующим образом:

```
class OrderAdmin(admin.ModelAdmin):
    # ...
    actions = [export_to_csv]
```

Откройте <http://127.0.0.1:8000/admin/orders/order/> в вашем браузере. В результате действие администратора должно выглядеть следующим образом:

Select order to change

Action: [Export to CSV](#) Go 1 of 19 selected

<input type="checkbox"/>	ID	FIRST NAME	LAST NAME	EMAIL	ADDRESS
<input checked="" type="checkbox"/>	19	Antonio	Melé	antonio.mele@gmail.com	Bank Street
<input type="checkbox"/>	18	Django	Reinhardt	email@domain.com	Music Street

Выберите некоторые заказы и выберите действие Export to CSV из окна выбора, затем нажмите кнопку Go. Ваш браузер загрузит созданный CSV-файл с именем `order.csv`. Откройте загруженный файл с помощью текстового редактора. Вы должны увидеть содержимое следующего формата, включая строку заголовка и строку для каждого выбранного вами объекта `Order`:

```
ID,first name,last name,email,address,postal  
code,city,created,updated,paid,braintree id  
3,antonio,Melé,antonio.mele@gmail.com,Bank Street,WS  
J11,London,25/02/2018,25/02/2018,True,2bwkx5b6  
...
```

Как вы можете видеть, создание действий администратора это довольно просто. Вы можете узнать больше о создании CSV-файлов с помощью Django в <https://docs.djangoproject.com/en/2.0/howto/outputting-csv/>.

Расширение сайта администратора с помощью пользовательских представлений

Иногда вам нужны настройки для сайта администрирования больше тех, что может предоставить `ModelAdmin`, создание действий администратора и переопределения шаблонов администратора. Если это так, то вам нужно создать пользовательское представление администратора. С помощью пользовательского представления вы можете создавать любую функциональность, в которой вы нуждаетесь. Вам просто нужно убедиться, что только пользователи с соответствующим разрешением могут получить доступ к вашему представлению и что вы поддерживаете внешний вид сайта администрирования, создавая шаблон который расширяет шаблон сайта администратора.

Давайте создадим пользовательское представление для отображения информации о заказе. Измените файл `views.py` приложения `orders` и добавьте к нему следующий код:

```
from django.contrib.admin.views.decorators import staff_member_required
from django.shortcuts import get_object_or_404
from .models import Order

@staff_member_required
def admin_order_detail(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    return render(request,
                  'admin/orders/order/detail.html',
                  {'order': order})
```

Декоратор `staff_member_required` проверяет, что поля `is_active` и `is_staff` пользователя, запрашивающего страницу, установлены в `True`. В этом представлении мы получаем объект `order` с данным идентификатором и визуализируем шаблон для отображения заказа.

Теперь отредактируйте файл `urls.py` приложения `orders` и добавьте к нему следующий паттерн URL:

```
path('admin/order/<int:order_id>', views.admin_order_detail,  
name='admin_order_detail'),
```

Создайте следующую структуру файлов внутри каталога `templates/` приложения `orders`:

```
admin/  
    orders/  
        order/  
            detail.html
```

Откройте шаблон `detail.html` и добавьте к нему следующее содержимое:

```
{% extends "admin/base_site.html" %}  
{% load static %}  
  
{% block extrastyle %}  
    <link rel="stylesheet" type="text/css" href="{% static "css/admin.css" %}"  
/>>  
{% endblock %}  
  
{% block title %}  
    Order {{ order.id }} {{ block.super }}  
{% endblock %}  
  
{% block breadcrumbs %}  
    <div class="breadcrumbs">  
        <a href="{% url "admin:index" %}">Home</a> &rsaquo;  
        <a href="{% url "admin:orders_order_changelist" %}">Orders</a>  
        &rsaquo;  
        <a href="{% url "admin:orders_order_change" order.id %}">Order {{
```

```

order.id }}>/a>
    &rsaquo; Detail

```

```

</div>
{%- endblock %}

{%- block content %}
<h1>Order {{ order.id }}</h1>
<ul class="object-tools">
    <li>
        <a href="#" onclick="window.print();">Print order</a>
    </li>
</ul>
<table>
    <tr>
        <th>Created</th>
        <td>{{ order.created }}</td>
    </tr>
    <tr>
        <th>Customer</th>
        <td>{{ order.first_name }} {{ order.last_name }}</td>
    </tr>
    <tr>
        <th>E-mail</th>
        <td><a href="mailto:{{ order.email }}">{{ order.email }}</a></td>
    </tr>
    <tr>
        <th>Address</th>
        <td>{{ order.address }}, {{ order.postal_code }} {{ order.city }}</td>
    </tr>
    <tr>
        <th>Total amount</th>
        <td>${{ order.get_total_cost }}</td>
    </tr>
    <tr>
        <th>Status</th>
        <td>{% if order.paid %}Paid{% else %}Pending payment{% endif %}</td>
    </tr>
</table>

<div class="module">
    <div class="tabular inline-related last-related">
        <table>
            <h2>Items bought</h2>
            <thead>
                <tr>
                    <th>Product</th>
                    <th>Price</th>
                    <th>Quantity</th>
                    <th>Total</th>
                </tr>
            </thead>

```

```
</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>
</div>
{% endblock %}
```

Это шаблон для отображения информации о заказе на сайте администрации. Этот шаблон расширяет шаблон `admin/base_site.html` сайта администрации Django, который содержит основную структуру HTML и стили CSS администратора. Мы загружаем пользовательский статический файл `css/admin.css`.

Чтобы использовать статические файлы, вам нужно получить их из кода, который идет с этой главой. Скопируйте статические файлы, расположенные в каталоге `static/` приложения `заказов orders`, и добавьте их в то же место в вашем проекте.

Мы используем блоки, определенные в родительском шаблоне, чтобы включить наше собственное содержимое. Мы показываем информацию о заказе и купленных товарах.

Если вы хотите расширить шаблон администратора, вам необходимо знать его структуру и идентифицировать существующие блоки. Вы можете найти все шаблоны администратора <https://github.com/django/django/tree/2.0/django/contrib/admin>

`n/templates/admin.`

Вы также можете переопределить шаблон администратора, если вам это нужно. Чтобы переопределить шаблон администратора, скопируйте его в каталог `templates`, сохраняя тот же относительный путь и имя файла. Административный сайт Django будет использовать ваш собственный шаблон вместо стандартного.

Наконец, добавим ссылку на каждый объект `Order` на странице отображения списка на сайте администрирования.

Отредактируйте файл `admin.py` приложения `orders` и добавьте к нему следующий код над классом `OrderAdmin`:

```
from django.urls import reverse
from django.utils.safestring import mark_safe

def order_detail(obj):
    return mark_safe('<a href="{}">View</a>'.format(
        reverse('orders:admin_order_detail', args=[obj.id])))
```

Это функция, которая принимает объект `Order` в качестве аргумента и возвращает ссылку HTML для `admin_order_detail` URL. Django по умолчанию удаляет вывод HTML. Мы должны использовать функцию `mark_safe`, чтобы избежать автоматического экранирования.

Используйте функцию `mark_safe`, чтобы избежать экранирования HTML. Когда вы используете `mark_safe`, обязательно избегайте ввода, который пришел от пользователя, чтобы избежать межсайтового скрипtingа.

Затем отредактируйте класс `OrderAdmin`, чтобы отобразить ссылку:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id',
                    'first_name',
                    # ...
                    'updated',
                    order_detail]
```

Откройте <http://127.0.0.1:8000/admin/orders/order/> в вашем браузере.
Каждая строка теперь включает ссылку View следующим образом:

PAID	CREATED	UPDATED	ORDER DETAIL
------	---------	---------	--------------

 Feb. 6, 2018, 1:35 a.m. Feb. 6, 2018, 1:45 a.m. [View](#)

Нажмите на ссылку View любого заказа для загрузки страницы с подробным описанием заказа. Вы должны увидеть страницу, такую как следующая:

Django administration

Home › Orders › Order 19 › Detail

Order 19 PRINT ORDER

Created Feb. 6, 2018, 1:35 a.m.

Customer Antonio Melé

E-mail antonio.mele@gmail.com

Address Jazz Street, 28027 Madrid

Total amount \$21.2

Status Paid

Items bought

PRODUCT	PRICE	QUANTITY	TOTAL
Tea powder	\$21.2	1	\$21.2
Total			\$21.2

Динамическое создание счетов-фактур PDF

Теперь, когда у нас есть полная система оплаты и контроля, мы можем создать счет-фактуру PDF для каждого заказа. Для создания файлов PDF существует несколько библиотек Python. Одной из популярных библиотек для создания PDF-файлов является Reportlab. Вы можете найти информацию о том, как создавать файлы PDF с помощью Reportlab в <https://docs.djangoproject.com/en/2.0/howto/outputting-pdf/>.

В большинстве случаев вам придется добавлять пользовательские стили и форматирование в файлы PDF. Вам будет удобнее отображать HTML-шаблон и преобразовывать его в файл PDF на уровне презентации. Мы будем следовать этому подходу и использовать модуль для создания PDF-файлов в Django. Мы будем использовать WeasyPrint, представляющую собой библиотеку Python, которая может создавать PDF-файлы из HTML-шаблонов.

Установка WeasyPrint

Во-первых, установите зависимости WeasyPrint для вашей ОС, которые вы найдете в <http://weasyprint.org/docs/install/#platforms>. Затем установите WeasyPrint с помощью `pip`, используя следующую команду:

```
pip install WeasyPrint==0.42.3
```

Создание шаблона PDF

Нам нужен документ HTML в качестве ввода для WeasyPrint. Мы собираемся создать HTML-шаблон с помощью Django и передать его в WeasyPrint для создания PDF-файла.

Создайте новый файл шаблона в каталоге `templates/orders/order/` приложения `orders` и назовите его `pdf.html`. Добавьте к нему следующий 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>
                <th>Cost</th>
            </tr>
        </thead>
        <tbody>
            {% for item in order.items.all %}<br>
```

```
<tr class="row{% cycle "1" "2" %}">
    <td>{{ item.product.name }}</td>
    <td class="num">${{ item.price }}</td>
    <td class="num">{{ item.quantity }}</td>
    <td class="num">${{ item.get_cost }}</td>
</tr>
{% endfor %}
<tr class="total">
    <td colspan="3">Total</td>
    <td class="num">${{ order.get_total_cost }}</td>
</tr>
</tbody>
</table>

<span class="{% if order.paid %}paid{% else %}pending{% endif %}">
    {% if order.paid %}Paid{% else %}Pending payment{% endif %}
</span>
</body>
</html>
```

Это шаблон для счета-фактуры PDF. В этом шаблоне мы отобразим все детали заказа в элементе HTML `<table>`, включая товар. Мы также включаем сообщение для отображения, если заказ был уплачен или платеж все еще находится на рассмотрении.

Отрисовывание PDF файлов

Мы собираемся создать представление для создания счетов-фактур PDF для существующих заказов с использованием сайта администрирования. Измените файл `views.py` в каталоге приложений `orders` и добавьте к нему следующий 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 = HttpResponseRedirect(content_type='application/pdf')
    response['Content-Disposition'] = 'filename=%s.pdf' % order.id
    weasyprint.HTML(string=html).write_pdf(response,
                                           stylesheets=[weasyprint.CSS(
                                               settings.STATIC_ROOT + 'css/pdf.css')])
    return response
```

Это представление для создания счета-фактуры PDF для заказа. Мы используем декоратор `staff_member_required`, чтобы убедиться, что только пользователи могут получить доступ к этому представлению. Мы получаем объект `order` с данным ID, и мы используем функцию `render_to_string()`, предоставленную Django для отрисовывания `orders/order/pdf.html`. Отображаемый HTML сохраняется в переменной `html`. Затем мы создаем новый объект `HttpResponse`, определяющий тип содержимого `application/pdf` и включающий заголовок `Content-Disposition`, чтобы указать имя файла. Мы используем WeasyPrint для создания PDF-файла из отображаемого HTML-кода и записи файла в объект `HttpResponse`. Мы используем статический файл `css/pdf.css`, чтобы добавить

стили CSS в сгенерированный PDF-файл. Мы загружаем его по локальному пути, используя параметр `STATIC_ROOT`. Наконец, мы возвращаем сгенерированный ответ.

Если вам не хватает стилей CSS, не забудьте скопировать статические файлы, расположенные в папке `static/` каталога `shop` в то же место вашего проекта.

Поскольку нам нужно использовать параметр `STATIC_ROOT`, мы должны добавить его в наш проект. Это путь проекта для размещения статических файлов. Откройте файл `settings.py` проекта `myshop` и добавьте следующий параметр:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

Затем выполните следующую команду:

```
python manage.py collectstatic
```

Вы должны увидеть вывод, который заканчивается следующим:

```
120 static files copied to 'code/myshop/static'.
```

Команда `collectstatic` копирует все статические файлы из ваших приложений в каталог, определенный в настройке `STATIC_ROOT`. Это позволяет каждому приложению предоставлять свои собственные статические файлы, используя каталог `static/`, содержащий их. Вы также можете предоставить дополнительные источники статических файлов в настройке `STATICFILES_DIRS`. Все каталоги, указанные в списке `STATICFILES_DIRS`, также будут скопированы в каталог `STATIC_ROOT` при выполнении `collectstatic`. Всякий раз, когда вы снова выполняете `collectstatic`, вас спросят, хотите ли вы переопределить существующие статические файлы.

Примечание переводчик: если у вас перестали отображаться правильные стили css, то скорее всего django загружает другой файл **base.css**, например предназначенный для сайта администрирования, а не файл с таким же именем для приложения **shop**. Для решения данной проблемы можно переименовать файл **base.css** со стилями для приложения **shop** например в **base_shop.css** и в файле шаблонов **shop/templates/shop/base.html** изменить строку загрузки стилей css с
href="`{% static 'css/base.css' %}`" на
href="`{% static 'css/base_shop.css' %}`" соответственно.

Измените файл `urls.py` в каталоге приложения `orders` и добавьте к нему следующий паттерн URL:

```
urlpatterns = [
    # ...
    path('admin/order/<int:order_id>/pdf/',
        views.admin_order_pdf,
        name='admin_order_pdf'),
]
```

Теперь мы можем отредактировать admin страницу отображения списка для модели `Order`, чтобы добавить ссылку на файл PDF для каждого результата. Измените файл `admin.py` в приложении `orders` и добавьте следующий код выше класса

`OrderAdmin`:

```
def order_pdf(obj):
    return mark_safe('<a href="{}">PDF</a>'.format(
        reverse('orders:admin_order_pdf', args=[obj.id])))
order_pdf.short_description = 'Invoice'
```

Если вы укажете атрибут `short_description`, Django будет использовать его для имени столбца.

Добавьте `order_pdf` в атрибут `list_display` класса `OrderAdmin` следующим образом:

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id',
                    # ...
                    order_detail,
                    order_pdf]
```

Откройте <http://127.0.0.1:8000/admin/orders/order/> в вашем браузере.
Каждая строка должна теперь включать ссылку на файл в
формате PDF следующим образом:

UPDATED	ORDER DETAIL	INVOICE
Feb. 11, 2018, 3:17 p.m.	View	PDF

Нажмите на PDF ссылку для любого заказа. Вы должны
увидеть сгенерированный PDF-файл, похожий на следующий,
для заказов, которые еще не были оплачены:

My Shop

Invoice no. 16

Feb 01, 2018

Bill to

Antonio Mele
antonio.mele@gmail.com
Jazz Street
28033, Madrid

Items bought

Product	Price	Quantity	Cost
Green tea	\$30	1	\$30
Total			\$30

PENDING PAYMENT

Для оплаченых заказов вы увидите следующий файл PDF:

My Shop

Invoice no. 19

Feb 06, 2018

Bill to

Antonio Melé
antonio.mele@gmail.com
Jazz Street
28027, Madrid

Items bought

Product	Price	Quantity	Cost
Tea powder	\$21.2	1	\$21.2
Total			\$21.2



Отправка файлов PDF по электронной почте

В случае если платеж будет успешным, мы отправим автоматическое электронное письмо нашим клиентам, включая сгенерированный счет в формате PDF. Откройте файл `views.py` приложения `payment` и добавьте в него следующие импорты:

```
from django.template.loader import render_to_string
from django.core.mail import EmailMessage
from django.conf import settings
import weasyprint
from io import BytesIO
```

Затем в представлении `payment_process` добавьте следующий код после строки `order.save()` с тем же уровнем отступов, как показано ниже:

```
def payment_process(request):
    # ...
    if request.method == 'POST':
        # ...
        if result.is_success:
            # ...
            order.save()
            # create invoice e-mail
            subject = 'My Shop - Invoice no. {}'.format(order.id)
            message = 'Please, find attached the invoice for your recent\
purchase.'
            email = EmailMessage(subject,
                                  message,
                                  'admin@myshop.com',
                                  [order.email])
            # generate 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)
        # attach PDF file
        email.attach('order_{}.pdf'.format(order.id),
                     out.getvalue(),
                     'application/pdf')
        # send e-mail
        email.send()

        return redirect('payment:done')
    else:
        return redirect('payment:canceled')
else:
    # ...
```

Мы используем класс `EmailMessage`, предоставляемый Django для создания объекта `email`. Затем мы преобразуем шаблон в переменную `html`. Мы создаем PDF-файл из визуализируемого шаблона и выводим его в экземпляр `BytesIO`, который является буфером байтов в памяти. Затем мы присоединяем сгенерированный PDF-файл к объекту `EmailMessage`, используя метод `attach()`, включая содержимое буфера `out` и, наконец, мы отправляем электронное письмо.

Не забудьте настроить параметры SMTP в файле проекта `settings.py` для отправки электронной почты. Вы можете обратиться к [глава 2, Улучшение вашего блога с помощью расширения функционала](#), чтобы увидеть рабочий пример конфигурации SMTP.

Теперь вы можете инициализировать новый процесс оплаты, чтобы получить счет-фактуру PDF на свой адрес электронной почты.

Резюме

В этой главе вы интегрировали платежный шлюз в свой проект. Вы настроили административный сайт Django и научились динамически генерировать файлы CSV и PDF.

В следующей главе вы познакомитесь с интернационализацией и локализацией проектов Django. Вы также создадите купонную систему и создадите механизм рекомендации товаров.

Глава 9

Расширение магазина

В предыдущей главе вы узнали, как интегрировать платежный шлюз в свой магазин. Вы также узнали, как создавать файлы CSV и PDF. В этой главе вы добавите в свой магазин купонную систему. Вы узнаете, как работают интернационализация и локализация, и вы создадите механизм рекомендаций.

В этой главе будут рассмотрены следующие вопросы:

- Создание купонной системы для применения скидок
- Добавление интернационализации к вашему проекту
- Использование Rosetta для управления переводами
- Перевод моделей с использованием django-parler
- Построение механизма рекомендаций по продуктам

Создание системы купонов

Многие интернет-магазины выдают купоны клиентам, которые могут быть применены для получения скидки при покупках. Онлайн-купон обычно состоит из кода, который предоставляется пользователям и он действительный на протяжении определенного интервала времени. Код может использоваться один или несколько раз.

Мы собираемся создать купонную систему для нашего магазина. Наши купоны будут действительны для клиентов, которые их используют в определенный период времени. Купоны не будут иметь никаких ограничений в отношении количества раз, когда они могут быть погашены, и они будут применяться к общей стоимости корзины покупок. Для этой функции нам нужно будет создать модель для хранения купонного кода, действительного временного интервала и скидки для применения.

Создайте новое приложение в проекте `myshop`, используя следующую команду:

```
python manage.py startapp coupons
```

Отредактируйте файл `settings.py` проекта `myshop` и добавьте приложение в настройку `INSTALLED_APPS` следующим образом:

```
INSTALLED_APPS = [  
    # ...  
    'coupons.apps.CouponsConfig',  
]
```

Новое приложение теперь активировано в нашем проекте

Django.

Построение модели купонов

Начнем с создания модели `Coupon`. Откройте файл `models.py` приложения `coupons` и добавьте к нему следующий код:

```
from django.db import models
from django.core.validators import MinValueValidator, \
    MaxValueValidator

class Coupon(models.Model):
    code = models.CharField(max_length=50,
                           unique=True)
    valid_from = models.DateTimeField()
    valid_to = models.DateTimeField()
    discount = models.IntegerField(
        validators=[MinValueValidator(0),
                   MaxValueValidator(100)])
    active = models.BooleanField()

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

Это модель, которую мы собираемся использовать для хранения купонов. Модель `Coupon` содержит следующие поля:

- `code`: Код, который пользователь должен ввести, чтобы применить купон к своей покупке.
- `valid_from`: Значение даты и времени, которое указывает, когда купон становится действительным.
- `valid_to`: Значение datetime, указывающее, когда купон становится недействительным.
- `discount`: Ставка скидок (это процент, поэтому он

принимает значения от 0 до 100). Мы используем валидаторы для этого поля, чтобы ограничить минимальные и максимально допустимые значения.

- `active`: Логическое значение, указывающее, активен ли купон.

Выполните следующую команду для генерации начальной миграции для приложения `coupons`:

```
python manage.py makemigrations
```

Вывод должен включать следующие строки:

```
Migrations for 'coupons':  
  coupons/migrations/0001_initial.py:  
    - Create model Coupon
```

Затем мы выполняем следующую команду для применения миграции:

```
python manage.py migrate
```

Вы должны увидеть вывод, который включает следующую строку:

```
Applying coupons.0001_initial... OK
```

Миграции теперь применяются к базе данных. Давайте добавим модель `Coupon` на сайт администрирования. Откройте файл `admin.py` приложения `coupons` и добавьте к нему следующий код:

```
from django.contrib import admin
```

```
from .models import Coupon

class CouponAdmin(admin.ModelAdmin):
    list_display = ['code', 'valid_from', 'valid_to',
                   'discount', 'active']
    list_filter = ['active', 'valid_from', 'valid_to']
    search_fields = ['code']
admin.site.register(Coupon, CouponAdmin)
```

Модель `Coupon` теперь зарегистрирована на сайте администрации. Убедитесь, что ваш локальный сервер работает `python manage.py runserver`. Откройте <http://127.0.0.1:8000/admin/coupons/coupon/add/> в своем браузере. Вы должны увидеть следующую форму:

Django administration

WELCOME, ADMIN. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Coupons > Coupons > Add coupon

Add coupon

Code:

Valid from:

Date: Today |

Time: Now |

Note: You are 1 hour ahead of server time.

Valid to:

Date: Today |

Time: Now |

Note: You are 1 hour ahead of server time.

Discount:

Active

[Save and add another](#)

[Save and continue editing](#)

SAVE

Заполните форму, чтобы создать новый купон, действительный для текущей даты, убедитесь, что вы установили флажок Active и нажмите кнопку SAVE.

Применение купона к корзине покупок

Мы можем хранить новые купоны и делать запросы для получения существующих купонов. Теперь нам нужно, чтобы клиенты применяли купоны для своих покупок.

Функциональность применения купона будет следующей:

1. Пользователь добавляет продукты в корзину покупок.
2. Пользователь может ввести код купона в форме, отображаемой на странице подробной информации о корзине покупок.
3. Когда пользователь вводит код купона и отправляет форму, мы ищем существующий купон с данным кодом, который в настоящее время действителен. Мы должны проверить, соответствует ли код купона тому, который введен пользователем, атрибут `active` равен `True`, и что текущие дата-время находятся между `valid_from` и `valid_to`.
4. Если купон найден, мы сохраняем его в сессии пользователя и показываем корзину, включая примененную к ней скидку и обновленную общую сумму.
5. Когда пользователь размещает заказ, мы сохраняем купон в данном заказе.

Создайте новый файл внутри каталога приложений `coupons` и назовите его `forms.py`. Добавьте к нему следующий код:

```
from django import forms

class CouponApplyForm(forms.Form):
    code = forms.CharField()
```

Это форма, которую мы собираемся использовать для ввода пользователем купонного кода. Измените файл `views.py` внутри приложения `coupons` и добавьте к нему следующий 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` проверяет купон и сохраняет его в сессии пользователя. Мы применяем декоратор `require_POST` к этому представлению, чтобы ограничить его запросами `POST`. В представлении мы выполняем следующие задачи:

1. Мы создаем экземпляр формы `CouponApplyForm`, используя введенные данные, и проверяем, что форма действительна.
2. Если форма действительна, мы получаем код введенный

пользователем в форму из словаря `cleaned_data`. Мы пытаемся получить объект `Coupon` с заданным кодом. Мы используем поиск `iexact`, чтобы выявить точное совпадение с учетом регистра. Купон должен быть активен в настоящий момент (`active=True`) и действителен для текущего времени. Мы используем функцию Django `timezone.now()` для получения текущей даты и времени, и мы сравниваем ее с полями `valid_from` и `valid_to`, выполняющими `lte` (меньше или равно) и `gte` (больше или равно) поиски соответственно.

3. Мы сохраняем идентификатор купона в сессии пользователя.
4. Мы перенаправляем пользователя на URL `cart_detail`, чтобы отобразить корзину с применяемыми купоном скидками.

Нам нужен паттерн URL для представления `coupon_apply`. Создайте новый файл внутри каталога приложения `coupons` и назовите его `urls.py`. Добавьте к нему следующий код:

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

app_name = 'coupons'

urlpatterns = [
    path('apply/', views.coupon_apply, name='apply'),
]
```

Затем отредактируйте основной `urls.py` проекта `myshop` и укажите паттерн URL-адресов `coupons` следующим образом:

```
urlpatterns = [
    # ...
]
```

```
    path('coupons/', include('coupons.urls', namespace='coupons')),
    path('', include('shop.urls', namespace='shop')),
]
```

Не забудьте разместить этот паттерн перед `shop.urls`.

Теперь отредактируйте файл `cart.py` Приложения `cart` включив следующий импорт:

```
from coupons.models import Coupon
```

Добавьте следующий код в конец метода `__init__()` класса `Cart` для инициализации купона из текущей сессии:

```
class Cart(object):
    def __init__(self, request):
        # ...
        # сохранить текущий применяемый купон
        self.coupon_id = self.session.get('coupon_id')
```

В этом коде мы пытаемся получить ключ `coupon_id` из текущего сессии и сохранить его значение в объекте `cart`. Добавьте следующие методы в объект `cart`:

```
class Cart(object):
    # ...
    @property
    def coupon(self):
        if self.coupon_id:
            return Coupon.objects.get(id=self.coupon_id)
        return None

    def get_discount(self):
        if self.coupon:
            return (self.coupon.discount / Decimal('100')) \
                * self.get_total_price()
        return Decimal('0')

    def get_total_price_after_discount(self):
        return self.get_total_price() - self.get_discount()
```

Эти методы заключаются в следующем:

- `coupon()`: Мы определяем этот метод как свойство. Если корзина содержит атрибут `coupon_id`, возвращается объект `Coupon` с данным идентификатором.
- `get_discount()`: Если в корзине есть купон, мы получаем его дисконтную ставку и возвращаем сумму, подлежащую вычету из общего объема корзины.
- `get_total_price_after_discount()`: Мы возвращает общую сумму корзины после вычитания суммы, возвращаемой методом `get_discount()`.

Класс `Cart` теперь готов обрабатывать купон, применяемый к текущему сеансу, и применять соответствующую скидку.

Включим систему купонов в подробное представление корзины. Измените файл `views.py` приложения `cart` и добавьте следующий импорт вверху файла:

```
from coupons.forms import CouponApplyForm
```

Далее внизу отредактируйте представление `cart_detail` и добавьте к нему новую форму следующим образом:

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
            initial={'quantity': item['quantity'],
                      'update': True})
    coupon_apply_form = CouponApplyForm()

    return render(request,
                  'cart/detail.html',
```

```
{'cart': cart,
'coupon_apply_form': coupon_apply_form})
```

Откройте шаблон `cart/detail.html` приложения `cart` и найдите следующие строки:

```
<tr class="total">
<td>Total</td>
<td colspan="4"></td>
<td class="num">${{ cart.get_total_price }}</td>
</tr>
```

Замените их следующим:

```
{% 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 код для отображения необязательного купона и его учетной ставки. Если в корзине есть купон, мы показываем первую строку, включая общую сумму тележки в качестве промежуточного итога. Затем мы используем вторую строку

для отображения текущего купона, применяемого к корзине. Наконец, мы показываем общую цену, включая любую скидку, вызывая метод

`get_total_price_after_discount()` объекта `cart`.

В том же файле введите следующий код после `</table>` HTML тега:

```
<p>Apply a coupon:</p>
<form action="{% url "coupons:apply" %}" method="post">
    {{ coupon_apply_form }}
    <input type="submit" value="Apply">
    {% csrf_token %}
</form>
```

Это отобразит форму, чтобы ввести код купона и применить его к текущей корзине.

Откройте `http://127.0.0.1:8000/` в своем браузере, добавьте товар в корзину, затем примените созданный вами купон, введя его код в форме. Вы должны увидеть, что корзина отображает скидку купона следующим образом:

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Tea powder	<input type="button" value="1"/> <input type="button" value="Update"/>	Remove	\$21.2	\$21.2
Subtotal					\$21.20
"SUMMER" coupon (10% off)					- \$2.12
Total					\$19.08

Apply a coupon:

Code:

[Continue shopping](#)

[Checkout](#)

Давайте добавим купон для следующего этапа процесса покупки. Откройте шаблон `orders/order/create.html` приложения заказов и найдите следующие строки:

```
<ul>
  {% for item in cart %}
    <li>
      {{ item.quantity }}x {{ item.product.name }}
      <span>${{ item.total_price }}</span>
    </li>
```

```
{% endfor %}  
</ul>
```

Замените их следующим 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>- ${{ cart.get_discount|floatformat:"2" }}</span>  
        </li>  
    {% endif %}  
</ul>
```

В окончательном заказе теперь должен быть указан использованный купон, если таковой имеется. Теперь найдите следующую строку:

```
<p>Total: ${{ cart.get_total_price }}</p>
```

Замените ее следующим:

```
<p>Total: ${{ cart.get_total_price_after_discount|floatformat:"2" }}</p>
```

Таким образом, общая цена будет также рассчитываться путем применения скидки купона.

Откройте <http://127.0.0.1:8000/orders/create/> в вашем браузере. Вы должны убедиться, что в заказе указан применяемый купон:

Your order

- | | |
|----------------------|----------|
| • 1x Tea powder | \$21.20 |
| • "SUMMER" (10% off) | - \$2.12 |

Total: \$19.08

Теперь пользователи могут применять купоны в своей корзине покупок. Тем не менее, нам по-прежнему необходимо хранить информацию о купонах в заказе, который создается когда пользователи проверяют корзину.

Применение купонов к заказам

Мы собираемся хранить купон, который был применен к каждому заказу. Во-первых, нам нужно изменить модель `Order`, чтобы сохранить связанный объект `Coupon`, если таковой имеется.

Измените файл `models.py` приложения `orders` и добавьте в него следующие импорты:

```
from decimal import Decimal
from django.core.validators import MinValueValidator, \
    MaxValueValidator
from coupons.models import Coupon
```

Затем добавьте следующие поля в модель `order`:

```
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 нам хранить дополнительный купон для заказа и процент скидки, применяемый по купону. Скидка хранится в соответствующем объекте `Coupon`, но мы включаем его в модель `Order`, чтобы сохранить его, если купон изменен или удален. Мы устанавливаем `on_delete` в `models.SET_NULL`, чтобы, если купон удаляется, в поле `coupon` было установлено значение `null`.

Нам нужно создать миграцию, чтобы включить новые поля модели `Order`. Выполните следующую команду из командной строки:

```
python manage.py makemigrations
```

Вы должны увидеть следующий вывод:

```
Migrations for 'orders':
  orders/migrations/0003_auto_20180307_2202.py:
    - Add field coupon to order
    - Add field discount to order
```

Примените новую миграцию с помощью следующей команды:

```
python manage.py migrate orders
```

Вы должны увидеть подтверждение о том, что новая миграция была применена. Изменения полей модели `Order` теперь синхронизируются с базой данных.

Вернитесь к файлу `models.py` и измените метод `get_total_cost()` модели `Order` следующим образом:

```
class Order(models.Model):
    # ...
    def get_total_cost(self):
        total_cost = sum(item.get_cost() for item in self.items.all())
        return total_cost - total_cost * \
            (self.discount / Decimal('100'))
```

Метод `get_total_cost()` модели `Order` теперь учитывает применяемую скидку, если она есть.

Измените файл `views.py` приложения `orders` и отредактируйте представление `order_create`, чтобы сохранить соответствующий купон и его скидку при создании нового заказа. Найдите

следующую строку:

```
order = form.save()
```

Замените ее следующим:

```
order = form.save(commit=False)
if cart.coupon:
    order.coupon = cart.coupon
    order.discount = cart.coupon.discount
order.save()
```

В новом коде мы создаем объект `Order`, используя метод `save()` формы `OrdercreateForm`. Мы не сохраняем его в базе данных, используя `commit=False`. Если корзина содержит купон, мы сохраним соответствующий купон и скидку, которая была применена. Затем мы сохраняем объект `order` в базе данных.

Запустите сервер разработки командой `python manage.py runserver`.

Откройте `http://127.0.0.1:8000/` в вашем браузере и совершите покупку с помощью купона, который вы создали. Когда вы закончите успешную покупку, вы можете перейти на `http://127.0.0.1:8000/admin/orders/order/` и проверить, что объект заказа содержит купон и примененную скидку следующим образом:

Braintree id: d31natz6

Coupon: SUMMER ↴ ⚡ + ✘

Discount: 10 ⌂

ORDER ITEMS			
PRODUCT	PRICE	QUANTITY	DELETE?
21			
3 Q Tea powder	21,2 ⌂	1 ⌂	<input type="checkbox"/>

Вы также можете изменить шаблон деталей заказа сайта администратора и заказа в формате PDF, чтобы отобразить примененный купон так же, как мы сделали для корзины.

Теперь мы можем добавить интернационализацию в наш проект.

Добавление интернационализации и локализации

Django предлагает полную поддержку интернационализации и локализации. Он позволяет вам перевести ваше приложение на несколько языков и применить языковое форматирование для дат, времени, чисел и часовых поясов. Давайте уточним разницу между интернационализацией и локализацией.

Интернационализация (часто сокращается до **i18n**) - это процесс адаптации программного обеспечения для потенциального использования разных языков и локалей, чтобы оно не было привязано к конкретному языку или локали. **Локализация** (сокращенно **l10n**) - это процесс фактического перевода программного обеспечения и адаптации его к определенному языку. Сам Django переведен на более чем 50 языков, используя его фреймворк интернационализации.

Интернационализация с помощью Django

Фреймворк интернационализации позволяет легко маркировать строки для перевода как в коде Python, так и в ваших шаблонах. Он использует набор инструментов GNU gettext для создания и управления файлами сообщений. Файл **message** - это простой текстовый файл, представляющий язык. Он содержит часть или все строки перевода, найденные в вашем приложении, и их соответствующие переводы для какого либо языка. Файлы сообщений имеют расширение `.po`.

Как только перевод сделан, файлы сообщений компилируются для быстрого доступа к переведенным строкам. Скомпилированные файлы переводов имеют расширение `.mo`.

