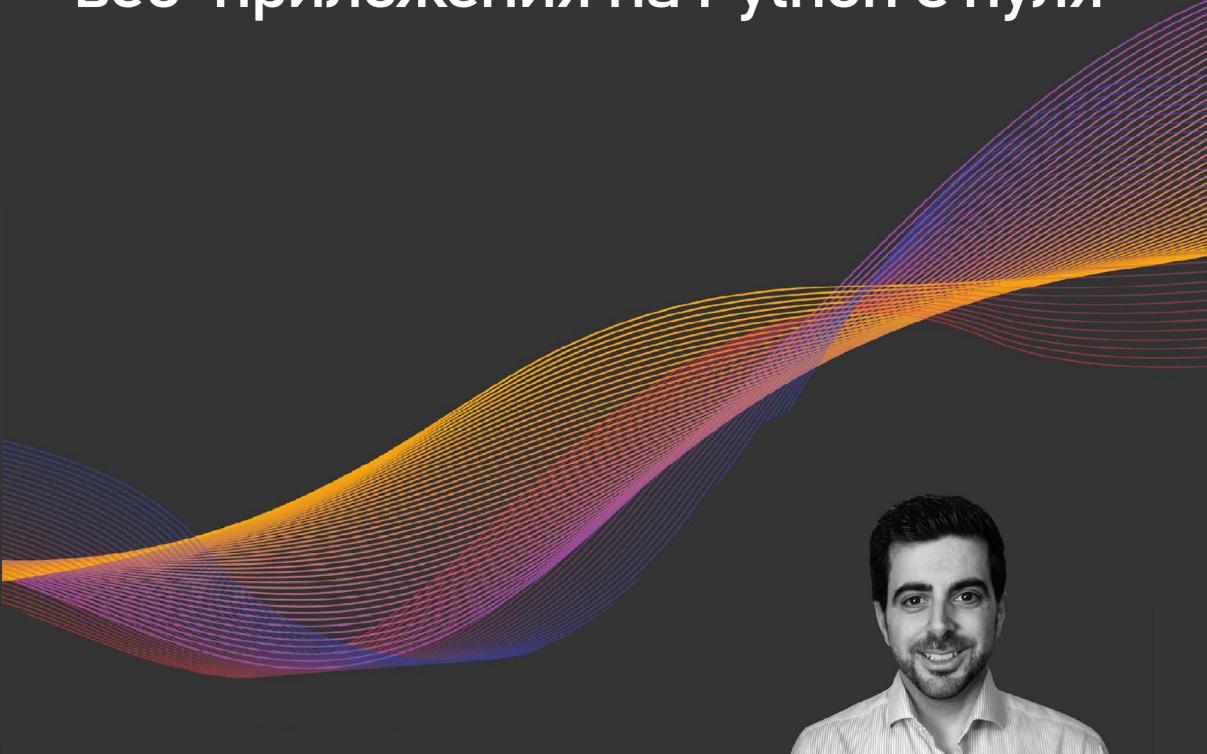


Django 4 в примерах

Разрабатывайте
мощные и надежные
веб-приложения на Python с нуля



Антонио Меле



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

Прочитав книгу, вы:

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

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

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

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

Интернет-магазин:
www.dmkpress.com

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

Packt

ДМК
издательство
www.dmk.ru

ISBN 978-5-93700-204-4



9 785937 002044 >

Антонио Меле

Django 4 в примерах

Antonio Melé

Django 4 By Example

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



BIRMINGHAM—MUMBAI

Антонио Меле

Django 4 в примерах

Разрабатывайте мощные и надежные
веб-приложения на Python с нуля



Москва, 2023

УДК 004.04
ББК 32.372
М47

Меле А.

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

ISBN 978-5-93700-204-4

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

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

УДК 004.04
ББК 32.372

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

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

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

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

Посвящается моей сестре

Содержание

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

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

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

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

П переопределение метода save() класса ModelForm	288
Р разработка букмаклера с помощью JavaScript	293
С создание представления детальной информации об изображениях.....	306
С создание миниатюр изображений с помощью easy-thumbnails	309
Д добавление асинхронных действий с помощью JavaScript	312
З загрузка JavaScript в DOM.....	314
З защита от подделки межсайтовых HTTP-запросов на JavaScript.....	315
В выполнение HTTP-запросов с помощью JavaScript	317
Д добавление бесконечной постраничной прокрутки в список изображений.....	323
Д дополнительные ресурсы	330
Резюме	331
Глава 7. Отслеживание действий пользователя	332
Р разработка системы подписки.....	333
Ф формирование взаимосвязей многие-ко-многим с промежуточной моделью.....	333
С создание представлений списка и детальной информации для профилей пользователей	336
Д добавление действий пользователя по подписке/отписке с помощью JavaScript	342
Р разработка типового приложения для потока активности	345
П применение фреймворка contenttypes	346
Д добавление обобщенных отношений в модели	347
И игнорирование повторных действий в потоке активности	351
Д добавление действий пользователя в поток активности.....	352
О отображение потока активности	355
О оптимизация наборов запросов, предусматривающих связанные объекты	355
П применение метода select_related()	356
П применение метода prefetch_related()	357
С создание шаблонов действий.....	357
И использование сигналов для денормализации количественных данных	361
Р работа с сигналами.....	361
К конфигурационные классы приложений	364
И использование меню отладочных инструментов Django.....	366
У установка меню отладочных инструментов Django.....	367
П панели меню отладочных инструментов Django	370
К команды меню отладочных инструментов Django	373
П подсчет просмотров изображений с помощью хранилища Redis.....	374
У установка платформы Docker	375
У установка хранилища Redis	375
И использование хранилища Redis вместе с Python	377
Х хранение просмотров изображений в хранилище Redis	379
Х хранение рейтинга в хранилище Redis.....	381
С следующие шаги с Redis	384
Д дополнительные ресурсы	385
Резюме	385

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

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

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

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

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

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

От издательства

Отзывы и пожелания

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

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

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

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

Нарушение авторских прав

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

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

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

Вступительное слово

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

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

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

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

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

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

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

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

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

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

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

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

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

– *Боб Белдербос*, соучредитель Pybites

Об авторе

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

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

О рецензенте

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

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

Предисловие

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

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

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

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

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

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

Для кого эта книга предназначена

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

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

О чем эта книга рассказывает

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

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

Каждая глава охватывает несколько функциональных возможностей Django.

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

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

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

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

использовать фреймворк сообщений и разработаете конкретно-прикладной бэкенд аутентификации.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Что нужно, чтобы извлечь максимум пользы из этой книги

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

Используемые обозначения

В этой книге используется ряд текстовых обозначений.

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

Блок исходного кода задается, как показано ниже:

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

admin.site.register(Post)
```

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

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

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

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

```
python manage.py runserver
```

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



Предупреждения или важные примечания помечаются так.



Советы и программные трюки выглядят так.

1

Разработка приложения для ведения блога

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

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

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

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

В данной главе будут освещены следующие темы:

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

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

Установка языка Python

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

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

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

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

```
python
```

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

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

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

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

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

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

Создание виртуальной среды Python

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

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

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

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

```
python -m venv my_env
```

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

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

```
py -m venv my_env
```

При этом будет использоваться программа быстрого запуска Python в Windows.

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

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

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

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

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

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

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

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

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

Установка веб-фреймворка Django

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

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

Установка Django с помощью pip

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

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

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

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

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

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

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



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

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

Новые функциональные возможности Django 4

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

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

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

Общий обзор веб-фреймворка Django

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

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

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

Главные компоненты веб-фреймворка

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

Обязанности в шаблоне архитектурного дизайна MTV Django распределены следующим образом:

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

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

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

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

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

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

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

Архитектура Django

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

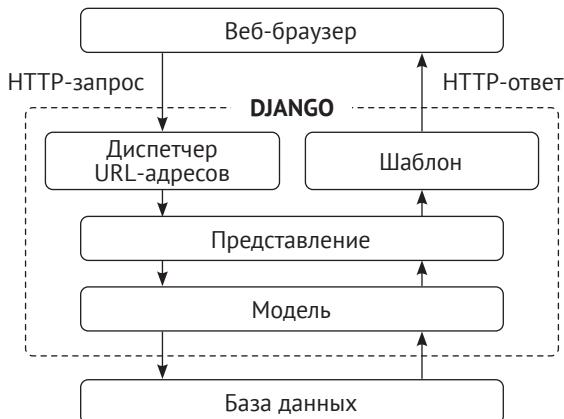


Рис. 1.1. Архитектура Django

Вот как Django оперирует HTTP-запросами и генерирует ответы:

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

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

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

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

Создание первого проекта

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

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

```
django-admin startproject mysite
```

Она создаст проект Django с именем `mysite`.



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

Давайте взглянем на сгенерированную структуру проекта:

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

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

¹ Син. зацепки, хуки. – Прим. перев.

² Англ. middleware. – Прим. перев.

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

Применение первоначальных миграций базы данных

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

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

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

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

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

Откройте приглашение командной оболочки и выполните следующие ниже команды:

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

Вы увидите результат, который заканчивается следующими ниже строками:

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

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

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

Запуск и выполнение сервера разработки

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

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

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

```
python manage.py runserver
```

Вы должны увидеть что-то вроде этого:

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

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

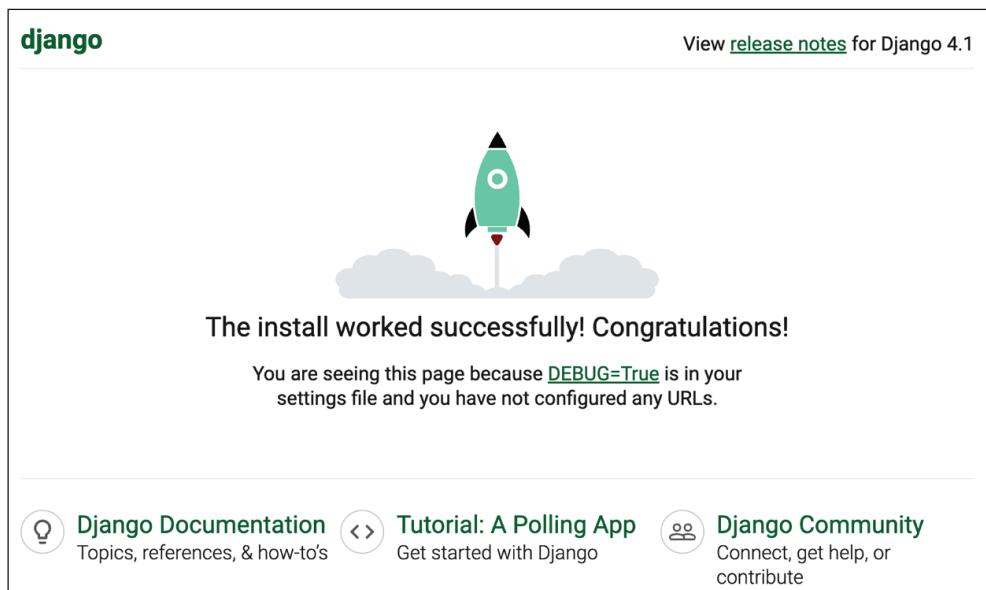


Рис. 1.2. Домашняя страница сервера разработки

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

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

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

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

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



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

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

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

Настроечные параметры проекта

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

Давайте рассмотрим некоторые настроечные параметры проекта.

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

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

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

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

Проекты и приложения

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

На рис. 1.3 показана структура проекта Django.

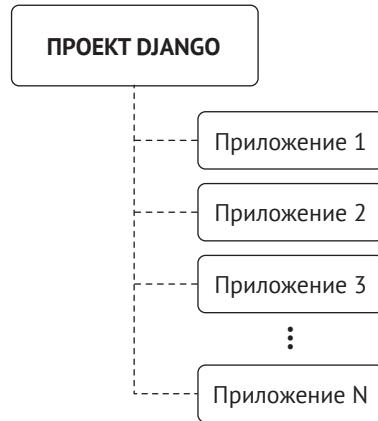


Рис. 1.3. Структура проекта/приложений Django

Создание приложения

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

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

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

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

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

Ниже приведено описание этих файлов:

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

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

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

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

Создание моделей данных блога

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

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

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

Создание модели поста

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

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