Настройки интернационализации и локализации

Django предоставляет несколько настроек для интернационализации. Наиболее важными являются следующие настройки:

- `USE_I18N`: Логическое значение, указывающее, включена ли система перевода Django. По умолчанию `True`.
- `USE_L10N`: Логическое значение, указывающее, включено ли локализованное форматирование. Когда активно, локализованные форматы используются для представления дат и цифр. Идет как `False` по умолчанию.
- `USE_TZ`: Логическое значение, определяющее, являются ли данные дата-время временной зоной. Когда вы создаете проект с помощью команды `startproject`, это значение равно `True`.
- `LANGUAGE_CODE`: Код языка по умолчанию для проекта. Это стандартный формат идентификатора языка, например, '`en-us`' для американского английского языка, или '`en-gb`' для английского языка. Для установки этого параметра требуется чтобы `use_i18n` было равно `True`. Вы можете найти список идентификаторов языка в

<http://www.i18nguy.com/unicode/language-identifiers.html>.

- `LANGUAGES`: Кортеж, содержащий доступные языки для проекта. Они состоят из двух кортежей **кода языка и имени языка**. Вы можете просмотреть список доступных языков в `django.conf.global_settings`. Когда вы выбираете языки, на которых будет доступен ваш сайт, вы устанавливаете `LANGUAGES` подмножество этого списка.
- `LOCALE_PATHS`: Список каталогов, в которых Django ищет файлы сообщений, содержащие переводы для этого проекта.
- `TIME_ZONE`: Страна, представляющая часовой пояс для проекта. Это значение устанавливается в 'utc' при создании нового проекта с помощью команды `startproject`. Вы можете установить его в любой другой часовой пояс, например 'Europe/Madrid'.

Это некоторые из параметров интернационализации и локализации. Вы можете найти полный список на <https://docs.djangoproject.com/en/2.0/ref/settings/#globalization-i18n-l10n>.

Команды управления интернационализацией

Django включает следующие команды для управления переводами:

- `makemessages`: Выполняется над деревом исходников, чтобы найти все строки, помеченные для перевода, и создает или обновляет файлы сообщений `.po` в каталоге `locale`. Для каждого языка создается один файл `.po`.
- `compilemessages`: Компилирует существующие файлы сообщений `.po` в файлы `.mo`, которые используются для извлечения переводов.

Вам понадобится инструментарий `gettext`, который сможет создавать, обновлять и компилировать файлы сообщений. Большинство дистрибутивов Linux включают в себя набор инструментов `gettext`. Если вы используете macOS X, возможно, самый простой способ установить его через Homebrew на <https://brew.sh/> с помощью команды `brew install gettext`. Вам также может потребоваться принудительно связать его командой `brew link gettext --force`. Для Windows выполните следующие действия: <https://docs.djangoproject.com/en/2.0/topics/i18n/translation/#gettext-on-windows>.

Как добавить переводы в проект Django

Давайте посмотрим на процесс интернационализации нашего проекта. Нам нужно будет сделать следующее:

1. Отметить строки для перевода в нашем Python-коде и наших шаблонах
2. Запустить команду `makemessages` для создания или обновления файлов сообщений, содержащих все строки перевода из нашего кода
3. Перевести строки, содержащиеся в файлах сообщений, и скомпилировать их с помощью команды управления `compilemessages`

Как Django определяет текущий язык

Django поставляется с промежуточным программным обеспечением, которое определяет текущий язык на основе данных запроса. Это промежуточное ПО `LocaleMiddleware`, которое находится в `django.middleware.locale.LocaleMiddleware` выполняет следующие задачи:

1. Если вы используете `i18_patterns`, то есть используете переведенные шаблоны URL, он ищет префикс языка в запрошенном URL для определения текущего языка.
2. Если языковой префикс не найден, он ищет существующий `LANGUAGE_SESSION_KEY` в сессии текущего пользователя.
3. Если язык не задан в сеансе, он ищет существующий файл cookie с текущим языком. Пользовательское имя для этого файла cookie может быть предоставлено в настройке `LANGUAGE_COOKIE_NAME`. По умолчанию имя для этого файла cookie `django_language`.
4. Если cookie не найден, он ищет HTTP-заголовок запроса `Accept-Language`.
5. Если заголовок `Accept-Language` не указывает язык, Django использует язык, определенный в настройке `LANGUAGE_CODE`.

По умолчанию Django будет использовать язык, определенный в параметре `LANGUAGE_CODE`, если вы не используете `LocaleMiddleware`. Процесс, описанный здесь, применяется только при

использовании этого промежуточного программного обеспечения.

Подготовка нашего проекта для интернационализации

Давайте подготовим наш проект для использования на разных языках. Мы собираемся создать английскую и испанскую версию для нашего магазина. Измените файл `settings.py` вашего проекта и добавьте к нему следующие настройки `LANGUAGES`. Поместите его рядом с настройкой `LANGUAGE_CODE`:

```
LANGUAGES = (
    ('en', 'English'),
    ('es', 'Spanish'),
)
```

Параметр `LANGUAGES` содержит два кортежа, состоящих из кода языка и имени. Коды языков могут быть специфичными для локали, например `en-us` или `en-gb`, или общие, например `en`. С помощью этой настройки мы укажем, что наше приложение будет доступно только на английском и испанском языках. Если мы не определим пользовательский параметр `LANGUAGES`, сайт будет доступен на всех языках, на которые Django переведен.

Сделайте настройку `LANGUAGE_CODE` следующей:

```
LANGUAGE_CODE = 'en'
```

Добавте `'django.middleware.locale.LocaleMiddleware'` к настройке `MIDDLEWARE`. Убедитесь, что это промежуточное программное обеспечение находится после `SessionMiddleware`, потому что в `LocaleMiddleware` необходимо использовать данные сессии. Оно также должно быть помещено перед `CommonMiddleware`, потому что последнему

нужен активный язык для разрешения запрошенного URL.
MIDDLEWARE должно выглядеть следующим образом:

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.locale.LocaleMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    # ...  
]
```

Порядок классов промежуточного программного обеспечения очень важен, поскольку каждое промежуточное программное обеспечение может зависеть от данных, установленных другим промежуточным программным обеспечением, выполненным ранее. Промежуточное ПО применяется для запросов в порядке появления в MIDDLEWARE и в обратном порядке для ответов.

Создайте следующую структуру каталогов внутри основного каталога проекта, рядом с файлом `manage.py`:

```
locale/  
  en/  
  es/
```

Каталог `locale` – это место, где будут находиться файлы сообщений для вашего приложения. Отредактируйте файл `settings.py` еще раз и добавьте в него следующие настройки:

```
LOCALE_PATHS = (  
    os.path.join(BASE_DIR, 'locale/'),  
)
```

Параметр `LOCALE_PATHS` указывает каталоги, в которых Django должен искать файлы перевода. Сначала появляются локальные пути, которые имеют наивысший приоритет.

Когда вы используете команду `makemessages` в каталоге проекта, файлы сообщений будут сгенерированы в созданном нами каталоге `locale/`. Однако для приложений, которые содержат свой собственный каталог `locale/`, файлы сообщений будут

сгенерированы в нем.

Перевод кода Python

Чтобы перевести литералы в коде Python, вы можете пометить строки для перевода, используя функцию `gettext()`, включенную в `django.utils.translation`. Эта функция преобразует сообщение и возвращает строку. Соглашение заключается в том, чтобы импортировать эту функцию как более короткий псевдоним с именем `_` (символ подчеркивания).

Вы можете найти всю документацию о переводах на <https://docs.djangoproject.com/en/2.0/topics/i18n/translation/>.

Стандартные переводы

Следующий код показывает, как отметить строку для перевода:

```
from django.utils.translation import gettext as _
output = _('Text to be translated.')
```

Ленивый (Lazy) перевод

Django включает версии **lazy** для всех своих функций перевода, которые имеют суффикс `_lazy()`. При использовании ленивых функций строки передаются при достижении значения, а не при вызове функции (поэтому этот перевод называется **ленивый**). Функции ленивого перевода полезны, когда строки, помеченные для перевода, находятся в путях, которые выполняются при загрузке модулей.

*Используя `gettext_lazy()` вместо `gettext()`, строки передаются при обращении к значению, а не когда вызывается функция. Django предлагает **lazy** версию для всех функций перевода.*

Переводы включаемых переменных

Строки, помеченные для перевода, могут включать заполнители для включения переменных в переводы. Следующий код является примером строки перевода с заполнителем:

```
from django.utils.translation import gettext as _
month = _('April')
day = '14'
output = _('Today is %(month)s %(day)s') % {'month': month,
                                             'day': day}
```

С помощью заполнителей вы можете изменить порядок текстовых переменных. Например, английский перевод предыдущего примера («Сегодня - 14 апреля») может быть «*Today is April 14*», а испанский - «*Hoy es 14 de Abril*». Всегда используйте интерполяцию строк вместо позиционной интерполяции, если для строки перевода имеется несколько параметров. Поступая таким образом, вы сможете изменить порядок текста заполнителя.

Множественные формы в переводах

Для множественных форм вы можете использовать `ngettext()` и `ngettext_lazy()`. Эти функции переводят единичные и множественные формы в зависимости от аргумента, который указывает количество объектов. В следующем 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 литералов в нашем Python-коде, пришло время применить переводы к нашему проекту.

Перевод вашего кода

Откройте файл `settings.py` вашего проекта, импортируйте функцию `gettext_lazy()` и измените настройку `LANGUAGES` следующим образом, чтобы перевести имена языков:

```
from django.utils.translation import gettext_lazy as _

LANGUAGES = (
    ('en', _('English')),
    ('es', _('Spanish')),
)
```

Здесь мы используем функцию `gettext_lazy()` вместо `gettext()`, чтобы избежать циклического импорта, таким образом переводя имена языков при доступе к ним.

Откройте оболочку и запустите следующую команду из каталога проекта:

```
django-admin makemessages --all
```

Вы должны увидеть следующий результат:

```
processing locale es
processing locale en
```

Посмотрите каталог `locale/`. Вы должны увидеть следующую файловую структуру:

```
en/
  LC_MESSAGES/
    django.po
```

```
| es/
|   LC_MESSAGES/
|     django.po
```

Для каждого языка создан файл сообщения .po. Откройте es/LC_MESSAGES/django.po с помощью текстового редактора. В конце файла вы должны увидеть следующее:

```
| #: myshop/settings.py:117
| msgid "English"
| msgstr ""

| #: myshop/settings.py:118
| msgid "Spanish"
| msgstr ""
```

Каждой строке перевода предшествует комментарий, содержащий подробную информацию о файле и строке, в которой он был найден. Каждый перевод включает две строки:

- `msgid`: Стока перевода, как она отображается в исходном коде.
- `msgstr`: Перевод языка, который по умолчанию пуст. Здесь вы должны ввести фактический перевод для данной строки.

Заполните `msgstr` переводы для данной строки `msgid` следующим образом:

```
| #: myshop/settings.py:117
| msgid "English"
| msgstr "Inglés"

| #: myshop/settings.py:118
| msgid "Spanish"
| msgstr "Español"
```

Сохраните измененный файл сообщения, откройте консоль и выполните следующую команду:

```
django-admin compilemessages
```

Если все будет хорошо, вы должны увидеть результат следующим образом:

```
processing file django.po in myshop/locale/en/LC_MESSAGES
processing file django.po in myshop/locale/es/LC_MESSAGES
```

Вывод дает вам информацию о файлах сообщений, которые компилируются. Еще раз просмотрите каталог `locale` проекта `myshop`. Вы должны увидеть следующие файлы:

```
en/
    LC_MESSAGES/
        django.mo
        django.po
es/
    LC_MESSAGES/
        django.mo
        django.po
```

Вы можете видеть, что скомпилированный файл сообщения `.mo` был сгенерирован для каждого языка.

Мы сами перевели названия языков. Теперь давайте переведем имена полей модели, которые отображаются на сайте. Отредактируйте файл `models.py` приложения `orders` и добавьте имена, помеченные для перевода полей 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)
# ...
```

Мы добавили имена для полей, которые отображаются, когда пользователь оформляет новый заказ. Это `first_name`, `last_name`, `email`, `address`, `postal_code` И `city`. Помните, что вы также можете использовать атрибут `verbose_name`, чтобы назвать поля.

Создайте следующую структуру каталогов внутри каталога приложений `orders`:

```
locale/
  en/
  es/
```

Создав каталог `locale`, строки перевода этого приложения будут сохранены в файле сообщения в этом каталоге вместо основного файла сообщений. Таким образом, вы можете создавать отдельные файлы перевода для каждого приложения.

Откройте консоль из каталога проекта и выполните следующую команду:

```
django-admin makemessages --all
```

Вы должны увидеть следующий результат:

```
processing locale es
processing locale en
```

Откройте файл `locale/es/LC_MESSAGES/django.po` приложения `order`, используя текстовый редактор. Вы увидите строки перевода для модели `Order`. Заполните следующие `msgstr` переводы для заданных `msgid` строк:

```
#: orders/models.py:10
msgid "first name"
msgstr "nombre"

#: orders/models.py:11
msgid "last name"
msgstr "apellidos"

#: orders/models.py:12
msgid "e-mail"
msgstr "e-mail"

#: orders/models.py:13
msgid "address"
msgstr "dirección"

#: orders/models.py:14
msgid "postal code"
msgstr "código postal"

#: orders/models.py:15
msgid "city"
msgstr "ciudad"
```

После того, как вы закончите добавлять переводы, сохраните файл.

Помимо текстового редактора, вы можете использовать Poedit для редактирования переводов. Poedit - это программное обеспечение для редактирования переводов, и оно использует gettext. Он доступен для Linux, Windows и MacOS X. Вы можете скачать Poedit из <https://poedit.net/>.

Давайте также переведем формы нашего проекта. `OrdercreateForm` приложения `orders` не нужно переводить, поскольку это `ModelForm` и использует атрибут `verbose_name` полей модели `Order` для меток полей формы. Мы собираемся перевести формы приложений

cart И coupons.

Откройте файл forms.py в каталоге приложения cart и добавьте атрибут label в поле quantity в CartAddProductForm, а затем пометьте это поле для перевода следующим образом:

```
from django import forms
from django.utils.translation import gettext_lazy as _

PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]

class CartAddProductForm(forms.Form):
    quantity = forms.TypedChoiceField(
        choices=PRODUCT_QUANTITY_CHOICES,
        coerce=int,
        label=_('Quantity'))
    update = forms.BooleanField(required=False,
                                initial=False,
                                widget=forms.HiddenInput)
```

Измените файл forms.py Приложения coupons и переведите форму CouponApplyForm следующим образом:

```
from django import forms
from django.utils.translation import gettext_lazy as _

class CouponApplyForm(forms.Form):
    code = forms.CharField(label=_('Coupon'))
```

Мы добавили метку в поле code и пометили его для перевода.

Перевод шаблонов

Django предлагает теги шаблона `{% trans %}` и `{% blocktrans %}` для перевода строк в шаблонах. Чтобы использовать теги перевода шаблона, вам нужно добавить `{% load i18n %}` в верхней части шаблона, чтобы загрузить их.

Тег шаблона `{% trans %}`

Тег шаблона `{% trans %}` позволяет вам пометить строку, константу или переменную для перевода. Внутри Django выполняет `gettext()` по данному тексту. Вот как отметить строку для перевода в шаблоне:

```
|  {% trans "Text to be translated" %}
```

Вы можете использовать `as` для хранения переведенного содержимого в переменной, которую вы можете использовать во всем шаблоне. Следующий пример хранит переведенный текст в переменной, называемой `greeting`:

```
|  {% trans "Hello!" as greeting %}
```

```
|  <h1>{{ greeting }}</h1>
```

Тег `{% trans %}` полезен для простых строк перевода, но он не может обрабатывать контент для перевода, который включает переменные.

Тег шаблона `{% blocktrans %}`

Тэг `{% blocktrans %}` позволяет вам помечать контент, включающий литералы и переменный контент с использованием заполнителей. В следующем примере показано, как использовать тег `{% blocktrans %}`, включая переменную `name` в содержимом для перевода:

```
{% blocktrans %}Hello {{ name }}!{% endblocktrans %}
```

Вы можете использовать `with` для включения выражений шаблонов, таких как доступ к атрибутам объектов или применение фильтров шаблонов к переменным. Для этого вам всегда нужно использовать заполнители. Вы не можете получить доступ к выражениям или атрибутам объектов внутри блока `blocktrans`. В следующем примере показано, как использовать `with` для включения атрибута объекта, к которому применяется фильтр `capfirst`:

```
{% blocktrans with name=user.name|capfirst %}
  Hello {{ name }}!
{% endblocktrans %}
```

Используйте тег `{% blocktrans %}` вместо `{% trans %}`, когда вам нужно включить переменную содержимого в вашей строке перевода.

Перевод шаблонов магазина

Отредактируйте шаблон `shop/base.html` приложения `shop`.

Убедитесь, что вы загрузили тег `i18n` в верхней части шаблона и отметили строки для перевода следующим образом:

```
{% load i18n %}  
{% load static %}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <title>  
        {% block title %}{% trans "My shop" %}{% endblock %}  
    </title>  
    <link href="{% static "css/base.css" %}" rel="stylesheet">  
</head>  
<body>  
    <div id="header">  
        <a href="/" class="logo">{% trans "My shop" %}</a>  
    </div>  
    <div id="subheader">  
        <div class="cart">  
            {% with total_items=cart|length %}  
            {% if cart|length > 0 %}  
                {% trans "Your cart" %}:  
                <a href="{% url "cart:cart_detail" %}">  
                    {% blocktrans with total_items_plural=total_items|pluralize  
                    total_price=cart.get_total_price %}  
                    {{ total_items }} item{{ total_items_plural }},  
                    ${{ total_price }}  
                {% endblocktrans %}  
                </a>  
            {% else %}  
                {% trans "Your cart is empty." %}  
            {% endif %}  
            {% endwith %}  
        </div>  
    </div>  
    <div id="content">  
        {% block content %}
```

```
    {% endblock %}  
  </div>  
</body>  
</html>
```

Обратите внимание на тег `{% blocktrans %}`, чтобы отобразить содержимое корзины. Ранее содержимое корзины было следующим:

```
{{ total_items }} item{{ total_items|pluralize }},  
${{ cart.get_total_price }}
```

Мы использовали `{% blocktrans with ... %}` для установки заполнителей для `total_items|pluralize` (здесь применяется шаблонный тег) и `cart.get_total_price` (метод объекта вызываемого здесь), что приводит к следующему:

```
{% blocktrans with total_items_plural=total_items|pluralize  
total_price=cart.get_total_price %}  
{{ total_items }} item{{ total_items_plural }},  
${{ total_price }}  
{% endblocktrans %}
```

Затем отредактируйте шаблон `shop/product/detail.html` приложения `shop` и загрузите теги `i18n` вверху, но после тега `{% extends %}`, который всегда должен быть первым тегом в шаблоне:

```
{% load i18n %}
```

Затем найдите следующую строку:

```
<input type="submit" value="Add to cart">
```

Замените ее следующим:

```
<input type="submit" value="{% trans "Add to cart" %}">
```

Теперь переведите шаблоны приложения orders.

Отредактируйте шаблон `orders/order/create.html` приложения orders и пометьте текст для перевода следующим образом:

```
{% extends "shop/base.html" %}
{% load i18n %}

{% block title %}
    {% trans "Checkout" %}
{% endblock %}

{% block content %}
<h1>{% trans "Checkout" %}</h1>

<div class="order-info">
    <h3>{% trans "Your order" %}</h3>
    <ul>
        {% for item in cart %}
            <li>
                {{ item.quantity }}x {{ item.product.name }}
                <span>${{ item.total_price }}</span>
            </li>
        {% endfor %}
        {% if cart.coupon %}
            <li>
                {% blocktrans with code=cart.coupon.code
                    discount=cart.coupon.discount %}
                    "{{ code }}" ({{ discount }}% off)
                {% endblocktrans %}
                <span>- ${{ cart.get_discount|floatformat:"2" }}</span>
            </li>
        {% endif %}
    </ul>
    <p>{% trans "Total" %}: ${{ cart.get_total_price_after_discount|floatformat:"2" }}</p>
</div>

<form action"." method="post" class="order-form">
    {{ form.as_p }}
    <p><input type="submit" value="{} trans "Place order" {}"></p>
    {% csrf_token %}
</form>
{% endblock %}
```

Взгляните на следующие файлы в коде, который сопровождает эту главу, чтобы увидеть, как строки были помечены для

перевода:

- Приложение `shop`: Шаблон `shop/product/list.html`
- Приложение `orders`: Шаблон `orders/order/created.html`
- Приложение `cart`: Шаблон `cart/detail.html`

Давайте обновим файлы сообщений, чтобы включить новые строки перевода. Откройте терминал и выполните следующую команду:

```
django-admin makemessages --all
```

Файлы `.po` находятся внутри каталога `locale` проекта `myshop`, и вы увидите, что приложение `orders` теперь содержит все строки, которые мы отметили для перевода.

Измените файлы перевода `.po` проекта и приложения `orders` и добавте испанский перевод в `msgstr`. Вы также можете использовать переведенные файлы `.po` из исходного кода, прилагаемого к этой главе.

Выполните следующую команду для компиляции файлов перевода:

```
django-admin compilemessages
```

Вы увидите следующий 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
```

Файл .mo, содержащий скомпилированные переводы, был сгенерирован для каждого файла перевода .po.

Использование интерфейса перевода Rosetta

Rosetta - стороннее приложение, которое позволяет редактировать переводы, используя тот же интерфейс, что и сайт администрирования Django. Rosetta упрощает редактирование файлов .po и обновляет скомпилированные файлы переводов. Давайте добавим его в наш проект.

Установите Rosetta через pip, используя следующую команду:

```
pip install django-rosetta==0.8.1
```

Затем добавьте 'rosetta' в параметр INSTALLED_APPS в файле settings.py вашего проекта следующим образом:

```
INSTALLED_APPS = [
    # ...
    'rosetta',
]
```

Вам нужно добавить URL-адреса Rosetta в вашу основную конфигурацию URL-адресов. Отредактируйте основной файл urls.py вашего проекта и добавьте к нему следующий паттерн URL:

```
urlpatterns = [
    # ...
    path('rosetta/', include('rosetta.urls')),
    path('', include('shop.urls', namespace='shop')),
]
```

Обязательно поместите его перед паттерном `shop.urls`, чтобы избежать нежелательного соответствия шаблону.

Откройте `http://127.0.0.1:8000/admin/` и войдите в систему с помощью суперпользователя. Затем перейдите на `http://127.0.0.1:8000/rosetta/` в своем браузере. В меню Filter нажмите THIRD PARTY, чтобы отобразите все доступные файлы сообщений, в том числе те, которые относятся к приложению `orders`. Вы должны увидеть список существующих языков следующим образом:

Rosetta

Home > Language selection

Filter: PROJECT THIRD PARTY DJANGO ALL

English

APPLICATION	PROGRESS	MESSAGES	TRANSLATED	FUZZY	OBsolete	FILE
Myshop	0%	12	0	0	0	/Users/zenx/dbe/myshop/locale/en/LC_MESSAGES/django.po
Orders	0%	13	0	0	0	/Users/zenx/dbe/myshop/orders/locale/en/LC_MESSAGES/django.po

Spanish

APPLICATION	PROGRESS	MESSAGES	TRANSLATED	FUZZY	OBsolete	FILE
Myshop	100%	12	12	0	0	/Users/zenx/dbe/myshop/locale/es/LC_MESSAGES/django.po
Orders	100%	13	13	0	0	/Users/zenx/dbe/myshop/orders/locale/es/LC_MESSAGES/django.po
Rosetta	73%	37	28	1	2	/Users/zenx/env/dbe3/lib/python3.6/site-packages/rosetta/locale/es/LC_MESSAGES/django.po

Нажмите ссылку Myshop в разделе Spanish для редактирования испанских переводов. Вы должны увидеть список строк перевода следующим образом:

Translate into Spanish

Display: UNTRANSLATED ONLY TRANSLATED ONLY FUZZY ONLY ALL

ORIGINAL	SPANISH	FUZZY OCCURRENCES(S)
Quantity	Cantidad	<input type="checkbox"/> cart/forms.py:12
Coupon	Cupón	<input type="checkbox"/> coupons/forms.py:6
English	Inglés	<input type="checkbox"/> myshop/settings.py:117
Spanish	Español	<input type="checkbox"/> myshop/settings.py:118
My shop	Mi tienda	<input type="checkbox"/> shop/templates/shop/base.html:7 shop/templates/shop/base.html:12
Your cart	Tu carro	<input type="checkbox"/> shop/templates/shop/base.html:18

Вы можете ввести переводы в столбце Spanish. Столбец OCCURRENCES(S) отображает файлы и строку кода, в которой была найдена каждая строка перевода.

Переводы, которые включают заполнители, будут выглядеть следующим образом:

```
%({total_items})s item%  
(total_items_plural)s,  
$%({total_price})s
```

```
%({total_items})s producto%  
(total_items_plural)s,  
$%({total_price})s
```



shop/templates/shop/base.html:20

Rosetta использует другой цвет фона для отображения заполнителей. Когда вы переводите контент, убедитесь, что вы держите placeholders непереведенными. Например, возьмите следующую строку:

```
%({total_items})s item%({total_items_plural})s, $%({total_price})s
```

Она переведена на испанский язык следующим образом:

```
%({total_items})s producto%({total_items_plural})s, $%({total_price})s
```

Вы можете взглянуть на исходный код, который прилагается к этой главе, чтобы использовать те же испанские переводы для вашего проекта.

Закончив редактирование переводов, нажмите кнопку `Save and translate next block`, чтобы сохранить переводы в файл `.po`. Rosetta компилирует файл сообщения при сохранении переводов, поэтому вам не нужно запускать команду `compilemessages`. Однако для записи файлов сообщений Rosetta требуется доступ на запись в каталоги `locale`. Убедитесь, что каталоги имеют соответствующие разрешения прав.

Если вы хотите, чтобы другие пользователи могли редактировать переводы, откройте <http://127.0.0.1:8000/admin/auth/group/add/> в своем браузере и создайте новую группу с именем `translators`. Затем выберите <http://127.0.0.1:8000/admin/auth/user/>, чтобы выбрать пользователей,

которым вы хотите предоставить разрешения, чтобы они могли редактировать переводы. При редактировании пользователя в разделе Permissions добавьте `translators` в группу Chosen Groups для каждого пользователя. Rosetta доступна только для суперпользователей или пользователей, принадлежащих группе `translators`.

Вы можете ознакомиться с документацией Rosetta <https://django-rosetta.readthedocs.io/en/latest/>.

Когда вы добавляете новые переводы в производственную среду, если вы обслуживаете Django с помощью реального веб-сервера, вам придется перезагрузить свой сервер после выполнения команды `compilemessages` или после сохранения переводов с Rosetta, чтобы изменения вступили в силу.

Неточные (Fuzzy) переводы

Возможно, вы заметили, что в Rosetta есть столбец FUZZY. Это не функция Rosetta; она предоставляется gettext. Если флаг `fuzzy` активен для перевода, он не будет включен в скомпилированные файлы сообщений. Этот флаг отмечает строки перевода, которые должны быть пересмотрены переводчиком. Когда файлы `.po` обновляются с новыми строками перевода, возможно, что некоторые строки перевода автоматически помечены как неточные. Это происходит, когда `gettext` находит некоторые `msgid`, которые были слегка изменены. `gettext` сопоставляет его с тем, что, по его мнению, является старым переводом, и помещает его как неточные для пересмотра. Затем переводчик должен просмотреть неточные переводы, удалить `fuzzy` флаг и скомпилировать файл перевода снова.

Шаблоны URL для интернационализации

Django предлагает возможности интернационализации для URL-адресов. Он включает две основные функции для интернационализированных URL-адресов:

- **Префикс языка в шаблонах URL:** Добавление префикса языка к URL-адресам для обслуживания каждой языковой версии под другим базовым URL-адресом
- **Переведенные шаблоны URL:** Перевод шаблонов URL-адресов, чтобы каждый URL-адрес отличался для каждого языка

Причиной для перевода URL-адресов является оптимизация вашего сайта для поисковых систем. Добавив префикс языка к вашим шаблонам, вы сможете индексировать URL-адрес для каждого языка, а не один URL-адрес для всех. Кроме того, переведя URL-адреса на каждый язык, вы предоставите поисковым системам URL-адреса, которые будут лучше ранжироваться для каждого языка.

Добавление префикса языка к шаблонам URL

Django позволяет добавить языковой префикс к вашим паттернам URL. Например, английскую версию вашего сайта можно обслуживать по пути, начинающемуся с `/en/`, а испанскую версии `/es/`.

Чтобы использовать языки в шаблонах URL-адресов, вы должны использовать `LocaleMiddleware`, предоставленный Django. Фреймворк будет использовать его для идентификации текущего языка из запрошенного URL-адреса. Вы добавили его ранее в настройку `MIDDLEWARE` вашего проекта, поэтому вам не нужно это делать сейчас.

Давайте добавим префикс языка к нашим шаблонам URL. Отредактируйте основной файл `urls.py` проекта `myshop` и добавьте `i18n_patterns()` следующим образом:

```
from django.conf.urls.i18n import i18n_patterns

urlpatterns = i18n_patterns(
    path('admin/', admin.site.urls),
    path('cart/', include('cart.urls', namespace='cart')),
    path('orders/', include('orders.urls', namespace='orders')),
    path('payment/', include('payment.urls', namespace='payment')),
    path('coupons/', include('coupons.urls', namespace='coupons')),
    path('rosetta/', include('rosetta.urls')),
    path('', include('shop.urls', namespace='shop')),
)
```

Вы можете комбинировать непереводимые стандартные шаблоны и шаблоны URL в `i18n_patterns`, чтобы некоторые шаблоны включали языковой префикс, а другие - нет. Тем не

менее, лучше всего использовать переведенные URL-адреса только во избежание того, чтобы небрежно переведенный URL-адрес соответствовал шаблону без трансляции.

Запустите сервер разработки и откройте `http://127.0.0.1:8000/` в своем браузере. Django выполнит шаги, описанные ранее в разделе *Как Django определяет текущий язык*, чтобы определить текущий язык, и перенаправит вас на запрошенный URL, включая префикс языка. Взгляните на URL 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` вашего браузера, если он является испанским или английским, в противном случае, определенный в ваших настройках по умолчанию `LANGUAGE_CODE` (`English`).

Перевод шаблонов URL

Django поддерживает переведенные строки в шаблонах URL. Вы можете использовать разные переводы для каждого языка для одного шаблона URL. Вы можете пометить шаблоны URL для перевода так же, как и литералы, используя функцию `gettext_lazy()`.

Отредактируйте основной файл `urls.py` проекта `myshop` и добавьте строки перевода в регулярные выражения шаблонов URL-адресов для `cart`, `orders`, `payment`, и `coupons` следующим образом:

```
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` и пометьте шаблоны URL для перевода следующим образом:

```
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    path(_('create/'), views.order_create, name='order_create'),
    # ...
]
```

Откройте файл `urls.py` приложения `payment` и измените код на следующий:

```
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'),
]
```

Нам не нужно переводить шаблоны URL-адресов приложения `shop`, поскольку они построены с переменными и не содержат никаких других литералов.

Откройте консоль и выполните следующую команду, чтобы обновить файлы сообщений с помощью новых переводов:

```
django-admin makemessages --all
```

Убедитесь, что сервер разработки запущен. Откройте <http://127.0.0.1:8000/en/rosetta/> в своем браузере и нажмите ссылку Myshop в разделе Spanish. Теперь вы увидите шаблоны URL для перевода. Вы можете щелкнуть по Untranslated only, чтобы видеть только строки, которые еще не были переведены. Теперь вы можете перевести URL-адреса.

Разрешение пользователям переключать язык

Поскольку мы обслуживаем контент, доступный на нескольких языках, мы должны позволить нашим пользователям переключать язык сайта. Мы собираемся добавить селектор языка на наш сайт. Селектор языка будет состоять из списка доступных языков, которые отображаются с использованием ссылок.

Откройте шаблон `shop/base.html` приложения `shop` и найдите следующие строки:

```
<div id="header">
    <a href="/" class="logo">{% trans "My shop" %}</a>
</div>
```

Замените их следующим 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>
```

```
{% endfor %}  
  </ul>  
</div>  
</div>
```

Вот как мы строим наш языковой селектор:

1. Во-первых, мы загружаем теги интернационализации, используя `{% load i18n %}`
2. Мы используем тег `{% get_current_language %}` для извлечения текущего языка
3. Мы получаем языки, определенные в настройках `LANGUAGES`, используя шаблонный тег `{% get_available_languages %}`
4. Мы используем тег `{% get_language_info_list %}` для обеспечения легкого доступа к языковым атрибутам
5. Мы создаем список HTML для отображения всех доступных языков и добавляем атрибут класса `selected` к текущему активному языку

Мы используем теги шаблонов, предоставляемые `i18n`, на основе языков, доступных в настройках вашего проекта. Теперь откройте `http://127.0.0.1:8000/` в своем браузере. Вы должны увидеть селектор языка в правом верхнем углу сайта следующим образом:

Tu carro está vacío.

Categorías

Productos

Todos

Tea

NO IMAGE
AVAILABLE



Green tea
\$30



Red tea
\$45,5

Tea powder
\$21,2

Теперь пользователи могут легко переключаться на
предпочитаемый язык.

Перевод моделей с помощью django-parler

Django не предоставляет решение для перевода моделей из коробки. Вы должны реализовать собственное решение для управления контентом, хранящимся на разных языках, или использовать сторонний модуль для перевода модели.

Существует несколько сторонних приложений, которые позволяют вам переводить поля модели. Каждое из них использует разный подход к хранению и доступу к переводам.

Одним из таких приложений является `django-parler`. Этот модуль предлагает очень эффективный способ перевода моделей, и он плавно интегрируется с административным сайтом Django.

`django-parler` создает отдельную таблицу базы данных для каждой модели, которая содержит переводы. Эта таблица включает все переведенные поля и внешний ключ для исходного объекта, к которому принадлежит перевод. Она также содержит поле языка, поскольку каждая строка хранит содержимое для одного языка.

Установка django-parler

Установите `django-parler` с помощью `pip` используя следующую команду:

```
pip install django-parler==1.9.2
```

Откройте файл `settings.py` вашего проекта и добавте `'parler'` к настройкам `INSTALLED_APPS` следующим образом:

```
INSTALLED_APPS = [
    # ...
    'parler',
]
```

Также добавьте следующий код в ваши настройки:

```
PARLER_LANGUAGES = {
    None: (
        {'code': 'en'},
        {'code': 'es'},
    ),
    'default': {
        'fallback': 'en',
        'hide_untranslated': False,
    }
}
```

Этот параметр определяет доступные языки `en` и `es` для `django-parler`. Мы указываем язык по умолчанию `en`, и мы указываем, что `django-parler` не должен скрывать не переведенный контент.

Перевод полей модели

Давайте добавим переводы для нашего каталога продуктов.

`django-parler` предоставляет класс модели `TranslatedModel` и `TranslatedFields` для перевода полей модели. Откройте файл `models.py` в каталоге приложения `shop` и добавьте следующий импорт:

```
from parler.models import TranslatableModel, TranslatedFields
```

Затем измените модель `Category`, чтобы сделать поля `name` и `slug` переводимыми следующим образом:

```
class Category(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=200,
                               db_index=True),
        slug = models.SlugField(max_length=200,
                               db_index=True,
                               unique=True)
    )
```

Модель `Category` теперь наследуется от `TranslatedModel` вместо `models.Model` и оба поля `name` и `slug` включены в `TranslatedFields`.

Измените модель `Product`, чтобы добавить переводы для полей `name`, `slug` и `description` следующим образом:

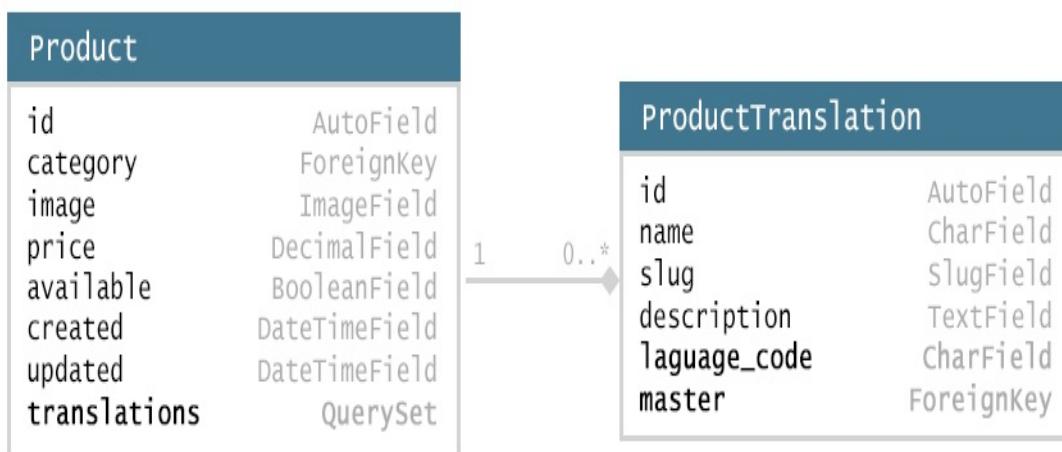
```
class Product(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=200, db_index=True),
        slug = models.SlugField(max_length=200, db_index=True),
        description = models.TextField(blank=True)
    )
    category = models.ForeignKey(Category,
```

```

        related_name='products')
image = models.ImageField(upload_to='products/%Y/%m/%d',
                           blank=True)
price = models.DecimalField(max_digits=10, decimal_places=2)
available = models.BooleanField(default=True)
created = models.DateTimeField(auto_now_add=True)
updated = models.DateTimeField(auto_now=True)

```

`django-parler` управляет переводами, создавая еще одну модель для каждой переводимой модели. Ниже вы можете увидеть поля модели `Product` и то, как будет выглядеть сгенерированная модель `ProductTranslation`:



Модель `ProductTranslation`, сгенерированная `django-parler`, включает в себя `name`, `slug`, и `description` переводимые поля, поле `language_code` и `ForeignKey` для объекта `Product`. Существует отношение «один ко многим» из `Product` в `ProductTranslation`. Объект `ProductTranslation` будет существовать для каждого доступного языка в каждом объекте `Product`.

Поскольку Django использует отдельную таблицу для переводов, есть некоторые функции Django, которые мы не можем использовать. Невозможно использовать порядок по умолчанию с помощью переведенного поля. Вы можете фильтровать по переводимым полям в запросах, но вы не

можете включить поле для перевода в опцию `ordering` `Meta`.

Измените файл `models.py` приложения `shop` и закомментируйте атрибут `ordering` класса `Category Meta`:

```
class Category(TranslatableModel):
    # ...
    class Meta:
        # ordering = ('name',)
        verbose_name = 'category'
        verbose_name_plural = 'categories'
```

Мы также должны закомментировать атрибуты `ordering` и `index_together` класса `Product Meta`. Текущая версия `django-parler` не предоставляет поддержку для проверки `index_together`.
Закомментируйте класс `Product Meta` следующим образом:

```
class Product(TranslatableModel):
    # ...

    # class Meta:
    #     ordering = ('-name',)
    #     index_together = (('id', 'slug'),)
```

Вы можете больше узнать о совместимости модуля `django-parler` с `Django` <https://django-parler.readthedocs.io/en/latest/compatibility.html>.

Интеграция переводов в сайт администрирования

`django-parler` плавно интегрируется с сайтом администрирования Django. Он включает класс `TranslatableAdmin`, который переопределяет класс `ModelAdmin`, предоставленный Django для управления переводами модели.

Отредактируйте файл `admin.py` приложения `shop` и добавьте в него следующий импорт:

```
from parler.admin import TranslatableAdmin
```

Измените классы `CategoryAdmin` и `ProductAdmin`, чтобы наследовать их от `TranslatableAdmin` вместо `ModelAdmin`. `django-parler` не поддерживает атрибут `prepopulated_fields`, но поддерживает метод `get_prepopulated_fields()`, который обеспечивает ту же функциональность. Отредактируйте файл `admin.py`, чтобы он выглядел следующим образом:

```
from django.contrib import admin
from .models import Category, Product
from parler.admin import TranslatableAdmin

@admin.register(Category)
class CategoryAdmin(TranslatableAdmin):
    list_display = ['name', 'slug']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}


@admin.register(Product)
class ProductAdmin(TranslatableAdmin):
    list_display = ['name', 'slug', 'price',
                    'available', 'created', 'updated']
```

```
list_filter = ['available', 'created', 'updated']
list_editable = ['price', 'available']

def get_prepopulated_fields(self, request, obj=None):
    return {'slug': ('name',)}
```

Мы адаптировали сайт администрирования для работы с новыми переведенными моделями. Теперь мы сможем синхронизировать базу данных с изменениями модели, которые мы сделали.

Создание миграции для переведенных моделей

Откройте консоль и запустите следующую команду, чтобы создать новую миграцию для модели переводов:

```
python manage.py makemigrations shop --name "translations"
```

Вы увидите следующий 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 field name from category
- Remove field slug from category
- Alter index_together for product (0 constraint(s))
- Add field master to producttranslation
- Add field master to categorytranslation
- Remove field description from product
- Remove field name from product
- Remove field slug from product
- Alter unique_together for producttranslation (1 constraint(s))
- Alter unique_together for categorytranslation (1 constraint(s))
```

Эта миграция автоматически включает модели `CategoryTranslation` и `ProductTranslation`, созданные динамически с помощью `djangoparler`. Важно отметить, что эта миграция удаляет предыдущие существующие поля из наших моделей. Это означает, что мы потеряем эти данные и вам нужно будет снова установить наши категории и продукты на сайте администратора после запуска.

Выполните следующую команду, чтобы применить миграцию:

```
python manage.py migrate shop
```

Вы увидите вывод, который заканчивается следующей строкой:

```
Applying shop.0002_translations... OK
```

Теперь наши модели синхронизировались с базой данных.

Запустите сервер разработки с помощью `python manage.py runserver` и откройте `http://127.0.0.1:8000/en/admin/shop/category/` в вашем браузере. Вы увидите, что существующие категории потеряли свое `name` и `slug` из-за удаления этих полей и вместо этого использовали переведенные, созданные `django-parler`. Нажмите на категорию, чтобы отредактировать ее. Вы увидите, что страница `Change category` включает в себя две разные вкладки: одну для английского и одну для испанских переводов:

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](#)

Обязательно заполните имя для потерянных категорий. Также добавьте испанский перевод для каждой из них и нажмите кнопку SAVE. Обязательно сохраните изменения перед переключением вкладок или вы потеряете их.

После внесения данных для существующих категорий откройте <http://127.0.0.1:8000/en/admin/shop/product/> и отредактируйте каждый из продуктов, содержащих английское и испанское name, slug, и description.

Адаптация представлений для переводов

Мы должны адаптировать наши представления `shop` для использования `QuerySet` с переводами. Выполните следующую команду, чтобы открыть терминал Python:

```
python manage.py shell
```

Давайте посмотрим, как вы можете извлекать и запрашивать поля перевода. Чтобы получить объект с полями, переведенными на определенный язык, вы можете использовать функцию Django `activate()` следующим образом:

```
>>> from shop.models import Product
>>> from django.utils.translation import activate
>>> activate('es')
>>> product=Product.objects.first()
>>> product.name
'Té verde'
```

Другой способ сделать это - использовать менеджер `language()`, предоставленный `django-parler` следующим образом:

```
>>> product=Product.objects.language('en').first()
>>> product.name
'Green tea'
```

Когда вы получаете доступ к переведенным полям, они разрешаются с использованием текущего языка. Вы можете установить другой текущий язык для доступа к этому конкретному переводу следующим образом:

```
>>> product.set_current_language('es')
>>> product.name
'Té verde'
>>> product.get_current_language()
'es'
```

При выполнении `QuerySet` с использованием `filter()` вы можете фильтровать с помощью связанных объектов перевода с синтаксисом `translations__` следующим образом:

```
>>> Product.objects.filter(translations__name='Green tea')
<TranslatableQuerySet [<Product: Té verde>]>
```

Давайте адаптируем представления каталога продуктов. Откройте файл `views.py` приложения `shop` и в представлении `product_list` найдите следующую строку:

```
category = get_object_or_404(Category, slug=category_slug)
```

Замените ее на следующее:

```
language = request.LANGUAGE_CODE
category = get_object_or_404(Category,
                             translations__language_code=language,
                             translations__slug=category_slug)
```

Затем отредактируйте представление `product_detail` и найдите следующую строку:

```
product = get_object_or_404(Product,
                            id=id,
                            slug=slug,
                            available=True)
```

Замените её следующим кодом:

```
language = request.LANGUAGE_CODE
```

```
product = get_object_or_404(Product,
                            id=id,
                            translations__language_code=language,
                            translations__slug=slug,
                            available=True)
```

Представления `product_list` и `product_detail` теперь адаптированы для извлечения объектов с использованием переведенных полей. Запустите сервер разработки и откройте `http://127.0.0.1:8000/es/` в своем браузере. Вы должны увидеть страницу списка продуктов, включая все продукты, переведенные на испанский язык:

Mi tienda Language: English español

Tu carro está vacío.

Productos

Categorías

Todos

Té

NO IMAGE AVAILABLE



Té verde \$30 Té rojo \$45,5 Té en polvo \$21,2

Теперь URL каждого продукта строится с использованием поля

`slug`, переведенного на текущий язык. Например, URL-адрес продукта на испанском языке – ЭТО `http://127.0.0.1:8000/es/2/te-rojo/`, тогда как на английском языке URL-адрес `http://127.0.0.1:8000/en/2/red-tea/`. Если вы перейдете на страницу сведений о продукте, вы увидите переведенный URL-адрес и содержимое выбранного языка, как показано в следующем примере:

Mi tienda Language: English español

Tu carro está vacío.



Té rojo
Té
\$45,5

Cantidad: Añadir al carro

El té pu-erh es conocido en Occidente también como té rojo (en chino: 普洱茶, pinyin: pǔ'ěrchá) y su nombre proviene de la región de Pu'er de Yunnan China, de donde procede. Se trata de un té inusual en China, siendo este el mayor productor del té rojo o pu-erh del mundo.

Если вы хотите узнать больше о `django-parler`, вы можете найти полную документацию на <https://django-parler.readthedocs.io/en/latest/>.

Вы узнали, как перевести код Python, шаблоны, паттерны URL

и поля модели. Чтобы завершить процесс интернационализации и локализации, нам нужно также использовать локализованное форматирование для дат, времени и чисел.

Локализация форматов

В зависимости от локали пользователя вы можете отображать даты, время и числа в разных форматах. Локализованное форматирование можно активировать, изменив параметр `USE_L10N` на `True` в файле `settings.py` вашего проекта.

Если параметр `USE_L10N` включен, Django будет пытаться использовать формат, специфичный для локали, всякий раз, когда он выдает значение в шаблоне. Вы можете видеть, что десятичные числа в английской версии вашего сайта отображаются с разделителем точки для десятичных знаков, тогда как в испанской версии они отображаются с запятой. Это связано с языковыми форматами, указанными для языкового стандарта `es` Django. Вы можете посмотреть испанскую конфигурацию форматирования на <https://github.com/django/django/blob/stable/2.0.x/django/conf/locale/es/formats.py>.

Обычно вы устанавливаете для параметра `USE_L10N` значение `True` и позволяете Django применять локализацию формата для каждой локали. Однако могут возникнуть ситуации, когда вы не хотите использовать локализованные значения. Это особенно актуально при применении JavaScript или JSON, который должен обеспечивать машиночитаемый формат.

Django предлагает тег шаблона `{% localize %}`, который позволяет включать/отключать локализацию фрагментов шаблона. Это дает вам возможность контролировать локализованное форматирование. Вам нужно будет загрузить теги `l10n`, чтобы использовать этот шаблонный тег. Ниже приведен пример того, как включить и отключить локализацию в шаблоне:

```
|  {% load l10n %}
```

```
{% localize on %}  
  {{ value }}  
{% endlocalize %}  
  
{% localize off %}  
  {{ value }}  
{% endlocalize %}
```

Django также предлагает фильтры шаблонов `localize` И `unlocalize`, чтобы применить или избежать локализации значения. Эти фильтры могут применяться следующим образом:

```
{{ value|localize }}  
{{ value|unlocalize }}
```

Вы также можете создавать файлы пользовательского формата, чтобы указать форматирование локали. Вы можете найти дополнительную информацию о локализации формата в <https://docs.djangoproject.com/en/2.0/topics/i18n/formatting/>.

Использование django-localflavor для проверки полей формы

`django-localflavor` является сторонним модулем, который содержит набор конкретных utils, проверки полей формы или полей модели, которые являются специфическими для каждой страны. Очень полезно проверять местные регионы, местные номера телефонов, номера удостоверений личности, номера социального страхования и т. д. Пакет организован в ряд модулей, названных в соответствии с кодами стран ISO 3166.

Установите `django-localflavor` используя следующую команду:

```
pip install django-localflavor==2.0
```

Измените файл `settings.py` вашего проекта и добавте `localflavor` к настройкам `INSTALLED_APPS`:

```
INSTALLED_APPS = [  
    # ...  
    'localflavor',  
]
```

Мы собираемся добавить поле почтового индекса Соединенных Штатов, чтобы при создании нового заказа использовался действующий почтовый индекс США.

Измените файл `forms.py` приложения `orders` следующим образом:

```
from django import forms
```

```
from .models import Order
from localflavor.us.forms import USZipCodeField

class OrderCreateForm(forms.ModelForm):
    postal_code = USZipCodeField()
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address',
                  'postal_code', 'city']
```

Мы импортируем поле `USZipCodeField` из `us` пакета `localflavor` и используем его для `postal_code` поля формы `OrderCreateForm`.

Запустите сервер разработки и откройте

`http://127.0.0.1:8000/en/orders/create/` в вашем браузере. Заполните все поля, введите трехбуквенный почтовый индекс и отправьте форму. Вы получите следующую ошибку проверки, которая вызывает `USZipCodeField`:

Enter a zip code in the format xxxxx or xxxxx-xxxx.

Это лишь краткий пример использования пользовательского поля из `localflavor` в вашем собственном проекте для целей проверки. Локальные компоненты, предоставляемые `localflavor`, очень полезны для адаптации вашего приложения к конкретным странам. Вы можете прочитать документацию `django-localflavor` и просмотреть все доступные локальные компоненты для каждой страны в <https://django-localflavor.readthedocs.io/en/latest/>.

Далее мы создадим механизм рекомендаций в нашем магазине.

Создание механизма рекомендаций

Механизм рекомендаций - это система, которая предсказывает предпочтение или рейтинг, которые пользователь дал бы элементу. Система выбирает релевантные элементы для пользователей на основе их поведения и знаний о них. В настоящее время системы рекомендаций используются во многих онлайн-сервисах. Они помогают пользователям, выбирая материал, который им может быть интересен из огромного количества доступных данных, которые не имеют к ним отношения. Предоставление хороших рекомендаций повышает вовлеченность пользователей. Сайты электронной коммерции также могут предлагать рекомендации по соответствующим продуктам, увеличивая их среднюю продажу.

Мы собираемся создать простой, но мощный механизм рекомендаций, который предлагает продукты, которые обычно покупаются вместе. Мы будем предлагать продукты, основанные на истории продаж, таким образом определяя продукты, которые обычно покупаются вместе. Мы собираемся предложить дополнительные продукты в двух разных сценариях:

- **Страница сведений о продукте:** Мы покажем список продуктов, которые обычно покупаются с данным продуктом. Он будет отображаться как: Пользователи, которые купили это, также купили X, Y, Z. Нам нужна структура данных, которая позволит хранить счетчик, когда каждый продукт был куплен вместе с

отображаемым продуктом.

- **Подробная страница корзины:** Основываясь на том, что пользователи добавляют в корзину, мы собираемся предложить продукты, которые обычно покупаются вместе с этими продуктами. В этом случае счет, который мы вычисляем для получения связанных продуктов, должен быть агрегирован.

Мы будем использовать Redis для хранения продуктов, которые будут куплены вместе. Помните, что вы уже использовали Redis в [главе 6](#), *Отслеживание действий пользователя*. Если вы еще не установили Redis, вы можете найти инструкции по установке в этой главе.

Рекомендация товаров на основе предыдущих покупок

Теперь мы будем рекомендовать продукты пользователям на основе того, что они добавили в корзину. Мы собираемся хранить ключ в Redis для каждого продукта, купленного на нашем сайте. Ключ продукта будет содержать отсортированный набор Redis с оценками. Мы будем увеличивать счет на 1 для каждого продукта, купленного вместе, каждый раз, когда будет завершена новая покупка.

Когда заказ успешно оплачен, мы сохраним ключ для каждого купленного продукта, включая отсортированный набор продуктов, принадлежащих одному и тому же заказу. Отсортированный набор позволит нам давать оценки для продуктов, которые покупаются вместе.

Не забудьте установить `redis-py` в свою среду, используя следующую команду:

```
pip install redis==2.10.6
```

Откройте файл `settings.py` вашего проекта и добавьте в него следующие настройки:

```
REDIS_HOST = 'localhost'  
REDIS_PORT = 6379  
REDIS_DB = 1
```

Это настройки, необходимые для установления соединения с сервером Redis. Создайте новый файл в каталоге приложений `shop` и назовите его `recommender.py`. Добавьте к нему следующий код:

```
import redis
from django.conf import settings
from .models import Product

# connect to redis
r = redis.StrictRedis(host=settings.REDIS_HOST,
                      port=settings.REDIS_PORT,
                      db=settings.REDIS_DB)

class Recommender(object):

    def get_product_key(self, id):
        return 'product:{}:purchased_with'.format(id)

    def products_bought(self, products):
        product_ids = [p.id for p in products]
        for product_id in product_ids:
            for with_id in product_ids:
                # get the other products bought with each product
                if product_id != with_id:
                    # increment score for product purchased together
                    r.zincrby(self.get_product_key(product_id),
                              with_id,
                              amount=1)
```

Это класс `Recommender`, который позволит нам хранить покупки товара и получать предложения для данного продукта или продуктов. Метод `get_product_key()` получает идентификатор объекта `Product` и создает ключ Redis для отсортированного набора, в котором хранятся связанные продукты, чтобы это выглядело как `product:[id]:purchased_with`.

Метод `products_bought()` получает список объектов `Product`, которые были куплены вместе (то есть принадлежат одному и тому же заказу). В этом методе мы выполняем следующие задачи:

1. Мы получаем идентификаторы продуктов для данного объекта `Product`.
2. Мы перебираем идентификаторы продуктов. Для каждого идентификатора мы перебираем идентификаторы продуктов и пропускаем один и тот же

продукт, чтобы мы получили продукты, которые были куплены вместе с каждым продуктом.

3. Мы получаем ключ продукта Redis для каждого продукта, купленного с помощью метода `get_product_id()`. Для продукта с идентификатором 33 этот метод возвращает ключ `product:33:purchased_with`. Это ключ для отсортированного набора, который содержит идентификаторы продуктов, купленных вместе с этим.
4. Мы увеличиваем оценку каждого идентификатора продукта, содержащегося в отсортированном наборе, на 1. Счет представляет собой время, когда другой продукт был куплен вместе с данным продуктом.

Таким образом, у нас есть метод хранения и счетчик продуктов, которые были куплены вместе. Теперь нам нужен метод для извлечения продуктов, которые покупаются вместе для списка данных продуктов. Добавьте следующий метод

`suggest_products_for()` В класс `Recommender`:

```
def suggest_products_for(self, products, max_results=6):  
    product_ids = [p.id for p in products]  
    if len(products) == 1:  
        # only 1 product  
        suggestions = r.zrange(  
            self.get_product_key(product_ids[0]),  
            0, -1, desc=True)[:max_results]  
    else:  
        # generate a temporary key  
        flat_ids = ''.join([str(id) for id in product_ids])  
        tmp_key = 'tmp_{}'.format(flat_ids)  
        # multiple products, combine scores of all products  
        # store the resulting sorted set in a temporary key  
        keys = [self.get_product_key(id) for id in product_ids]  
        r.zunionstore(tmp_key, keys)  
        # remove ids for the products the recommendation is for  
        r.zrem(tmp_key, *product_ids)  
        # get the product ids by their score, descendant sort  
        suggestions = r.zrange(tmp_key, 0, -1,
```

```
        desc=True)[:max_results]
    # remove the temporary key
    r.delete(tmp_key)
suggested_products_ids = [int(id) for id in suggestions]

# get suggested products and sort by order of appearance
suggested_products =
list(Product.objects.filter(id__in=suggested_products_ids))
suggested_products.sort(key=lambda x: suggested_products_ids.index(x.id))
return suggested_products
```

Метод `suggest_products_for()` получает следующие параметры:

- `products`: Это список объектов `Product` для получения рекомендаций. Он может содержать один или несколько продуктов.
- `max_results`: Это целое число, представляющее максимальное количество возвращаемых рекомендаций.

В этом методе мы выполняем следующие действия:

1. Мы получаем идентификаторы продуктов для данного объекта `Product`.
2. Если предоставляется только один продукт, мы получаем идентификаторы продуктов, которые были куплены вместе с данным продуктом, упорядоченные по общему количеству раз, когда они были куплены вместе. Для этого мы используем команду Redis `ZRANGE`. Мы ограничиваем количество результатов числом, указанным в атрибуте `max_results` по умолчанию (6).
3. Если указано более одного продукта, мы создаем временный ключ Redis, созданный с идентификаторами

продуктов.

4. Мы объединяем и суммируем все оценки для элементов, содержащихся в отсортированном наборе каждого из данных продуктов. Это делается с помощью команды Redis `ZUNIONSTORE`. Команда `ZUNIONSTORE` выполняет объединение отсортированных наборов с заданными ключами и сохраняет агрегированную сумму баллов элементов в новом ключе Redis. Подробнее об этой команде можно узнать в <https://redis.io/commands/ZUNIONSTORE>. Мы сохраняем агрегированные баллы во временном ключе.
5. Поскольку мы объединяем баллы, мы можем получить те же продукты, на которые мы получаем рекомендации. Мы удаляем их из генерированного отсортированного набора с помощью команды `ZREM`.
6. Мы извлекаем идентификаторы продуктов из временного ключа, упорядоченные по их счету, используя команду `ZRANGE`. Мы ограничиваем число результатов числом, указанным в атрибуте `max_results`. Затем мы удаляем временный ключ.
7. Наконец, мы получаем объекты `Product` с указанными идентификаторами и сортируем продукты в том же порядке.

Для практических целей давайте также добавим метод очистки рекомендаций. Добавьте следующий 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))
```

Давайте попробуем наш механизм рекомендаций. Убедитесь, что вы добавили несколько объектов `Product` в базу данных и инициализировали сервер Redis, используя следующую команду из консоли в каталоге Redis:

```
| src/redis-server
```

Откройте другую консоль и запустите следующую команду:

```
| python manage.py shell
```

Убедитесь, что в вашей базе данных есть как минимум четыре разных продукта. Извлеките четыре разных продукта по их названию:

```
>>> from shop.models import Product
>>> black_tea = Product.objects.get(translations__name='Black tea')
>>> red_tea = Product.objects.get(translations__name='Red tea')
>>> green_tea = Product.objects.get(translations__name='Green tea')
>>> tea_powder = Product.objects.get(translations__name='Tea powder')
```

Затем добавьте некоторые тестовые покупки в механизм рекомендаций:

```
>>> 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])
```

Мы сохранили следующие оценки:

```
| black_tea:  red_tea (2), tea_powder (2), green_tea (1)
```

```
red_tea:    black_tea (2), tea_powder (1), green_tea (1)
green_tea:  black_tea (1), tea_powder (1), red_tea(1)
tea_powder: black_tea (2), red_tea (1), green_tea (1)
```

Давайте активируем язык для извлечения переведенных товаров и получим рекомендации по продукту для покупки вместе с данным товаром:

```
>>> from django.utils.translation import activate
>>> activate('en')
>>> r.suggest_products_for([black_tea])
[<Product: Tea powder>, <Product: Red tea>, <Product: Green tea>]
>>> r.suggest_products_for([red_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Green tea>]
>>> r.suggest_products_for([green_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Red tea>]
>>> r.suggest_products_for([tea_powder])
[<Product: Black tea>, <Product: Red tea>, <Product: Green tea>]
```

Вы можете видеть, что заказ с рекомендованными продуктами основан на их оценке. Давайте дадим рекомендации для нескольких продуктов 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 можете видеть, что порядок предлагаемых продуктов соответствует агрегированным баллам. Например, предлагаемые продукты `black_tea` И `red_tea` С `tea_powder` ($2+1$) И `green_tea` ($1+1$).

Мы проверили, что наш алгоритм рекомендаций работает должным образом. Давайте отобразим рекомендации для товаров на нашем сайте.

Измените файл `views.py` приложения `shop`. Добавьте

функциональность, чтобы получить максимум четырёх рекомендуемых продукта в представлении `product_detail` следующим образом:

```
from .recommender import Recommender

def product_detail(request, id, slug):
    language = request.LANGUAGE_CODE
    product = get_object_or_404(Product,
                                id=id,
                                translations__language_code=language,
                                translations__slug=slug,
                                available=True)
    cart_product_form = CartAddProductForm()

    r = Recommender()
    recommended_products = r.suggest_products_for([product], 4)

    return render(request,
                  'shop/product/detail.html',
                  {'product': product,
                   'cart_product_form': cart_product_form,
                   'recommended_products': recommended_products})
```

Откройте шаблон `shop/product/detail.html` приложения `shop` и добавьте следующий код ПОСЛЕ `{{ product.description|linebreaks }}`:

```
{% if recommended_products %}
<div class="recommendations">
    <h3>{% trans "People who bought this also bought" %}</h3>
    {% for p in recommended_products %}
        <div class="item">
            <a href="{{ p.get_absolute_url }}>
                
            </a>
            <p><a href="{{ p.get_absolute_url }}>{{ p.name }}</a></p>
        </div>
    {% endfor %}
</div>
{% endif %}
```

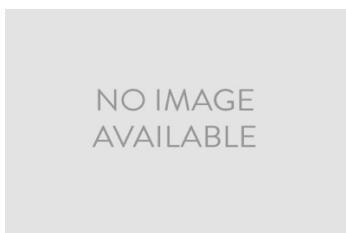
Запустите сервер разработки и откройте <http://127.0.0.1:8000/en/> в

своем браузере. Нажмите на любой продукт, чтобы просмотреть его данные. Вы должны увидеть, что рекомендуемые продукты отображаются под товаром, как показано на следующем снимке экрана:



The screenshot shows a product page for 'Tea powder'. At the top right, the product name 'Tea powder' is displayed in bold black text. Below it, the category 'Tea' is shown in blue. The price '\$21.2' is prominently displayed in large black font. To the right of the price is a 'Quantity' selector with a value of '1' and an 'Add to cart' button. The main image shows a mound of green tea powder on a white surface.

People who bought this also bought



Black tea



Red tea



Green tea

Мы также будем включать рекомендации по продуктам в корзину. Рекомендация будет основана на продуктах, добавленных пользователем в корзину.

Откройте `views.py` в приложении `cart`, импортируйте класс `Recommender` и отредактируйте представление `cart_detail`, чтобы сделать его похожим на следующее:

```

from shop.recommender import Recommender

def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
            initial={'quantity': item['quantity'],
                      'update': True})

    coupon_apply_form = CouponApplyForm()

    r = Recommender()
    cart_products = [item['product'] for item in cart]
    recommended_products = r.suggest_products_for(cart_products,
                                                    max_results=4)

    return render(request,
                  'cart/detail.html',
                  {'cart': cart,
                   'coupon_apply_form': coupon_apply_form,
                   'recommended_products': recommended_products})

```

Измените шаблон `cart/detail.html` приложения `cart` и добавьте следующий код сразу после тега HTML `</table>`:

```

{% if recommended_products %}
<div class="recommendations cart">
    <h3>{% trans "People who bought this also bought" %}</h3>
    {% for p in recommended_products %}
        <div class="item">
            <a href="{{ p.get_absolute_url }}>
                
            </a>
            <p><a href="{{ p.get_absolute_url }}>{{ p.name }}</a></p>
        </div>
    {% endfor %}
</div>
{% endif %}

```

Откройте `http://127.0.0.1:8000/en/` в вашем браузере и добавьте пару продуктов в корзину. Когда вы переходите на `http://127.0.0.1:8000/en/cart/`, вы должны увидеть агрегированные рекомендации по продуктам для товаров в корзине следующим

образом:

Your shopping cart

Image	Product	Quantity	Remove	Unit price	Price
	Tea powder	<input type="button" value="1"/> <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$21.2	\$21.2
	Green tea	<input type="button" value="1"/> <input type="button" value="Update"/>	<input type="button" value="Remove"/>	\$30	\$30
Total					\$51.20

People who bought this also bought



Black tea



Red tea

Apply a coupon:

Coupon:

Поздравляем! Вы создали полноценный механизм рекомендаций, используя Django и Redis.

Резюме

В этой главе вы создали купонную систему с использованием сессий. Вы узнали, как работают интернационализация и локализация. Вы также создали механизм рекомендаций с помощью Redis.

В следующей главе вы начнете новый проект. Вы создадите платформу электронного обучения с Django, используя представления на основе классов, и вы создадите собственную систему управления контентом.

Глава 10

Создание платформы электронного обучения (E-Learning)

В предыдущей главе вы добавили интернационализацию в свой проект интернет-магазина. Вы также создали купонную систему и механизм рекомендаций по продуктам. В этой главе вы создадите новый проект. Вы создадите платформу электронного обучения, создав пользовательский интерфейс **Content Management System (CMS)**.

В этой главе вы узнаете, как:

- Создать fixtures для ваших моделей
- Использовать наследование модели
- Создать настраиваемые поля модели
- Использовать базовые классы представлений и mixins
- Создать formsets
- Управлять группами и разрешениями
- Создать CMS

Настройка проекта электронного обучения

Нашим последним проектом будет платформа для электронного обучения. В этой главе мы собираемся создать гибкую CMS, которая позволит инструкторам создавать курсы и управлять их содержимым.

Сначала создайте виртуальную среду для своего нового проекта и активируйте ее следующими командами:

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

Установите Django в свою виртуальную среду с помощью следующей команды:

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

Мы собираемся управлять загрузкой изображений в нашем проекте, поэтому нам также нужно установить `Pillow` с помощью следующей команды:

```
pip install Pillow==5.1.0
```

Создайте новый проект, используя следующую команду:

```
django-admin startproject educa
```

Войдите в новый каталог `educa` и создайте новое приложение,

используя следующие команды:

```
cd educa
django-admin startapp courses
```

Откройте файл `settings.py` проекта `educa` и добавьте `courses` к настройке `INSTALLED_APPS` следующим образом:

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

Приложение `courses` теперь активно для проекта. Давайте определим модели и контент для курсов.

Создание моделей курса

Наша платформа электронного обучения будет предлагать курсы по различным предметам. Каждый курс будет разделен на настраиваемое количество модулей, и каждый модуль будет содержать настраиваемое количество контента. Они будут содержать контент различных типов: текст, файл, изображение или видео. В следующем примере показано, как будет выглядеть структура данных нашего 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 создадим модели курса. Измените файл `models.py` приложения `courses` и добавьте к нему следующий код:

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

class Subject(models.Model):
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)

    class Meta:
        ordering = ['title']

    def __str__(self):
        return self.title

class Course(models.Model):
```

```

owner = models.ForeignKey(User,
                         related_name='courses_created',
                         on_delete=models.CASCADE)
subject = models.ForeignKey(Subject,
                           related_name='courses',
                           on_delete=models.CASCADE)
title = models.CharField(max_length=200)
slug = models.SlugField(max_length=200, unique=True)
overview = models.TextField()
created = models.DateTimeField(auto_now_add=True)

class Meta:
    ordering = ['-created']

def __str__(self):
    return self.title


class Module(models.Model):
    course = models.ForeignKey(Course,
                               related_name='modules',
                               on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)

    def __str__(self):
        return self.title

```

Это исходные модели `Subject`, `Course`, и `Module`. Поле модели `Course` выглядит следующим образом:

- `owner`: Инструктор, создавший этот курс.
- `subject`: Тема, к которой относится этот курс. Поле `ForeignKey` указывает на модель `Subject`.
- `title`: Название курса.
- `slug`: Slug курса. Он будет использоваться в URL-адресах позже.
- `overview`: Это столбец `TextField`, который включает в себя обзор курса.

- `created`: Дата и время создания курса. Оно будет автоматически установлено Django при создании новых объектов `auto_now_add=True`.

Каждый курс делится на несколько модулей. Поэтому модель `Module` содержит поле `ForeignKey`, которое указывает на модель `Course`.

Откройте консоль и выполните следующую команду для создания начальной миграции для этого приложения:

```
python manage.py makemigrations
```

Вы увидите следующий результат:

```
Migrations for 'courses':  
  0001_initial.py:  
    - Create model Course  
    - Create model Module  
    - Create model Subject  
    - Add field subject to course
```

Затем выполните следующую команду, чтобы применить все миграции к базе данных:

```
python manage.py migrate
```

Вы должны увидеть выходные данные, включающие все применяемые миграции. Вывод будет содержать следующую строку:

```
Applying courses.0001_initial... ok
```

Модели нашего `courses` были синхронизированы с базой данных.

Регистрация моделей на сайте администрирования

Давайте добавим модели курса на сайт администрирования. Измените файл `admin.py` в каталоге приложений `courses` добавив к нему следующий 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 для программы курса теперь зарегистрированы на сайте администрирования. Помните, что мы используем декоратор `@admin.register()` для регистрации моделей на сайте администрирования.

Использование fixtures для представления исходных данных для моделей

Иногда вам может потребоваться предварительно заполнить вашу базу данных жестко закодированными данными. Это полезно, чтобы автоматически включать исходные данные в настройку проекта, а не добавлять их вручную. Django поставляется с простым способом загрузки и выгрузки данных из базы в файлы, которые называется **fixtures**.