```
from django.db import models

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

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

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

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

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



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

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

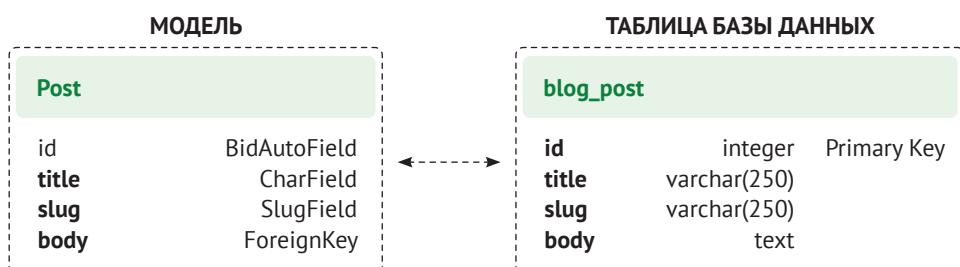


Рис. 1.4. Соответствие изначальной модели `Post` и таблицы базы данных

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

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

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

Добавление полей даты/времени

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

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

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

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

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

В модель `Post` были добавлены следующие ниже поля:

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

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

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

Определение предустановленного порядка сортировки

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

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

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

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

    class Meta:
        ordering = ['-publish']

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

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

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

Добавление индекса базы данных

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

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

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

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

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

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

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



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

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

Активация приложения

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

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

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

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

Добавление поля статуса

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

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

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

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

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

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

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

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

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

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

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



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

Давайте посмотрим, как взаимодействовать с вариантами статуса.

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

```
python manage.py shell
```

Затем наберите такие строки:

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

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

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

Наберите следующую ниже строку:

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

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

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

Наберите следующую ниже строку:

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

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

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

Наберите такую строку:

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

Вы получите имена вариантов, как показано ниже:

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

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

Добавление взаимосвязи многие-к-одному

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

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

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

class Post(models.Model):

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

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

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

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

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

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

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

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

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

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

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

Создание и применение миграций

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

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

Прежде всего необходимо создать первоначальную миграцию модели `Post`.

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

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

Вы должны получить результат, аналогичный приведенному ниже:

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

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

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

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

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

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

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

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

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

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

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

Создаются следующие три индекса базы данных:

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

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

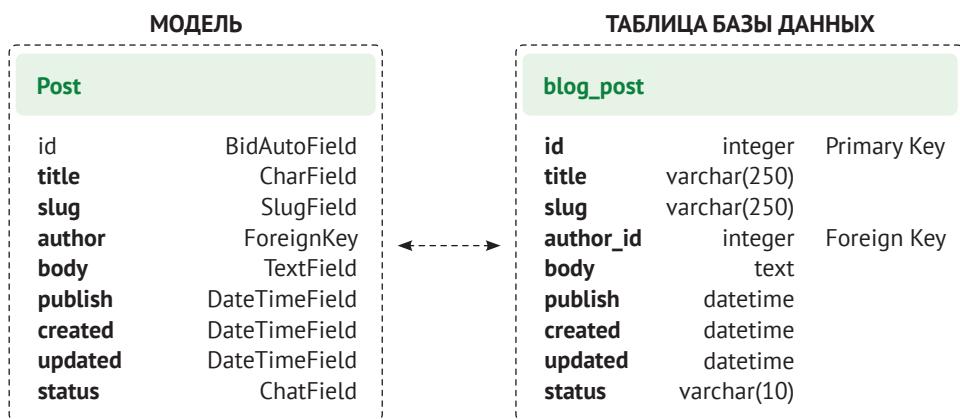


Рис. 1.5. Полное соответствие модели `Post` и таблицы базы данных

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

Давайте синхронизируем базу данных с новой моделью.

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

```
python manage.py migrate
```

Вы получите результат, который заканчивается следующей ниже строкой:

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

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

После применения миграций база данных отражает текущее состояние моделей.

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

Создание сайта администрирования для моделей

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

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

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

Создание суперпользователя

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

```
python manage.py createsuperuser
```

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

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

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

И вы увидите такое сообщение об успехе:

```
Superuser created successfully.
```

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

Сайт администрирования

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

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



Рис. 1.6. Экран входа на сайт администрирования

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

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

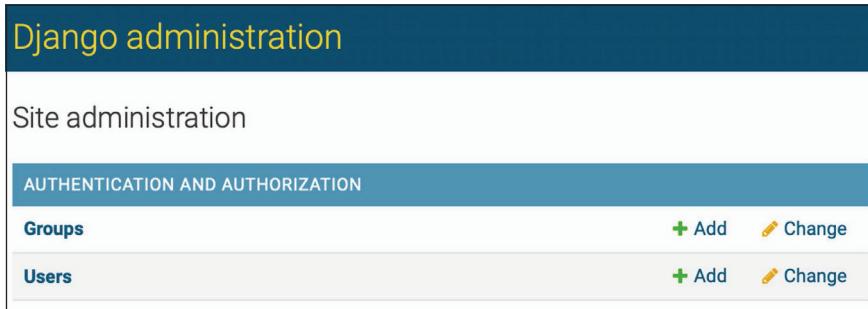


Рис. 1.7. Индексная страница сайта администрирования

Добавление моделей на сайт администрирования

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

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

admin.site.register(Post)
```

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

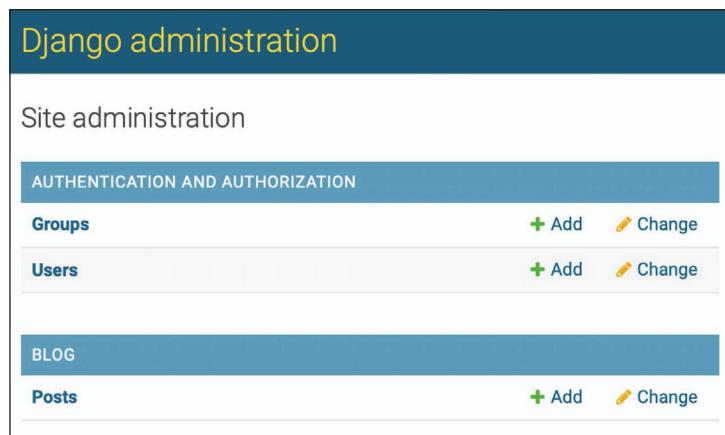


Рис. 1.8. Модель `Post` приложения `blog`, включенная в индексную страницу сайта администрирования

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

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

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

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

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

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

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

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

Адаптация внешнего вида моделей под конкретно-прикладную задачу

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

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

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

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

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

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

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

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

```

from django.contrib import admin
from .models import Post

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

```

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

Action:	TITLE	SLUG	AUTHOR	PUBLISH	STATUS
<input type="checkbox"/>	Who was Django Reinhardt?	who-was-django-reinhardt	admin	Jan 1, 2022, 11:39 p.m.	Draft

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

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

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

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

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

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

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

Author:	1	🔍
---------	---	---

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

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

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

Работа с наборами запросов QuerySet и менеджерами

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

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

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

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

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

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

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

Создание объектов

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

```
python manage.py shell
```

Затем наберите следующие ниже строки:

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

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

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

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

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

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

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

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

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

```
post.save()
```

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

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

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

Обновление объектов

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

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

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



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

Извлечение объектов

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

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

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

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

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

Применение метода `filter()`

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

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

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

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

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

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



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

Применение метода `exclude()`

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

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

Применение метода `order_by()`

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

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

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

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

Удаление объектов

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

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

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

Когда вычисляются наборы запросов QuerySet

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

Наборы запросов QuerySet вычисляются только в следующих ниже случаях:

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

Создание модельных менеджеров

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

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

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

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

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

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

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

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

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

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

Следующей ниже командой снова запустите сервер разработки из командной оболочки:

```
python manage.py shell
```

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

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

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

Разработка представлений списка и детальной информации

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

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

Создание представлений списка постов и детальной информации о посте

Давайте начнем с создания представления списка постов на странице.

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

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

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

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

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

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

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

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

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

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

```
from django.http import HttpResponse

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

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

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

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

Применение функции сокращенного доступа `get_object_or_404()`

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

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

```
from django.shortcuts import render, get_object_or_404

# ...
```

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

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

Добавление шаблонов URL-адресов представлений

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

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

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

app_name = 'blog'

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

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

¹ Также пространство имен. – Прим. перев.

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

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

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



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

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

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

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

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

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

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

Создание шаблонов представлений

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

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

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

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

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

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

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

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

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

Создание базового шаблона

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

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

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

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

Создание шаблона списка постов

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

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

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

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

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

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



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

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

Доступ к приложению

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

```
python manage.py runserver
```

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

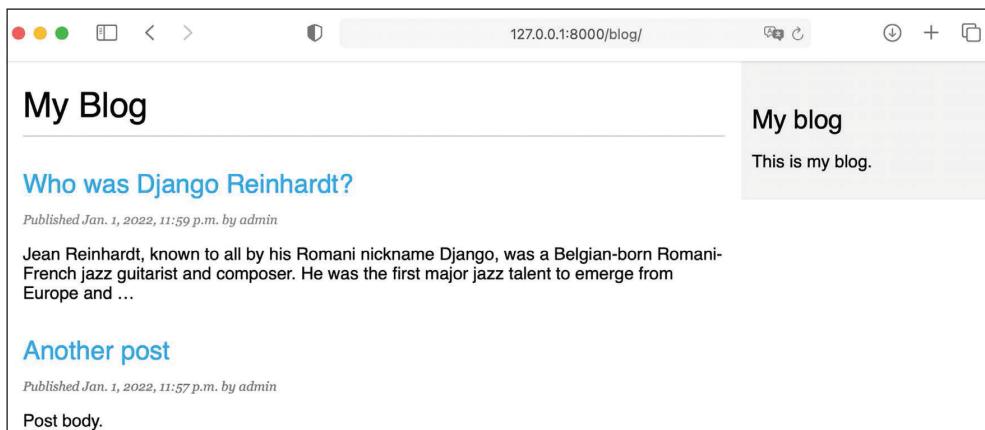


Рис. 1.14. Страница представления списка постов

Создание шаблона детальной информации о посте

Далее отредактируйте файл `post/detail.html`:

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

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

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


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


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

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



Рис. 1.15. Страница представления детальной информации о посте

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

Цикл запроса/ответа

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

Рассмотрим процесс запроса/ответа Django.