Django поддерживает декодеры в форматах JSON, XML или YAML. Мы собираемся создать fixture для включения нескольких начальных объектов `Subject` для нашего проекта.

Во-первых, создайте суперпользователя, используя следующую команду:

```
python manage.py createsuperuser
```

Затем запустите сервер разработки, используя следующую команду:

```
python manage.py runserver
```

Откройте <http://127.0.0.1:8000/admin/courses/subject/> в вашем браузере. Создайте несколько объектов, используя сайт администрирования. Страница отображения списка должна выглядеть следующим образом:

Django administration

WELCOME, ADMIN. [VIEW SITE / CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Courses > Subjects

Select subject to change

[ADD SUBJECT +](#)

Action: Go 0 of 4 selected

<input type="checkbox"/>	TITLE	▲ SLUG
<input type="checkbox"/>	Mathematics	mathematics
<input type="checkbox"/>	Music	music
<input type="checkbox"/>	Physics	physics
<input type="checkbox"/>	Programming	programming

4 subjects

Выполните следующую команду из консоли:

```
python manage.py dumpdata courses --indent=2
```

Вы увидите вывод, похожий на следующий:

```
[  
{  
    "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` выдает данные из базы данных в стандартный вывод, по умолчанию в формате JSON. Результирующая структура данных включает информацию о модели и ее полях, чтобы Django мог ее загрузить в базу данных.

Вы можете ограничить вывод к моделям приложения, указав имена приложений в команде или указав отдельные модели для вывода данных с использованием формата `app.Model`. Вы также можете указать формат, используя флаг `--format`. По

умолчанию `dumpdata` выводит сериализованные данные на стандартный вывод. Однако вы можете указать выходной файл, используя флаг `--output`. Флаг `--indent` позволяет указать отступ. Для получения дополнительной информации о параметрах `dumpdata` запустите `python manage.py dumpdata --help`.

Сохраните этот дамп в файл привязки в каталог `fixtures/` используя следующие команды:

```
mkdir courses/fixtures
python manage.py dumpdata courses --indent=2 --
output=courses/fixtures/subjects.json
```

Запустите сервер разработки и воспользуйтесь сайтом администрирования для удаления созданных вами тем. Затем загрузите fixture в базу данных, используя следующую команду:

```
python manage.py loaddata subjects.json
```

Все объекты `Subject`, включенные в fixture, загружаются в базу данных.

По умолчанию Django ищет файлы в каталоге `fixtures/` каждого приложения, но вы можете указать полный путь к файлу fixture для команды `loaddata`. Вы также можете использовать параметр `FIXTURE_DIRS`, чтобы указать дополнительные каталоги Django для поиска fixtures.

Fixtures не только полезны для настройки исходных данных, но также для предоставления выборочных данных для вашего приложения или данных, необходимых для ваших тестов.

Вы можете прочитать о том, как использовать fixtures для тестирования на <https://docs.djangoproject.com/en/2.0/topics/testing/tools/#fixture-loading>.

Если вы хотите загрузить fixtures в модельные миграции,

ознакомьтесь с документацией Django о миграции данных. Вы можете найти документацию для переноса данных на <https://docs.djangoproject.com/en/2.0/topics/migrations/#data-migrations>.

Создание моделей для разнообразного контента

Мы планируем добавлять различные типы контента в учебные модули, такие как тексты, изображения, файлы и видео. Нам нужна универсальная модель данных, которая позволит нам хранить этот контент. В [Глава 6, Отслеживание действий пользователя](#), вы поняли удобство использования общих отношений для создания внешних ключей, которые могут указывать на объекты любой модели. Мы собираемся создать модель `Content`, которая предоставит содержимое модулей и определит общее отношение для связывания любого типа контента.

Откройте файл `models.py` приложения `courses` и добавьте следующие импорты:

```
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey
```

Затем добавьте следующий 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`. Модель содержит различный тип контента, поэтому мы определяем поле `ForeignKey` к модели `Module`. Мы также

установили общее отношение для ассоциирования объектов из разных моделей, представляющих различные типы контента. Помните, что нам нужно три разных поля для создания общих отношений. В нашей модели `content` это:

- `content_type`: Поле `ForeignKey` модели `ContentType`
- `object_id`: Поле `PositiveIntegerField` для хранения первичного ключа связанного объекта
- `item`: Поле `GenericForeignKey` к связанному объекту путем объединения двух предыдущих полей

Только поля `content_type` и `object_id` имеют соответствующий столбец в таблице базы данных этой модели. Поле `item` позволяет вам напрямую загрузить или установить связанный объект, а его функциональность будет построена поверх двух других полей.

Мы собираемся использовать различные модели для каждого типа контента. Наши модели контента будут иметь некоторые общие поля, но они будут отличаться фактическими данными, которые смогут хранить.

Использование наследования модели

Django поддерживает наследование модели. Оно работает аналогично стандартным наследованиям классов в Python. Django предлагает следующие три варианта использования наследования модели:

- **Абстрактные модели:** Полезно, если вы хотите поместить некоторую общую информацию в несколько моделей. Для абстрактной модели не создается таблица базы данных.
- **Мульти-табличное наследование модели:** Применимо, когда каждая модель в иерархии считается полной моделью сама по себе. Для каждой модели создается таблица базы данных.
- **Прокси-модели:** Полезно, когда вам нужно изменить поведение модели, например, включив дополнительные методы, изменив менеджер по умолчанию или используя разные мета-параметры. Для прокси-моделей не создается таблица базы данных.

Давайте подробнее рассмотрим каждый из них.

Абстрактные модели

Абстрактная модель - это базовый класс, в котором вы определяете поля, которые хотите включить во все дочерние модели. Django не создает таблицу базы данных для абстрактных моделей. Для каждой дочерней модели создается таблица базы данных, включая поля, унаследованные от абстрактного класса, и те, которые определены в дочерней модели.

Чтобы пометить модель как абстрактную, вам нужно включить `abstract=True` в свой класс `Meta`. Django поймет, что это абстрактная модель и не будет создавать для нее таблицу базы данных.

Чтобы создать дочерние модели, вы должны наследоваться от абстрактной модели.

В следующем примере показана 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 этом случае Django создаст таблицу только для модели `Text`, включая поля `title`, `created` и `body`.

Много-табличное наследование модели

В много-табличном наследовании, каждая модель соответствует таблице базы данных. Django создает поле `OneToOneField` для отношения в модели потомка к его родительскому элементу.

Чтобы использовать много-табличное наследование, вы должны наследовать существующую модель. Django создаст таблицы базы данных для исходной и наследуемой модели. В следующем примере показано наследование нескольких таблиц:

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class Text(BaseContent):
    body = models.TextField()
```

Django будет включать автоматически созданное поле `OneToOneField` в модели `Text` и создаст таблицу базы данных для каждой модели.

Прокси-модели

Прокси-модели используются для изменения поведения модели, например, путем включения дополнительных методов или различных мета-опций. Обе модели работают в таблице базы данных исходной модели. Чтобы создать прокси-модель, добавьте `proxy=True` в класс `Meta` модели.

В следующем примере показано, как создать 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 мы определяем модель `orderedContent`, которая является прокси-моделью для модели `content`. Эта модель предоставляет порядок по умолчанию для `QuerySet` и дополнительный метод `created_delta()`. Обе модели, `Content` и `OrderedContent` работают в одной таблице базы данных, а объекты доступны через ORM любой модели.

Создание моделей контента

Модель `Content` нашего приложения `courses` содержит общее отношение для связывания с ней различных типов контента. Мы создадим другую модель для каждого типа контента. Все модели контента будут иметь некоторые общие поля и дополнительные поля для хранения пользовательских данных. Мы собираемся создать абстрактную модель, которая предоставляет общие поля для всех моделей контента.

Измените файл `models.py` приложения `courses` и добавьте к нему следующий 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 этом коде мы определяем абстрактную модель с именем `ItemBase`. Поэтому мы установили `abstract=True` в класс `Meta`. В этой модели мы определяем поля `owner`, `title`, `created` и `updated`. Это общие поля которые будут использоваться для всех типов контента. Поле `owner` позволяет нам сохранять, какой пользователь создал контент. Поскольку это поле определено в абстрактном классе, нам нужно другое `related_name` для каждой подмодели. Django позволяет указать местозаполнитель для имени класса `model` в атрибуте `related_name` как `%s`. Таким образом, `related_name` для каждой дочерней модели будет сгенерировано автоматически. Поскольку мы используем `'%(class)s_related'` как `related_name`, обратное отношение для дочерних моделей будет `text_related`, `file_related`, `image_related`, и `video_related` соответственно.

Мы определили четыре различные модели контента, которые наследуются от абстрактной модели `ItemBase`. Это следующие:

- `Text`: Сохранение текстового содержимого
- `File`: Чтобы хранить файлы, например PDF
- `Image`: Для хранения файлов изображений
- `Video`: Для хранение видео; мы используем поле `URLField`, чтобы предоставить URL-адрес видео, в порядке его добавления

Каждая дочерняя модель содержит поля, определенные в классе `ItemBase`, в дополнение к своим собственным полям. Будет создана таблица базы данных для моделей `Text`, `File`, `Image`, и `Video` соответственно. Не будет создано таблицы базы данных, связанной с моделью `ItemBase`, поскольку она является абстрактной моделью.

В рание созданной модели `Content` измените поле `content_type` следующим образом:

```
content_type = models.ForeignKey(ContentType,
                                 on_delete=models.CASCADE,
                                 limit_choices_to={'model__in':(
                                     'text',
                                     'video',
                                     'image',
                                     'file'))}
```

Мы добавляем аргумент `limit_choices_to` для ограничения объектов `ContentType`, которые могут использоваться для общих отношений. Мы используем поле `model__in` для фильтрации запроса к объектам `ContentType` с помощью атрибута `model`, который содержит `'text'`, `'video'`, `'image'`, ИЛИ `'file'`.

Давайте создадим миграцию для новых моделей, которые мы добавили. Выполните следующую команду из командной строки:

```
python manage.py makemigrations
```

Вы увидите следующий результат:

```
Migrations for 'courses':
courses/migrations/0002_content_file_image_text_video.py
- Create model Content
- Create model File
- Create model Image
- Create model Text
- Create model Video
```

Затем выполните следующую команду для применения новой миграции:

```
python manage.py migrate
```

Результат, который вы увидите, должен заканчиваться следующей строкой:

Applying courses.0002_content_file_image_text_video... OK

Мы создали модели, которые подходят для добавления разнообразного контента в модули курса. Однако в наших моделях все еще чего-то не хватает. Курсовые модули и содержимое должны следовать определенному порядку. Нам нужно поле, которое позволяет нам легко их упорядочить.

Создание настраиваемых полей модели

Django поставляется с полным набором полей модели, которые вы можете использовать для создания своих моделей. Однако вы также можете создавать свои собственные поля модели для хранения пользовательских данных или изменения поведения существующих полей.

Нам нужно поле, которое позволяет определить последовательность для объектов. Простым способом перечисления последовательности объектов с использованием существующих полей Django является добавление к вашим моделям `PositiveIntegerField`. Используя целые числа, мы можем легко указать порядок объектов. Мы можем создать собственное упорядоченное поле, которое наследуется от `PositiveIntegerField` и обеспечивает дополнительное поведение.

Есть две соответствующие функции, которые мы создадим в нашем поле заказа:

- **Автоматически присваивать значение последовательности, если конкретный порядковый номер не указан:** При сохранении нового объекта без определенного порядкового номера наше поле должно автоматически присваивать номер, который идет после последнего существующего упорядоченного объекта. Если есть два объекта с порядковыми номерами 1 и 2, соответственно, при сохранении третьего объекта мы должны автоматически

присваивать ему номер 3, если не был определен конкретный порядковый номер.

- **Упорядочивать объекты по отношению к другим полям:** Курсовые модули будут упорядочены в отношении курса, к которому они принадлежат, а модули содержимого относительно модуля, к которому они принадлежат.

Создайте новый файл `fields.py` в каталоге приложения `courses` и добавьте к нему следующий код:

Это наш пользовательский `orderField`. Он наследуется от поля `PositiveIntegerField`, предоставленного Django. В нашем поле `orderField` используется необязательный параметр `for_fields`, который позволяет нам указывать поля, которые должны быть упорядочены.

Наше поле переопределяет метод `pre_save()` поля `PositiveIntegerField`, который выполняется перед сохранением поля в базе данных. В этом методе мы выполняем следующие действия:

1. Мы проверяем, существует ли значение для этого поля в экземпляре модели. Мы используем `self.attname`, который является именем атрибута, заданным полем модели. Если значение атрибута отличается от `None`, мы вычисляем порядковый номер, который мы должны дать ему следующим образом:
 1. Мы создаем `QuerySet` для извлечения всех объектов для поля модели. Мы извлекаем класс модели, к которому принадлежит поле, путем доступа к `self.model`.
 2. Мы фильтруем `QuerySet` по текущему значению полей для модели, которые определены в параметре `for_fields` поля, если таковые имеются. Таким образом, мы вычисляем порядок по заданным полям.
3. Мы извлекаем объект с наивысшим порядковым номером `last_item = qs.latest(self.attname)` из базы данных. Если объект не найден, мы предполагаем, что этот объект является первым и

присваивает ему номер 0.

4. Если объект найден, мы добавляем 1 к найденному самому большому порядковому номеру.
 5. Мы присваиваем вычисленный порядковый номер значению поля в экземпляре модели с помощью `setattr()` и возвращаем его.
-
2. Если экземпляр модели имеет порядковый номер для текущего поля, мы ничего не делаем.

Когда вы создаете пользовательские поля модели, сделайте их обобщенными. Избегайте жесткого кодирования, ваше поле должно работать в любой модели.

Вы можете найти дополнительную информацию о написании пользовательских полей модели в <https://docs.djangoproject.com/en/2.0/howto/custom-model-fields/>.

Добавление порядкового номера к модулю и объекту контента

Давайте добавим новое поле в наши модели. Отредактируйте файл `models.py` приложения `courses` и добавте класс `OrderField` и поле в модель `Module` следующим образом:

```
from .fields import OrderField

class Module(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['course'])
```

Мы назовем новое поле `order`, и мы указываем, что порядковый номер вычисляется по курсу, устанавливая `for_fields=['course']`. Это означает, что номер нового модуля будет увеличен на 1 по сравнению с последним порядковым номером модуля того же объекта `Course`. Теперь вы можете отредактировать метод `__str__()` модели `Module`, чтобы упорядочить его следующим образом:

```
class Module(models.Model):
    # ...
    def __str__(self):
        return '{}. {}'.format(self.order, self.title)
```

Содержимое модуля также должно быть упорядочено. Добавьте поле `OrderField` в модель `Content` следующим образом:

```
class Content(models.Model):
    # ...
    order = OrderField(blank=True, for_fields=['module'])
```

На этот раз мы указали, что порядок вычисляется относительно поля `module`. Наконец, давайте добавим порядковый номер по умолчанию для обеих моделей. Добавьте следующий класс `Meta` в модели `Module` и `Content`:

```
class Module(models.Model):
    # ...
    class Meta:
        ordering = ['order']

class Content(models.Model):
    # ...
    class Meta:
        ordering = ['order']
```

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 '{}. {}'.format(self.order, self.title)

class Content(models.Model):
    module = models.ForeignKey(Module,
                               related_name='contents',
                               on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType,
                                     on_delete=models.CASCADE,
                                     limit_choices_to={'model__in':(
                                         'text',
                                         'video',
                                         'image',
                                         '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 модели, которая отображает новые поля для порядкового номера. Откройте консоль и выполните следующую команду:

```
python manage.py makemigrations courses
```

Вы увидите следующий результат:

```
You are trying to add a non-nullable field 'order' to content without a  
default; we can't do that (the database needs something to populate existing  
rows).  
Please select a fix:  
1) Provide a one-off default now (will be set on all existing rows with a  
null value for this column)  
2) Quit, and let me add a default in models.py  
Select an option:
```

Django сообщает нам, что мы должны предоставить значение по умолчанию для нового поля `order` для уже существующих строк в базе данных. Если бы поле имело `null=True`, оно принимало бы нулевые значения, а Django автоматически создавал миграцию вместо запроса значения по умолчанию. Мы можем указать значение по умолчанию или отменить миграцию и добавить атрибут `default` в поле `order` в файле `models.py` перед созданием миграции.

Ведите `1` и нажмите *Enter*, чтобы предоставить значение по умолчанию для существующих записей. Вы увидите следующий результат:

```
Please enter the default value now, as valid Python  
The datetime and django.utils.timezone modules are available, so you can do  
e.g. timezone.now  
Type 'exit' to exit this prompt  
>>>
```

Ведите `0`, чтобы задать значение по умолчанию для существующих записей и нажмите *Enter*. Django запросит у вас значение по умолчанию для модели `Module`. Выберите первый вариант и введите `0` в качестве значения по умолчанию снова. Наконец, вы увидите вывод, похожий на следующий:

```
Migrations for 'courses':  
courses/migrations/0003_auto_20180326_0704.py  
- Change Meta options on content  
- Change Meta options on module  
- Add field order to content  
- Add field order to module
```

Затем примените новые миграции с помощью следующей команды:

```
python manage.py migrate
```

Вывод команды сообщает вам, что миграция была успешно применена, как показано ниже.:

```
Applying courses.0003_auto_20180326_0704... OK
```

Давайте проверим наше новое поле. Откройте консоль следующей командой:

```
python manage.py shell
```

Создайте новый курс следующим образом:

```
>>> 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')
```

Мы создали курс в базе данных. Теперь добавим модули к курсу и посмотрим, как их порядок будет автоматически рассчитан. Мы создаем исходный модуль и проверяем его номер:

```
>>> m1 = Module.objects.create(course=c1, title='Module 1')
>>> m1.order
0
```

`OrderField` устанавливает значение `0`, поскольку это первый объект `Module`, созданный для данного курса. Теперь мы создаем второй модуль для одного и того же курса:

```
>>> m2 = Module.objects.create(course=c1, title='Module 2')
>>> m2.order
1
```

`OrderField` вычисляет следующее значение порядкового номера, добавляя `1` к самому большому значению для существующих объектов. Давайте создадим третий модуль, указав определенный порядковый номер:

```
>>> m3 = Module.objects.create(course=c1, title='Module 3', order=5)
>>> m3.order
5
```

Если мы укажем произвольный порядковый номер, поле `OrderField` не будет использоваться, а значение будет равно указанному порядковому номеру.

Добавим четвертый модуль:

```
>>> m4 = Module.objects.create(course=c1, title='Module 4')
>>> m4.order
6
```

Порядковый номер для этого модуля был автоматически

установлен. Поле `OrderField` не гарантирует, что все порядковые значения будут последовательными. Тем не менее, оно принимает существующие значения и всегда назначает следующий порядковый номер на основе самого высокого существующего значения.

Давайте создадим второй курс и добавим к нему 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
```

Чтобы рассчитать порядковый номер нового модуля, поле учитывает только существующие модули, принадлежащие к одному и тому же курсу. Так как это первый модуль второго курса, результирующий номер `0`. Это связано с тем, что мы указали `for_fields=['course']` в поле `order` модели `Module`.

Поздравляем! Вы успешно создали свое первое настраиваемое поле модели.

Создание CMS

Теперь, когда мы создали универсальную модель данных, мы собираемся создать CMS. CMS позволит преподавателям создавать курсы и управлять их содержанием. Нам необходимо предоставить следующие функции:

- Вход в систему CMS
- Перечисление курсов, созданных инструктором
- Создание, редактирование и удаление курсов
- Добавление модулей в курс и изменение их порядка
- Добавление различных типов контента в каждый модуль и изменение порядка содержимого

Добавление системы аутентификации

Мы собираемся использовать фреймворк аутентификации Django на нашей платформе. Преподаватели и ученики будут экземплярами модели Django `user`, поэтому они смогут войти на сайт с использованием представлений аутентификации

`django.contrib.auth.`

Отредактируйте основной файл `urls.py` проекта `educa` и включите представления `login` и `logout` фреймворка аутентификации Django :

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

urlpatterns = [
    path('accounts/login/', auth_views.LoginView.as_view(), name='login'),
    path('accounts/logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('admin/', admin.site.urls),
]
```

Создание шаблонов аутентификации

Создайте следующую файловую структуру внутри каталога приложения `courses`:

```
templates/
    base.html
    registration/
        login.html
        logged_out.html
```

Перед созданием шаблонов аутентификации нам необходимо подготовить базовый шаблон для нашего проекта. Измените файл шаблона `base.html` и добавьте к нему следующее содержимое:

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>{% block title %}Educa{% endblock %}</title>
    <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
    <div id="header">
        <a href="/" class="logo">Educa</a>
        <ul class="menu">
            {% if request.user.is_authenticated %}
                <li><a href="{% url "logout" %}">Sign out</a></li>
            {% else %}
                <li><a href="{% url "login" %}">Sign in</a></li>
            {% endif %}
        </ul>
    </div>
    <div id="content">
```

```
{% block content %}  
    {% endblock %}  
</div>  
  
<script src="https://ajax.googleapis.com/ajax/libs/jquery/  
3.3.1/jquery.min.js"></script>  
<script>  
    $(document).ready(function() {  
        {% block domready %}  
        {% endblock %}  
    });  
</script>  
</body>  
</html>
```

Это базовый шаблон, который будет расширен остальными шаблонами. В этом шаблоне мы определяем следующие блоки:

- `title`: Блок для других шаблонов для добавления настраиваемого заголовка для каждой страницы.
- `content`: Основной блок для содержимого. Все шаблоны, расширяющие базовый шаблон, должны добавлять контент в этот блок.
- `domready`: Расположен внутри функции `$document.ready()` jQuery. Это позволяет нам выполнять код, когда DOM закончил загрузку.

Стили CSS, используемые в этом шаблоне, находятся в каталоге `static/` приложения `courses` в коде, который поставляется вместе с этой главой. Скопируйте каталог `static/` в такой же каталог вашего проекта, чтобы использовать их.

Отредактируйте шаблон `registration/login.html` и добавьте к нему следующий 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 в систему для представления `login` в Django.

Откройте шаблон `registration/logged_out.html` и добавьте к нему следующий 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 <a href="{% url "login" %}">log-in again</a>.</p>
</div>
{% endblock %}
```

Это шаблон, который будет отображаться пользователю после выхода из системы. Запустите сервер разработки с помощью следующей команды:

```
| python manage.py runserver
```

Откройте <http://127.0.0.1:8000/accounts/login/> в вашем браузере. Вы должны увидеть страницу входа в систему:

The screenshot shows a login interface for a system named 'EDUCA'. At the top, there is a green header bar with the word 'EDUCA' on the left and 'Sign out' on the right. Below the header, the page title 'Log-in' is centered. A message 'Please, use the following form to log-in:' is displayed above two input fields. The first field is labeled 'Username:' and the second is labeled 'Password:', both followed by empty input boxes. At the bottom of the form is a green rectangular button with the text 'LOG-IN' in white capital letters.

EDUCA

Sign out

Log-in

Please, use the following form to log-in:

Username:

Password:

LOG-IN

Создание представлений на основе базовых классов

Мы собираемся создать представления для создания, редактирования и удаления курсов. Для этого мы будем использовать представления на основе классов. Откройте файл `views.py` приложения `courses` и добавьте к нему следующий 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(ManageCourseListView, self).get_queryset()
        return qs.filter(owner=self.request.user)
```

Это представление `ManageCourseListView`. Оно наследуется от базового класса Django `ListView`. Мы переопределяем метод `get_queryset()` представления для получения только курсов, созданных текущим пользователем. Чтобы пользователи не редактировали, не обновляли или не удаляли те курсы, которые они не создавали, нам также необходимо переопределить метод `get_queryset()` в представлениях создания, обновления и удаления. Когда вам нужно предоставить конкретное поведение для нескольких представлений на основе классов, рекомендуется использовать *mixins*.

Использование mixins для представлений на основе базовых классов

Mixins - это особый вид множественного наследования для класса. Вы можете использовать их для обеспечения общей дискретной функциональности, которая добавляется к другим миксинам, позволяет определить поведение класса.

Существуют две основные ситуации для использования mixins:

- Вы хотите предоставить несколько дополнительных функций для класса
- Вы хотите использовать определенную функцию в нескольких классах

Django поставляется с некоторыми миксинами, которые обеспечивают дополнительную функциональность для ваших представлений на основе классов. Вы можете узнать больше о миксинах на <https://docs.djangoproject.com/en/2.0/topics/class-based-views/mixins/>.

Мы собираемся создать класс mixin, который включает в себя общее поведение и использовать его для просмотров курса. Отредактируйте файл `views.py` приложения `courses` и измените его следующим образом:

```
from django.urls import reverse_lazy
from django.views.generic.list import ListView
from django.views.generic.edit import CreateView, UpdateView, \
```

```

DeleteView
from .models import Course

class OwnerMixin(object):
    def get_queryset(self):
        qs = super(OwnerMixin, self).get_queryset()
        return qs.filter(owner=self.request.user)

class OwnerEditMixin(object):
    def form_valid(self, form):
        form.instance.owner = self.request.user
        return super(OwnerEditMixin, self).form_valid(form)

class OwnerCourseMixin(OwnerMixin):
    model = Course

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin):
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')
    template_name = 'courses/manage/course/form.html'

class ManageCourseListView(OwnerCourseMixin, ListView):
    template_name = 'courses/manage/course/list.html'

class CourseCreateView(OwnerCourseEditMixin, CreateView):
    pass

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):
    pass

class CourseDeleteView(OwnerCourseMixin, DeleteView):
    template_name = 'courses/manage/course/delete.html'
    success_url = reverse_lazy('manage_course_list')

```

В этом коде мы создаем миксины `OwnerMixin` и `OwnerEditMixin`. Мы будем использовать их вместе с представлениями `ListView`, `CreateView`, `UpdateView` и `DeleteView`, предоставленными Django. `OwnerMixin` реализует следующий метод:

- `get_queryset()`: Этот метод используется представлениями для получения базового `QuerySet`. Наш `mixin` переопределит этот метод для фильтрации объектов атрибутом `owner` для извлечения объектов, принадлежащих текущему пользователю (`request.user`).

`OwnerEditMixin` реализует следующий метод:

- `form_valid()`: Этот метод используется представлениями, использующими миксин `ModelFormMixin`, то есть представления с формами или с формами на основе моделей, такими как `CreateView` и `UpdateView`. `form_valid()` выполняются, когда отправленная форма действительна. Поведение по умолчанию для этого метода заключается в сохранении экземпляра (для `modelforms`) и перенаправления пользователя на `success_url`. Мы переопределяем этот метод, чтобы автоматически установить текущего пользователя в атрибуте `owner` сохраняемого объекта. Поступая таким образом, мы автоматически устанавливаем владельца для объекта, когда он сохраняется.

Наш класс `OwnerMixin` может использоваться для представлений, которые взаимодействуют с любой моделью, содержащей атрибут `owner`.

Мы также определяем класс `OwnerCourseMixin`, который наследует `OwnerMixin` и предоставляет следующий атрибут для дочерних представлений:

- `model`: Модель, используемая для `QuerySet`. Используется всеми представлениями.

Мы определяем mixin `OwnerCourseEditMixin` со следующими атрибутами:

- `fields`: Поля модели для построения модельной формы в

`CreateView` И `UpdateView` представлениях.

- `success_url`: Используется `CreateView` И `UpdateView` для перенаправления пользователя после успешной отправки формы. Мы используем URL-адрес с именем `manage_course_list`, который мы собираемся создать позже.

Наконец, мы создаем следующие представления, как подкласс `OwnerCourseMixin`:

- `ManageCourseListView`: Перечисляет курсы, созданные пользователем. Он наследует `OwnerCourseMixin` И `ListView`.
- `CourseCreateView`: Использует модель для создания нового объекта `Course`. Он использует поля, определенные в `OwnerCourseEditMixin`, чтобы создать форму модели, а также подклассы `CreateView`.
- `CourseUpdateView`: Позволяет редактировать существующий объект `Course`. Он наследует `OwnerCourseEditMixin` И `UpdateView`.
- `CourseDeleteView`: Наследует от `OwnerCourseMixin` и базового класса `DeleteView`. Определяет `success_url` для перенаправления пользователя после удаления объекта.

Работа с группами и разрешениями

Мы создали основные представления для управление курсами. В настоящее время любой пользователь может получить доступ к этим представлениям. Мы хотим ограничить эти представления, чтобы только преподаватели имели разрешение на создание и управление курсами. Фреймворк аутентификации Django включает в себя систему прав, позволяющую назначать права пользователям и группам. Мы собираемся создать группу для пользователей-инструкторов и дать ей права на создание, обновление и удаление курсов.

Запустите сервер разработки и откройте

`http://127.0.0.1:8000/admin/auth/group/add/` в своем браузере, чтобы создать новый объект `Group`. Добавьте имя `Instructors` и выберите все права для приложения `courses`, кроме тех, что указаны в модели `Subject`, как показано ниже:

Add group

Name:	Instructors	
Permissions:	<p>Available permissions ?</p> <p><input type="text"/> Filter</p> <ul style="list-style-type: none">admin log entry Can add log entryadmin log entry Can change log entryadmin log entry Can delete log entryauth group Can add groupauth group Can change groupauth group Can delete groupauth permission Can add permissionauth permission Can change permissionauth permission Can delete permissionauth user Can add userauth user Can change user	<p>Chosen permissions ?</p> <ul style="list-style-type: none">courses content Can add contentcourses content Can change contentcourses content Can delete contentcourses course Can add coursecourses course Can change coursecourses course Can delete coursecourses file Can add filecourses file Can change filecourses file Can delete filecourses image Can add imagecourses image Can change image <p>Remove all</p>
Choose all ?		
Hold down "Control", or "Command" on a Mac, to select more than one.		
Save and add another Save and continue editing SAVE		

Как вы можете видеть, для каждой модели есть три разных права: *can add*, *can change*, и *can delete*. Выбрав права для этой группы, нажмите кнопку SAVE.

Django автоматически создает права для моделей, но вы также можете создавать пользовательские разрешения. Вы можете больше узнать о добавлении пользовательских прав на <https://docs.djangoproject.com/en/2.0/topics/auth/customizing/#custom-permissions>.

Откройте <http://127.0.0.1:8000/admin/auth/user/add/> и создайте нового пользователя. Отредактируйте пользователя и добавьте его в группу Instructors, как показано ниже:

Groups:

The screenshot shows the 'Groups' section of the Django Admin interface. On the left, there is a sidebar titled 'Available groups' with a search bar labeled 'Filter'. On the right, there is a main pane titled 'Chosen groups' containing the group 'Instructors'. A green '+' button is located in the top right corner of the main pane. Below the main pane, there are two small circular arrows, one pointing left and one pointing right, likely for navigating between pages or filtering results.

Пользователи наследуют права доступа групп, к которым они принадлежат, но вы также можете добавлять индивидуальные разрешения конкретному пользователю с использованием сайта администрирования. Пользователи, у которых `is_superuser` установлено на `True`, получают все права автоматически.

Ограничение доступа к представлениям на основе базовых классов

Мы собираемся ограничить доступ к представлениям, чтобы только пользователи с соответствующими разрешениями могли добавлять, изменять или удалять объекты `course`. Мы будем использовать следующие две миксины, предоставляемые `django.contrib.auth`, чтобы ограничить доступ к представлениям:

- `LoginRequiredMixin`: Реплицирует функциональность декоратора `login_required`.
- `PermissionRequiredMixin`: Предоставляет доступ к представлению пользователям, имеющим конкретное разрешение. Помните, что суперпользователи автоматически имеют все разрешения.

Отредактируйте файл `views.py` приложения `courses` и добавьте следующий импорт:

```
from django.contrib.auth.mixins import LoginRequiredMixin, \
    PermissionRequiredMixin
```

Сделайте `OwnerCourseMixin` наследуемым от `LoginRequiredMixin` следующим образом:

```
class OwnerCourseMixin(OwnerMixin, LoginRequiredMixin):  
    model = Course
```

```
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')
```

Затем добавьте атрибут `permission_required` в представления создания, обновления и удаления, как показано ниже:

```
class CourseCreateView(PermissionRequiredMixin,
                      OwnerCourseEditMixin,
                      CreateView):
    permission_required = 'courses.add_course'

class CourseUpdateView(PermissionRequiredMixin,
                      OwnerCourseEditMixin,
                      UpdateView):
    permission_required = 'courses.change_course'

class CourseDeleteView(PermissionRequiredMixin,
                      OwnerCourseMixin,
                      DeleteView):
    template_name = 'courses/manage/course/delete.html'
    success_url = reverse_lazy('manage_course_list')
    permission_required = 'courses.delete_course'
```

`PermissionRequiredMixin` проверяет, имеет ли пользователь доступ к представлению, в соответствии с разрешением указанным в атрибуте `permission_required`. Теперь наши представления доступны только пользователям, имеющим соответствующие разрешения.

Давайте создадим URL-адреса этих представлений. Создайте новый файл внутри каталога приложений `courses` и назовите его `urls.py`. Добавьте к нему следующий 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 для представлений курса: list, create, edit, и delete. Отредактируйте основной файл `urls.py` проекта `educa` и включите шаблоны URL-адреса приложения `courses`, как показано ниже:

```
from django.urls import path, include

urlpatterns = [
    path('accounts/login/', auth_views.LoginView.as_view(), name='login'),
    path('accounts/logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('admin/', admin.site.urls),
    path('course/', include('courses.urls')),
]
```

Нам нужно создать шаблоны для этих представлений. Создайте следующие каталоги и файлы в каталоге `templates`/директории `courses`:

```
courses/
  manage/
    course/
      list.html
      form.html
      delete.html
```

Откройте `courses/manage/course/list.html` и добавьте следующий код в него:

```
{% 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>
                <a href="{% url "course_edit" course.id %}">Edit</a>
                <a href="{% url "course_delete" course.id %}">Delete</a>
            </p>
        </div>
    {% empty %}
    <p>You haven't created any courses yet.</p>
    {% endfor %}
    <p>
        <a href="{% url "course_create" %}" class="button">Create new
course</a>
    </p>
</div>
{% endblock %}
```

Это шаблон для представления `ManageCourseListView`. В этом шаблоне мы перечислим курсы, созданные текущим пользователем. Мы включаем ссылки для редактирования или удаления каждого курса и ссылку для создания новых курсов.

Запустите сервер разработки с помощью команды `python manage.py runserver`. Откройте `http://127.0.0.1:8000/accounts/login/?next=/course/mine/` в своем браузере и войдите в систему под пользователем, который принадлежит к группе `Instructors`. После входа в систему вы будете перенаправлены на `http://127.0.0.1:8000/course/mine/` и вы должны увидеть следующую страницу:

My courses

You haven't created any courses yet.

[CREATE NEW COURSE](#)

На этой странице будут отображены все курсы, созданные текущим пользователем.

Давайте создадим шаблон, который отобразит форму для создания и обновления представлений курса. Откройте шаблон `courses/manage/course/form.html` и добавьте следующий код:

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

{% block title %}
    {% if object %}
        Edit course "{{ object.title }}"
    {% else %}
        Create a new course
    {% endif %}
{% endblock %}

{% block content %}
<h1>
    {% if object %}
        Edit course "{{ object.title }}"
    {% else %}
        Create a new course
    {% endif %}
</h1>
<div class="module">
    <h2>Course info</h2>
```

```
<form action"." method="post">
{{ form.as_p }}
{% csrf_token %}
<p><input type="submit" value="Save course"></p>
</form>
</div>
{% endblock %}
```

Шаблон `form.html` используется для представлений `CourseCreateView` и `CourseUpdateView`. В этом шаблоне мы проверяем, находится ли в контексте переменная `object`. Если в контексте существует `object`, мы знаем, что мы обновляем существующий курс, и используем его в названии страницы. В противном случае мы создаем новый объект `Course`.

Откройте `http://127.0.0.1:8000/course/mine/` в вашем браузере и нажмите на кнопку **CREATE NEW COURSE**. Вы должны увидеть следующую страницу:

Create a new course

Course info

Subject:

Title:

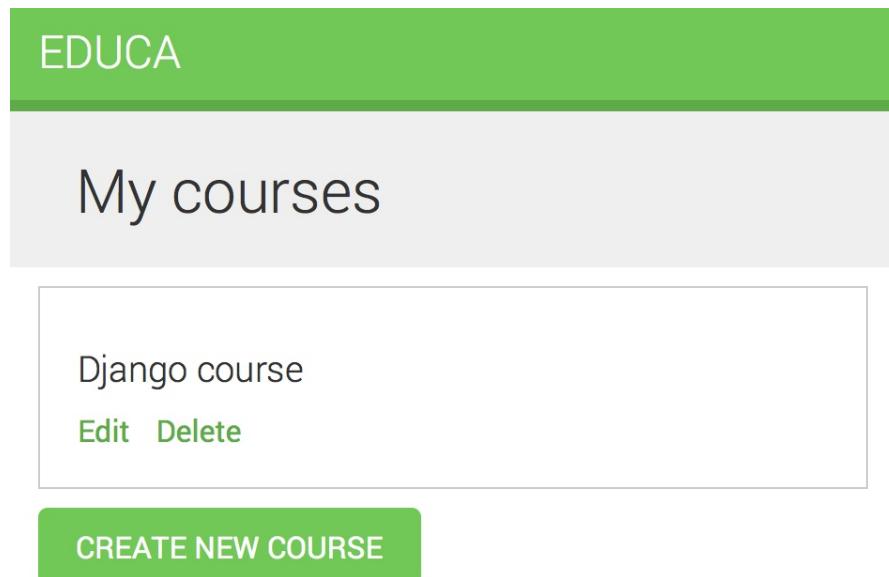
Slug:

Overview:

SAVE COURSE

Заполните форму и нажмите кнопку SAVE COURSE. Курс будет сохранен, и вы будете перенаправлены на страницу списка

курсов. Она должна выглядеть следующим образом:



Затем щелкните ссылку `Edit` для выбранного курса. Вы снова увидите форму, но на этот раз вы редактируете существующий объект `Course` вместо того, чтобы создавать новый.

Наконец, откройте шаблон `courses/manage/course/delete.html` и добавьте следующий код:

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

{% block title %}Delete course{% endblock %}

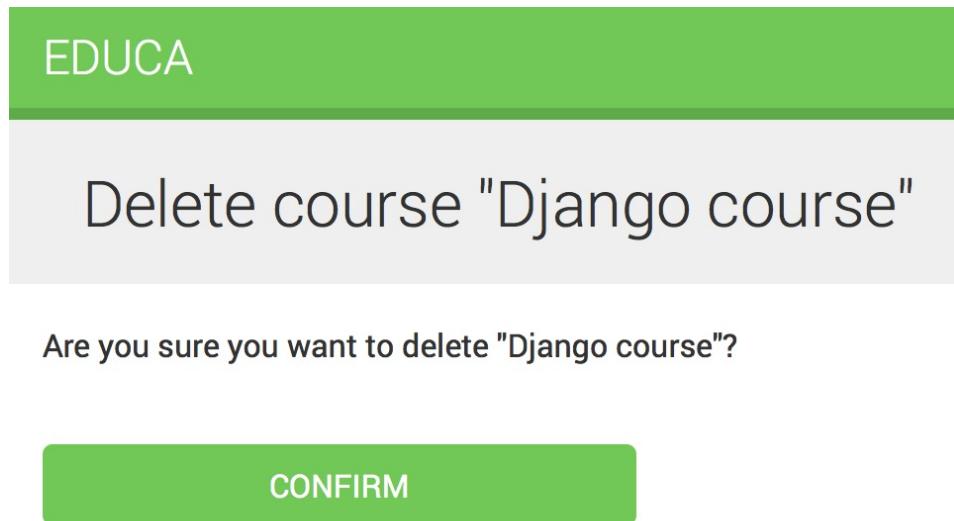
{% block content %}
    <h1>Delete course "{{ object.title }}"</h1>

    <div class="module">
        <form action="" method="post">
            {% csrf_token %}
            <p>Are you sure you want to delete "{{ object }}"?</p>
            <input type="submit" class="button" value="Confirm">
        </form>
    </div>
{% endblock %}
```

Это шаблон для представления `CourseDeleteView`. Это

представление наследуется от `DeleteView`, предоставленного Django, который ожидает подтверждения пользователя для удаления объекта.

Откройте ваш браузер и нажмите ссылку Delete вашего курса. Вы должны увидеть следующую страницу подтверждения:



Нажмите кнопку CONFIRM. Курс будет удален, и вы снова будете перенаправлены на страницу списка курсов.

Теперь инструкторы могут создавать, редактировать и удалять курсы. Затем мы должны предоставить им CMS для добавления модулей и содержимого в курсы. Мы начнем с управления модулями курса.

Управление модулями курса и содержимым

Мы собираемся создать систему для управления модулями курса и их содержимым. Нам нужно будет создавать формы, которые могут использоваться для управления несколькими модулями для каждого курса и различными типами контента для каждого модуля. Оба модуля и содержимое должны будут следовать определенному порядку, и мы должны иметь возможность их переупорядочивать с помощью CMS.

Использование наборов форм (formset) для модулей курса

Django имеет слой абстракции для работы с несколькими формами на одной странице. Эти наборы форм известны как *formset*. Formset управляют несколькими экземплярами определенной формы `Form` или `ModelForm`. Все формы подаются сразу, а formset заботится о первоначальном количестве форм для отображения, ограничивая максимальное количество форм, которые могут быть представлены, и проверяя все формы.

Formset включает метод `is_valid()` для проверки всех форм одновременно. Вы также можете предоставить начальные данные для форм и указать, сколько дополнительных пустых форм отобразить.

Вы можете узнать больше о formset на <https://docs.djangoproject.com/en/2.0/topics/forms/formsets/> и о модели formset на <https://docs.djangoproject.com/en/2.0/topics/forms/modelforms/#model-formsets>.

Поскольку курс делится на переменное количество модулей, имеет смысл использовать formset для управления ими. Создайте файл `forms.py` в каталоге приложений `courses` и добавьте к нему следующий 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 formset `ModuleFormSet`. Мы создаем его с помощью функции `inlineformset_factory()`, предоставляемой Django. Встроенный formset - это небольшая абстракция поверх набора форм, которая упрощают работу со связанными объектами. Эта функция позволяет нам динамически строить модельный formset для объектов `Module`, связанных с объектом `course`.

Мы используем следующие параметры для построения formset:

- `fields`: Поля, которые будут включены в каждую форму formset.
- `extra`: Позволяет нам установить количество пустых дополнительных форм для отображения в formset.
- `can_delete`: Если вы установите значение `True`, Django будет содержать логическое поле для каждой формы, которое будет отображаться как входной флагок. Он позволяет отмечать объекты, которые вы хотите удалить.

Отредактируйте файл `views.py` приложения `courses` добавив к нему следующий код:

```
from django.shortcuts import redirect, get_object_or_404
from django.views.generic.base import TemplateResponseMixin, View
from .forms import ModuleFormSet

class CourseModuleUpdateView(TemplateResponseMixin, View):
    template_name = 'courses/manage/module/formset.html'
    course = None

    def get_formset(self, data=None):
```

```

        return ModuleFormSet(instance=self.course,
                             data=data)

    def dispatch(self, request, pk):
        self.course = get_object_or_404(Course,
                                         id=pk,
                                         owner=request.user)
        return super(CourseModuleUpdateView,
                     self).dispatch(request, pk)

    def get(self, request, *args, **kwargs):
        formset = self.get_formset()
        return self.render_to_response({'course': self.course,
                                       'formset': formset})

    def post(self, request, *args, **kwargs):
        formset = self.get_formset(data=request.POST)
        if formset.is_valid():
            formset.save()
            return redirect('manage_course_list')
        return self.render_to_response({'course': self.course,
                                       'formset': formset})

```

Представление `CourseModuleUpdateView` обрабатывает `formset` для добавления, обновления и удаления модулей для определенного курса. Это представление наследуется от следующих миксинов и представлений:

- `TemplateResponseMixin`: Этот mixin отвечает за рендеринг шаблонов и возврат HTTP-ответа. Для этого требуется атрибут `template_name`, который указывает на визуализируемый шаблон и предоставляет метод `render_to_response()`, чтобы передать ему контекст и отобразить шаблон.
- `view`: Основное представление базовых классов, предоставленное Django.

В этом представлении мы реализуем следующие методы:

- `get_formset()`: Мы определяем этот метод, чтобы избежать повторения кода для построения formset. Мы создаем объект `ModuleFormSet` для данного объекта `course` с дополнительными данными.
- `dispatch()`: Этот метод предоставляется классом `view`. Он принимает HTTP-запрос и его параметры и пытается делегировать нижестоящему методу, который соответствует используемому методу HTTP: запрос `GET` делегируется методу `get()` а `POST` запросит `post()`, соответственно. В этом методе мы используем функцию `get_object_or_404()`, чтобы получить объект `course` для данного параметра `id`, который принадлежит текущему пользователю. Мы включаем этот код в метод `dispatch()`, потому что нам нужно получить курс для запросов `GET` и `POST`. Мы сохраняем его в атрибуте `course` представления, чтобы сделать его доступным для других методов.
- `get()`: Выполняется для запросов `GET`. Мы создаем пустой `formset ModuleFormSet` и переносим его в шаблон вместе с текущим объектом `course`, используя метод `render_to_response()`, предоставленный `TemplateResponseMixin`.
- `post()`: Выполняется для запросов `POST`. В этом методе мы выполняем следующие действия:
 1. Мы создаем экземпляр `ModuleFormSet`, используя представленные данные.
 2. Мы выполняем `formset` метод `is_valid()` для проверки всех его форм.
 3. Если `formset` действителен, мы сохраняем его,

вызывая метод `save()`. На данный момент любые изменения, такие как добавление, обновление или маркировка модулей для удаления, применяются к базе данных. Затем мы перенаправляем пользователей на URL `manage_course_list`. Если `formset` недействителен, мы создаем шаблон для отображения любых ошибок.

Отредактируйте файл `urls.py` приложения `courses` и добавьте к нему следующий шаблон URL:

```
path('<pk>/module/',
     views.CourseModuleUpdateView.as_view(),
     name='course_module_update'),
```

Создайте новый каталог внутри каталога шаблонов `courses/manage/` и назовите его `module`. Создайте шаблон `courses/manage/module/formset.html` и добавьте к нему следующий 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 action="" method="post">
{{ formset }}
{{ formset.management_form }}
{% csrf_token %}
<input type="submit" class="button" value="Save modules">
</form>
</div>
{% endblock %}
```

В этом шаблоне мы создаем HTML-элемент `<form>`, в который мы включаем `formset`. Мы также включаем форму управления для `formset` с переменной `{{formset.management_form}}`. Менеджер форм включает скрытые поля для управления начальным, итоговым, минимальным и максимальным количеством форм. Как вы видите, создать набор форм очень просто.

Откройте шаблон `courses/manage/course/list.html` и добавьте следующую ссылку для URL `course_module_update` ниже ссылок редактирования и удаления курса:

```
<a href="{% url "course_edit" course.id %}">Edit</a>
<a href="{% url "course_delete" course.id %}">Delete</a>
<a href="{% url "course_module_update" course.id %}">Edit
modules</a>
```

Мы добавили ссылку для редактирования модулей курса. Откройте `http://127.0.0.1:8000/course/mine/` в своем браузере. Создайте курс и нажмите ссылку Edit modules. Вы должны увидеть `formset` следующим образом:

Edit "Django course"

Course modules

Title:

Description:

Delete:

Title:

Description:

Delete:

SAVE MODULES

Formset включает форму для каждого объекта `Module`, содержащегося в курсе. После этого отображаются две пустые дополнительные формы, потому что мы устанавливаем `extra=2` для `ModuleFormSet`. Когда вы сохраните formset, Django будет

включать еще два дополнительных поля для добавления новых модулей.

Добавление контента в модули курса

Теперь нам нужен способ добавления контента в учебные модули. У нас есть четыре разных типа контента: текст, видео, изображение и файл. Мы можем рассмотреть возможность создания четырех разных представлений для создания контента, по одному для каждой модели. Тем не менее мы собираемся использовать более общий подход и создать представление, которое обрабатывает создание или обновление объектов любой модели контента.

Откройте файл `views.py` приложения `courses` и добавьте в него следующий код:

```
from django.forms.models import modelform_factory
from django.apps import apps
from .models import Module, Content

class ContentCreateUpdateView(TemplateResponseMixin, View):
    module = None
    model = None
    obj = None
    template_name = 'courses/manage/content/form.html'

    def get_model(self, model_name):
        if model_name in ['text', 'video', 'image', 'file']:
            return apps.get_model(app_label='courses',
                                  model_name=model_name)
        return None

    def get_form(self, model, *args, **kwargs):
        Form = modelform_factory(model, exclude=['owner',
                                                'order',
                                                'created',
                                                'updated'])
        return Form(*args, **kwargs)
```

```
def dispatch(self, request, module_id, model_name, id=None):
    self.module = get_object_or_404(Module,
                                    id=module_id,
                                    course__owner=request.user)
    self.model = self.get_model(model_name)
    if id:
        self.obj = get_object_or_404(self.model,
                                    id=id,
                                    owner=request.user)
    return super(ContentCreateUpdateView,
                self).dispatch(request, module_id, model_name, id)
```

Это первая часть `contentCreateUpdateView`. Она позволит нам создавать и обновлять содержимое различных моделей. Это представление определяет следующие методы:

- `get_model()`: Здесь мы проверяем, является ли данное имя модели одной из четырех моделей контента: `Text`, `Video`, `Image`, или `File`. Затем мы используем модуль приложений Django для получения фактического класса для данного имени модели. Если данное имя модели не действительно, мы возвращаемся `None`.
- `get_form()`: Мы создаем динамическую форму с помощью функции `modelform_factory()` фреймворка форм. Поскольку мы собираемся создать форму для моделей `Text`, `Video`, `Image` и `File`, мы используем параметр `exclude`, чтобы указать поля для исключения из формы а все остальные атрибуты включить автоматически. Поступая таким образом, нам не обязательно знать, какие поля включать в зависимости от модели.
- `dispatch()`: Она получает следующие параметры URL и сохраняет соответствующий модуль, модель и объект контента в качестве атрибутов класса:

- `module_id`: Идентификатор модуля, с которым будет связан/использоваться контент.
- `model_name`: Название модели контента для создания/обновления.
- `id`: Идентификатор объекта, который обновляется. Либо `None` для создания новых объектов.

Добавьте следующие методы `get()` и `post()` к `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})
```

Эти методы заключаются в следующем:

- `get()`: Выполняется при получении запроса `GET`. Мы создаем форму модели для экземпляра `Text`, `Video`, `Image`,

или `file`, который обновляется. В противном случае мы не передаем экземпляр для создания нового объекта, так как `self.obj` равен `None`, если идентификатор не указан.

- `post()`: Выполняется при получении запроса `POST`. Мы строим модельную форму, передавая ей любые переданные данные и файлы. Затем мы ее проверяем. Если форма действительна, мы создаем новый объект и назначаем `request.user` его владельцем, прежде чем сохранить в базе данных. Мы проверяем параметр `id`. Если идентификатор не указан, мы знаем, что пользователь создает новый объект вместо обновления существующего. Если это новый объект, мы создаем объект `Content` для данного модуля и сопоставляем ему новый контент.

Откройте файл `urls.py` приложения `courses` и добавьте следующие URL паттерны в него:

```
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 содержат следующее:

- `module_content_create`: Для создания новых текстов, видео, изображений или файлов и добавления их в модуль. Он включает параметры `module_id` и `model_name`. Первый позволяет связать новый объект контента с данным

модулем. Последняя специфицирует модель контента для построения формы.

- `module_content_update`: Обновление существующего текста, видео, изображения или файла. Он включает параметры `module_id` и `model_name` и параметр `id` для идентификации обновляемого контента.

Создайте новый каталог внутри каталога шаблонов `courses/manage/` и назовите его `content`. Создайте шаблон `courses/manage/content/form.html` и добавьте к нему следующий код:

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

{% block title %}
    {% if object %}
        Edit content "{{ object.title }}"
    {% else %}
        Add a new content
    {% endif %}
{% endblock %}

{% block content %}
<h1>
    {% if object %}
        Edit content "{{ object.title }}"
    {% else %}
        Add a new content
    {% endif %}
</h1>
<div class="module">
    <h2>Course info</h2>
    <form action="" method="post" enctype="multipart/form-data">
        {{ form.as_p }}
        {% csrf_token %}
        <p><input type="submit" value="Save content"></p>
    </form>
</div>
{% endblock %}
```

Это шаблон для представления `ContentCreateUpdateView`. В этом

шаблоне мы проверяем, находится ли в контексте переменная `object`. Если в контексте существует `object`, мы обновляем существующий объект. В противном случае мы создаем новый объект.

Мы включаем `enctype="multipart/form-data"` в HTML-элемент `<form>` потому что форма содержит загрузку файла для моделей контента `File` и `Image`.

Запустите сервер разработки, откройте `http://127.0.0.1:8000/course/mine/`, нажмите `Edit modules` для существующего курса и создайте модуль. Откройте консоль Python командой `python manage.py shell` и получите идентификатор последнего созданного модуля следующим образом:

```
>>> from courses.models import Module  
>>> Module.objects.latest('id').id  
6
```

Запустите сервер разработки и откройте `http://127.0.0.1:8000/course/module/6/content/image/create/` в своем браузере, заменив идентификатор модуля тем, который вы получили ранее. Вы увидите форму для создания объекта `Image`, как показано ниже:

Add a new content

Course info

Title:

File:

no file selected

SAVE CONTENT

Пока еще не отправляйте форму. Если вы попытаетесь это сделать, она не сработает, потому что мы еще не определили URL `module_content_list`. Мы собираемся создать это позже.

Нам также нужно представление для удаления содержимого. Измените файл `views.py` приложения `courses` и добавьте следующий 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 данным идентификатором; он удаляет связанный объект `Text`, `Video`, `Image` или `File` и, наконец, он удаляет объект `Content` и перенаправляет пользователя на URL `module_content_list`, чтобы перечислить другое содержимое модуля.

Откройте файл `urls.py` приложения `courses` и добавьте в него следующий URL паттерн:

```
path('content/<int:id>/delete/',
      views.ContentDeleteView.as_view(),
      name='module_content_delete'),
```

Теперь инструкторы могут легко создавать, обновлять и удалять содержимое.

Управление модулями и контентом

Мы добавили представления для создания, редактирования и удаления модулей и содержимого курса. Теперь нам нужны представления для отображения всех модулей и содержимого курса и для отображения списка контента для определенного модуля.

Откройте файл `views.py` приложения `courses` и добавьте следующий код в него:

```
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` получает объект `Module` с идентификатором, который принадлежит текущему пользователю, и отображает шаблон с данным модулем.

Отредактируйте файл `urls.py` приложения `courses` и добавьте к нему следующий шаблон URL:

```
path('module/<int:module_id>',
      views.ModuleContentListView.as_view(),
      name='module_content_list'),
```

Создайте новый шаблон внутри каталога

templates/courses/manage/module/ И назовите его content_list.html. Добавьте к нему следующий код:

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

{% block title %}
    Module {{ module.order|add:1 }}: {{ module.title }}
{% endblock %}

{% block content %}
    {% with course=module.course %}
        <h1>Course "{{ course.title }}"</h1>
        <div class="contents">
            <h3>Modules</h3>
            <ul id="modules">
                {% for m in course.modules.all %}
                    <li data-id="{{ m.id }}" {% if m == module %}class="selected"{% endif %}>
                        <a href="{% url "module_content_list" m.id %}">
                            <span>
                                Module <span class="order">{{ m.order|add:1 }}</span>
                            </span>
                            <br>
                            {{ m.title }}
                        </a>
                    </li>
                {% empty %}
                    <li>No modules yet.</li>
                {% endfor %}
            </ul>
            <p><a href="{% url "course_module_update" course.id %}">
                Edit modules</a></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>
                            <a href="#">Edit</a>
                            <form action="{% url "module_content_delete" content.id %}"
                                method="post">
                                <input type="submit" value="Delete">
                                {% csrf_token %}
                            </form>
                        {% endwith %}
                    </div>
                {% empty %}
                    <div>No contents yet.</div>
                {% endfor %}
            </div>
        </div>
    {% endwith %}
{% endblock %}
```

```
        {% endwith %}
    </div>
    {% empty %}
        <p>This module has no contents yet.</p>
    {% endfor %}
</div>
<h3>Add new content:</h3>
<ul class="content-types">
    <li><a href="{% url "module_content_create" module.id "text" %}">
        Text</a></li>
    <li><a href="{% url "module_content_create" module.id "image" %}">
        Image</a></li>
    <li><a href="{% url "module_content_create" module.id "video" %}">
        Video</a></li>
    <li><a href="{% url "module_content_create" module.id "file" %}">
        File</a></li>
</ul>
</div>
{% endwith %}
{% endblock %}
```

Это шаблон, который отображает все модули для курса и содержимое выбранного модуля. Мы перебираем модули курса, чтобы отображать их на боковой панели. Мы перебираем содержимое модуля и получаем доступ к `content.item`, чтобы получить связанный `Text`, `Video`, `Image` ИЛИ `File`. Мы также включаем ссылки для создания нового текста, видео, изображения или содержимого файла.

Мы хотим знать, к какому типу принадлежит каждый из элементов объекта: текст, видео, изображение ИЛИ файл. Нам нужно имя модели чтобы создать URL-адрес для редактирования объекта. Помимо этого, мы могли бы отображать каждый элемент в шаблоне по-разному, в зависимости от типа содержимого. Мы можем получить модель для объекта из класса `Meta` модели, обратившись к атрибуту `_meta` объекта. Тем не менее Django не разрешает доступ к переменным или атрибутам шаблона, начинающимся с подчеркивания, чтобы предотвратить получение частных атрибутов или вызовов частных методов. Мы можем решить эту проблему, написав специальный фильтр шаблонов.

Создайте следующую файловую структуру внутри каталога приложения `courses`:

```
templatetags/  
    __init__.py  
    course.py
```

Откройте файл `course.py` и добавьте к нему следующий 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 шаблон фильтра `model_name`. Мы можем применить его в шаблонах как `object|model_name` чтобы получить имя модели для объекта.

Откройте шаблон `templates/courses/manage/module/content_list.html` и добавьте следующую строку под тегом шаблона `{% extends %}`:

```
{% load course %}
```

Это загрузит теги шаблона `course`. Затем найдите строки:

```
<p>{{ item }}</p>  
<a href="#">Edit</a>
```

Замените их следующими:

```
<p>{{ item }} ({{ item|model_name }})</p>  
<a href="{% url "module_content_update" module.id item|model_name item.id
```

```
| %}">Edit</a>
```

Теперь мы показываем модель элемента в шаблоне и используем имя модели для создания ссылки на редактируемый объект. Измените шаблон `courses/manage/course/list.html` и добавьте ссылку на URL-адрес `module_content_list` следующим образом:

```
<a href="{% url "course_module_update" course.id %}">Edit modules</a>
{% if course.modules.count > 0 %}
    <a href="{% url "module_content_list" course.modules.first.id %}">
        Manage contents</a>
    {% endif %}
```

Новая ссылка позволяет пользователям получать доступ к содержимому первого модуля курса, если такой имеются.

Откройте `http://127.0.0.1:8000/course/mine/` и нажмите ссылку `Manage contents` на курс, содержащий хотя бы один модуль. Вы увидите страницу, похожую на следующую:

Course "Django course"

Modules

MODULE 1
Introduction to Django

[Edit modules](#)

Module 1: Introduction to Django

Module contents:

This module has no contents yet.

Add new content:

[Text](#)

[Image](#)

[Video](#)

[File](#)

Когда вы нажимаете на модуль на левой боковой панели, его содержимое отображается в основной области. Шаблон также включает ссылки для добавления нового текста, видео, изображения или содержимого файла для отображаемого модуля. Добавьте в модуль несколько различных типов контента и посмотрите результат. Содержимое появится после Module contents, как показано в следующем примере:

Course "Django course"

Modules

MODULE 1

Introduction to Django

MODULE 2

Configuring Django

[Edit modules](#)

Module 2: Configuring Django

Module contents:

Setting up Django (text)

[Edit](#)

[Delete](#)

Example settings.py (image)

[Edit](#)

[Delete](#)

Add new content:

[Text](#)

[Image](#)

[Video](#)

[File](#)

Переупорядочивание модулей и контента

Нам нужно предоставить простой способ изменения порядка модулей курса и их содержимого. Мы будем использовать виджеты drag-n-drop JavaScript, чтобы наши пользователи могли изменить порядок модулей курса, перетащив их. Когда пользователи заканчивают перетаскивание модуля, мы запустим асинхронный запрос (AJAX), чтобы сохранить новый порядок модуля.

Использование mixins из django-braces

`django-braces` является сторонним модулем, который содержит коллекцию общих миксинов для Django. Эти миксины предоставляют дополнительные функции для представлений на основе классов. Вы можете увидеть список всех миксинов, предоставленных `django-braces` на <https://django-braces.readthedocs.io/>.

Мы будем использовать следующие миксины `django-braces`:

- `CsrfExemptMixin`: Чтобы избежать проверки токена CSRF в запросах `POST`. Нам это нужно для выполнения запросов AJAX `POST` без создания `csrf_token`.
- `JsonRequestResponseMixin`: Разбирает данные запроса как JSON, а также сериализует ответ как JSON и возвращает ответ HTTP с типом содержимого `application/json`.

Установите `django-braces` через `pip` используя следующую команду:

```
pip install django-braces==1.13.0
```

Нам нужно представление, которое получает новый порядок идентификаторов модулей, закодированных в JSON. Отредактируйте файл `views.py` приложения `courses` и добавьте к нему следующий код:

```
from braces.views import CsrfExemptMixin, JsonRequestResponseMixin
```

```
class ModuleOrderView(CsrftokenExemptMixin,
                      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`.

Мы можем создать аналогичное представление для упорядочивания контента модуля. Добавьте следующий 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` и добавьте следующие шаблоны URL-адресов:

```
path('module/order/',
      views.ModuleOrderView.as_view(),
      name='module_order'),

path('content/order/',
      views.ContentOrderView.as_view(),
      name='content_order'),
```

Наконец, нам нужно реализовать функцию drag-n-drop в шаблоне. Для этого мы будем использовать библиотеку пользовательского интерфейса jQuery. Пользовательский интерфейс jQuery построен поверх jQuery и предоставляет набор взаимодействий, эффектов и виджетов интерфейса. Мы

будем использовать элемент `sortable`. Во-первых, нам нужно загрузить jQuery UI в базовый шаблон. Откройте файл `base.html`, расположенный в каталоге `templates/` приложения `courses`, и добавьте jQuery UI под скриптом для загрузки jQuery следующим образом:

```
<script  
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">  
</script>  
<script src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.12.1/jquery-  
ui.min.js"></script>
```

Мы загружаем библиотеку пользовательского интерфейса jQuery сразу после фреймворка jQuery. Теперь отредактируйте шаблон `courses/manage/module/content_list.html` и добавьте следующий код к нему в нижней части шаблона:

```
{% block domready %}  
$( '#modules' ).sortable({  
    stop: function(event, ui) {  
        modules_order = {};  
        $( '#modules' ).children().each(function(){  
            // update the order field  
            $(this).find('.order').text($(this).index() + 1);  
            // associate the module's id with its order  
            modules_order[$(this).data('id')] = $(this).index();  
        });  
        $.ajax({  
            type: 'POST',  
            url: '{% url "module_order" %}',  
            contentType: 'application/json; charset=utf-8',  
            dataType: 'json',  
            data: JSON.stringify(modules_order)  
        });  
    }  
});  
  
$( '#module-contents' ).sortable({  
    stop: function(event, ui) {  
        contents_order = {};  
        $( '#module-contents' ).children().each(function(){  
            // associate the module's id with its order  
            contents_order[$(this).data('id')] = $(this).index();  
        });  
    }  
});
```

```
$.ajax({
    type: 'POST',
    url: '{% url "content_order" %}',
    contentType: 'application/json; charset=utf-8',
    dataType: 'json',
    data: JSON.stringify(contents_order),
});
}
});
{% endblock %}
```

Этот JavaScript код находится в блоке `{% block domready %}`, и поэтому он будет включен в событие `$(document).ready()` jQuery, которое мы определили в шаблон `base.html`. Это гарантирует, что наш код JavaScript будет выполнен после загрузки страницы. Мы определяем элемент `sortable` для списка модулей на боковой панели, а другой для списка содержимого модуля. Оба работают аналогичным образом. В этом коде мы выполняем следующие задачи:

1. Сначала мы определяем элемент `sortable` для HTML-элемента `modules`. Помните, что мы используем `#modules`, поскольку jQuery использует нотацию CSS для селекторов.
2. Мы указываем функцию для события `stop`. Это событие запускается каждый раз, когда пользователь заканчивает сортировку элемента.
3. Мы создаем пустой словарь `modules_order`. Ключами для этого словаря будет идентификатор модулей, а значения будут назначены для каждого модуля.
4. Мы перебираем дочерние элементы `#module`. Мы пересчитываем отображаемый порядок для каждого модуля и получаем его атрибут `data-id`, который содержит идентификатор модуля. Мы добавляем идентификатор в качестве ключа словаря `modules_order` и новый индекс

модуля в качестве значения.

5. Мы выполняем запрос AJAX POST к `content_order` URL, включая сериализованные данные JSON `modules_order` в запросе. Соответствующий `ModuleOrderView` выполняет обновление порядка модулей.

Элемент `sortable` для упорядочения содержимого очень похож на этот. Вернитесь в свой браузер и перезагрузите страницу. Теперь вы сможете щелкнуть и перетащить оба модуля и содержимое, чтобы изменить порядок, как в следующем примере:

Course "Django course"

Modules

MODULE 1
Introduction to Django

MODULE 2
Configuring Django

Edit modules

Module 2: Configuring Django

contents:

Setting up Django (text)	Edit	Delete
Example settings.py (image)	Edit	Delete

Add new content:

[Text](#)

[Image](#)

[Video](#)

[File](#)

Великолепно! Теперь вы можете изменить порядок как модулей курса, так и содержимого модуля.

Резюме

В этой главе вы узнали, как создать универсальную CMS. Вы использовали наследование модели и создали поле настраиваемой модели. Вы также работали с классами и миксинами. Вы создали набор форм и систему для управления различными типами контента.

В следующей главе вы создадите систему регистрации учеников. Вы также будете отображать различные типы контента, и вы узнаете, как работать с фреймворком кэша Django.

Глава 11

Отображение и кеширование контента

В предыдущей главе вы использовали наследование моделей и общие отношения для создания гибких моделей контента курса. Вы также создали систему управления курсом с использованием представлений на основе классов, форм и запроса AJAX для содержимого. В этой главе вы:

- Создадите общедоступные представления для отображения информации о курсе
- Создадите систему регистрации учеников
- Научитесь управлять зачислением учащихся на курсы
- Будите предоставлять разнообразный контент курса
- Научитесь работать с содержимым кэша используя фреймворк кэша

Мы начнем с создания учебного каталога для студентов, чтобы просмотреть существующие курсы и иметь возможность записаться на них.

Отображение курсов

Для нашего каталога курсов мы должны создать следующие функции:

- Список всех доступных курсов (отфильтрованных по предмету, но это необязательно)
- Отображение обзора одного курса

Откройте файл `views.py` приложения `courses` и добавьте следующий 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 наследуется от `TemplateResponseMixin` и `View`. В этом представлении мы выполняем следующие задачи:

1. Мы извлекаем все предметы, включая общее количество курсов для каждого из них. Мы используем метод ORM `annotate()` с помощью функции агрегации `Count()`, чтобы включить общее количество курсов для каждого предмета.
2. Мы извлекаем все доступные курсы, включая общее количество модулей, содержащихся в каждом курсе.
3. Если задан `slug` параметр в URL-адресе предмета, мы получаем соответствующий этому предмету объект, и мы ограничиваем запрос курсами, принадлежащими данному предмету.
4. Мы используем метод `render_to_response()`, предоставляемый `TemplateResponseMixin`, чтобы отобразить объекты в шаблоне и вернуть HTTP-ответ.

Давайте создадим подробное представление для отображения обзора одного курса. Добавьте следующий 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 наследуется от общего `DetailView`, предоставленного Django. Мы указываем атрибуты `model` и `template_name`. Django класс `DetailView` ожидает первичный ключ (`pk`) или `slug` параметр URL для извлечения одного объекта для данной модели. Затем он отображает шаблон, указанный в `template_name`, включая объект в `context` как `object`.

Отредактируйте основной файл `urls.py` проекта `educa` и добавьте к нему следующий шаблон URL:

```
from courses.views import CourseListView

urlpatterns = [
    # ...
    path('', CourseListView.as_view(), name='course_list'),
]
```

Мы добавляем URL шаблон `course_list` в основной файл проекта `urls.py`, потому что мы хотим отобразить список курсов в URL `http://127.0.0.1:8000/` а все остальные URL-адреса для приложения `courses` имеют префикс `/course/`.

Отредактируйте файл `urls.py` приложения `courses` и добавьте следующие шаблоны URL:

```
path('subject/<slug:subject>/',
      views.CourseListView.as_view(),
      name='course_list_subject'),

path('<slug:slug>',
      views.CourseDetailView.as_view(),
      name='course_detail'),
```

Мы определяем следующие шаблоны URL:

- `course_list_subject`: Для отображения всех курсов для предмета
- `course_detail`: Для отображения обзора одного курса

Давайте создадим шаблоны для представлений `CourseListView` и `CourseDetailView`.