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

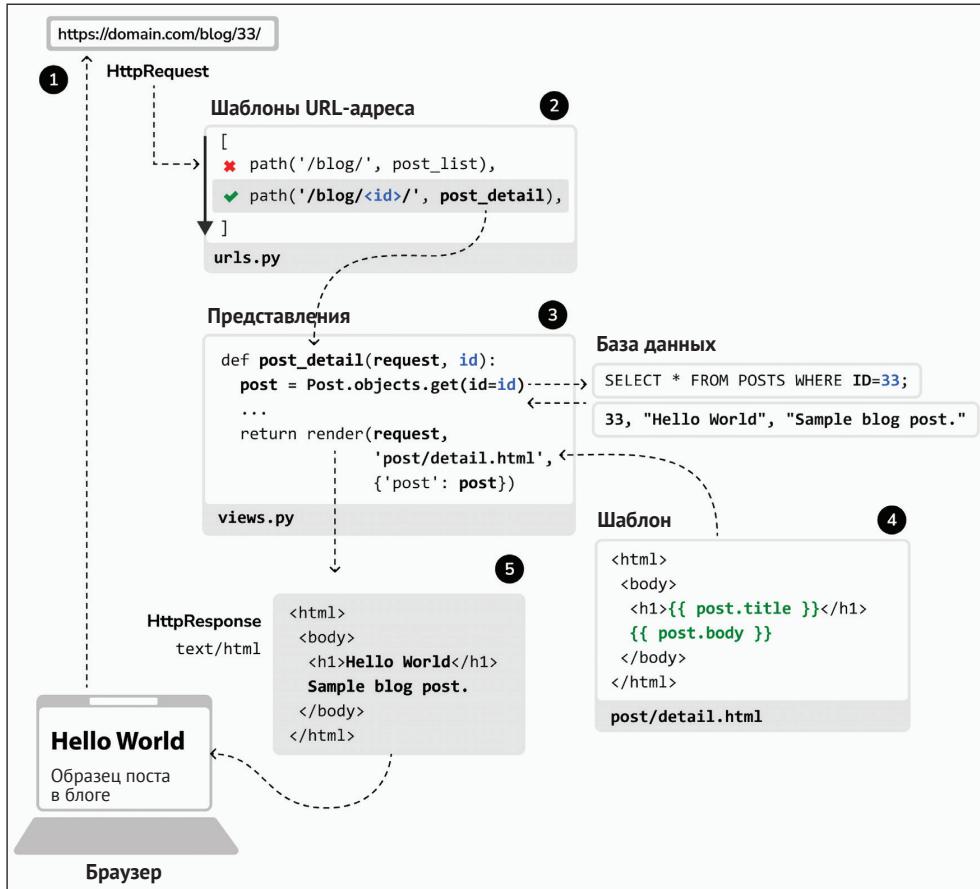


Рис. 1.16. Цикл запроса/ответа Django

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

Дополнительные ресурсы

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

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

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

Резюме

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

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

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

Присоединяйтесь к нам на Discord

Читайте эту книгу вместе с другими пользователями и автором.

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

<https://packt.link/django>



2

Усовершенствование блога за счет продвинутых функциональностей

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

В данной главе будут рассмотрены следующие темы:

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

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

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

Использование канонических URL-адресов для моделей

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

Мы будем использовать URL-адрес `post_detail`, определенный в шаблонах URL-адресов приложения, чтобы формировать канонический URL-адрес для объектов `Post`. Django предоставляет различные функции-резольверы `URL-fhtcjd1`, которые позволяют формировать URL-адреса динамически, используя их имя и любые требуемые параметры. Мы будем использовать функцию-утилиту `reverse()2` модуля `django.urls`.

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

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

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

class Post(models.Model):

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

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

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

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

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

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

    def __str__(self):
        return self.title

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

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

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

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

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

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

строкой

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

Теперь файл `blog/post/list.html` должен выглядеть следующим образом:

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

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

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

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

```
python manage.py runserver
```

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

Создание дружественных для поисковой оптимизации URL-адресов постов

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

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

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

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

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

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

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

Выполните следующую ниже команду в командной оболочке:

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

Вы должны получить следующий ниже результат:

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

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

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

```
python manage.py migrate
```

Вы получите результат, который заканчивается такой строкой:

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

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

Видоизменение шаблонов URL-адресов

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

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

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

строками

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

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

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

app_name = 'blog'

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

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

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

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

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

Видоизменение представлений

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

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

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

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

Видоизменение канонического URL-адреса постов

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

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

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

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

```
python manage.py runserver
```

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



Рис. 2.1. Страница представления детальной информации о посте

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

Добавление постраничной разбивки

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

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

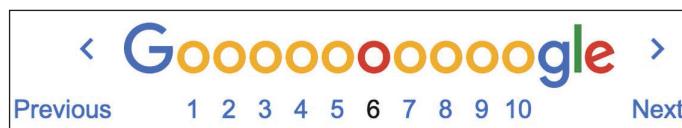


Рис. 2.2. Постранично разбитые ссылки Google
на страницы результатов поиска

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

Добавление постраничной разбивки в представление списка постов

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

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

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

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

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

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

Создание шаблона постраничной разбивки

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

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

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

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

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

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

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

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

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

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

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

```
python manage.py runserver
```

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

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

The screenshot shows a blog interface with a header "My Blog". Below it is a sidebar with "My blog" and "This is my blog.". The main content area displays three blog posts:

- Notes on Duke Ellington**
Published Jan. 3, 2022, 1:19 p.m. by admin
Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...
- Who was Miles Davis?**
Published Jan. 2, 2022, 1:18 p.m. by admin
Miles Davis was an American trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.
- Who was Django Reinhardt?**
Published Jan. 1, 2022, 11:59 p.m. by admin
Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...

At the bottom left, there is a link "Page 1 of 2. Next".

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

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

The screenshot shows the second page of the blog results. The browser address bar indicates the URL is 127.0.0.1:8000/blog/?page=2. The page content is identical to the first page, displaying the same three posts. The sidebar "My blog" and "This is my blog." remains visible.

Рис. 2.4. Вторая страница результатов

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

Обработка ошибок постраничной разбивки

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

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

EmptyPage at /blog/

That page contains no results

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

Рис. 2.5. Страница с ошибкой `EmptyPage`

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

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

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

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

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

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

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

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



Рис. 2.6. Последняя страница результатов

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

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



Рис. 2.7. Страница ошибки PageNotAnInteger

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

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

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

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

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

Теперь постраничная разбивка постов блога полностью реализована.

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

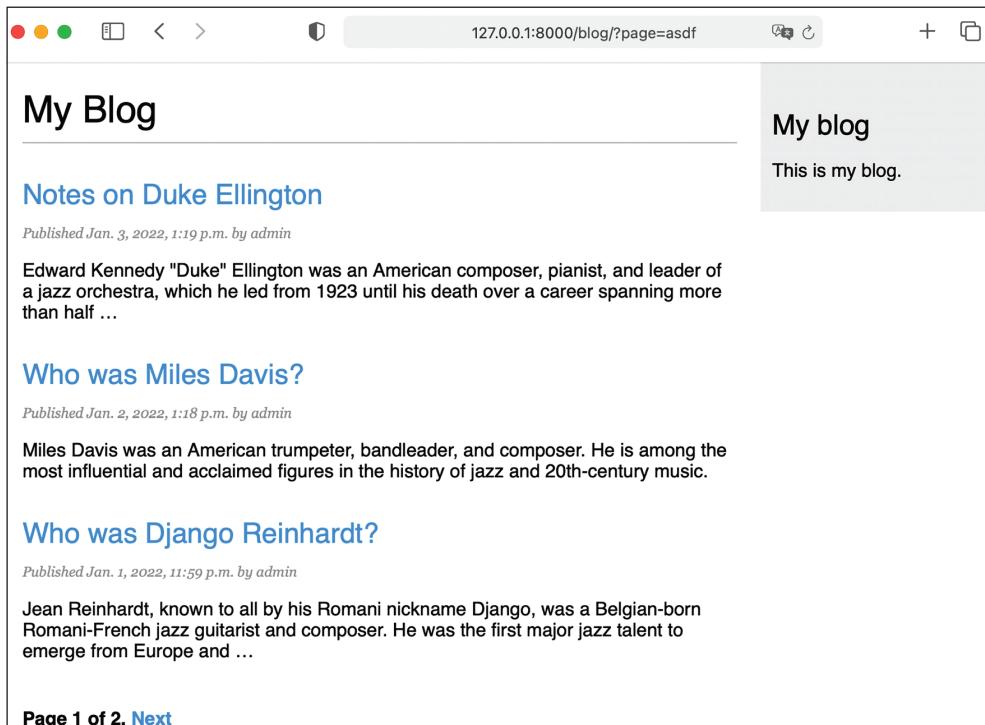


Рис. 2.8. Первая страница результатов

Разработка представлений на основе классов

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

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

Зачем использовать представления на основе классов

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

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

Использование представления на основе класса для отображения списка постов

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

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

```
from django.views.generic import ListView

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

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

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

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

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

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

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

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

{% block content %}
<h1>My Blog</h1>
{% for post in posts %}
<h2>
<a href="{{ post.get_absolute_url }}">
{{ post.title }}
</a>
</h2>
<p class="date">
Published {{ post.publish }} by {{ post.author }}

```

```
</p>
{{ post.body|truncatewords:30|linebreaks }}
{% endfor %}
{% include "pagination.html" with page=page_obj %}
{% endblock %}
```

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

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

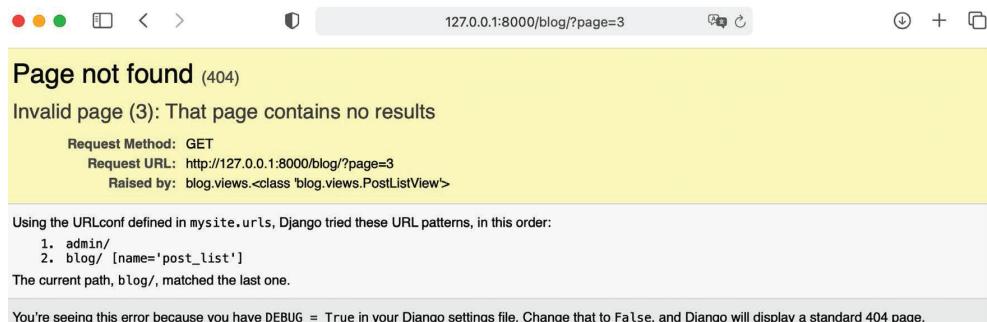


Рис. 2.9. HTTP-ответ с кодом состояния 404 (Страница не найдена)

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

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

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

Рекомендация постов по электронной почте

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

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

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

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

Разработка форм с помощью Django

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

Django поставляется с двумя базовыми классами для разработки форм:

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

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

```
from django import forms

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

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



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

Форма содержит следующие ниже поля:

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

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

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

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

Работа с формами в представлениях

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

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

```
from .forms import EmailPostForm

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

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

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

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

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

```
form = EmailPostForm()
```

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

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

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



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

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

Отправка электронных писем с помощью Django

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

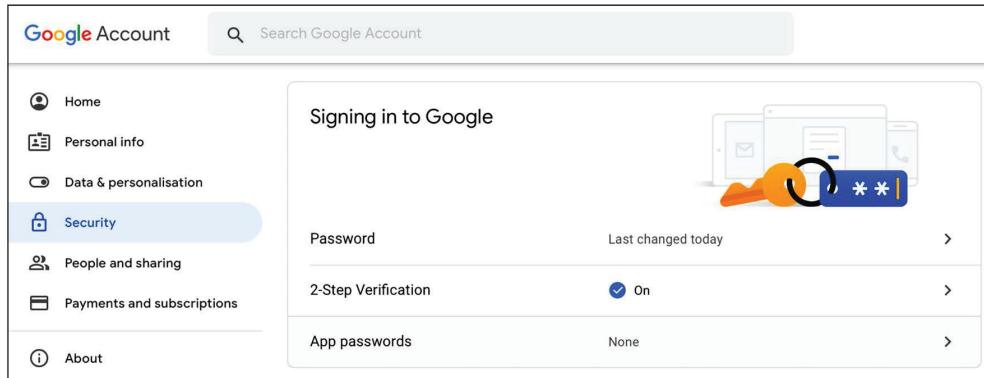


Рис. 2.10. Вход в Google для учетных записей Google

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

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

The screenshot shows a form for generating an app password. At the top, there's a back arrow and the title 'App passwords'. Below that is a descriptive text: 'App passwords let you sign in to your Google Account from apps on devices that don't support 2-Step Verification. You'll only need to enter it once so you don't need to remember it.' A 'Learn more' link is also present. The main part of the form has two dropdown menus: 'Select app' (with options: Mail, Calendar, Contacts, YouTube, Other (Custom name), and a currently selected 'Other (Custom name)' option) and 'Select device' (with a dropdown arrow). A large 'GENERATE' button is located at the bottom right of the form area.

Рис. 2.11. Форма для генерирования нового пароля приложения Google

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

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

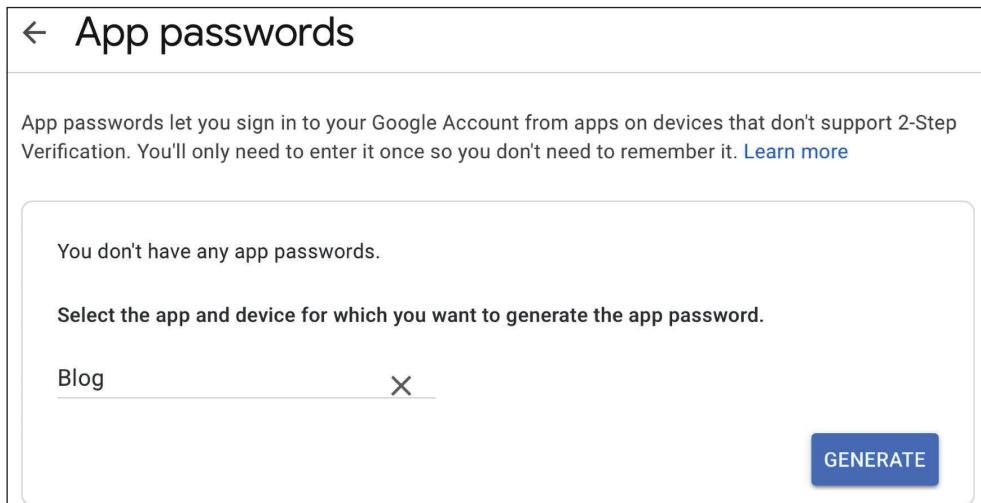


Рис. 2.12. Форма для генерирования нового пароля приложения Google

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

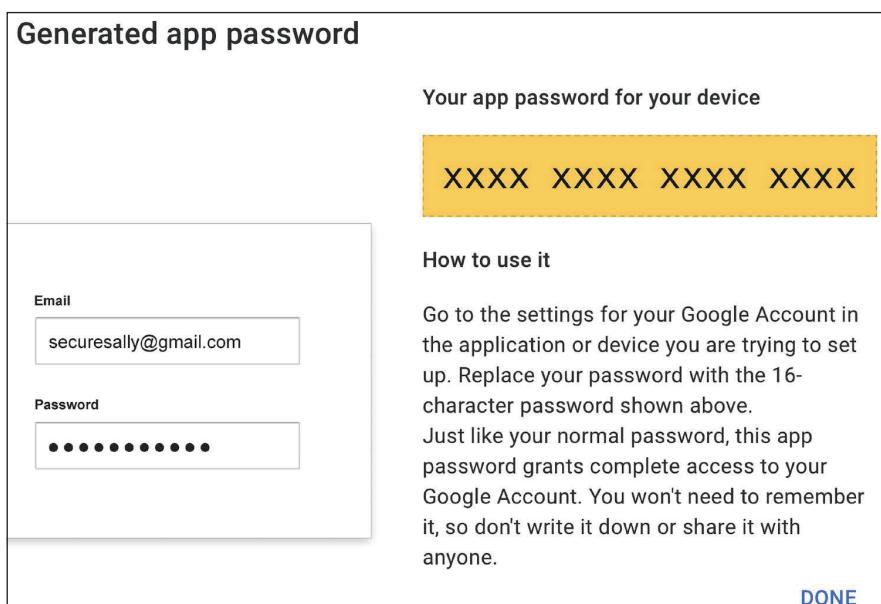


Рис. 2.13. Сгенерированный пароль приложения Google

Скопируйте сгенерированный пароль приложения.

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

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

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

```
python manage.py shell
```

Исполните следующий ниже исходный код в оболочке Python:

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

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

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

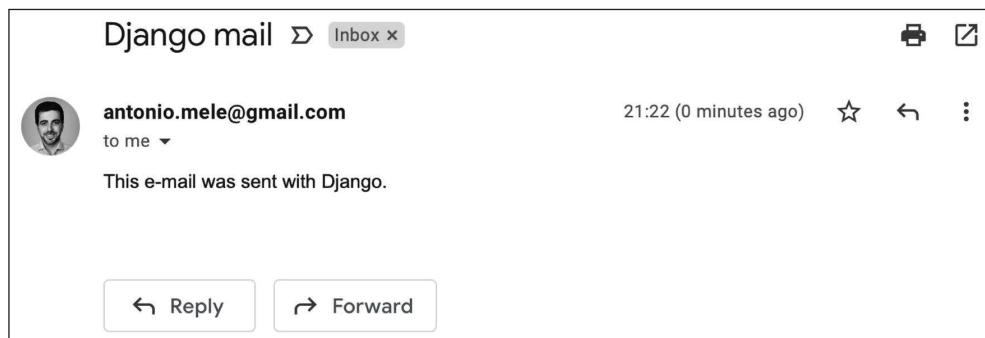


Рис. 2.14. Отправленное тестовое электронное письмо отображается в Gmail

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

Давайте добавим эту функциональность в представление post_share.

Отправка электронных писем в представлениях

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

```
from django.core.mail import send_mail

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

    sent = False

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

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

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

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

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

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

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

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

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

app_name = 'blog'

urlpatterns = [
    # представления поста
    path('', views.post_list, name='post_list'),
    path('', views.PostListView.as_view(), name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
]
```

Прорисовка форм в шаблонах

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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



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

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

```
{% extends "blog/base.html" %}  
  
{% block title %}{{ post.title }}{% endblock %}  
  
{% block content %}  
    <h1>{{ post.title }}</h1>  
    <p class="date">  
        Published {{ post.publish }} by {{ post.author }}  
    </p>  
    {{ post.body|linebreaks }}  
    <p>  
        <a href="{% url "blog:post_share" post.id %}">  
            Share this post  
        </a>  
    </p>  
{% endblock %}
```

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

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

```
python manage.py runserver
```

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

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

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

The screenshot shows a blog post titled "Notes on Duke Ellington". Above the post, there is a sidebar with the text "My blog" and "This is my blog.". Below the title, the post is published on "Jan. 3, 2022, 1:19 p.m. by admin". The post content reads: "Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half a century." At the bottom of the post, there is a blue button labeled "Share this post".

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

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

The screenshot shows a sharing form titled "Share 'Notes on Duke Ellington' by e-mail". It has fields for "Name" (with a redacted input), "Email" (with a redacted input), and "To" (with a redacted input). There is also a "Comments:" text area and a "SEND E-MAIL" button.

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

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

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



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

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

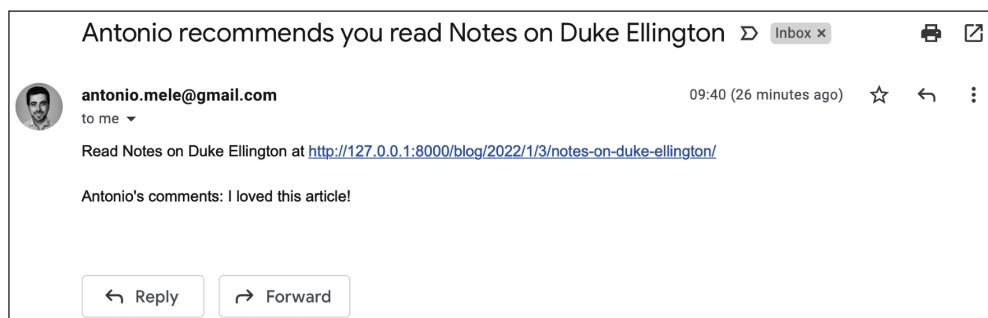


Рис. 2.18. Отправленное тестовое электронное письмо отображается в Gmail

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

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

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

The screenshot shows a web form titled "Share 'Notes on Duke Ellington' by e-mail". The form includes fields for Name (Antonio), Email (Invalid), To (empty), and Comments (empty). A red error message "Enter a valid email address." is displayed above the Email field. A second error message "This field is required." is displayed above the To field. A blue "SEND E-MAIL" button is at the bottom left.

Name:
Antonio

Email:
Invalid

• Enter a valid email address.
• This field is required.

To:

Comments:

SEND E-MAIL

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

Создание системы комментариев

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

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

Разработка модели комментария

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

Откройте файл `models.py` приложения `blog` и добавьте в него следующий ниже исходный код:

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

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

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

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

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

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

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

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

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

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

Выполните следующую ниже команду из командной оболочки:

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

Вы должны увидеть следующий ниже результат:

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

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

Выполните следующую ниже команду, чтобы применить существующие миграции:

```
python manage.py migrate
```

Вы получите результат, который содержит следующую ниже строку:

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

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

Добавление комментариев на сайт администрирования

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

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

```
from .models import Post, Comment

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

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

```
python manage.py runserver
```

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

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

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

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

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

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

Теперь появилась возможность управлять экземплярами комментариев с помощью сайта администрирования.

Создание форм из моделей

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

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

```
from .models import Comment

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

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

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

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

Оперирование формами `ModelForm` в представлениях

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

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

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

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

# ...

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

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

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

В этом представлении реализованы следующие ниже действия.

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

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

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



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

6. Пост назначается созданному комментарию:

```
comment.post = post
```

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

```
comment.save()
```

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

Давайте создадим шаблон URL-адреса этого представления.