Создайте следующую файловую структуру внутри каталога `templates/courses/` приложения курсов:

```
course/
  list.html
  detail.html
```

Откройте шаблон `courses/course/list.html` и запишите в него следующий 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 %}>
            <a href="{% url "course_list" %}">All</a>
        </li>
        {% for s in subjects %}
            <li {% if subject == s %}class="selected"{% endif %}>
                <a href="{% url "course_list_subject" s.slug %}">
                    {{ s.title }}
                    <br><span>{{ s.total_courses }} courses</span>
                </a>
            </li>
        {% endfor %}
    </ul>
</div>
<div class="module">
    {% for course in courses %}
        {% with subject=course.subject %}
            <h3><a href="{% url "course_detail" course.slug %}">
                {{ course.title }}</a></h3>
            <p>
                <a href="{% url "course_list_subject" subject.slug %}">
                    {{ subject }}</a>.
                {{ course.total_modules }} modules.
            </p>
        {% endwith %}
    {% endfor %}
</div>
```

```
Instructor: {{ course.owner.get_full_name }}
```

```
</p>
```

```
{% endwith %}
```

```
{% endfor %}
```

```
</div>
```

```
{% endblock %}
```

Это шаблон для перечисления доступных курсов. Мы создаем список HTML для отображения всех объектов `Subject` и создаем ссылку на URL-адрес `course_list_subject` для каждого из них. Мы добавляем HTML-класс `selected`, чтобы выделить текущий объект, если он есть. Мы перебираем каждый объект `course`, отображая общее количество модулей и имя инструктора.

Запустите сервер разработки и откройте `http://127.0.0.1:8000/` в своем браузере. Вы должны увидеть страницу, похожую на следующую:

All courses

Subjects

All

Mathematics

1 COURSES

Music

0 COURSES

Physics

0 COURSES

Programming

2 COURSES

Django course

Programming. 2 modules. Instructor: Antonio Melé

Python for beginners

Programming. 2 modules. Instructor: Laura Marlon

Algebra basics

Mathematics. 4 modules. Instructor: Laura Marlon

Левая панель содержит все предметы, включая общее количество курсов для каждого из них. Вы можете щелкнуть любой объект, чтобы отфильтровать отображаемые курсы.

Откройте шаблон `courses/course/detail.html` и добавьте в него следующий код:

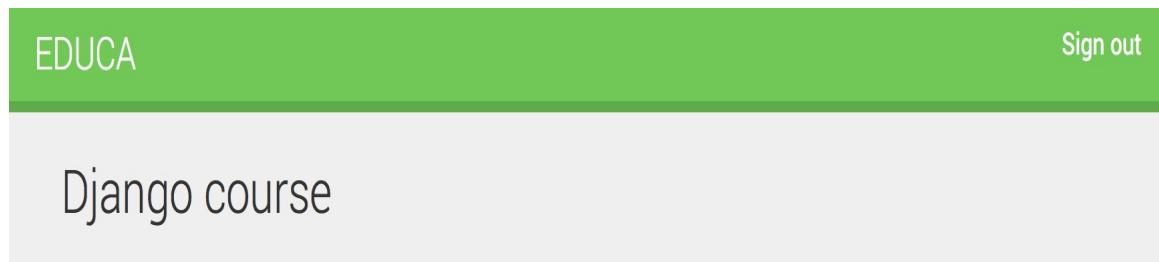
```
{% extends "base.html" %}

{% block title %}
    {{ object.title }}
{% endblock %}

{% block content %}
    {% with subject=course.subject %}
        <h1>
            {{ object.title }}
        </h1>
    {% endwith %}
{% endblock %}
```

```
<div class="module">
    <h2>Overview</h2>
    <p>
        <a href="{% url "course_list_subject" subject.slug %}">
            {{ subject.title }}</a>.
        {{ course.modules.count }} modules.
        Instructor: {{ course.owner.get_full_name }}
    </p>
    {{ object.overview|linebreaks }}
</div>
{% endwith %}
{% endblock %}
```

В этом шаблоне мы показываем обзор и детали для одного курса. Откройте <http://127.0.0.1:8000/> в своем браузере и щелкните по одному из курсов. Вы должны увидеть страницу со следующей структурой:



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.

Мы создали общедоступную область для показа курсов. Теперь мы должны разрешить пользователям регистрироваться в качестве студентов и записываться на курсы.

Добавление регистрации учащихся

Создайте новое приложение, используя следующую команду:

```
python manage.py startapp students
```

Измените файл `settings.py` проекта `educa` и добавьте новое приложение в настройку `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [  
    # ...  
    'students.apps.StudentsConfig',  
]
```

Создание представления регистрации учащихся

Откройте файл `views.py` приложения `students` и добавьте следующий 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(StudentRegistrationView,
                      self).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 учащимся регистрироваться на нашем сайте. Мы используем базовый класс `CreateView`, который предоставляет функциональные возможности для создания объектов модели. Для этого представления требуются следующие атрибуты:

- `template_name`: Путь шаблона для отображения этого представления.
- `form_class`: Форма для создания объектов, которая должна

быть `ModelForm`. Мы используем `Django UserCreationForm` в качестве регистрационной формы для создания объектов `User`.

- `success_url`: URL-адрес для перенаправления пользователя, когда форма успешно отправлена. Мы вернемся к URL `student_course_list`, который мы собираемся создать в разделе *Доступ к содержимому курса* для перечисления студентов посещающих курсы.

Метод `form_valid()` выполняется, когда были отправлены действительные данные формы. Он должен вернуть HTTP-ответ. Мы переопределяем этот метод для аутентификации пользователя после успешной регистрации.

Создайте новый файл в каталоге приложений `students` и назовите его `urls.py`. Добавьте к нему следующий код:

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

urlpatterns = [
    path('register/',
        views.StudentRegistrationView.as_view(),
        name='student_registration'),
]
```

Затем отредактируйте основной `urls.py` проекта `educa` и включите URL-адреса для приложения `students`, добавив следующий шаблон в конфигурацию ваших URL-адресов:

```
urlpatterns = [
    # ...
    path('students/', include('students.urls')),
]
```

Создайте следующую файловую структуру внутри каталога приложений `students`:

```
templates/
    students/
        student/
            registration.html
```

Отредактируйте шаблон `students/student/registration.html` добавив к нему следующий 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 action="" method="post">
            {{ form.as_p }}
            {% csrf_token %}
            <p><input type="submit" value="Create my account"></p>
        </form>
    </div>
{% endblock %}
```

Запустите сервер разработки и откройте `http://127.0.0.1:8000/students/register/` в своем браузере. Вы должны увидеть регистрационную форму следующим образом:

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

Обратите внимание, что URL `student_course_list`, указанный в атрибуте `success_url` в представлении `StudentRegistrationView` еще не существует. Если вы отправите форму, Django не найдет URL для перенаправления после успешной регистрации. Мы создадим этот URL-адрес в разделе *Доступ к содержимому курса*.

Зачисление на курсы

После того, как пользователи создадут учетную запись, они смогут зарегистрироваться на курсах. Чтобы хранить заявки, нам необходимо создать отношения «многие ко многим» между моделями `Course` и `User`.

Измените файл `models.py` приложения `courses` и добавьте следующее поле в модель `Course`:

```
students = models.ManyToManyField(User,
                                 related_name='courses_joined',
                                 blank=True)
```

Из консоли выполните следующую команду чтобы создать миграции для этого изменения:

```
python manage.py makemigrations
```

Вы увидите результат:

```
Migrations for 'courses':
courses/migrations/0004_course_students.py
- Add field students to course
```

Затем выполните следующую команду, чтобы применить миграции:

```
python manage.py migrate
```

Вы должны увидеть вывод, который заканчивается следующей строкой:

```
Applying courses.0004_course_students... OK
```

Теперь мы можем связать студентов с курсами, на которые они зачислены. Давайте создадим функциональность чтобы студенты могли записаться на курсы.

Создайте новый файл в каталоге приложения `students` и назовите его `forms.py`. Добавьте к нему следующий код:

```
from django import forms
from courses.models import Course

class CourseEnrollForm(forms.Form):
    course = forms.ModelChoiceField(queryset=Course.objects.all(),
                                    widget=forms.HiddenInput)
```

Мы собираемся использовать эту форму для того, чтобы студенты могли поступить на курсы. Поле `course` предназначено для курса, в котором пользователь регистрируется. Поэтому оно `ModelChoiceField`. Мы используем виджет `HiddenInput`, потому что мы не будем показывать это поле пользователю. Мы собираемся использовать эту форму в представлении `CourseDetailView`, чтобы отобразить кнопку для регистрации.

Откройте файл `views.py` приложения `students` и добавте следующий 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(StudentEnrollCourseView,
                    self).form_valid(form)
```

```
def get_success_url(self):
    return reverse_lazy('student_course_detail',
                        args=[self.course.id])
```

Это представление `StudentEnrollCourseView`. Оно обрабатывает студентов регистрирующихся на курсах. Представление наследуется от миксины `LoginRequiredMixin`, чтобы доступ к представлению могли получить только зарегистрированные пользователи. Оно также наследует представление `Django FormView`, поскольку мы обрабатываем представление формы. Мы используем форму `CourseEnrollForm` для атрибута `form_class`, а также определяем атрибут `course` для хранения данного объекта `course`. Когда форма действительна, мы добавляем текущего пользователя к учащимся, обучающимся на курсе.

Метод `get_success_url()` возвращает URL-адрес, по которому пользователь будет перенаправлен, если форма была успешно отправлена. Этот метод эквивалентен атрибуту `success_url`. Мы вернемся к URL `student_course_detail`, который мы создадим в следующем разделе *Доступ к содержимому курса*, чтобы отобразить содержимое курса.

Отредактируйте файл `urls.py` приложения `students` и добавьте к нему следующий шаблон URL:

```
path('enroll-course/',
      views.StudentEnrollCourseView.as_view(),
      name='student_enroll_course'),
```

Давайте добавим форму кнопки регистрации на странице обзора курса. Откройте файл `views.py` приложения `courses` и измените `CourseDetailView`, 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(CourseDetailView,
                    self).get_context_data(**kwargs)
    context['enroll_form'] = CourseEnrollForm(
        initial={'course':self.object})
    return context
```

Мы используем метод `get_context_data()` чтобы включить форму регистрации в `context` для визуализации шаблона. Мы инициализируем скрытое поле курса формы с текущим объектом `course`, чтобы он мог быть отправлен напрямую.

Откройте шаблон `courses/course/detail.html` и найдите следующую сторку:

```
 {{ object.overview|linebreaks }}
```

Замените ее следующим кодом:

```
 {{ object.overview|linebreaks }}
{% if request.user.is_authenticated %}
    <form action="{% url "student_enroll_course" %}" method="post">
        {{ enroll_form }}
        {% csrf_token %}
        <input type="submit" class="button" value="Enroll now">
    </form>
{% else %}
    <a href="{% url "student_registration" %}" class="button">
        Register to enroll
    </a>
{% endif %}
```

Это кнопка для зачисления на курсы. Если пользователь аутентифицирован, мы показываем кнопку регистрации, включая скрытую форму, указывающую на URL `student_enroll_course`. Если пользователь не аутентифицирован, мы показываем ссылку для регистрации на платформе.

Убедитесь, что сервер разработки запущен, откройте <http://127.0.0.1:8000/> в своем браузере и щелкните по курсу. Если вы вошли в систему, вы должны увидеть кнопку ENROLL NOW, расположенную ниже обзора курса, следующим образом:

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](#)

Если вы не вошли в систему, вы увидите вместо этого кнопку REGISTER TO ENROLL

Доступ к содержимому курса

Нам нужно представление для отображения курсов, и представление для доступа к фактическому содержимому курса. Измените файл `views.py` приложения `students` и добавьте к нему следующий код:

```
from django.views.generic.list import ListView
from courses.models import Course

class StudentCourseListView(LoginRequiredMixin, ListView):
    model = Course
    template_name = 'students/course/list.html'

    def get_queryset(self):
        qs = super(StudentCourseListView, self).get_queryset()
        return qs.filter(students__in=[self.request.user])
```

Это представление для студентов, оно перечисляет курсы, на которые те зарегистрированы. Оно наследуется от `LoginRequiredMixin`, чтобы убедиться, что только зарегистрированные пользователи могут получить доступ к представлению. Оно также наследуется от базового класса `ListView` для отображения списка объектов `course`. Мы переопределяем метод `get_queryset()` для извлечения только курсов, на которые зачислен пользователь; мы фильтруем `QuerySet` с помощью поля `ManyToManyField` для этого студента.

Затем добавьте следующий код в файл `views.py`:

```
from django.views.generic.detail import DetailView

class StudentCourseDetailView(DetailView):
    model = Course
    template_name = 'students/course/detail.html'
```

```
def get_queryset(self):
    qs = super(StudentCourseDetailView, self).get_queryset()
    return qs.filter(students__in=[self.request.user])

def get_context_data(self, **kwargs):
    context = super(StudentCourseDetailView,
                    self).get_context_data(**kwargs)
    # получить объект курс
    course = self.get_object()
    if 'module_id' in self.kwargs:
        # получить текущий модуль
        context['module'] = course.modules.get(
            id=self.kwargs['module_id'])
    else:
        # получить первый модуль
        context['module'] = course.modules.all()[0]
    return context
```

Это `StudentCourseDetailView`. Мы переопределяем метод `get_queryset()`, чтобы ограничить базовый `QuerySet` курсами, на которых пользователь зарегистрирован. Мы также переопределяем метод `get_context_data()`, чтобы установить модуль курса в `context`, если указан параметр URL `module_id`. В противном случае мы устанавливаем первый модуль курса. Таким образом, учащиеся смогут перемещаться по модулю внутри курса.

Измените файл `urls.py` приложения `students` и добавьте к нему следующие шаблоны URL:

```
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'),
```

Создайте следующую файловую структуру внутри каталога `templates/students/`:

```
course/
    detail.html
    list.html
```

Отредактируйте шаблон `students/course/list.html` и добавьте к нему следующий код:

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

{% block title %}My courses{% endblock %}

{% block content %}
    <h1>My courses</h1>

    <div class="module">
        {% for course in object_list %}
            <div class="course-info">
                <h3>{{ course.title }}</h3>
                <p><a href="{% url "student_course_detail" course.id %}">
                    Access contents</a></p>
            </div>
        {% empty %}
        <p>
            You are not enrolled in any courses yet.
            <a href="{% url "course_list" %}">Browse courses</a>
            to enroll in a course.
        </p>
        {% endfor %}
    </div>
{% endblock %}
```

Этот шаблон отображает курсы, на которые пользователь зарегистрирован. Помните, что когда новый студент успешно регистрируется на платформе, он будет перенаправлен на URL `student_course_list`. Давайте также перенаправим учащихся на этот URL-адрес при входе на платформу.

Отредактируйте файл `settings.py` проекта `educa` и добавьте к нему следующий код:

```
from django.urls import reverse_lazy
LOGIN_REDIRECT_URL = reverse_lazy('student_course_list')
```

Это параметр, используемый модулем auth для перенаправления пользователя после успешного входа в систему, если в запросе отсутствует следующий параметр. После успешного входа в систему ученики будут перенаправлены на URL student_course_list, чтобы просмотреть курсы, на которые они записаны.

Откройте шаблон students/course/detail.html и добавьте к нему следующий 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 %}>
                    <a href="{% url "student_course_detail_module"
                    object.id m.id %}">
                        <span>
                            Module <span class="order">{{ m.order|add:1 }}</span>
                        </span>
                        <br>
                        {{ m.title }}
                    </a>
                </li>
            {% empty %}
                <li>No modules yet.</li>
            {% endfor %}
        </ul>
    </div>
    <div class="module">
        {% for content in module.contents.all %}
```

```
{% with item=content.item %}  
    <h2>{{ item.title }}</h2>  
    {{ item.render }}  
{% endwith %}  
{% endfor %}  
</div>  
{% endblock %}
```

Это шаблон для зарегистрированных учащихся чтобы они имели доступ к содержимому курса. Сначала мы создаем список HTML, включающий все модули курса и выделям текущий модуль. Затем мы перебираем содержимое текущего модуля и получаем доступ к каждому элементу содержимого, чтобы отобразить его с помощью `{{ item.render }}`. После этого мы добавим метод `render()` к модели контента. Этот метод позаботится о корректном отображении контента.

Отображение различных типов контента

Нам необходимо предоставить способ отображения каждого типа контента. Отредактируйте файл `models.py` приложения `courses` и добавьте метод `render()` в модель `ItemBase`:

```
from django.template.loader import render_to_string
from django.utils.safestring import mark_safe

class ItemBase(models.Model):
    # ...

    def render(self):
        return render_to_string('courses/content/{}.html'.format(
            self._meta.model_name), {'item': self})
```

Этот метод использует функцию `render_to_string()` для отображения шаблона и возврата содержимого в виде строки. Каждый вид контента создается с использованием шаблона, названного в честь модели содержимого. Мы используем `self._meta.model_name` для динамической генерации соответствующего имени шаблона для каждой модели контента. Метод `render()` предоставляет общий интерфейс для отображения разнообразного контента.

Создайте следующую структуру файлов внутри каталога `templates/courses/` приложения `courses`:

```
content/
    text.html
    file.html
    image.html
    video.html
```

Откройте шаблон `courses/content/text.html` и добавьте этот код:

```
    {{ item.content|linebreaks|safe }}
```

Откройте шаблон `courses/content/file.html` и добавьте следующее:

```
<p><a href="{{ item.file.url }}" class="button">Download file</a></p>
```

Отредактируйте шаблон `courses/content/image.html` и запишите в него:

```
<p></p>
```

Для файлов, загруженных с помощью `ImageField` и `FileField` нам нужно настроить наш проект для работы с медиафайлами на сервере разработки. Откройте файл `settings.py` вашего проекта и добавьте к нему следующий код:

```
MEDIA_URL = '/media/'  
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

Помните, что `MEDIA_URL` - это базовый URL для загрузки медиафайлов, а `MEDIA_ROOT` - это локальный путь, где находятся файлы.

Отредактируйте основной файл `urls.py` вашего проекта и добавьте следующие импорты:

```
from django.conf import settings  
from django.conf.urls.static import static
```

Затем допишите следующие строки в конце файла:

```
if settings.DEBUG:  
    urlpatterns += static(settings.MEDIA_URL,
```

```
|     document_root=settings.MEDIA_ROOT)
```

Теперь ваш проект готов к загрузке и обслуживанию медиафайлов. Сервер разработки Django будет отвечать за обслуживание медиафайлов во время разработки (то есть, когда параметр `DEBUG` установлен в `True`). Помните, что сервер разработки не подходит для использования в производственных целях. Вы узнаете, как настроить производственную среду в [главе 13, "Продвижение в жизнь"](#).

Мы также должны создать шаблон для отображения объектов `Video`. Мы будем использовать `django-embed-video` для встраивания видеоконтента. `django-embed-video` является сторонним приложением Django, которое позволяет вставлять видео в свои шаблоны из таких источников, как YouTube или Vimeo, просто используя общедоступный URL видео.

Установите пакет с помощью следующей команды:

```
| pip install django-embed-video==1.1.2
```

Измените файл `settings.py` вашего проекта и добавьте приложение в `INSTALLED_APPS`, установив следующее:

```
| INSTALLED_APPS = [  
|     # ...  
|     'embed_video',  
| ]
```

Документацию приложения `django-embed-video` можно найти на <https://django-embed-video.readthedocs.io/en/latest/>.

Откройте шаблон `courses/content/video.html` и добавьте следующий код:

```
| {% load embed_video_tags %}  
| {% video item.url "small" %}
```

Теперь запустите сервер разработки и откройте <http://127.0.0.1:8000/course/mine/> в вашем браузере.

Зайдите на сайт под пользователем, входящим в группу `Instructors`, и добавьте несколько экземпляров на курс. Чтобы включить видеоконтент, вы можете просто скопировать любой URL-адрес YouTube, например <https://www.youtube.com/watch?v=bgV39D1mz2U>, и добавить его к полю формы `url`.

После добавления содержимого к курсу откройте <http://127.0.0.1:8000/>, кликните курс и нажмите на кнопку ENROLL NOW. Вы должны быть зачислены на курс и перенаправлены на URL `student_course_detail`. На следующем скриншоте показан пример содержимого курса:

Introduction to Django

Modules

MODULE 1
Introduction to Django

MODULE 2
Configuring Django

MODULE 3
Your first Django project

MODULE 4
Django URLs

Why Django?

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.

Django video



Великолепно! Вы создали интерфейс для визуализации различных типов содержимого курса.

Использование фреймворка кэша

HTTP-запросы к вашему веб-приложению обычно влекут за собой доступ к базе данных, обработку данных и отображение шаблонов. Это намного дороже с точки зрения обработки, чем обслуживание статического веб-сайта.

Накладные расходы в некоторых запросах могут быть значительными, когда ваш сайт начинает получать все больше и больше трафика. Здесь кэширование становится необходимым. Кэшируя запросы, результаты расчета или отображаемые данные в HTTP-запросе, вы избежите дорогостоящих операций в следующих запросах. Это приводит к сокращению времени отклика и меньшей нагрузке на стороне сервера.

Django включает надежную систему кеша, которая позволяет кэшировать данные с разной степенью детализации. Вы можете кэшировать один запрос, вывод определенного представления, частей отображаемого содержимого шаблона или всего вашего сайта. Элементы, по умолчанию, хранятся в системе кэширования определенное время. Вы можете указать тайм-аут по умолчанию для кэшированных данных.

Вот как вы обычно будете использовать структуру кэша, когда ваше приложение получит HTTP-запрос:

1. Пробуете найти запрошенные данные в кеше
2. Если найдены, вернете кэшированные данные
3. Если не найдены, выполняете следующие действия:

1. Выполняете запрос или обработку, необходимые для получения данных
2. Сохраняете сгенерированные данные в кеше
3. Возвращаете данные

Вы можете прочитать подробную информацию о кеш-системе Django на <https://docs.djangoproject.com/en/2.0/topics/cache/>.

Доступные бэкэнды кэша

Django поставляется с несколькими кеш-серверами. Вот они:

- `backends.memcached.MemcachedCache` ИЛИ `backends.memcached.PyLibMCCache`:
Бэкэнд Memcached. Memcached - это быстрый и эффективный сервер кеша на основе памяти.
Использование бэкэнда зависит от выбранных вами привязок Memcached Python.
- `backends.db.DatabaseCache`: Использует базу данных как кеш-систему.
- `backends.filebased.FileBasedCache`: Использует систему хранения файлов. Сериализует и сохраняет каждое значение кеша в виде отдельного файла.
- `backends.locmem.LocMemCache`: Бэкэнд локальной памяти. Это бэкэнда по умолчанию.
- `backends.dummy.DummyCache`: Бэкэнд-заглушка, предназначенный только для разработки. Он реализует интерфейс кеша, фактически не кэшируя что-либо. Этот кеш предназначен для каждого процесса и потокобезопасен.

Для обеспечения оптимальной производительности используйте бэкэнд на основе памяти, такой как Memcached.

Установка Memcached

Мы будем использовать бэкэнд Memcached. Memcached запускается в памяти и под него выделяется определенный объем оперативной памяти. Когда выделенное ОЗУ заполнено, Memcached начинает удаление самых старых данных для хранения новых данных.

Загрузите Memcached из <https://memcached.org/downloads>. Если вы используете Linux, вы можете установить Memcached, используя следующую команду:

```
./configure && make && make test && sudo make install
```

Если вы используете macOS X, вы можете установить Memcached с через Homebrew менеджер с помощью команды `brew install memcached`. Вы можете скачать Homebrew из <https://brew.sh/>.

После установки Memcached откройте консоль и запустите его используя следующую команду:

```
memcached -l 127.0.0.1:11211
```

По умолчанию Memcached будет работать на порту 11211. Однако вы можете указать собственный хост и порт с помощью параметра `-l`. Вы можете найти более подробную информацию о Memcached по адресу <https://memcached.org>.

После установки Memcached вы должны установить программу для связи с Python. Вы можете сделать это с помощью следующей команды:

```
pip install python-memcached==1.59
```

Настройки кеша

Django предоставляет следующие настройки кеша:

- `CACHES`: Словарь, содержащий все доступные кеши для проекта.
- `CACHE_MIDDLEWARE_ALIAS`: Псевдоним кеша.
- `CACHE_MIDDLEWARE_KEY_PREFIX`: Префикс для использования ключей кеша.
Установите префикс, чтобы избежать коллизий ключей, если вы используете один и тот же кеш между несколькими сайтами.
- `CACHE_MIDDLEWARE_SECONDS`: По умолчанию используется количество секунд для кэширования страниц.

Систему кэширования для проекта можно настроить с помощью параметра `CACHES`. Этот параметр является словарем, который позволяет указать конфигурацию для нескольких кешей. Каждый кеш, включенный в словарь `CACHES`, может устанавливать следующие данные:

- `BACKEND`: Бэкэнд для использования кэша.
- `KEY_FUNCTION`: Стока, содержащая путь к вызову, который принимает префикс, версию и ключ в качестве аргументов и возвращает окончательный ключ кеша.

- `KEY_PREFIX`: Префикс строки для всех ключей кеша, чтобы избежать коллизий.
- `LOCATION`: Расположение кеша. В зависимости от бэкэда это может быть каталог, хост и порт или имя для кэшапамяти.
- `OPTIONS`: Любые дополнительные параметры, которые должны быть переданы в бэкэнд кэша.
- `TIMEOUT`: Тайм-аут по умолчанию, в секундах, для хранения ключей кеша. 300 секунд по умолчанию, что составляет пять минут. Если установлено значение `None`, ключи кеша не потеряют своей актуальности.
- `VERSION`: Номер версии по умолчанию для ключей кеша. Полезно для управления версиями кешей.

Добавление Memcached в ваш проект

Давайте настроим кеш для нашего проекта. Откройте файл `settings.py` проекта `educa` и добавьте к нему следующий код:

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',  
        'LOCATION': '127.0.0.1:11211',  
    }  
}
```

Мы используем бэкэнд `MemcachedCache`. Мы указываем его местоположение, используя нотацию `address:port`. Если у вас несколько экземпляров Memcached, вы можете использовать список для `LOCATION`.

Мониторинг Memcached

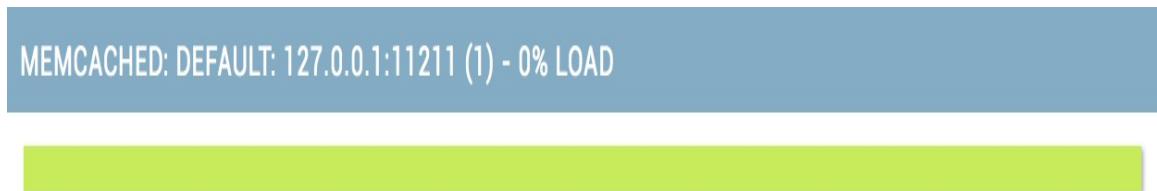
Чтобы контролировать Memcached, мы будем использовать сторонний пакет с именем `django-memcache-status`. Это приложение отображает статистику для ваших экземпляров Memcached на сайте администрирования. Установите его с помощью следующей команды:

```
pip install django-memcache-status==1.3
```

Откройте файл `settings.py` и добавте '`memcache_status`' к настройкам `INSTALLED_APPS`:

```
INSTALLED_APPS = [  
    # ...  
    'memcache_status',  
]
```

Убедитесь, что Memcached запущен, запустите сервер разработки в другом окне консоли и откройте `http://127.0.0.1:8000/admin/` в своем браузере. Войдите в административный сайт как суперпользователь. Вы должны увидеть следующий блок:



MEMCACHED: DEFAULT: 127.0.0.1:11211 (1) - 0% LOAD

На этом графике показано использование кеша. Зеленый цвет представляет собой свободный кеш, а красный обозначает использованное пространство. Если вы нажмете заголовок

окна, он отобразит подробную статистику вашего экземпляра Memcached.

Мы создали Memcached для нашего проекта и можем его контролировать. Давайте начнем кэшировать данные!

Уровни кэширования

Django предоставляет следующие уровни кэширования, перечисленные здесь, по возрастанию порядка детализации:

- **Низко-уровневый API кэша:** Обеспечивает самую высокую степень детализации. Позволяет кэшировать определенные запросы или вычисления.
- **Кэширование представлений:** Обеспечивает кэширование отдельных представлений.
- **Кэширование шаблонов:** Позволяет кэшировать фрагменты шаблона.
- **Кэширование сайта:** Кэш самого высокого уровня. Он кэширует весь сайт.

Перед реализацией кэширования подумайте о своей стратегии кэширования. Сначала сосредоточьтесь на дорогих запросах или вычислениях, которые не рассчитываются для каждого пользователя.

Использование низкоуровневого API кэша

Низкоуровневый API-интерфейс кэша позволяет хранить объекты в кеше с любой детализацией. Он находится в `django.core.cache`. Вы можете импортировать его так:

```
from django.core.cache import cache
```

Для этого используется кеш по умолчанию. Это эквивалентно кешам `caches['default']`. Доступ к определенному кешу также возможен через его псевдоним:

```
from django.core.cache import caches
my_cache = caches['alias']
```

Давайте посмотрим, как работает API кеширования. Откройте консоль командой `python manage.py shell` и выполните следующий код:

```
>>> from django.core.cache import cache
>>> cache.set('musician', 'Django Reinhardt', 20)
```

Мы задействуем кеш-память по умолчанию и используем `set(key, value, timeout)` для хранения ключа с именем `'musician'` со значением, которое является строкой `'Django Reinhardt'` в течение 20 секунд. Если мы не укажем тайм-аут, Django использует тайм-аут по умолчанию, указанный для бэкэнда кэша в настройке `CACHES`. Теперь выполните следующий код:

```
>>> cache.get('musician')
```

```
'Django Reinhardt'
```

Мы извлекаем ключ из кеша. Подождите 20 секунд и выполните тот же код:

```
>>> cache.get('musician')
```

На этот раз значение не возвращается. Срок жизни ключа кеша 'musician' истек, и метод `get()` возвращает `None`, потому что ключ больше не находится в кеше.

Всегда избегайте сохранения значения `None` в ключе кеша, потому что вы не сможете отличить фактическое значение кеша от отсутствующего.

Давайте кэшируем `QuerySet` со следующим кодом:

```
>>> from courses.models import Subject
>>> subjects = Subject.objects.all()
>>> cache.set('all_subjects', subjects)
```

Мы выполняем `QuerySet` в модели `Subject` и сохраняем возвращенные объекты в '`all_subjects`'. Получаем кэшированные данные:

```
>>> cache.get('all_subjects')
<QuerySet [<Subject: Mathematics>, <Subject: Music>, <Subject: Physics>,
<Subject: Programming>]>
```

Мы собираемся кэшировать некоторые запросы в наших представлениях. Отредактируйте файл `views.py` приложения `courses` и добавьте следующий импорт:

```
from django.core.cache import cache
```

В методе `get()` `CourseListView` найдите следующую строку:

```
subjects = Subject.objects.annotate(  
    total_courses=Count('courses'))
```

Замените ее на следующие:

```
subjects = cache.get('all_subjects')  
if not subjects:  
    subjects = Subject.objects.annotate(  
        total_courses=Count('courses'))  
    cache.set('all_subjects', subjects)
```

В этом коде мы пытаемся получить ключ `all_subjects` из кеша, используя `cache.get()`. Он возвращает `None`, если данный ключ не найден. Если ключ не найден (не кэширован или истек тайм-аут кеширования), мы выполняем запрос для извлечения всех объектов `Subject` и их количества курсов, и мы кэшируем результат, используя `cache.set()`.

Запустите сервер разработки и откройте <http://127.0.0.1:8000/> в вашем браузере. Когда представление вызывается, ключ кеша не будет найден и `QuerySet` исполнится. Откройте <http://127.0.0.1:8000/admin/> в своем браузере и разверните статистику Memcached. Вы должны увидеть данные об использовании для кеша, аналогичного следующему экрану:

MEMCACHED: DEFAULT: 127.0.0.1:11211 (1) - 0% LOAD



Miss Ratio 38%



Avg GET by item 1

Avg GET by seconds/minutes 0/0

Detailed Statistics:

Pid 12606

Uptime 0y, 0d, 0h, 43m, 12s

Time 03/30/18 17:13:02

Version 1.4.20

Libevent 2.0.21-stable

Взгляните на Curr Items, который должен быть 1. Это показывает, что в кеше хранится один элемент. Get Hits показывает, сколько команд закончилось успешно, а Get Misses показывает запросы на получение отсутствующих ключей. Miss Ratio рассчитывается с использованием обоих из них.

Теперь вернитесь к <http://127.0.0.1:8000/>, используя ваш браузер, и повторно загрузите страницу несколько раз. Если вы сейчас просмотрите статистику кеша, вы увидите несколько больше чтений (Get Hits и Cmd Get будет увеличиваться).

Кэширование на основе динамических данных

Часто вам будет необходимо кэширование, основанное на динамических данных. В этих случаях вам нужно создать динамические ключи, которые содержат всю информацию, необходимую для однозначной идентификации кэшированных данных. Откройте файл `views.py` приложения `courses` и измените представление `courseListView`, чтобы он выглядел следующим образом:

В этом случае мы также кэшируем как все курсы, так и курсы, отфильтрованные по теме. Мы используем ключ кеша `all_courses` для хранения всех курсов, если объект не указан. Если есть предмет, мы динамически строим ключ с помощью

```
'subject_{}_courses'.format(subject.id).
```

Важно отметить, что вы не можете использовать кэшированный `QuerySet` для создания других `QuerySets`, поскольку то, что вы кэшировали, на самом деле является результатом `QuerySet`. Поэтому вы не можете сделать следующее:

```
courses = cache.get('all_courses')
courses.filter(subject=subject)
```

Вместо этого вам нужно создать базовый `QuerySet` `Course.objects.annotate(total_modules=Count('modules'))`, который не будет выполняться до тех пор, пока он не будет запущен принудительно, и использовать его для дальнейшего ограничения `QuerySet` с `all_courses.filter(subject=subject)`, если данные не были найдены в кеше.

Кэширование фрагментов шаблона

Кэширование фрагментов шаблона - это подход более высокого уровня. Вам нужно загрузить теги шаблона кэша в свой шаблон, используя `{% load cache %}`. Затем вы сможете использовать тег шаблона `{% cache %}` для кэширования определенных фрагментов шаблона. Обычно вы используете тег шаблона следующим образом:

```
{% cache 300 fragment_name %}  
...  
{% endcache %}
```

Тег `{% cache %}` имеет два обязательных аргумента: тайм-аут, в секундах и имя для фрагмента. Если вам нужно кэшировать содержимое в зависимости от динамических данных, вы можете сделать это, передав дополнительные аргументы тегу шаблона `{% cache %}`, чтобы однозначно идентифицировать фрагмент.