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

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

app_name = 'blog'

urlpatterns = [
    # представления поста
    # path('', views.post_list, name='post_list'),
    path('', views.PostListView.as_view(), name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>',
         views.post_detail,
         name='post_detail'),
    path('<int:post_id>/share/',
         views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
         views.post_comment, name='post_comment'),
]
```

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

Создание шаблонов комментарной формы

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

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

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

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

Файловая структура должна выглядеть следующим образом:

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

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

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

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

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

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

Теперь файловая структура должна выглядеть следующим образом:

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

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

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

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

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

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

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

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

Добавление комментариев в представление детальной информации о посте

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

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

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

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

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

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

Добавление комментариев в шаблон детальной информации о посте

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

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

Мы начнем с добавления общего числа комментариев к посту.

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

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

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

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
  <p>
    <a href="{% url "blog:post_share" post.id %}">
      Share this post
    </a>
  </p>
  {% with comments.count as total_comments %}
    <h2>
      {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
  {% endwith %}
{% endblock %}
```

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



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

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

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

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

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

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

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

{% block content %}
<h1>{{ post.title }}</h1>
<p class="date">
    Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|linebreaks }}
<p>
    <a href="{% url "blog:post_share" post.id %}">
        Share this post
    </a>
</p>
{% with comments.count as total_comments %}
<h2>
    {{ total_comments }} comment{{ total_comments|pluralize }}
</h2>
```

```

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

```

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

Наконец, давайте добавим форму комментария в шаблон.

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

```

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

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

{% block content %}
<h1>{{ post.title }}</h1>
<p class="date">
  Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|linebreaks }}
<p>
  <a href="{% url "blog:post_share" post.id %}">
    Share this post
  </a>
</p>
{% with comments.count as total_comments %}
  <h2>
    {{ total_comments }} comment{{ total_comments|pluralize }}
  </h2>

```

```
{% endwith %}  
{% for comment in comments %}  
  <div class="comment">  
    <p class="info">  
      Comment {{ forloop.counter }} by {{ comment.name }}  
      {{ comment.created }}  
    </p>  
    {{ comment.body|linebreaks }}  
  </div>  
  {% empty %}  
  <p>There are no comments.</p>  
  {% endfor %}  
  {% include "blog/post/includes/comment_form.html" %}  
  {% endblock %}
```

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

Notes on Duke Ellington

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

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

[Share this post](#)

0 comments

There are no comments yet.

Add a new comment

Name:

Email:

Body:

[ADD COMMENT](#)

My blog

This is my blog.

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

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

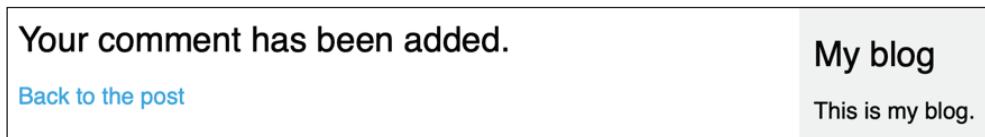


Рис. 2.23. Страница успешного добавления комментария

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

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

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

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

2 comments

Comment 1 by Antonio Jan. 3, 2022, 7:58 p.m.

I didn't know that!

Comment 2 by Bienvenida Jan. 3, 2022, 9:13 p.m.

I really like this article.

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

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

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

2 comments

Рис. 2.26. Список комментариев на сайте администрирования

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

Change comment

Comment by Antonio on Notes on Duke Ellington

Post: Notes on Duke Ellington

Name: Antonio

Email: test_account@gmail.com

Body: I didn't know that!

Active

Delete **Save and add another** **Save and continue editing** **SAVE**

Рис. 2.27. Редактирование комментария на сайте администрирования

Вы будете перенаправлены на список комментариев. В столбце **Active** отобразится неактивный значок комментария, как показано на рис. 2.28:

The comment "Comment by Antonio on Notes on Duke Ellington" was changed successfully.

Select comment to change

Action: Go 0 of 2 selected

<input type="checkbox"/>	NAME	EMAIL	POST	CREATED	ACTIVE
<input type="checkbox"/>	Antonio	test_account@gmail.com	Notes on Duke Ellington	Jan. 3, 2022, 7:58 p.m.	
<input type="checkbox"/>	Bienvenida	test_account2@gmail.com	Notes on Duke Ellington	Jan. 3, 2022, 9:13 p.m.	

2 comments

Рис. 2.28. Активные/неактивные комментарии на сайте администрирования

Если вернуться к представлению детальной информации о посте, то можно заметить, что неактивный комментарий больше не отображается и не учитывается при подсчете общего числа активных комментариев к посту:

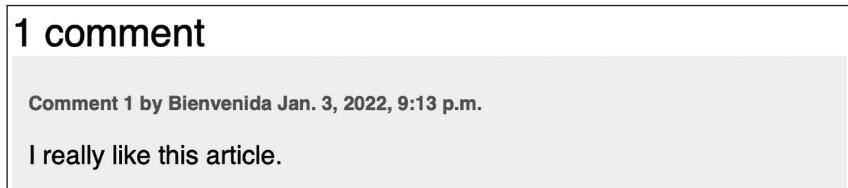


Рис. 2.29. Один активный комментарий, отображаемый на странице детальной информации о посте

Благодаря полю **Active** можно деактивировать неуместные комментарии и не показывать их в своих постах.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter02>.
- Функции-утилиты для URL-адресов: <https://docs.djangoproject.com/en/4.1/ref/urlresolvers/>.
- Конверторы путей URL-адресов: <https://docs.djangoproject.com/en/4.1/topics/http/urls/#path-converters>.
- Встроенный в Django класс постраничной разбивки: <https://docs.djangoproject.com/en/4.1/ref/paginator/>.
- Введение в представления на основе классов: <https://docs.djangoproject.com/en/4.1/topics/class-based-views/intro/>.
- Отправка электронных писем с помощью Django: <https://docs.djangoproject.com/en/4.1/topics/email/>.
- Типы полей формы в Django: <https://docs.djangoproject.com/en/4.1/ref/forms/fields/>.
- Работа с формами: <https://docs.djangoproject.com/en/4.1/topics/forms/>.
- Создание форм из моделей: <https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/>.
- Модельные взаимосвязи многие-к-одному: https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_one/.

Резюме

В данной главе вы научились определять для моделей канонические URL-адреса. Вы создали дружественные для поисковой оптимизации URL-адреса постов блога и реализовали постраничную разбивку объектов для списка постов. Вы также научились работать с формами Django и моделировать формы. Вы создали систему рекомендации постов по электронной почте и систему комментариев для своего блога.

В следующей главе вы создадите в своем блоге систему тегирования. Вы научитесь формировать сложные наборы запросов, чтобы извлекать объекты по сходству. Вы освоите создание конкретно-прикладных шаблонных тегов и фильтров. Вы также разработаете конкретно-прикладную карту сайта и новостную ленту для постов блога и реализуете функциональность полно-текстового поиска постов.

3

Расширение приложения для ведения блога

В предыдущей главе были рассмотрены основы форм и создание комментарийной системы. Вы также научились отправлять электронные письма с помощью Django. В этой главе вы расширите свое приложение для ведения блога другими популярными используемыми на блоговых платформах функциональными возможностями, такими как тегирование, рекомендация схожих постов, предоставление читателям новостной RSS-ленты и поиск постов. В ходе разработки этих функциональностей вы узнаете о новых компонентах и функциональностях Django.

В данной главе будут рассмотрены следующие темы:

- интегрирование сторонних приложений;
- использование приложения `django-taggit` для реализации системы тегирования;
- формирование сложных наборов запросов для рекомендации схожих постов;
- создание конкретно-прикладных шаблонных тегов и фильтров для показа на боковой панели списка последних постов и постов, получивших наибольшее число комментариев;
- создание карты сайта с использованием фреймворка карт сайтов;
- формирование новостной RSS-ленты с использованием фреймворка синдицированных новостных лент;
- установка базы данных PostgreSQL;
- реализация полнотекстового поискового механизма с использованием Django и PostgreSQL.

Исходный код к этой главе находится по адресу <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter03>.

Все используемые в этой главе пакеты Python включены в файл `requirements.txt` в исходном коде к данной главе. Следуйте инструкциям по уста-

новке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Добавление функциональности тегирования

Очень распространенной функциональностью в блогах является категоризация постов посредством тегов. Теги позволяют классифицировать контент неиерархическим образом, используя простые ключевые слова. Тег – это просто метка или ключевое слово, которое можно назначать постам. Мы создадим систему тегирования, интегрировав в проект стороннее приложение Django по тегированию.

`django-taggit` – это приспособленное для реиспользования приложение, которое в первую очередь предлагает модель `Tag` и менеджер для удобного добавления тегов в любую модель. Исходный код приложения доступен для просмотра на странице <https://github.com/jazzband/django-taggit>.

Сначала необходимо установить приложение `django-taggit` с помощью `pip`, выполнив следующую ниже команду:

```
pip install django-taggit==3.0.0
```

Затем откройте файл `settings.py` проекта `mysite` и добавьте `taggit` в настроечный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
    'taggit',
]
```

Откройте файл `models.py` приложения `blog` и добавьте предоставляемый приложением `django-taggit` менеджер `TaggableManager` в модель `Post`, используя следующий ниже исходный код:

```
from taggit.managers import TaggableManager

class Post(models.Model):
    # ...
    tags = TaggableManager()
```

Менеджер `tags` позволит добавлять, извлекать и удалять теги из объектов `Post`.

Приведенная ниже схема показывает модели данных, определенные в приложении `django-taggit` для создания тегов и хранения схожих тегированных объектов:

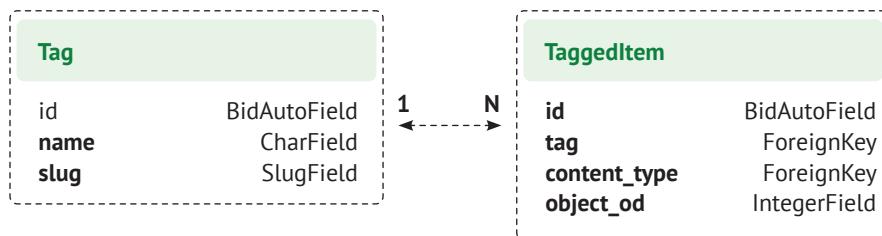


Рис. 3.1. Модель Tag приложения django-taggit

Модель `Tag` используется для хранения тегов. Она содержит поля `name` и `slug`.

Модель `TaggedItem` используется для хранения схожих тегированных объектов. В ней есть поле `ForeignKey` для схожего объекта `Tag`. Она содержит внешний ключ (`ForeignKey`) к объекту `ContentType` и целочисленное поле (`IntegerField`) для хранения соответствующего `id` тегированного объекта. Поля `content_type` и `object_id` в сочетании образуют обобщенное отношение с любой моделью в проекте. Такой подход позволяет создавать взаимосвязи между экземпляром модели `Tag` и экземпляром любой другой модели своих собственных приложений. Об обобщенных отношениях вы узнаете в главе 7 «Отслеживание действий пользователя».

Выполните следующую ниже команду в командной оболочке, чтобы создать миграцию изменений в модели:

```
python manage.py makemigrations blog
```

Вы должны получить такой результат:

```
Migrations for 'blog':
  blog/migrations/0004_post_tags.py
    - Add field tags to post
```

Теперь выполните следующую ниже команду, чтобы создать необходимые таблицы базы данных для моделей приложения `django-taggit` и синхронизировать изменения в своей модели:

```
python manage.py migrate
```

Вы увидите результат, говорящий о том, что миграции были применены, как показано ниже:

```
Applying taggit.0001_initial... OK
Applying taggit.0002_auto_20150616_2121... OK
```

```
Applying taggit.0003_taggeditem_add_unique_index... OK
Applying taggit.0004_alter_taggeditem_content_type_alter_taggeditem_tag... OK
Applying taggit.0005_auto_20220424_2025... OK
Applying blog.0004_post_tags... OK
```

Теперь база данных синхронизирована с моделями `taggit`, и можно начать использовать функциональности приложения `django-taggit`.

Давайте сейчас проинспектируем способы применения менеджера `tags`.

Откройте оболочку Django, выполнив следующую ниже команду в командной строке системной оболочки:

```
python manage.py shell
```

Выполните следующий ниже исходный код, чтобы получить один из постов (пост с ИД 1):

```
>>> from blog.models import Post
>>> post = Post.objects.get(id=1)
```

Затем добавьте в него несколько тегов и извлеките его теги, чтобы проверить успешность их добавления:

```
>>> post.tags.add('music', 'jazz', 'django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>, <Tag: django>]>
```

Наконец, удалите тег и еще раз проверьте список тегов:

```
>>> post.tags.remove('django')
>>> post.tags.all()
<QuerySet [<Tag: jazz>, <Tag: music>]>
```

Добавлять, извлекать или удалять теги из модели с помощью определенного нами менеджера действительно легко.

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/taggit/tag/` в своем браузере.

Вы увидите страницу администрирования со списком объектов Tag приложения taggit:

The screenshot shows a Django admin page titled "Select tag to change". At the top right is a button labeled "ADD TAG +". Below it is a search bar with a magnifying glass icon and a "Search" button. Underneath is a table header with columns "NAME" and "SLUG". The table contains three rows of data:

NAME	SLUG
django	django
jazz	jazz
music	music

Below the table, the text "3 tags" is displayed. At the bottom left of the page is a link "3 tags".