Откройте `/students/course/detail.html` приложения `students`. Добавьте следующий код вверху сразу после тега `{% extends %}`:

```
{% load cache %}
```

Затем найдите следующие 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 %}
```

Мы кэшируем этот фрагмент шаблона с помощью имени `module_contents` и передаем ему текущий объект `Module`. Таким образом, мы однозначно идентифицируем фрагмент. Это важно, чтобы избежать кэширования содержимого модуля и обслуживания неправильного содержимого при запросе другого модуля.

Если параметр `USE_I18N` равен `true`, кеш промежуточного программного обеспечения для каждого сайта будет использовать активный язык. Если вы используете тег шаблона `{% cache %}`, вы должны использовать одну из переменных для перевода, доступных в шаблонах, для достижения того же результата, например `{% cache 600 name request.LANGUAGE_CODE %}`.

Кэширование представлений

Вы можете кэшировать вывод отдельных представлений с помощью декоратора `cache_page`, расположенного в `django.views.decorators.cache`. Декоратору требуется аргумент `timeout` (в секундах).

Давайте используем его в наших представлениях.

Отредактируйте файл `urls.py` приложения `students` и добавьте следующий импорт:

```
from django.views.decorators.cache import cache_page
```

Затем примените декоратор `cache_page` к шаблонам URL

`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 результат для `StudentCourseDetailView` кэшируется в течение 15 минут.

В кэше каждого представления используется URL-адрес для создания ключа кеша. Несколько URL-адресов, указывающих на одно и то же представление, будут кэшироваться отдельно.

Использование кеша для всего сайта

Это кеш самого высокого уровня. Он позволяет кэшировать весь сайт.

Чтобы разрешить кеш для всего сайта, отредактируйте файл `settings.py` вашего проекта и добавьте классы `UpdateCacheMiddleware` и `FetchFromCacheMiddleware` в `MIDDLEWARE`, следующим образом:

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.cache.UpdateCacheMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.cache.FetchFromCacheMiddleware',  
    # ...  
]
```

Помните, что middlewares выполняются в прямом порядке во время фазы запроса и в обратном порядке во время фазы ответа. `UpdateCacheMiddleware` помещается перед `CommonMiddleware`, потому что он запускается во время ответа, когда middlewares выполняются в обратном порядке. `FetchFromCacheMiddleware` размещается после `CommonMiddleware` намеренно, потому что ему необходимо получить доступ к данным запроса, установленным последними.

Затем добавьте следующие параметры в файл `settings.py`:

```
CACHE_MIDDLEWARE_ALIAS = 'default'  
CACHE_MIDDLEWARE_SECONDS = 60 * 15 # 15 minutes  
CACHE_MIDDLEWARE_KEY_PREFIX = 'educa'
```

В этих настройках мы используем кеш-память по умолчанию для нашего промежуточного программного обеспечения кэша, и мы устанавливаем глобальный тайм-аут кэша 15 минут. Мы также указываем префикс для всех ключей кеша, чтобы избежать коллизий, если мы используем один и тот же основной сервер Memcached для нескольких проектов. Наш сайт теперь будет кэшировать и возвращать кэшированный контент для всех запросов `get`.

Мы сделали это, чтобы проверить функциональность кеша для целого сайта. Однако кеш для всего сайта не подходит для нас, поскольку представления управления курсом должны показывать обновленные данные, чтобы мгновенно отражать любые изменения. Лучшим подходом к нашему проекту является кэширование шаблонов или представлений, которые используются для отображения содержимого курса для учащихся.

Мы познакомились с методами, предоставляемыми Django для кэширования данных. Вы должны правильно определить свою стратегию кэширования и расставить приоритеты по самым дорогостоящим `QuerySet` или вычислениям.

Резюме

В этой главе мы создали публичные представления для курсов, и систему для студентов, чтобы зарегистрироваться и записаться на курсы. Мы установили Memcached и реализовали разные уровни кеша.

В следующей главе мы создадим API RESTful для нашего проекта.

Глава 12

Создание API

В предыдущей главе вы построили систему регистрации учащихся и зачисления на курсы. Вы создали представления для отображения содержимого курса и узнали, как использовать фреймворк кеша Django. В этой главе вы узнаете, как сделать следующее:

- Создать RESTful API
- Проверить подлинность и разрешения для представлений API
- Создать API настроек представлений и маршрутизаторов

Создание RESTful API

Возможно, вы захотите создать интерфейс для других сервисов чтобы они взаимодействовали с вашим веб-приложением.

Создавая API, вы можете разрешать третьим сторонам получать информацию и работать с вашим программным обеспечением.

Существует несколько способов структурирования вашего API, но рекомендуется следовать принципам REST. Архитектура REST происходит от **Передачи репрезентативных состояний**. API RESTful основаны на ресурсах. Ваши модели предоставляют ресурсы и HTTP-методы, такие как `GET`, `POST`, `PUT`, или `DELETE` чтобы извлекать, создавать, обновлять или удалять объекты. В этом контексте также используются коды ответов HTTP. В результате ответа на HTTP-запрос могут возвращаться разные коды HTTP, например, `2xx` в случае успеха, `4xx` для ошибок и т.д.

Наиболее распространенными форматами для обмена данными в RESTful API являются JSON и XML. Мы построим REST API с сериализацией JSON для нашего проекта. Наш API предоставит следующие функции:

- Извлечь предметы
- Получить доступные курсы
- Получить содержимое курса
- Записаться на курс

Мы можем создавать API с нуля с помощью Django, используя пользовательские представления. Однако есть несколько сторонних модулей, которые упрощают создание API для вашего проекта, наиболее популярными среди которых являются Django REST framework.

Установка Django REST фреймворка

Django REST фреймворк позволяет вам легко создавать REST API для вашего проекта. Вы можете найти всю информацию о REST фреймворке на <https://www.django-rest-framework.org/>.

Откройте консоль и установите фреймворк с помощью следующей команды:

```
pip install djangorestframework==3.8.2
```

Откройте файл `settings.py` проекта `educa` и добавьте следующие настройки к `INSTALLED_APPS`, чтобы активировать `rest_framework`:

```
INSTALLED_APPS = [
    # ...
    'rest_framework',
]
```

Затем добавьте следующий код в файл `settings.py`:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES':
        ['rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly']
}
```

Вы можете указать определенную конфигурацию для своего API с помощью параметра `REST_FRAMEWORK`. REST фреймворк предлагает широкий диапазон настроек для установки поведения по умолчанию. Параметр `DEFAULT_PERMISSION_CLASSES`

указывает разрешения по умолчанию для чтения, создания, обновления или удаления объектов. Мы устанавливаем `DjangoModelPermissionsOrAnonReadOnly` как единственный класс разрешений по умолчанию. Этот класс использует систему разрешений Django, позволяющую пользователям создавать, обновлять или удалять объекты, предоставляя доступ только для чтения анонимным пользователям. Подробнее о разрешениях вы узнаете позже, в разделе *Добавление разрешений в представления*.

Для получения полного списка доступных настроек для REST-фреймворка вы можете посетить <https://www.django-rest-framework.org/api-guide/settings/>.

Определение сериализаторов

После настройки REST фреймворка нам нужно указать, как будут преобразованы наши данные. Выходные данные должны быть сериализованы в определенном формате, а входные данные будут де-сериализованы для обработки. В фреймворке предусмотрены следующие классы для создания сериализаторов отдельных объектов:

- `Serializer`: Обеспечивает сериализацию для обычных экземпляров Python класса
- `ModelSerializer`: Обеспечивает сериализацию для экземпляров модели
- `HyperlinkedModelSerializer`: То же, что и `ModelSerializer`, но он представляет отношения объектов со ссылками, а не с первичными ключами

Давайте построим наш первый сериализатор. Создайте следующую структуру файлов внутри каталога приложения `courses`:

```
api/
    __init__.py
    serializers.py
```

Мы разместим все функциональные возможности API внутри каталога `api`, чтобы все было хорошо организовано. Откройте

файл `serializers.py` и добавьте в него следующий код:

```
from rest_framework import serializers
from ..models import Subject

class SubjectSerializer(serializers.ModelSerializer):
    class Meta:
        model = Subject
        fields = ['id', 'title', 'slug']
```

Это сериализатор для модели `Subject`. Сериализаторы определяются аналогично Django классам `Form` и `ModelForm`. Класс `Meta` позволяет указать модель для сериализации и поля, которые будут включены для сериализации. Все поля модели будут включены, если вы не установите атрибут `fields`.

Протестируем наш сериализатор. Откройте командную строку и запустите консоль Django с помощью следующей команды:

```
python manage.py shell
```

Выполните следующий 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 этом примере мы получаем объект `Subject`, создаем экземпляр `SubjectSerializer` и получаем доступ к сериализованным данным. Вы можете видеть, что данные модели преобразуются в собственные типы данных Python.

Понимание парсеров и рендеринга

Сериализованные данные должны отображаться в определенном формате до того, как вы вернете их в ответе HTTP. Аналогично, когда вы получаете HTTP-запрос, вам необходимо проанализировать входящие данные и десериализовать их, прежде чем вы сможете работать с ним. Структура REST включает средства рендеринга и парсеров для их обработки.

Посмотрим, как разбирать входящие данные. Выполните следующий код в консоли Python:

```
>>> 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'}
```

Учитывая строковый ввод JSON, вы можете использовать класс `JSONParser`, предоставляемый фреймворком REST, чтобы преобразовать его в объект Python.

Фреймворк REST также включает классы `Renderer`, которые позволяют отформатировать ответы API. Фреймворк определяет, какой рендерер использовать посредством анализа содержимого. Он проверяет заголовок `Accept` запроса для определения ожидаемого типа содержимого для ответа. Необязательно, средство визуализации определяется суффиксом формата URL-адреса. Например, для запуска JSON-запроса будет запускаться `JSONRenderer`.

Вернитесь к консоли и выполните следующий код для рендеринга объекта `serializer` из предыдущего примера сериализатора:

```
>>> from rest_framework.renderers import JSONRenderer  
>>> JSONRenderer().render(serializer.data)
```

Вы увидите следующий результат:

```
b'{"id":4,"title":"Programming","slug":"programming"}'
```

Мы используем `JSONRenderer` для рендеринга сериализованных данных в JSON. По умолчанию фреймворк REST использует два разных средства визуализации: `JSONRenderer` И `BrowsableAPIRenderer`. Последний предоставляет веб-интерфейс, чтобы легко просматривать ваш API. Вы можете изменить классы рендеринга по умолчанию с помощью параметра `DEFAULT_RENDERER_CLASSES` параметра `REST_FRAMEWORK`.

Вы можете найти дополнительную информацию о средствах визуализации и парсерах на <https://www.django-rest-framework.org/api-guide/renderers/> И <https://www.django-rest-framework.org/api-guide/parsers/>.

Создание списка и детального представлений

Фреймворк REST поставляется с набором общих представлений и миксин, которые вы можете использовать для создания представлений API. Они предоставляют функциональные возможности для извлечения, создания, обновления или удаления объектов модели. Вы можете видеть все общие миксины и представления, представленные фреймворком REST, в <https://www.django-rest-framework.org/api-guide/generic-views/>.

Давайте создадим список и подробное представления для извлечения объектов `Subject`. Создайте новый файл внутри каталога `courses/api/` и назовите его `views.py`. Добавьте к нему следующий код:

```
from rest_framework import generics
from ..models import Subject
from .serializers import SubjectSerializer

class SubjectListView(generics.ListAPIView):
    queryset = Subject.objects.all()
    serializer_class = SubjectSerializer

class SubjectDetailView(generics.RetrieveAPIView):
    queryset = Subject.objects.all()
    serializer_class = SubjectSerializer
```

В этом коде мы используем общие представления `ListAPIView` и `RetrieveAPIView` фреймворка REST. Мы включаем параметр URL `pk` для подробного представления чтобы извлечь объект для данного первичного ключа. Оба представления имеют следующие атрибуты:

- `queryset`: Базовый `querySet` для извлечения объектов
- `serializer_class`: Класс для сериализации объектов

Давайте добавим шаблоны URL для наших представлений. Создайте новый файл внутри каталога `courses/api/`, назовите его `urls.py` и добавте в него следующий код:

```
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 основной файл `urls.py` проекта `educa` и добавте паттерн API следующим образом:

```
urlpatterns = [
    # ...
    path('api/', include('courses.api.urls', namespace='api')),
]
```

Мы используем пространство имен `api` для наших URL-адресов API. Убедитесь, что ваш сервер работает `python manage.py runserver`. Откройте консоль и извлеките URL `http://127.0.0.1:8000/api/subjects/` при помощи `curl` следующим образом:

```
curl http://127.0.0.1:8000/api/subjects/
```

Вы получите ответ, похожий на следующий:

```
[  
    {"id":1,"title":"Mathematics","slug":"mathematics"},  
    {"id":2,"title":"Music","slug":"music"},  
    {"id":3,"title":"Physics","slug":"physics"},  
    {"id":4,"title":"Programming","slug":"programming"}  
]
```

HTTP-ответ содержит список объектов `Subject` в формате JSON. Если ваша операционная система не поставляется с `curl`, его можно загрузить из <https://curl.haxx.se/dlwiz/>. Вместо `curl` вы также можете использовать любой другой инструмент для отправки пользовательских HTTP-запросов, таких как расширение браузера, например Postman, которое вы можете получить из <https://www.getpostman.com/>.

Откройте `http://127.0.0.1:8000/api/subjects/` в вашем браузере. Вы увидите API-интерфейс REST фреймворка в браузере:

Subject List

OPTIONS

GET ▾

GET /api/subjects/

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"  
  }  
]
```

Этот HTML-интерфейс предоставляется рендерером `BrowsableAPIRenderer`. Он отображает заголовки и содержимое и

позволяет выполнять запросы. Вы также можете получить доступ к подробному представлению API для объекта `Subject`, указав его ID в URL-адресе. Откройте <http://127.0.0.1:8000/api/subjects/1/> в своем браузере. Вы увидите один объект `Subject`, отображаемый в формате JSON.

Создание вложенных сериализаторов

Мы собираемся создать сериализатор для модели `Course`.

Откройте файл `api/serializers.py` приложения `courses` и добавьте к нему следующий код:

```
from ..models import Course

class CourseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Course
        fields = ['id', 'subject', 'title', 'slug', 'overview',
                  'created', 'owner', 'modules']
```

Давайте посмотрим, как сериализуется объект `course`. Откройте консоль, запустите `python manage.py shell` и наберите следующий 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, включенными в `CourseSerializer`. Вы можете видеть, что связанные объекты менеджера `modules` сериализуются как список первичных ключей, а именно:

```
"modules": [6, 7, 9, 10]
```

Мы хотим включить больше информации о каждом модуле, поэтому нам нужно сериализовать объекты `Module` и вложить их. Измените предыдущий код файла `api/serializers.py` приложения `courses`, чтобы он выглядел следующим образом:

```
from rest_framework import serializers
from ..models import Module

class ModuleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Module
        fields = ['order', 'title', 'description']

class CourseSerializer(serializers.ModelSerializer):
    modules = ModuleSerializer(many=True, read_only=True)

    class Meta:
        model = Course
        fields = ['id', 'subject', 'title', 'slug', 'overview',
                  'created', 'owner', 'modules']
```

Мы определяем `ModuleSerializer`, чтобы обеспечить сериализацию для модели `Module`. Затем добавляем атрибут `modules` в `CourseSerializer` для вставки сериализатора `ModuleSerializer`. Мы устанавливаем `many=True`, чтобы указать, что мы сериализуем несколько объектов. Параметр `read_only` указывает, что это поле доступно только для чтения и не должно быть включено ни в какие входные данные для создания или обновления объектов.

Откройте консоль и снова создайте экземпляр `CourseSerializer`. Будет отрисован атрибут `data` с помощью `jsonRenderer`. На этот раз перечисленные модули сериализуются с вложенным сериализатором `ModuleSerializer` следующим образом:

```
"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."  
    },  
    ...  
]
```

Вы можете больше узнать о сериализаторах на <https://www.djangoproject.org/api-guide/serializers/>.

Создание пользовательских представлений

Фреймворк REST предоставляет класс `APIView`, который создает функциональность API поверх Django класса `View`. Класс `APIView` отличается от `View` тем, что использует пользовательские объекты REST фреймворка `Request` и `Response` и обрабатывает `APIException` для возврата соответствующих HTTP-ответов. Он также имеет встроенную систему аутентификации и авторизации для управления доступом к представлениям.

Мы собираемся создать представление для пользователей, чтобы записаться на курсы. Откройте файл `api/views.py` приложения `courses` и добавьте к нему следующий код:

```
from django.shortcuts import get_object_or_404
from rest_framework.views import APIView
from rest_framework.response import Response
from ..models import Course

class CourseEnrollView(APIView):
    def post(self, request, pk, format=None):
        course = get_object_or_404(Course, pk=pk)
        course.students.add(request.user)
        return Response({'enrolled': True})
```

Представление `CourseEnrollView` обрабатывает регистрацию пользователей на курсах. Предыдущий код выглядит следующим образом:

1. Мы создаем настраиваемое представление, наследуемое от класса `APIView`.
2. Мы определяем метод `post()` для POST запросов. Никакой

другой HTTP метод не сможет получить доступ к этому представлению.

3. Мы ожидаем, что параметр URL `pk` содержит идентификатор курса. Мы извлекаем курс с помощью заданного параметра `pk` и вызываем исключение `404`, если он не найден.
4. Мы добавляем текущего пользователя к `students` с отношением «многие ко многим» к объекту `course` и возвращаем успешный ответ.

Отредактируйте файл `api/urls.py` и добавьте следующий шаблон URL для представления `CourseEnrollView`:

```
path('courses/<pk>/enroll/',
      views.CourseEnrollView.as_view(),
      name='course_enroll'),
```

Теоретически, мы могли бы теперь выполнить запрос POST для регистрации текущего пользователя на курсе. Тем не менее, мы должны иметь возможность идентифицировать пользователя и не разрешать пользователям, не прошедшим аутентификацию, доступ к этому представлению. Посмотрим, как работают аутентификация и разрешения API.

Обработка аутентификации

REST фреймворк предоставляет классы проверки подлинности для идентификации пользователя, выполняющего запрос.

Если аутентификация прошла успешно, фреймворк устанавливает аутентифицированный объект `User` в `request.user`. Если ни один пользователь не аутентифицирован, вместо него устанавливается экземпляр `Django AnonymousUser`.

REST фреймворк обеспечивает следующие аутентификационные бекенды:

- `BasicAuthentication`: Это базовая аутентификация HTTP. Логин и пароль отправляются клиентом в HTTP заголовке `Authorization`, закодированном в Base64. Вы можете узнать больше об этом на https://en.wikipedia.org/wiki/Basic_access_authentication.
- `TokenAuthentication`: Это аутентификация на основе токенов. Для хранения токенов пользователей используется модель `Token`. Пользователи включают токен в HTTP-заголовке `Authorization` для аутентификации.
- `SessionAuthentication`: В этом случае для проверки подлинности используется сервер сессий Django. Этот бэкэнд полезен для выполнения аутентифицированных запросов AJAX к API с внешнего интерфейса вашего веб-сайта.
- `RemoteUserAuthentication`: Это позволяет делегировать

аутентификацию на ваш веб-сервер, через настройку переменной среды `REMOTE_USER`.

Вы можете создать собственный сервер аутентификации из класса `BaseAuthentication`, предоставляемого фреймворком REST, и переопределения метода `authenticate()`.

Вы можете установить аутентификацию для каждого представления или установить ее глобально с помощью параметра `DEFAULT_AUTHENTICATION_CLASSES`.

Аутентификация определяет только пользователя, выполняющего запрос. Она не может позволять или запрещать доступ к представлениям. Вы должны использовать разрешения для ограничения доступа к представлениям.

Вы можете найти всю информацию об аутентификации в <https://www.djangoproject.com/en/2.0/topics/auth/default/>.

Давайте добавим `BasicAuthentication` к нашему представлению. Отредактируйте файл `api/views.py` приложения `courses` и добавьте атрибут `authentication_classes` в `CourseEnrollView` следующим образом:

```
from rest_framework.authentication import BasicAuthentication

class CourseEnrollView(APIView):
    authentication_classes = (BasicAuthentication,)
    # ...
```

Пользователи будут идентифицированы с учетными данными, установленными в заголовке `Authorization` HTTP-запроса.

Добавление разрешений в представления

Фреймворк REST включает в себя систему разрешений для ограничения доступа к представлениям. Вот некоторые из встроенных разрешений фреймворка REST:

- `AllowAny`: Неограниченный доступ, независимо от того, аутентифицирован ли пользователь или нет.
- `IsAuthenticated`: Разрешает доступ только проверенным пользователям.
- `IsAuthenticatedOrReadOnly`: Полный доступ для аутентифицированных пользователей. Анонимным пользователям разрешено выполнять только такие методы чтения, как `GET`, `HEAD` ИЛИ `OPTIONS`.
- `DjangoModelPermissions`: Разрешения привязаны к `django.contrib.auth`. Для представления требуется атрибут `queryset`. Разрешение только у авторизованных пользователей с дозволенными разрешениями доступа к модели.
- `DjangoObjectPermissions`: Разрешения Django для каждого объекта.

Если пользователям отказывают в разрешении, они обычно получают один из следующих кодов ошибок HTTP:

- HTTP 401: Неавторизированы
- HTTP 403: Доступ запрещен

Вы можете узнать больше о разрешениях на <https://www.django-rest-framework.org/api-guide/permissions/>.

Отредактируйте файл `api/views.py` приложения `courses` и добавьте атрибут `permission_classes` в `CourseEnrollView` следующим образом:

```
from rest_framework.authentication import BasicAuthentication
from rest_framework.permissions import IsAuthenticated

class CourseEnrollView(APIView):
    authentication_classes = (BasicAuthentication,)
    permission_classes = (IsAuthenticated,)
    # ...
```

Мы включаем разрешение `IsAuthenticated`. Это предотвратит доступ анонимных пользователей к представлению. Теперь мы можем выполнить POST запрос для нашего нового метода API.

Убедитесь, что сервер разработки запущен. Откройте консоль и выполните следующую команду:

```
curl -i -X POST http://127.0.0.1:8000/api/courses/1/enroll/
```

Вы получите следующий ответ:

```
HTTP/1.1 401 Unauthorized
...
{"detail": "Authentication credentials were not provided."}
```

Мы получаем код HTTP 401, поскольку мы не аутентифицированы. Давайте используем базовую аутентификацию с одним из наших пользователей. Выполните следующую команду, заменив `student:password` на учетные данные

существующего пользователя:

```
curl -i -X POST -u student:password  
http://127.0.0.1:8000/api/courses/1/enroll/
```

Вы получите следующий ответ:

```
HTTP/1.1 200 OK  
...  
{"enrolled": true}
```

Вы можете войти на сайт администрирования и проверить, что пользователь теперь зарегистрирован на курсе.

Создание наборов представлений и маршрутизаторов

`ViewSets` позволяют определить взаимодействие вашего API и позволить REST-фреймворку динамически создавать URL-адреса с помощью объекта `Router`. С помощью наборов представлений вы можете избежать повторения логики для нескольких представлений. Элементы представления включают действия для типичных операций создания, извлечения, обновления и удаления, такие как `list()`, `create()`, `retrieve()`, `update()`, `partial_update()` и `destroy()`.

Давайте создадим набор представлений для модели `Course`. Отредактируйте файл `api/views.py` и добавьте к нему следующий код:

```
from rest_framework import viewsets
from .serializers import CourseSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer
```

Мы переопределяем класс `ReadOnlyModelViewSet`, который предоставляет действия только для чтения `list()` и `retrieve()` для обоих объектов списка или для извлечения одного объекта. Отредактируйте файл `api/urls.py` и создайте маршрутизатор для нашего представления следующим образом:

```
from django.urls import path, include
from rest_framework import routers
```

```
from . import views

router = routers.DefaultRouter()
router.register('courses', views.CourseViewSet)

urlpatterns = [
    # ...
    path('', include(router.urls)),
]
```

Мы создаем объект `DefaultRouter` и регистрируем наш набор представлений с помощью префикса `courses`. Маршрутизатор берет на себя ответственность за создание URL-адресов автоматически для нашего набора представлений.

Откройте `http://127.0.0.1:8000/api/` в вашем браузере. Вы увидите, что маршрутизатор перечисляет все наборы представлений в своем базовом URL-адресе, как показано на следующем скриншоте:

The screenshot shows a browser interface with a title bar "Api Root". Below it, a message says "The default basic root view for DefaultRouter". On the right, there are two buttons: "OPTIONS" and "GET ▾". A large button below them contains the text "GET /api/". Underneath, the response is displayed in a code block:

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "courses": "http://127.0.0.1:8000/api/courses/"
}
```

Вы можете открыть <http://127.0.0.1:8000/api/courses/>, чтобы получить список курсов.

Вы можете узнать больше о наборах представлений в <https://www.djangoproject-rest-framework.org/api-guide/viewsets/>. Вы также можете найти дополнительную информацию о маршрутизаторах на <https://www.djangoproject-rest-framework.org/api-guide/routers/>.

Добавление дополнительных действий к наборам представлений

Вы можете добавить дополнительные действия к наборам представлений. Давайте изменим наше предыдущее представление `CourseEnrollView` в пользовательское действие набора представлений. Отредактируйте файл `api/views.py` и измените класс `CourseViewSet`, чтобы он выглядел следующим образом:

```
from rest_framework.decorators import detail_route

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer

    @detail_route(methods=['post'],
                 authentication_classes=[BasicAuthentication],
                 permission_classes=[IsAuthenticated])
    def enroll(self, request, *args, **kwargs):
        course = self.get_object()
        course.students.add(request.user)
        return Response({'enrolled': True})
```

Мы добавляем собственный метод `enroll()`, который представляет собой дополнительное действие для этого набора представлений. Этот код работает следующим образом:

1. Мы используем декоратор фреймворка `detail_route`, чтобы указать, что это действие, которое должно выполняться на одном объекте.
2. Декоратор позволяет добавлять пользовательские

атрибуты для действия. Мы указываем, что для этого представления разрешен только метод `post` и задан класс аутентификации и разрешений.

3. Мы используем `self.get_object()` для извлечения объекта `Course`.
4. Мы добавляем текущего пользователя к `students` с отношением `many-to-many` и возвращаем настраиваемый ответ в случае успеха.

Измените файл `api/urls.py` и удалите следующий URL-адрес, поскольку он больше не нужен:

```
path('courses/<pk>/enroll/',
      views.CourseEnrollView.as_view(),
      name='course_enroll'),
```

Затем отредактируйте файл `api/views.py` и удалите класс `CourseEnrollView`.

URL-адрес для регистрации на курсах теперь автоматически генерируется маршрутизатором. URL-адрес остается тем же, поскольку он построен динамически с использованием нашего имени действия `enroll`.

Создание пользовательских разрешений

Мы хотим, чтобы учащиеся имели доступ к содержимому курсов, в которых они участвуют. Только учащиеся, обучающиеся на курсе, должны иметь доступ к его содержимому. Лучший способ сделать это - создать собственный класс разрешений. Django предоставляет класс `BasePermission`, который позволяет вам определить следующие методы:

- `has_permission()`: Проверка разрешения на уровне представления
- `has_object_permission()`: Проверка разрешения на уровне экземпляра

Эти методы должны возвращать `True` для предоставления доступа или `False` в противном случае. Создайте новый файл внутри каталога `classes/api/` и назовите его `permissions.py`. Добавьте к нему следующий код:

```
from rest_framework.permissions import BasePermission

class IsEnrolled(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.students.filter(id=request.user.id).exists()
```

Мы переопределили класс `BasePermission` и переопределяем `has_object_permission()`. Мы проверяем, что пользователь, выполняющий запрос, присутствует в отношении `students`

объекта `course`. Теперь мы будем использовать разрешение `IsEnrolled`.

Сериализация содержимого курса

Нам необходимо сериализовать содержимое курса. Модель `Content` включает общий внешний ключ, который позволяет нам связывать объекты разных моделей контента. Тем не менее, мы добавили общий метод `render()` для всех моделей контента в предыдущей главе. Мы можем использовать этот метод для предоставления отображаемого содержимого нашему API.

Откройте файл `api/serializers.py` приложения `courses` и добавьте следующий код в него:

```
from ..models import Content

class ItemRelatedField(serializers.RelatedField):
    def to_representation(self, value):
        return value.render()

class ContentSerializer(serializers.ModelSerializer):
    item = ItemRelatedField(read_only=True)

    class Meta:
        model = Content
        fields = ['order', 'item']
```

В этом коде мы определяем настраиваемое поле путем переопределения поля сериализатора `RelatedField`, предоставленного фреймворком REST, и переопределения метода `to_representation()`. Мы определяем сериализатор `ContentSerializer` для модели `Content` и используем настраиваемое поле для внешнего ключа.

Нам нужен альтернативный сериализатор для модели `Module`,

который включает ее содержимое, и расширенный сериализатор `Course`. Измените файл `api/serializers.py` и добавьте к нему следующий 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. Измените файл `api/views.py` и добавьте следующий метод в класс `CourseViewSet`:

```
from .permissions import IsEnrolled
from .serializers import CourseWithContentsSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    # ...
    @detail_route(methods=['get'],
                  serializer_class=CourseWithContentsSerializer,
                  authentication_classes=[BasicAuthentication],
                  permission_classes=[IsAuthenticated,
                                      IsEnrolled])
    def contents(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)
```

Описание этого метода выглядит следующим образом:

- Мы используем декоратор `detail_route`, чтобы указать, что это действие выполняется на одном объекте.

- Мы указываем, что для этого действия разрешен только метод `GET`.
- Мы используем новый класс сериализатора `CourseWithContentsSerializer`, который включает в себя визуализированное содержимое курса.
- Мы используем как `isAuthenticated`, так и наши пользовательские разрешения `isEnrolled`. Поступая таким образом, мы гарантируем, что только пользователи, включенные в курс, смогут получить доступ к его содержимому.
- Мы используем существующее действие `retrieve()`, чтобы вернуть объект `Course`.

Откройте `http://127.0.0.1:8000/api/courses/1/contents/` в вашем браузере. Вы получите доступ к представлению если введете правильные учетные данные, вы увидите, что каждый модуль курса включает отображаемый HTML для содержимого курса, а именно:

```
{  
    "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    }\n}\n
```

Вы создали простой API, который позволяет другим службам программно обращаться к сайту. REST фреймворк также позволяет обрабатывать создание и редактирование объектов с помощью набора `ModelViewSet`. Мы рассмотрели основные аспекты фреймворка Django REST, но вы можете найти дополнительную информацию об его функциях в обширной документации на <https://www.django-rest-framework.org/>.

Резюме

В этой главе вы создали RESTful API для взаимодействия внешних сервисов с вашим веб-приложением.

В следующей главе вы узнаете, как создать производственную среду с помощью uWSGI и NGINX. Вы также узнаете, как реализовать собственное промежуточное программное обеспечение и создавать собственные команды управления.

Глава 13

Продвигаем в жизнь

В предыдущей главе вы создали RESTful API для своего проекта. В этой главе мы узнаем, как создать производственную среду для нашего проекта, рассмотрев следующие темы:

- Настройка рабочей среды
- Создание настраиваемого промежуточного программного обеспечения
- Внедрение пользовательских команд управления

Создание производственной среды

Пришло время развернуть проект Django в производственной среде. Мы собираемся следовать этим шагам, чтобы получить рабочий проект:

1. Настроить параметры проекта для производственной среды
2. Использовать базу данных PostgreSQL
3. Настроить веб-сервер с помощью uWSGI и NGINX
4. Сервер статистики
5. Защитить наш сайт с помощью SSL

Управление настройками для множества сред

В реальных проектах вам придется иметь дело с несколькими средами. У вас будет, по крайней мере, локальная и производственная среда, но у вас могут быть и другие среды, для тестирования или предварительной подготовки.

Некоторые параметры проекта будут общие для всех сред, но другие должны быть переопределены для каждой среды. Давайте настроим параметры проекта для нескольких сред, сохраняя при этом все ясно организованно.

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 дополнительные файлы внутри папки `settings/`, чтобы новый каталог выглядел следующим образом:

```
settings/
    __init__.py
    base.py
    local.py
    pro.py
```

Эти файлы нужны для следующего:

- `base.py`: Базовый файл настроек, содержащий общие настройки (ранее `settings.py`)
- `local.py`: Пользовательские настройки для вашей локальной среды

- `pro.py`: Пользовательские настройки рабочей среды

Откройте файл `settings/base.py` и замените:

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

Следующим:

```
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(os.path.join(__file__,
os.pardir))))
```

Мы переместили наши файлы настроек в каталог на один уровень ниже, поэтому нам нужно в `BASE_DIR` указать расположение родительского каталога. Мы сделаем это, указав на родительский каталог с `os.pardir`.