Рис. 3.2. Представление списка с изменениями тегов на сайте администрирования

Кликните по тегу jazz. Вы увидите следующее:

The screenshot shows a Django admin page titled "Change tag" for the tag "jazz". At the top right is a button labeled "HISTORY". The page has two input fields: "Name" (containing "jazz") and "Slug" (containing "jazz"). Below these is a section titled "TAGGED ITEMS" with a table header "Tagged item". A single row is shown: "Who was Django Reinhardt? tagged with jazz". To the right of this row is a checkbox labeled "Delete". Further down are two dropdown menus: "Content type" (set to "blog | post") and "Object ID" (set to "1").

Рис. 3.3. Поле схожих тегов объекта Post

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/blog/post/1/change/>, чтобы отредактировать пост с ИД 1.

Вы увидите, что теперь в посты вставлено новое поле **Tags**, как показано ниже, в котором можно легко редактировать теги:

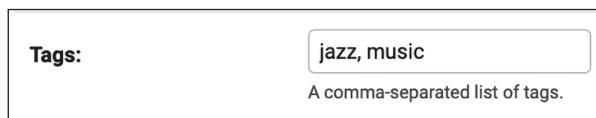


Рис. 3.4. Поле схожих тегов объекта Post

Сейчас необходимо отредактировать свои посты в блоге, чтобы отобразить теги.

Откройте шаблон `blog/post/list.html` и добавьте в него следующий ниже исходный код HTML, выделенный жирным шрифтом:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
<h1>My Blog</h1>
{% for post in posts %}
<h2>
<a href="{{ post.get_absolute_url }}>
{{ post.title }}
</a>
</h2>
<p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
<p class="date">
Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|truncatewords:30|linebreaks }}
{% endfor %}
{% include "pagination.html" with page=page_obj %}
{% endblock %}
```

Шаблонный фильтр `join` работает так же, как метод Python `string.join()`, чтобы конкатенировать элементы с заданной строкой.

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере. Под заголовком каждого поста вы должны увидеть список тегов:

Who was Django Reinhardt?

Tags: music, jazz

Published Jan. 1, 2022, 11:59 p.m. by admin

Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...

Рис. 3.5. Элемент списка постов, включающий схожие теги

Далее мы отредактируем представление `post_list`, чтобы пользователи имели возможность отображать список всех постов, помеченных конкретным тегом.

Откройте `views.py` файл приложения `blog`, импортируйте модель `Tag` из приложения `django-taggit` и измените представление `post_list`, как показано ниже, чтобы при необходимости фильтровать посты по тегу. Новый исходный код выделен жирным шрифтом:

```
from taggit.models import Tag

def post_list(request, tag_slug=None):
    post_list = Post.published.all()
    tag = None
    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        post_list = post_list.filter(tags__in=[tag])
    # Постраничная разбивка с 3 постами на страницу
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    try:
        posts = paginator.page(page_number)
    except PageNotAnInteger:
        # Если page_number не целое число, то
        # выдать первую страницу
        posts = paginator.page(1)
    except EmptyPage:
        # Если page_number находится вне диапазона, то
        # выдать последнюю страницу результатов
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                  {'posts': posts,
                   'tag': tag})
```

Теперь представление `post_list` работает следующим образом.

- Представление принимает опциональный параметр `tag_slug`, значение которого по умолчанию равно `None`. Этот параметр будет передан в URL-адресе.
- Внутри указанного представления формируется изначальный набор запросов, извлекающий все опубликованные посты, и если имеется слаг данного тега, то берется объект `Tag` с данным слагом, используя функцию сокращенного доступа `get_object_or_404()`.
- Затем список постов фильтруется по постам, которые содержат данный тег. Поскольку здесь используется взаимосвязь многие-ко-многим, необходимо фильтровать записи по тегам, содержащимся в заданном списке, который в данном случае содержит только один элемент. Здесь используется операция `__in` поиска по полю. Взаимосвязи многие-ко-многим возникают, когда несколько объектов модели ассоциированы с несколькими объектами другой модели. В нашем приложении пост может иметь несколько тегов, и тег может быть связан с несколькими постами. О методах создания взаимосвязи многие-ко-многим вы узнаете в главе 6 «Распространение контента на веб-сайте». Подробнее о взаимосвязях многие-ко-многим можно узнать на странице https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/.
- Наконец, теперь функция `render()` передает новую переменную `tag` в шаблон.

Напомним, что итерируемые наборы запросов `QuerySet` являются ленивыми. Наборы запросов, служащие для извлечения постов, будут оцениваться только при прокручивании списка постов в цикле во время прорисовки шаблона.

Откройте файл `urls.py` приложения `blog`, закомментируйте основанный на классе шаблон URL-адреса `PostListView` и раскомментируйте представление `post_list`, как показано ниже:

```
path('', views.post_list, name='post_list'),
# path('', views.PostListView.as_view(), name='post_list'),
```

Добавьте следующий ниже дополнительный шаблон URL-адреса, чтобы отображать список постов по тегу:

```
path('tag/<slug:tag_slug>', 
      views.post_list, name='post_list_by_tag'),
```

Как вы видите, оба шаблона указывают на одно и то же представление, но у них разные имена. Первый шаблон будет вызывать представление `post_list` без каких-либо опциональных параметров, тогда как второй шаблон будет вызывать это представление с параметром `tag_slug`. Конвертер путей `slug` используется для сочетания с параметром, представленным строковым литералом в нижнем регистре с буквами или цифрами ASCII, а также символами дефиса и подчеркивания.

Теперь файл urls.py приложения blog должен выглядеть следующим образом:

```
from django.urls import path
from . import views

app_name = 'blog'

urlpatterns = [
    # представления поста
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
    path('tag/<slug:tag_slug>/',
        views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>',
        views.post_detail,
        name='post_detail'),
    path('<int:post_id>/share/',
        views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
        views.post_comment, name='post_comment'),
]
```

Поскольку используется представление post_list, отредактируйте шаблон blog/post/list.html, видоизменив постраничную разбивку, чтобы использовать объект posts:

```
{% include "pagination.html" with page=posts %}
```

Добавьте в шаблон blog/post/list.html следующие ниже строки, выделенные жирным шрифтом:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
<h1>My Blog</h1>
{% if tag %}
<h2>Posts tagged with "{{ tag.name }}"</h2>
{% endif %}
{% for post in posts %}
<h2>
<a href="{{ post.get_absolute_url }}>
{{ post.title }}
</a>
```

```

</h2>
<p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
<p class="date">
    Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|truncatewords:30|linebreaks }}
{% endfor %}
{% include "pagination.html" with page=posts %}
{% endblock %}

```

Если пользователь зайдет в блог, то он увидит список всех постов. Если он будет фильтровать по постам, помеченным конкретным тегом, то увидит тег, по которому он фильтрует.

Теперь отредактируйте шаблон `blog/post/list.html`, изменив вид отображения тегов, как показано ниже. Новые строки выделены жирным шрифтом:

```

{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
    <h1>My Blog</h1>
    {% if tag %}
        <h2>Posts tagged with "{{ tag.name }}"</h2>
    {% endif %}
    {% for post in posts %}
        <h2>
            <a href="{{ post.get_absolute_url }}">
                {{ post.title }}
            </a>
        </h2>
        <p class="tags">
            Tags:
            {% for tag in post.tags.all %}
                <a href="{% url "blog:post_list_by_tag" tag.slug %}">
                    {{ tag.name }}
                </a>
                {% if not forloop.last %}, {% endif %}
            {% endfor %}
        </p>
        <p class="date">
            Published {{ post.publish }} by {{ post.author }}
        </p>
        {{ post.body|truncatewords:30|linebreaks }}
    {% endfor %}
    {% include "pagination.html" with page=posts %}
{% endblock %}

```

В приведенном выше исходном коде прокручиваются в цикле все теги поста, отображающие конкретно-прикладную ссылку на URL-адрес, чтобы фильтровать посты по этому тегу. URL-адрес формируется с помощью тега `{% url "blog:post_list_by_tag" tag.slug %}`, используя имя URL-адреса и тег `slug` в качестве его параметра. Теги отделяются запятыми.

Пройдите по URL-адресу `http://127.0.0.1:8000/blog/tag/jazz/` в своем браузере, и вы увидите список постов, отфильтрованных по этому тегу, примерно как показано ниже:

The screenshot shows a blog interface titled 'My Blog'. On the right, there's a sidebar with the title 'My blog' and the subtitle 'This is my blog.' Below the sidebar, the main content area displays a post titled 'Posts tagged with "jazz"'. Underneath the title, it says 'Who was Django Reinhardt?'. The post has a 'Tags: music , jazz' section. Below that, the text 'Published Jan. 1, 2022, 11:59 p.m. by admin' is shown. The main content of the post is: 'Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...'. At the bottom of the content area, it says 'Page 1 of 1.'

Рис. 3.6. Пост, отфильтрованный по тегу «jazz»

Извлечение постов по сходству

Теперь, когда мы реализовали тегирование постов блога, с помощью тегов можно делать много интересных вещей. Теги позволяют классифицировать посты неиерархическим образом. Посты на схожие темы будут иметь несколько общих тегов. Мы разработаем функциональность отображения схожих постов по числу имеющихся у них общих тегов. При таком подходе при чтении пользователем поста ему можно будет предлагать почитать другие родственные посты.

Для того чтобы получить посты, схожие с конкретным постом, необходимо выполнить следующие действия:

- 1) извлечь все теги текущего поста;
- 2) получить все посты, помеченные любым из этих тегов;
- 3) исключить текущий пост из этого списка, чтобы не рекомендовать тот же самый пост;
- 4) упорядочить результаты по числу общих тегов, которое есть у текущего поста;

- 5) в случае двух или более постов с одинаковым числом тегов рекомендовать самый последний пост;
- 6) ограничить запрос числом постов, которые вы хотите рекомендовать.

Эти шаги транслируются в сложный набор запросов QuerySet, который вставляется в представление post_detail.

Откройте файл views.py приложения blog и добавьте следующую ниже инструкцию импорта в верхнюю его часть:

```
from django.db.models import Count
```

Это функция агрегирования Count из Django ORM-преобразователя. Данная функция позволит выполнять агрегированный подсчет тегов. Модуль django.db.models содержит следующие ниже функции агрегирования:

- Avg: среднее значение;
- Max: максимальное значение;
- Min: минимальное значение;
- Count: общее количество объектов.

Об агрегировании можно узнать на странице <https://docs.djangoproject.com/en/4.1/topics/db/aggregation/>.