Откройте файл `settings/local.py` и добавьте следующие строки:

```
from .base import *

DEBUG = True

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

Это файл настроек для нашей локальной среды. Мы импортируем все настройки, определенные в файле `base.py`, и переопределяем только некоторые настройки для этой среды. Мы скопировали настройки `DEBUG` и `DATABASES` из файла `base.py`, так как они будут установлены для каждой среды. Вы можете удалить настройки `DATABASES` и `DEBUG` из файла настроек `base.py`.

Откройте файл `settings/pro.py` и добавьте следующее:

```
from .base import *

DEBUG = False

ADMINS = (
    ('Antonio M', 'email@mydomain.com'),
)

ALLOWED_HOSTS = ['*']

DATABASES = {
    'default': {
    }
}
```

Это настройки для производственной среды. Давайте подробнее рассмотрим каждую из них:

- `DEBUG`: Установка `DEBUG` в `False` должна быть обязательной для любой производственной среды. Несоблюдение этого требования приведет к тому, что информация о трассировке и конфиденциальные данные конфигурации будут открыты для всех.
- `ADMINS`: Если `DEBUG = False`, и представление вызывает исключение, вся информация будет отправлена по электронной почте людям, указанным в настройке `ADMINS`. Обязательно замените кортеж `name/email` вашей собственной информацией.
- `ALLOWED_HOSTS`: Django разрешает только хостам, включенными в этот список, обслуживать приложение. Это мера безопасности. Мы добавляем символ звездочки * для обозначения всех имен хостов. Мы ограничим имена хостов, которые могут быть

использованы для обслуживания приложения позже.

- `DATABASES`: Мы просто оставляем эту настройку пустой. В дальнейшем мы планируем установить базу данных для производственной среды.

При работе с несколькими средами создайте базовый файл настроек и файл настроек для каждой среды. Файлы настроек среды должны наследовать общие настройки и переопределять настройки для конкретной среды.

Мы установили параметры проекта в другом месте, вместо файла настроек `settings.py` по умолчанию. Вы не сможете выполнять какие-либо команды с помощью инструмента `manage.py`, если вы не укажете используемый модуль настроек. Вам нужно будет добавить флаг `--settings` при запуске команд управления из консоли или установить переменную среды `DJANGO_SETTINGS_MODULE`.

Откройте консоль и выполните следующую команду:

```
export DJANGO_SETTINGS_MODULE=educa.settings.pro
```

Это установит переменную среды `DJANGO_SETTINGS_MODULE` для текущего сеанса консоли. Если вы хотите избежать выполнения этой команды для каждой новой консоли, добавьте эту команду в конфигурацию вашей консоли в файлах `.bashrc` или `.bash_profile`. Если вы не установите эту переменную, вам нужно будет запускать команды управления, включая флаг `--settings`, как показано ниже:

```
python manage.py migrate --settings=educa.settings.pro
```

Вы успешно организовали настройки для работы в нескольких средах.

Использование PostgreSQL

Во всей этой книге мы в основном использовали базу данных SQLite. SQLite простая и быстрая в настройке, но для производственной среды вам понадобится более мощная база данных, такая как PostgreSQL, MySQL или Oracle. Вы уже узнали, как установить PostgreSQL и настроить базу данных PostgreSQL в [главе 3, *Расширение вашего приложения*](#) *блог*. Если вам нужно установить PostgreSQL, вы можете прочитать раздел [Установка PostgreSQL](#) [глава 3, *Расширение вашего приложения*](#) *блог*.

Давайте создадим пользователя PostgreSQL. Откройте терминал и запустите следующие команды для создания пользователя базы данных:

```
su postgres  
createuser -dP educa
```

Примечание переводчика: возможно вам придется использовать sudo
sudo su postgres

Вам будет предложено ввести пароль и разрешения, которые вы хотите предоставить этому пользователю. Введите желаемый пароль и разрешения, а затем создайте новую базу данных следующей командой:

```
createdb -E utf8 -U educa educa
```

Затем отредактируйте файл `settings/pro.py` и измените параметр `DATABASES`, чтобы он выглядел следующим образом:

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

Замените предыдущие данные на имена базы данных и учетные данные для созданного вами пользователя. Новая база данных пуста. Для применения всех миграций базы данных выполните следующую команду:

```
python manage.py migrate
```

Примечание переводчика: возможно вам придется установить модуль **psycopg2** выполнив команду:
pip install psycopg2==2.7.4

Наконец, создайте суперпользователя следующей командой:

```
python manage.py createsuperuser
```

Проверка вашего проекта

Django содержит команду управления `check` для проверки вашего проекта в любой момент. Эта команда проверяет приложения, установленные в вашем проекте Django, и выводит любые ошибки или предупреждения. Если вы включите опцию `--deploy`, будут активированы дополнительные проверки, относящиеся только к использованию продукта. Откройте консоль и выполните следующую команду для проверки:

```
python manage.py check --deploy
```

Вы увидите результат без ошибок, но содержащий несколько предупреждений. Это означает, что проверка прошла успешно, но вы должны изучить предупреждения, чтобы узнать, есть ли что-то еще, что вы можете сделать, чтобы сделать ваш проект более безопасным для производства. Мы не собираемся углубляться в это, но имейте в виду, что вы должны проверить свой проект перед развертыванием в производстве, чтобы выявить проблемы.

Обслуживание Django через WSGI

Основной платформой развертывания Django является WSGI. **WSGI** означает **Интерфейс шлюза веб-сервера**, и он является стандартом для обслуживания приложений Python в Интернете.

Когда вы создаете новый проект с помощью команды `startproject`, Django создает файл `wsgi.py` внутри вашего каталога проектов. Этот файл содержит приложение WSGI, которое является точкой доступа к вашему сайту. WSGI используется для запуска вашего проекта с сервером разработки Django и развертывания вашего приложения с выбранным вами сервером в рабочей среде.

Вы можете узнать больше о WSGI на <https://wsgi.readthedocs.io/en/latest/>.

Установка uWSGI

В этой книге вы используете сервер разработки Django для запуска проектов в своей локальной среде. Однако для развертывания приложения в рабочей среде необходим настоящий веб-сервер.

uWSGI - чрезвычайно быстрый сервер приложений Python. Он связывается с вашим приложением Python с использованием спецификации WSGI. uWSGI переводит веб-запросы в формат, который может обрабатывать ваш проект Django.

Установите uWSGI, используя следующую команду:

```
pip install uwsgi==2.0.17
```

Для создания uWSGI вам понадобится компилятор C, например `gcc` или `clang`. В среде Linux вы можете установить его с помощью команды `apt-get install build-essential`.

Если вы используете macOS X, вы можете установить uWSGI с помощью диспетчера пакетов Homebrew с помощью команды `brew install uwsgi`. Если вы хотите установить uWSGI в Windows, вам понадобится Cygwin <https://www.cygwin.com>. Однако желательно использовать uWSGI в средах на базе UNIX.

Вы можете прочитать документацию uWSGI по адресу <https://uwsgi-docs.readthedocs.io/en/latest/>.

Конфигурация uWSGI

Вы можете запустить uWSGI из командной строки. Откройте консоль и запустите следующую команду из каталога проекта `educa`:

```
sudo uwsgi --module=educa.wsgi:application \
--env=DJANGO_SETTINGS_MODULE=educa.settings.pro \
--master --pidfile=/tmp/project-master.pid \
--http=127.0.0.1:8000 \
--uid=1000 \
--virtualenv=/home/env/educa/
```

Возможно, вам придется добавить `sudo` к этой команде, если у вас нет необходимых разрешений.

С помощью этой команды мы запускаем uWSGI на нашем `localhost` со следующими параметрами:

- Мы используем WSGI `educa.wsgi:application`.
- Мы загружаем настройки для производственной среды.
- Мы используем нашу виртуальную среду. Замените путь в параметре `virtualenv` на путь к вашему каталогу виртуальной среды. Если вы не используете виртуальную среду, вы можете пропустить эту опцию.

Если вы не выполняете команду в каталоге проекта, включите опцию `--chdir=/path/to/educa/` с указанием пути к вашему проекту.

Откройте `http://127.0.0.1:8000/` в своем браузере. Вы должны

увидеть сгенерированный HTML-код без каких-либо таблиц стилей CSS или загружаемых изображений. Это так поскольку мы не настроили uWSGI для обслуживания статических файлов.

uWSGI позволяет вам определить пользовательскую конфигурацию в файле `.ini`. Это более удобно, чем передача параметров через командную строку.

Создайте следующую структуру файлов внутри основного каталога `educa/`:

```
config/
    uwsgi.ini
```

Отредактируйте файл `uwsgi.ini` и добавьте к нему следующий код:

```
[uwsgi]
# variables
projectname = educa
base = /home/projects/educa

# configuration
master = true
virtualenv = /home/env/%(projectname)
pythonpath = %(base)
chdir = %(base)
env = DJANGO_SETTINGS_MODULE=%(projectname).settings.pro
module = educa.wsgi:application
socket = /tmp/%(projectname).sock
```

В файле `.ini` мы определяем следующие переменные:

- `projectname`: Название нашего проекта Django, которое у нас `educa`.
- `base`: Абсолютный путь к проекту `educa`. Замените его

абсолютным путем к вашему проекту.

Это пользовательские переменные, которые мы будем использовать в параметрах uWSGI. Вы можете указать любые другие переменные, которые вам нравятся, если их имена отличаются от параметров uWSGI.

Мы установили следующие параметры:

- `master`: Включить мастер-процесс.
- `virtualenv`: Путь к вашей виртуальной среде. Замените его на ваш путь.
- `pythonpath`: Путь для доступа к вашему Python.
- `chdir`: Путь к вашему каталогу проектов, так чтобы uWSGI переходил в этот каталог перед загрузкой приложения.
- `env`: Переменные среды. Мы включаем переменную `DJANGO_SETTINGS_MODULE`, указывающую на настройки рабочей среды.
- `module`: Модуль WSGI для использования. Мы устанавливаем его в `application`, содержащийся в модуле `wsgi` нашего проекта.
- `socket`: Сокет UNIX/TCP для привязки сервера.

Опция `socket` предназначена для связи с некоторыми сторонними маршрутизаторами, такими как NGINX, в то время как опция `http` предназначена для uWSGI чтобы принимать входящие HTTP-запросы и маршрутизировать их самостоятельно. Мы будем запускать uWSGI с помощью сокета, поскольку мы собираемся настроить NGINX как наш веб-

сервер и общаться с uWSGI через сокет.

Вы можете найти список доступных опций uWSGI в <https://uwsgi-docs.readthedocs.io/en/latest/Options.html>.

Теперь вы можете запустить uWSGI с вашей настройкой конфигурации, используя эту команду:

```
uwsgi --ini config/uwsgi.ini
```

Теперь вы не сможете получить доступ к вашему экземпляру uWSGI из своего браузера, поскольку он работает через сокет.

Установка NGINX

Когда вы обслуживаете веб-сайт, вам придется обслуживать динамический контент, но вы также будите обслуживать статические файлы, такие как CSS, файлы JavaScript и изображения. Хотя uWSGI способен обслуживать статические файлы, он добавляет ненужные накладные расходы на HTTP-запросы, и поэтому рекомендуется установить для этого веб-сервер, например NGINX.

NGINX - это веб-сервер, ориентированный на высокую параллельность, производительность и низкое использование памяти. NGINX также действует как обратный прокси-сервер, принимает HTTP-запросы и маршрутизирует их на разные серверы. Как правило, вы будете использовать веб-сервер, например, NGINX, для эффективного и быстрого обслуживания статических файлов, а динамические запросы вы будете перенаправлять на uWSGI-обработчики. Используя NGINX, вы также можете использовать возможности обратного прокси.

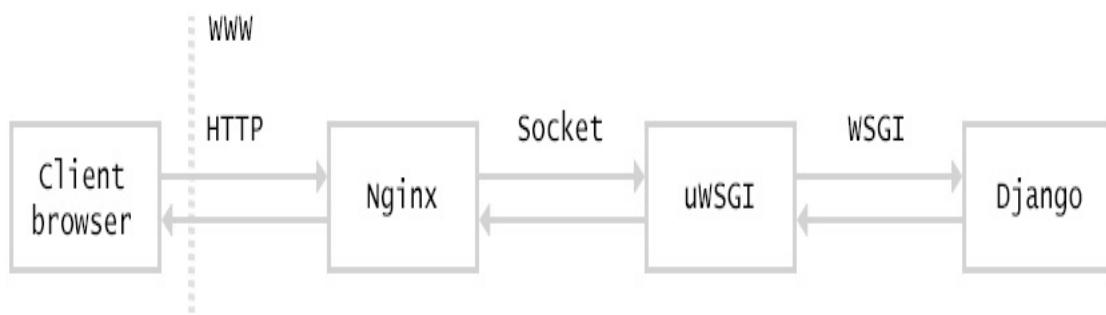
Установите NGINX с помощью следующей команды:

```
sudo apt-get install nginx
```

Если вы используете macOS X, вы можете установить NGINX с помощью команды `brew install nginx`. Вы можете найти исполняемые файлы NGINX для Windows на <https://nginx.org/en/download.html>.

Производственная среда

Следующая диаграмма показывает, как будет выглядеть наша конечная производственная среда:



Когда клиентский браузер отправит HTTP-запрос, произойдет следующее:

1. NGINX получает HTTP-запрос.
 2. Если запрашивается статический файл, NGINX обслуживает статический файл напрямую. Если запрашивается динамическая страница, NGINX делегирует запрос на uWSGI через сокет.
 3. uWSGI передает запрос Django для обработки.
- Полученный HTTP-ответ передается обратно в NGINX, который, в свою очередь, передает его обратно в клиентский браузер.

Конфигурация NGINX

Создайте новый файл в каталоге config/ и назовите его nginx.conf. Добавьте к нему следующий код:

```
# the upstream component nginx needs to connect to
upstream educa {
    server unix:///tmp/educa.sock;
}

server {
    listen      80;
    server_name www.educaproject.com educaproject.com;

    location / {
        include      /etc/nginx/uwsgi_params;
        uwsgi_pass  educa;
    }
}
```

Это базовая конфигурация для NGINX. Мы настроили восходящий поток с именем educa, который указывает на сокет, созданный uWSGI. Мы используем директиву server и добавляем следующую конфигурацию:

- Мы указываем, чтобы NGINX прослушивал порт 80.
- Мы устанавливаем имя сервера как www.educaproject.com, так и educaproject.com. NGINX будет обслуживать входящие запросы для обоих доменов.
- Мы указываем, что все под путем / должно быть направлено в сокет educa (uWSGI). Мы также включаем параметры конфигурации uWSGI по умолчанию,

которые поставляются с NGINX.

Вы можете найти документацию NGINX на <https://nginx.org/en/docs/>.

Основной файл конфигурации NGINX находится в `/etc/nginx/nginx.conf`. Он включает в себя любые файлы конфигурации, найденные в `/etc/nginx/sites-enabled/`. Чтобы заставить NGINX загружать ваш настраиваемый файл конфигурации, откройте консоль и создайте символьическую ссылку следующим образом:

```
sudo ln -s /home/projects/educa/config/nginx.conf /etc/nginx/sites-enabled/educa.conf
```

Замените `/home/projects/educa/` на абсолютный путь вашего проекта. Затем откройте консоль и запустите uWSGI, если вы еще не запустили ее:

```
uwsgi --ini config/uwsgi.ini
```

Откройте вторую консоль и запустите NGINX следующей командой:

```
service nginx start
```

Поскольку мы используем не настоящее доменное имя, нам необходимо перенаправить его на наш локальный хост. Отредактируйте файл `/etc/hosts` и добавьте в него следующие строки:

```
127.0.0.1 educaproject.com  
127.0.0.1 www.educaproject.com
```

Поступая таким образом, мы маршрутизуем оба имени хоста

на наш локальный сервер. На рабочем сервере вам не нужно будет этого делать, поскольку у вас будет фиксированный IP-адрес, и вы укажете имя вашего сервера в конфигурации DNS вашего домена.

Откройте `http://educaproject.com/` в своем браузере. Вы должны увидеть ваш сайт, который все еще не загружает статические файлы. Наша производственная среда почти готова.

Примечание переводчика: Если ваш сайт не открывается в браузере, то возможно вам не хватает разрешения для файла сокета `educa.sock` в каталоге `/tmp`. Выполните следующие команды:

chmod 777 /tmp/educa.sock

Потом перезапустите nginx:

service nginx restart

Теперь вы можете ограничить хосты вашего проекта Django. Отредактируйте файл производственных настроек `settings/pro.py` вашего проекта и измените настройку `ALLOWED_HOSTS` следующим образом:

```
ALLOWED_HOSTS = ['educaproject.com', 'www.educaproject.com']
```

Django теперь будет обслуживать ваше приложение только в том случае, если оно работает под любым из этих имен. Вы можете больше узнать о настройке в <https://docs.djangoproject.com/en/2.0/ref/settings/#allowed-hosts>.

Обслуживание статических и медиа файлов

NGINX очень хорошо обслуживает статический контент. Для лучшей производительности мы будем использовать NGINX для обслуживания статических файлов в нашей производственной среде. Мы настроим NGINX для обслуживания как статических файлов нашего приложения, так и медиафайлов, загруженных для содержимого курса.

Откройте файл `settings/base.py` и добавьте следующий код в него:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

Нам нужно экспортировать статические файлы в Django. Команда `collectstatic` копирует статические файлы из всех приложений и сохраняет их в каталоге `STATIC_ROOT`. Откройте консоль и выполните следующую команду:

```
python manage.py collectstatic
```

Вы увидите такой результат:

```
160 static files copied to '/educa/static'.
```

Теперь отредактируйте файл `config/nginx.conf` и добавьте следующий код внутри директивы `server`:

```
location /static/ {
    alias /home/projects/educa/static/;
}
```

```
location /media/ {  
    alias /home/projects/educa/media/;  
}
```

Не забудьте заменить путь `/home/projects/educa/` на абсолютный путь к вашей директории проекта. Эти директивы сообщают NGINX обслуживать статические файлы, расположенные в `/static/` и `/media/`. Эти пути следующие:

- `/static/`: Этот путь соответствует конфигурации в настройке `STATIC_URL`, а его целевой путь соответствует значению параметра `STATIC_ROOT`. Мы используем его для обслуживания статических файлов нашего приложения.
- `/media/`: Этот путь соответствует конфигурации в настройке `MEDIA_URL`, а его целевой путь соответствует значению параметра `MEDIA_ROOT`. Мы используем его для медиафайлов, загруженных в содержимое курса.

Перезагрузите NGINX следующей командой, чтобы отслеживать новые пути:

```
service nginx reload
```

Откройте `http://educaproject.com/` в вашем браузере. Ваш сайт должен правильно отображать статические ресурсы, такие как таблицы стилей CSS и изображения. NGINX теперь обслуживает статические файлы напрямую, а не перенаправляет запросы статических файлов на uWSGI.

Прекрасно! Вы успешно настроили NGINX для обслуживания статических файлов.

Защита соединения при помощи SSL

Протокол **Secure Sockets Layer (SSL)** становится нормой для обслуживания веб-сайтов через безопасное соединение.

Настоятельно рекомендуется, чтобы вы обслуживали ваши веб-сайты под HTTPS. Мы собираемся настроить SSL-сертификат в NGINX для безопасного обслуживания нашего сайта.

Создание сертификата SSL

Создайте новую папку в каталоге проекта `educa` и назовите ее `ssl`. Затем создайте SSL-сертификат из командной строки с помощью следующей команды:

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout  
ssl/educa.key -out ssl/educa.crt
```

Мы генерируем приватный ключ и 2048-битный SSL-сертификат, действительный в течение одного года. Вам будет предложено ввести данные следующим образом:

```
Country Name (2 letter code) [AU]:  
State or Province Name (full name) [Some-State]:  
Locality Name (eg, city) []:  
Organization Name (eg, company) [Internet Widgits Pty Ltd]:  
Organizational Unit Name (eg, section) []:  
Common Name (e.g. server FQDN or YOUR name) []: educaproject.com  
Email Address []: email@domain.com
```

Вы можете заполнить запрошенные данные своей собственной информацией. Наиболее важным полем является Common Name. Вы должны указать доменное имя для сертификата. Мы используем `educaproject.com`.

Это приведет к созданию внутри каталога `ssl/` файла приватного ключа `educa.key` и файла `educa.crt`, который является сертификатом.

Настройка NGINX для использования SSL

Откройте файл `nginx.conf` и отредактируйте директиву `server`, чтобы включить SSL следующим образом:

```
server {  
    listen          80;  
    listen          443 ssl;  
    ssl_certificate /home/projects/educa/ssl/educa.crt;  
    ssl_certificate_key /home/projects/educa/ssl/educa.key;  
    server_name      www.educaproject.com educaproject.com;  
    # ...  
}
```

Благодаря предыдущему коду наш сервер теперь прослушивает как HTTP через порт 80, так и HTTPS через порт 443. Мы указываем путь к сертификату SSL в `ssl_certificate` и ключ сертификата в `ssl_certificate_key`.

Перезапустите NGINX с помощью следующей команды:

```
sudo service nginx restart
```

NGINX загрузит новую конфигурацию. Откройте <https://educaproject.com/> в своем браузере. Вы должны увидеть предупреждающее сообщение, подобное следующему:



This Connection is Untrusted

You have asked Firefox to connect securely to educaproject.com, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

[Get me out of here!](#)

► Technical Details

▼ I Understand the Risks

If you understand what's going on, you can tell Firefox to start trusting this site's identification. **Even if you trust the site, this error could mean that someone is tampering with your connection.**

Don't add an exception unless you know there's a good reason why this site doesn't use trusted identification.

[Add Exception...](#)

Сообщение может отличаться в зависимости от вашего браузера. Оно предупреждает вас о том, что ваш сайт не использует доверенный сертификат: браузер не может проверить личность вашего сайта. Это связано с тем, что мы подписали наш собственный сертификат вместо того, чтобы получать его из доверенного **Центра сертификации (СА)**. Когда вы владеете реальным доменом, вы можете обратиться к доверенным ЦС, для получения сертификата SSL, чтобы браузеры могли проверить его личность.

Если вы хотите получить доверенный сертификат для реального домена, вы можете обратиться к проекту *Let's Encrypt*, созданному Linux Foundation. Это совместный проект, целью которого является упрощенное бесплатное получение и обновление доверенных сертификатов SSL. Вы можете найти

дополнительную информацию по адресу <https://letsencrypt.org>.

Нажмите кнопку Add Exception, чтобы ваш браузер знал, что вы доверяете этому сертификату. Вы увидите, что браузер отображает значок блокировки рядом с URL следующим образом:



Если вы нажмете на значок блокировки, будут отображены данные SSL-сертификата.

Настройка нашего проекта для SSL

Django поставляется с определенными настройками для поддержки SSL. Измените файл настроек `settings/pro.py` и добавьте в него:

```
| SECURE_SSL_REDIRECT = True  
| CSRF_COOKIE_SECURE = True
```

Эти настройки значат следующее:

- `SECURE_SSL_REDIRECT`: Должны ли HTTP-запросы перенаправляться на HTTPS
- `CSRF_COOKIE_SECURE`: Должен ли быть создан безопасный файл cookie для защиты от подделки запросов на межсайтовый запрос

Поздравляем! Вы настроили производственную среду, которая предложит отличную производительность для обслуживания вашего проекта.

Создание настраиваемого промежуточного программного обеспечения

Вы уже знакомы с настройкой `MIDDLEWARE`, которая содержит промежуточное программное обеспечение для вашего проекта. Вы можете думать о ней как о низко-уровневой системе плагинов, что позволит вам реализовать работу с процессами, которые выполняются во время запроса/ответа. Каждое промежуточное программное обеспечение отвечает за некоторые конкретные действия, которые будут выполняться для всех HTTP-запросов или ответов.

Избегайте добавления дорогостоящей обработки в middleware, поскольку они выполняются в каждом отдельном запросе.

Когда HTTP-запрос получен, промежуточное ПО выполняются в порядке появления в настройке `MIDDLEWARE`. Когда HTTP-ответ был сгенерирован Django, ответ проходит через все промежуточные ПО назад в обратном порядке.

Промежуточное программное обеспечение может быть записано как функция следующим образом:

```
def my_middleware(get_response):

    def middleware(request):
        # Code executed for each request before
        # the view (and later middleware) are called.

        response = get_response(request)

        # Code executed for each request/response after
        # the view is called.
```

```
    return response  
  
    return middleware
```

Фабрика промежуточного программного обеспечения является вызываемой, она принимает `get_response` и возвращает промежуточное программное обеспечение. Промежуточное ПО является вызываемым, оно принимает запрос и возвращает ответ, как и представление. Вызываемый `get_response` может быть следующим промежуточным программным обеспечением в цепочке или представлением.

Если какое-либо промежуточное ПО возвращает ответ, не вызывая `get_response`, оно прерывает процесс, никакие дополнительные посредники не выполняются (в том числе и представления), и ответ возвращается через те же слои, через какие передавался запрос.

Порядок посредников в настройке `MIDDLEWARE` очень важен, потому что промежуточное ПО может зависеть от данных, установленных в запросе другими промежуточными ПО, которые были выполнены ранее.

При добавлении нового промежуточного программного обеспечения в настройку `MIDDLEWARE` не забудьте поместить его в нужном порядке. Middlewares выполняются в порядке появления в настройках во время фазы запроса и в обратном порядке для ответов.

Дополнительную информацию о промежуточном программном обеспечении можно найти на странице <https://docs.djangoproject.com/en/2.0/topics/http/middleware/>.

Создание промежуточного ПО субдомена

Мы собираемся создать настраиваемое промежуточное программное обеспечение, чтобы позволить курсам быть доступными через пользовательский субдомен. Каждый URL-адрес подробного описания курса, который выглядит как `https://educaproject.com/course/django/`, также будет доступен через субдомен, который использует курс `slug`, например `https://django.educaproject.com/`. Пользователи смогут использовать субдомен для доступа к деталям курса. Любые запросы на субдомены будут перенаправлены на каждый соответствующий URL-адрес курса.

Промежуточное ПО может находиться в любом месте вашего проекта. Однако рекомендуется создать файл `middleware.py` в каталоге приложения.

Создайте новый файл в каталоге приложений `courses` и назовите его `middleware.py`. Добавьте к нему следующий 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):
    """
    Provides subdomains for courses
    """
    def middleware(request):
        host_parts = request.get_host().split('.')
        if len(host_parts) > 2 and host_parts[0] != 'www':
            # получить курс для данного субдомена
            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)
    return middleware
```

```
        args=[course.slug])
    # перенаправить текущий запрос в представление course_detail
    url = '{}://{}{}'.format(request.scheme,
                             '.'.join(host_parts[1:]),
                             course_url)
    return redirect(url)

    response = get_response(request)
    return response

return middleware
```

Когда HTTP-запрос получен, мы выполняем следующие задачи:

1. Мы получаем имя хоста, которое используется в запросе, и разделяем его на части. Например, если пользователь обращается к `mycourse.educaproject.com`, мы генерируем список `['mycourse', 'educaproject', 'com']`.
2. Мы проверяем, содержит ли хост имя субдомена, через определение, получено ли более двух элементов. Если имя хоста включает субдомен, и это не `www`, мы пытаемся получить курс со `slug` в субдомене.
3. Если курс не найден, мы вызываем исключение HTTP 404. В противном случае мы перенаправляем браузер на URL-адрес курса.

Измените файл `settings.py` (**примечание переводчика:** возможно имелся в виду файл `base.py`) проекта и добавьте `'courses.middleware.SubdomainCourseMiddleware'` в нижней части списка `MIDDLEWARE` следующим образом:

```
MIDDLEWARE = [
    # ...
    'courses.middleware.subdomain_course_middleware',
]
```

Теперь наше промежуточное программное обеспечение будет выполнено в каждом запросе.

Помните, что имена хостов, разрешенные для обслуживания нашего проекта Django, указаны в настройке `ALLOWED_HOSTS`.
Давайте изменим этот параметр так, чтобы любой возможный субдомен `educaproject.com` разрешалось обслуживать нашим приложениям.

Откройте файл `settings/pro.py` и измените настройку `ALLOWED_HOSTS` следующим образом:

```
ALLOWED_HOSTS = ['.educaproject.com']
```

Значение, начинающееся с точки, используется в качестве подстановочного поддомена. `'.educaproject.com'` будет соответствовать `educaproject.com` и любому поддомену для этого домена, например `course.educaproject.com` И `django.educaproject.com`.

Обслуживание нескольких субдоменов с помощью NGINX

Нам нужно, чтобы NGINX мог обслуживать наш сайт с любым возможным субдоменом. Отредактируйте файл `config/nginx.conf` и замените эту строку:

```
server_name www.educaproject.com educaproject.com;
```

На следующую:

```
server_name *.educaproject.com educaproject.com;
```

Благодаря звездочке это правило применяется ко всем субдоменам `educaproject.com`. Чтобы протестировать наше промежуточное ПО локально, нам нужно добавить любые поддомены, которые мы хотим проверить, к `/etc/hosts`. Для тестирования промежуточного программного обеспечения с объектом `course` с помощью `slug django` добавьте следующую строку в файл `/etc/hosts`:

```
127.0.0.1 django.educaproject.com
```

Откройте <https://django.educaproject.com/> в вашем браузере. Промежуточное программное обеспечение найдет курс по субдомену и перенаправит ваш браузер на <https://educaproject.com/course/django/>.

Внедрение пользовательских команд управления

Django позволяет вашим приложениям регистрировать пользовательские команды управления для утилиты `manage.py`. Например, мы использовали команды управления `makemessages` и `compilemessages` в [главе 9](#), *Расширение магазина* для создания и компиляции файлов перевода.

Команда управления состоит из модуля Python, содержащего класс `Command`, который наследуется от `django.core.management.base.BaseCommand` или одного из его подклассов. Вы можете создавать простые команды или передавать в них позиционные и необязательные аргументы в качестве входных данных.

Django ищет команды управления в каталоге `management/commands/` для каждого приложения зарегистрированного в настройке `INSTALLED_APPS`. Каждый найденный модуль будет зарегистрирован как команда управления, названная так же как и модуль.

Вы можете узнать больше о пользовательских командах управления на <https://docs.djangoproject.com/en/2.0/howto/custom-management-commands/>.

Мы собираемся создать пользовательскую команду управления, чтобы напомнить студентам о поступлении, по крайней мере, на один курс. Команда отправит напоминание по электронной почте пользователям, которые были зарегистрированы дольше указанного периода, но которые еще не записались ни на один курс.

Создайте следующую файловую структуру внутри каталога приложений `students`:

```
management/
    __init__.py
    commands/
        __init__.py
        enroll_reminder.py
```

Откройте файл `enroll_reminder.py` и добавьте к нему следующий код:

```
import datetime
from django.conf import settings
from django.core.management.base import BaseCommand
from django.core.mail import send_mass_mail
from django.contrib.auth.models import User
from django.db.models import Count

class Command(BaseCommand):
    help = 'Sends an e-mail reminder to users registered more \
            than N days that are not enrolled into any courses yet'

    def add_arguments(self, parser):
        parser.add_argument('--days', dest='days', type=int)

    def handle(self, *args, **options):
        emails = []
        subject = 'Enroll in a course'
        date_joined = datetime.date.today() - \
            datetime.timedelta(days=options['days'])
        users = User.objects.annotate(course_count=Count('courses_joined'))\
            .filter(course_count=0, date_joined__lte=date_joined)
        for user in users:
            message = "Dear {},\n\n We noticed that you didn't\
                enroll in any courses yet. What are you waiting\
                for?".format(user.first_name)
            emails.append((subject,
                           message,
                           settings.DEFAULT_FROM_EMAIL,
                           [user.email]))
        send_mass_mail(emails)
        self.stdout.write('Sent {} reminders'.format(len(emails)))
```

Это наша команда `enroll_reminder`. Предыдущий код работает

следующим образом:

- Класс `Command` наследуется от `BaseCommand`.
- Мы включаем атрибут `help`. Этот атрибут предоставляет краткое описание команды, которое напечатается, если вы запустите `python manage.py help enroll_reminder`.
- Мы используем метод `add_arguments()` для добавления аргумента с именем `--days`. Этот аргумент используется для указания минимального количества дней, когда, не зачисленный на какой-либо курс пользователь должен быть зарегистрирован, чтобы ему отправить напоминание.
- Функция `handle()` содержит команду. Мы получаем атрибут `days`, полученный из командной строки. Мы извлекаем пользователей, которые были зарегистрированы больше, чем указанные дни, и которые еще не записались на какие-либо курсы. Мы достигаем этого путем аннотации `QuerySet` с общим количеством курсов, на которые каждый пользователь зарегистрирован. Мы генерируем электронное письмо с напоминанием для каждого пользователя и добавляем его в список `email`. Наконец, мы отправляем электронные письма с помощью функции `send_mass_mail()`, которая оптимизирована для открытия одного SMTP-соединения для отправки всех сообщений электронной почты вместо того, чтобы создавать одно соединение для каждого отправляемого сообщения.

Вы создали свою первую команду управления. Откройте консоль и запустите команду:

```
python manage.py enrollReminder --days=20
```

Если у вас нет локального сервера SMTP, вы можете обратиться к [главе 2, Улучшение вашего блога с помощью расширения функционала](#) где мы настроили параметры SMTP для нашего первого проекта Django. В качестве альтернативы вы можете добавить следующий параметр в файл `settings.py` (**примечание переводчика:** возможно `base.py`), чтобы вывести исходящие сообщения Django на стандартный вывод на время разработки:

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

Давайте сделаем так, чтобы сервер выполнял нашу команду каждый день в 8 часов. Если вы используете систему на базе UNIX, такую как Linux или macOS X, откройте консоль и запустите `crontab -e`, чтобы отредактировать `crontab`. Добавьте к нему следующую строку:

```
0 8 * * * python /path/to/educa/manage.py enrollReminder --days=20 --  
settings=educa.settings.pro
```

Если вы не знакомы с **cron**, вы можете найти сведения о нем на <http://www.unixgeeks.org/security/newbie/unix/cron-1.html>.

Если вы используете Windows, вы можете планировать задачи с помощью Планировщика заданий. Вы можете найти дополнительную информацию о нем на [https://msdn.microsoft.com/en-us/library/windows/desktop/aa383614\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa383614(v=vs.85).aspx).

Другой способ периодически выполнять действия - это создавать задачи и планировать их с помощью Celery. Помните, что мы использовали Celery в [главе 7, Создание интернет-магазина](#) для выполнения асинхронных задач. Вместо того,

чтобы создавать команды управления и планировать их с помощью cron, вы можете создавать асинхронные задачи и выполнять их с помощью планировщика Celery. Вы можете узнать больше о планировании периодических заданий с помощью Celery на <https://celery.readthedocs.io/en/latest/userguide/periodic-tasks.html>.

Используйте команды управления для автономных скриптов, которые вы хотите запускать с помощью cron или панели управления планировщика Windows.

Django также включает утилиту для вызова команд управления с использованием Python. Вы можете запускать команды управления из своего кода следующим образом:

```
from django.core import management  
management.call_command('enroll_reminder', days=20)
```

Поздравляем! Теперь вы можете создавать собственные команды управления для своих приложений и планировать их запуск при необходимости.

Резюме

В этой главе вы сконфигурировали производственную среду с использованием uWSGI и NGINX. Вы также внедрили специальное промежуточное программное обеспечение, и вы узнали, как создавать пользовательские команды управления.

Вы дошли до конца этой книги. Поздравляем! Вы получили навыки, необходимые для успешного создания веб-приложений с помощью Django. Эта книга провела вас через процесс разработки реальных проектов и интеграции Django с другими технологиями. Теперь вы готовы создать свой собственный проект Django, будь то простой прототип или крупномасштабное веб-приложение.

Удачи вам с Django!

Другие книги, которыми вы можете наслаждаться

If you enjoyed this book, you may be interested in these other books by Packt:

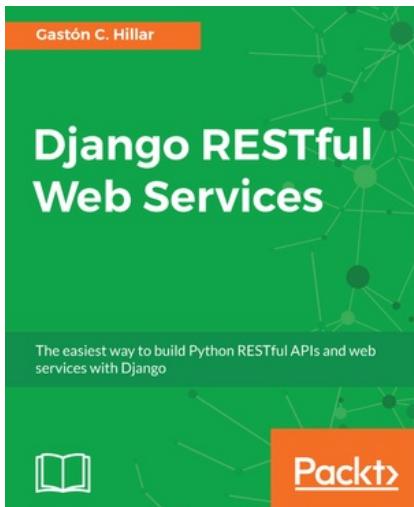


Python Programming Blueprints
Daniel Furtado, Marcus Pennington

ISBN: 978-1-78646-816-1

- Learn object-oriented and functional programming concepts while developing projects
- The dos and don'ts of storing passwords in a database
- Develop a fully functional website using the popular Django framework
- Use the BeautifulSoup library to perform web scrapping
- Get started with cloud computing by building microservice and serverless applications in AWS

- Develop scalable and cohesive microservices using the Nameko framework
- Create service dependencies for Redis and PostgreSQL



Django RESTful Web Services

Gastón C. Hillar

ISBN: 978-1-78883-392-9

- The best way to build a RESTful Web Service or API with Django and the Django REST Framework
- Develop complex RESTful APIs from scratch with Django and the Django REST Framework
- Work with either SQL or NoSQL data sources
- Design RESTful Web Services based on application requirements
- Use third-party packages and extensions to perform common tasks

- Create automated tests for RESTful web services
- Debug, test, and profile RESTful web services with Django and the Django REST Framework

Оставте отзыв - пусть другие читатели узнают, что вы думаете

Поделитесь своими мыслями об этой книге с другими, оставив обзор на сайте, на котором вы ее купили. Если вы купили книгу на Amazon, пожалуйста, оставьте нам честный обзор на странице этой книги. Это крайне важно, чтобы другие потенциальные читатели могли видеть и использовать ваше беспристрастное мнение для принятия решения о покупке, мы сможем понять что наши клиенты думают о наших продуктах, и наши авторы смогут видеть ваши отзывы. Это займет всего несколько минут вашего времени, но ценно для других потенциальных клиентов, наших авторов и Packt. Спасибо!