Откройте файл views.py приложения blog и добавьте следующие ниже строки в представление post_detail. Новые строки выделены жирным шрифтом:

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                           status=Post.Status.PUBLISHED,
                           slug=post,
                           publish__year=year,
                           publish__month=month,
                           publish__day=day)
    # Список активных комментариев к этому посту
    comments = post.comments.filter(active=True)

    # Форма для комментариев пользователей
    form = CommentForm()

    # Список схожих постов
    post_tags_ids = post.tags.values_list('id', flat=True)
    similar_posts = Post.published.filter(tags__in=post_tags_ids)\n        .exclude(id=post.id)
    similar_posts = similar_posts.annotate(same_tags=Count('tags'))\n        .order_by('-same_tags', '-publish')[:4]

    return render(request,
                  'blog/post/detail.html',
                  {'post': post,
```

```
'comments': comments,
'form': form,
'similar_posts': similar_posts})
```

Приведенный выше исходный код делает следующее:

- 1) извлекается Python'овский список идентификаторов тегов текущего поста. Набор запросов `QuerySet values_list()` возвращает кортежи со значениями заданных полей. Ему передается параметр `flat=True`, чтобы получить одиночные значения, такие как `[1, 2, 3, ...]`, а не одноэлементные кортежи, такие как `[(1,), (2,), (3,) ...]`;
- 2) берутся все посты, содержащие любой из этих тегов, за исключением текущего поста;
- 3) применяется функция агрегирования `Count`. Ее работа – генерировать вычисляемое поле – `same_tags`, – которое содержит число тегов, общих со всеми запрошенными тегами;
- 4) результат упорядочивается по числу общих тегов (в убывающем порядке) и по `publish`, чтобы сначала отображать последние посты для постов с одинаковым числом общих тегов. Результат нарезается, чтобы получить только первые четыре поста;
- 5) объект `similar_posts` передается в контекстный словарь для функции `render()`.

Теперь отредактируйте шаблон `blog/post/detail.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
<h1>{{ post.title }}</h1>
<p class="date">
    Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|linebreaks }}
<p>
    <a href="{% url "blog:post_share" post.id %}">
        Share this post
    </a>
</p>

<h2>Similar posts</h2>
{% for post in similar_posts %}
<p>
    <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
</p>
{% empty %}
```

```

There are no similar posts yet.
{% endfor %}

{% with comments.count as total_comments %}
<h2>
{{ total_comments }} comment{{ total_comments|pluralize }}
</h2>
{% endwith %}
{% for comment in comments %}
<div class="comment">
<p class="info">
Comment {{ forloop.counter }} by {{ comment.name }}
{{ comment.created }}
</p>
{{ comment.body|linebreaks }}
</div>
{% empty %}
<p>There are no comments yet.</p>
{% endfor %}
{% include "blog/post/includes/comment_form.html" %}
{% endblock %}

```

Страница детальной информации о посте должна выглядеть, как показано ниже:

Who was Django Reinhardt?

Published Jan. 1, 2022, 11:59 p.m. by admin

Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and remains the most significant.

[Share this post](#)

Similar posts

There are no similar posts yet.

Рис. 3.7. Страница детальной информации о посте, включающая список схожих постов

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/blog/post/> в своем браузере, отредактируйте пост, у которого нет тегов, добавив теги `music` и `jazz`, как показано ниже:

Who was Miles Davis?

Title: Who was Miles Davis?

Slug: who-was-miles-davis

Author: 1 Q admin

Body: Miles Davis was an American trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.

Publish:

Date: 2022-01-02 Today |

Time: 13:18:11 Now |

Note: You are 2 hours ahead of server time.

Status: Published

Tags: jazz, music

A comma-separated list of tags.

Рис. 3.8. Добавление тегов «jazz» и «music» в пост

Отредактируйте еще один пост, добавив тег jazz, как показано ниже:

Notes on Duke Ellington

Title: Notes on Duke Ellington

Slug: notes-on-duke-ellington

Author: 1 Q admin

Body: Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half a century.

Publish:

Date: 2022-01-03 Today |

Time: 13:19:33 Now |

Note: You are 2 hours ahead of server time.

Status: Published

Tags: jazz

A comma-separated list of tags.

Рис. 3.9. Добавление тега «jazz» в пост

Теперь страница детальной информации о первом посте должна выглядеть, как показано ниже:

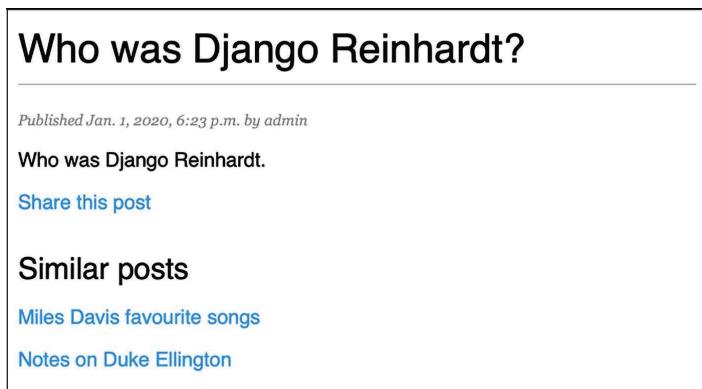


Рис. 3.10. Страница детальной информации о посте, включающая список схожих постов

Посты, рекомендованные в разделе **Similar posts** (Схожие посты), появляются в убывающем порядке в зависимости от числа общих тегов с изначальным постом.

Теперь появилась возможность успешно рекомендовать читателям схожие посты. Приложение django-taggit также содержит менеджер `similar_objects()`, который можно использовать для извлечения объектов на основе общих тегов. Со всеми менеджерами приложения django-taggit можно ознакомиться на странице <https://django-taggit.readthedocs.io/en/latest/api.html>.

Кроме того, список тегов можно добавить и в шаблон детальной информации о посте, точно таким же образом, как это было сделано в шаблоне `blog/post/list.html`.

Создание конкретно-прикладных шаблонных тегов и фильтров

Django предлагает разнообразный набор встроенных шаблонных тегов, таких как `{% if %}` или `{% block %}`. В главе 1 «Разработка приложения для ведения блога» и главе 2 «Усовершенствование блога за счет продвинутых функциональностей» использовались разные шаблонные теги. Полный справочный материал по встроенным шаблонным тегам и фильтрам находится на странице <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/>.

Django также позволяет создавать свои собственные шаблонные теги для выполнения действий, адаптированных под конкретно-прикладные зада-

чи. Конкретно-прикладные шаблонные теги бывают весьма кстати, когда появляется необходимость добавлять в свои шаблоны функциональность, которая не охватывается ключевым набором шаблонных тегов Django. Это может быть тег для исполнения набора запросов QuerySet или любой обработки на стороне сервера, которую вы хотите реиспользовать в шаблонах. Например, можно было бы разработать шаблонный тег для отображения списка последних постов, опубликованных в блоге. Этот список можно было бы вставлять в боковую панель, чтобы он всегда был виден, независимо от обрабатывающего запрос представления.

Реализация конкретно-прикладных шаблонных тегов

Django предоставляет следующие вспомогательные функции, которые позволяют легко создавать шаблонные теги:

- `simple_tag`: обрабатывает предоставленные данные и возвращает строковый литерал;
- `inclusion_tag`: обрабатывает предоставленные данные и возвращает прорисованный шаблон.

Шаблонные теги должны находиться внутри приложений Django.

Внутри каталога приложения `blog` создайте новый каталог, назовите его `templatetags` и добавьте в него пустой файл `__init__.py`. Создайте еще один файл в той же папке и назовите его `blog_tags.py`. Файловая структура приложения `blog` должна выглядеть, как показано ниже:

```
blog/
    __init__.py
    models.py
    ...
    templatetags/
        __init__.py
        blog_tags.py
```

Очень важно то, как файл называется. Вы будете использовать имя этого модуля для загрузки тегов в шаблоны.

Создание простого шаблонного тега

Давайте начнем с создания простого тега, чтобы получать общее число опубликованных в блоге постов.

Отредактируйте только что созданный вами файл `templatetags/blog_tags.py`, добавив следующий ниже исходный код:

```
from django import template
from ..models import Post

register = template.Library()

@register.simple_tag
def total_posts():
    return Post.published.count()
```

Мы создали простой шаблонный тег, который возвращает число опубликованных в блоге постов.

Для того чтобы быть допустимой библиотекой тегов, в каждом содержащем шаблонные теги модуле должна быть определена переменная с именем `register`. Эта переменная является экземпляром класса `template.Library`, и она используется для регистрации шаблонных тегов и фильтров приложения.

В приведенном выше исходном коде тег `total_posts` был определен с помощью простой функции Python. В функцию был добавлен декоратор `@register.simple_tag`, чтобы зарегистрировать ее как простой тег. Django будет использовать имя функции в качестве имени тега. Если есть потребность зарегистрировать ее под другим именем, то это можно сделать, указав атрибут `name`, например `@register.simple_tag(name='my_tag')`.



После добавления нового модуля шаблонных тегов потребуется перезапустить сервер разработки, чтобы новые теги и фильтры стали доступны для использования в шаблонах.

Прежде чем использовать конкретно-прикладные шаблонные теги, необходимо сделать их доступными для шаблона с помощью тега `{% load %}`. Как упоминалось ранее, нужно использовать имя модуля Python, содержащего ваши шаблонные теги и фильтры.

Отредактируйте шаблон `blog/templates/base.html`, добавив тег `{% load blog_tags %}` в верхней его части, чтобы загрузить свой модуль шаблонных тегов. Затем примените созданный вами тег для отображения общего числа постов, как показано ниже. Новые строки выделены жирным шрифтом:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
    <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
    <div id="content">
```

```
{% block content %}  
{% endblock %}  
</div>  
<div id="sidebar">  
    <h2>My blog</h2>  
    <p>  
        This is my blog.  
        I've written {% total_posts %} posts so far.  
    </p>  
</div>  
</body>  
</html>
```

Для того чтобы отслеживать добавленные в проект новые файлы, нужно перезапустить сервер. Остановите сервер разработки с помощью **Ctrl+C** и запустите его снова, используя следующую ниже команду:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/blog/> в своем браузере. На боковой панели вы должны увидеть общее число постов сайта, как показано ниже:

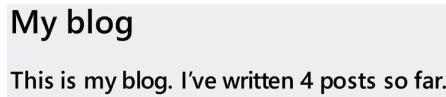


Рис. 3.11. Общее число опубликованных постов, вставленных в боковую панель

Если вы увидите следующее ниже сообщение об ошибке, то весьма вероятно, что вы не перезапустили сервер разработки.

```
TemplateSyntaxError at /blog/2022/1/1/who-was-django-reinhardt/  
'blog_tags' is not a registered tag library. Must be one of:  
admin_list  
admin_modify  
admin_urls  
cache  
i18n  
l10n  
log  
static  
tz
```

Рис. 3.12. Сообщение об ошибке, когда библиотека шаблонных тегов не зарегистрирована

Шаблонные теги позволяют обрабатывать любые данные и добавлять их в любой шаблон независимо от исполняемого представления. При этом для отображения результатов в своих шаблонах можно выполнять наборы запросов или обрабатывать любые данные.

Создание шаблонного тега включения

Мы создадим еще один тег, чтобы отображать последние посты на боковой панели блога. На этот раз мы реализуем тег включения. Используя тег включения, можно отображать шаблон с контекстными переменными, возвращаемыми вашим шаблонным тегом.

Отредактируйте файл `templatetags/blog_tags.py`, добавив следующий ниже исходный код:

```
@register.inclusion_tag('blog/post/latest_posts.html')
def show_latest_posts(count=5):
    latest_posts = Post.published.order_by('-publish')[count]
    return {'latest_posts': latest_posts}
```

В приведенном выше исходном коде мы зарегистрировали шаблонный тег, применяя декоратор `@register.inclusion_tag`. Используя `blog/post/latest_posts.html`, был указан шаблон, который будет прорисовываться возвращенными значениями. Шаблонный тег будет принимать optionalный параметр `count`, который по умолчанию равен 5. Этот параметр позволяет задавать число отображаемых постов. Данная переменная используется для того, чтобы ограничивать результаты запроса `Post.published.order_by('-publish')[count]`.

Обратите внимание, что приведенная выше функция возвращает не простое значение, а словарь переменных. Теги включения должны возвращать словарь значений, который используется в качестве контекста для прорисовки заданного шаблона. Только созданный шаблонный тег позволяет задавать optionalное число отображаемых постов как `{% show_latest_posts 3 %}`.

Теперь создайте новый файл шаблона в разделе `blog/post/` и назовите его `latest_posts.html`.

Отредактируйте новый шаблон `blog/post/latest_posts.html`, добавив следующий ниже исходный код:

```
<ul>
    {% for post in latest_posts %}
        <li>
            <a href="{{ post.get_absolute_url }}>{{ post.title }}</a>
        </li>
    {% endfor %}
</ul>
```

В приведенном выше исходном коде отображается неупорядоченный список постов, используя возвращаемую вашим шаблонным тегом переменную `latest_posts`. Теперь отредактируйте шаблон `blog/base.html`, добавив новый шаблонный тег, чтобы отображать последние три поста, как показано ниже. Новые строки выделены жирным шрифтом:

```
{% load blog_tags %}  
{% load static %}  
<!DOCTYPE html>  
<html>  
<head>  
    <title>{% block title %}{% endblock %}</title>  
    <link href="{% static "css/blog.css" %}" rel="stylesheet">  
</head>  
<body>  
    <div id="content">  
        {% block content %}  
        {% endblock %}  
    </div>  
    <div id="sidebar">  
        <h2>My blog</h2>  
        <p>  
            This is my blog.  
            I've written {% total_posts %} posts so far.  
        </p>  
        <h3>Latest posts</h3>  
        {% show_latest_posts 3 %}  
    </div>  
</body>  
</html>
```

Здесь вызывается шаблонный тег, передающий число отображаемых постов, и шаблон прорисовывается прямо на месте с заданным контекстом.

Затем вернитесь в свой браузер и обновите страницу. Теперь боковая панель должна выглядеть следующим образом:



Рис. 3.13. Боковая панель блога,
включая последние опубликованные посты

Создание шаблонного тега, возвращающего набор запросов

Наконец, мы создадим простой шаблонный тег, который возвращает значение. Мы сохраним результат в реиспользуемой переменной, не выводя его напрямую. Мы создадим тег, чтобы отображать посты с наибольшим числом комментариев.

Отредактируйте файл `templatetags/blog_tags.py`, добавив следующую ниже инструкцию импорта и шаблонный тег:

```
from django.db.models import Count

@register.simple_tag
def get_most_commented_posts(count=5):
    return Post.published.annotate(
        total_comments=Count('comments')
    ).order_by('-total_comments')[:count]
```

В приведенном выше шаблонном теге с помощью функции `annotate()` формируется набор запросов `QuerySet`, чтобы агрегировать общее число комментариев к каждому посту. Функция агрегирования `Count` используется для сохранения количества комментариев в вычисляемом поле `total_comments` по каждому объекту `Post`. Набор запросов `QuerySet` упорядочивается по вычисляемому полю в убывающем порядке. Также предоставляется опциональная переменная `count`, чтобы ограничивать общее число возвращаемых объектов.

В дополнение к `Count` Django предлагает функции агрегирования `Avg`, `Max`, `Min` и `Sum`. Подробнее о функциях агрегирования можно почитать на странице <https://docs.djangoproject.com/en/4.1/topics/db/aggregation/>.

Далее отредактируйте шаблон `blog/base.html`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
    <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
    <div id="content">
        {% block content %}
        {% endblock %}
    </div>
    <div id="sidebar">
        <h2>My blog</h2>
        <p>
            This is my blog.
        </p>
```

```
I've written {% total_posts %} posts so far.  
</p>  
<h3>Latest posts</h3>  
{% show_latest_posts 3 %}  
<h3>Most commented posts</h3>  
{% get_most_commented_posts as most_commented_posts %}  
<ul>  
    {% for post in most_commented_posts %}  
        <li>  
            <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>  
        </li>  
    {% endfor %}  
</ul>  
</div>  
</body>  
</html>
```

В приведенном выше исходном коде результат сохраняется в конкретно-прикладной переменной, используя аргумент `as`, за которым следует имя переменной. В качестве шаблонного тега используется `{% get_most_commented_posts as most_commented_posts %}`, чтобы сохранить результат шаблонного тега в новой переменной с именем `most_commented_posts`. Затем возвращенные посты отображаются, используя HTML-элемент в виде неупорядоченного списка.

Теперь откройте свой браузер и обновите страницу, чтобы увидеть итоговый результат. Он должен выглядеть следующим образом:

The screenshot shows a blog interface with a main content area and a sidebar.

Main Content Area:

- My Blog** (Section header)
- Notes on Duke Ellington** (Post title)
 - Tags: jazz
 - Published Jan. 3, 2022, 1:19 p.m. by admin
 - Text: Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...
- Who was Miles Davis?** (Post title)
 - Tags: music , jazz
 - Published Jan. 2, 2022, 1:18 p.m. by admin
 - Text: Miles Davis was an American trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.
- Who was Django Reinhardt?** (Post title)
 - Tags: music , jazz
 - Published Jan. 1, 2022, 11:59 p.m. by admin
 - Text: Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...

Page 1 of 2. Next (Navigation link at the bottom of the list).

Sidebar:

- My blog**
This is my blog. I've written 4 posts so far.
- Latest posts**
 - Notes on Duke Ellington
 - Who was Miles Davis?
 - Who was Django Reinhardt?
- Most commented posts**
 - Notes on Duke Ellington
 - Who was Django Reinhardt?
 - Another post
 - Who was Miles Davis?

Рис. 3.14. Представление списка постов, включая полную боковую панель с последними и наиболее прокомментированными постами

Теперь у вас есть четкое понимание того, как разрабатывать конкретно-прикладные шаблонные теги. Подробнее о них можно почитать на странице <https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/>.

Реализация конкретно-прикладных шаблонных фильтров

Django имеет целый ряд встроенных шаблонных фильтров, которые позволяют изменять переменные в шаблонах. Это функции Python, которые принимают один или два параметра, значение переменной, к которой применяется фильтр, и optionalный аргумент. Они возвращают значение, которое может быть отображено или обработано еще одним фильтром.

Фильтр записывается как {{ variable|my_filter }}. Фильтры с аргументом записываются как {{ variable|my_filter:"foo" }}. Например, фильтр capfirst можно использовать для приведения первого символа значения в верхний регистр, например {{ value|capfirst }}. Если значение равно django, то на выходе будет Django. К переменной можно применять столько фильтров, сколько потребуется, например {{ variable|filter1|filter2 }}, и каждый фильтр будет применен к результату, сгенерированному предыдущим фильтром.

Со списком встроенных шаблонных фильтров можно ознакомиться по адресу <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#built-in-filter-reference>.

Создание шаблонного фильтра для поддержки синтаксиса Markdown

Мы создадим конкретно-прикладной фильтр, который позволит использовать синтаксис упрощенной разметки Markdown в постах блога, а затем в шаблонах конвертировать текст поста в HTML.

Markdown – это синтаксис форматирования обычного текста, который очень прост в использовании и предназначен для конвертирования в HTML. Посты можно писать, используя простой синтаксис Markdown, и получать автоматическую конвертацию контента в исходный код HTML. Изучить синтаксис Markdown намного проще, чем изучить HTML. Используя Markdown, можно сделать так, чтобы другие не сведущие в технологиях участники легко писали посты для вашего блога. С основами формата Markdown можно ознакомиться на странице <https://daringfireball.net/projects/markdown/basics>.

Сначала установите модуль Python markdown посредством pip, используя следующую ниже команду в командной оболочке:

```
pip install markdown==3.4.1
```

Затем отредактируйте файл `templatetags/blog_tags.py`, вставив следующий ниже исходный код:

```
from django.utils.safestring import mark_safe
import markdown

@register.filter(name='markdown')
def markdown_format(text):
    return mark_safe(markdown.markdown(text))
```

Шаблонные фильтры регистрируются таким же образом, как и шаблонные теги. Во избежание конфликта имен между именем функции и модулем `markdown` мы дали функции имя `markdown_format`, а фильтру – имя `markdown` для использования в шаблонах, в частности `{{ variable|markdown }}`.

Django экранирует генерируемый фильтрами исходный код HTML; символы HTML-сущностей заменяются их кодированными в HTML символами. Например, `<p>` преобразовывается в `&ltp&gt` (символ *меньше*, символ *p*, символ *больше*).

Мы используем предоставляемую веб-фреймворком Django функцию `mark_safe`, чтобы помечать результат как безопасный для прорисовки в шаблоне исходный код HTML. По умолчанию Django не будет доверять никакому исходному коду HTML и будет экранировать его перед его вставкой в результат. Единственными исключениями являются переменные, которые помечены как безопасные, чтобы тем самым избежать экранирования. Такое поведение не дает Django выводить потенциально опасный исходный код HTML и позволяет создавать исключения, дабы возвращать безопасный исходный код HTML.

Отредактируйте шаблон `blog/post/detail.html`, добавив следующий ниже новый исходный код, выделенный жирным шрифтом:

```
{% extends "blog/base.html" %}
{% load blog_tags %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
<h1>{{ post.title }}</h1>
<p class="date">
    Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|markdown }}
<p>
    <a href="{% url "blog:post_share" post.id %}">
        Share this post
    </a>
</p>
<h2>Similar posts</h2>
{% for post in similar_posts %}
    <p>
```

```

<a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
</p>
{% empty %}
    There are no similar posts yet.
{% endfor %}

{% with comments.count as total_comments %}
    <h2>
        {{ total_comments }} comment{{ total_comments|pluralize }}
    </h2>
{% endwith %}
{% for comment in comments %}
    <div class="comment">
        <p class="info">
            Comment {{ forloop.counter }} by {{ comment.name }}
            {{ comment.created }}
        </p>
        {{ comment.body|linebreaks }}
    </div>
{% empty %}
    <p>There are no comments yet.</p>
{% endfor %}

{% include "blog/post/includes/comment_form.html" %}
{% endblock %}

```

Фильтр `linebreaks` шаблонной переменной `{{ post.body }}` был заменен фильтром `markdown`. Данный фильтр не только преобразовывает разрывы строк в теги `<p>`, но и конвертирует форматирование Markdown в HTML.

Отредактируйте шаблон `blog/post/list.html`, добавив следующий ниже новый исходный код, выделенный жирным шрифтом:

```

{% extends "blog/base.html" %}
{% load blog_tags %}

{% block title %}My Blog{% endblock %}

{% block content %}
    <h1>My Blog</h1>
    {% if tag %}
        <h2>Posts tagged with "{{ tag.name }}"</h2>
    {% endif %}
    {% for post in posts %}

```

```
<h2>
    <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
    </a>
</h2>
<p class="tags">
    Tags:
    {% for tag in post.tags.all %}
        <a href="{% url "blog:post_list_by_tag" tag.slug %}">
            {{ tag.name }}
        </a>
        {% if not forloop.last %}, {% endif %}
    {% endfor %}
</p>
<p class="date">
    Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|markdown|truncatewords_html:30 }}
{% endfor %}
{% include "pagination.html" with page=posts %}
{% endblock }
```

В шаблонную переменную {{ post.body }} был добавлен новый фильтр `markdown`. Этот фильтр преобразовывает контент в формате Markdown в формат HTML. Поэтому мы заменили приведенный выше фильтр `truncatewords` фильтром `truncatewords_html`. Данный фильтр усекает строку после определенного числа слов, избегая незакрытых HTML-тегов.

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/admin/blog/post/add/` в своем браузере и создайте новый пост со следующим ниже текстом:

```
This is a post formatted with markdown
-----
*This is emphasized* and **this is more emphasized**.
```

Here is a list:

- * One
- * Two
- * Three

And a [link to the Django website](<https://www.djangoproject.com/>).

Форма должна выглядеть следующим образом:

Add post

Title: Markdown post

Slug: markdown-post

Author: 1

Body:

This is a post formatted with markdown

This is emphasized and **this is more emphasized**.

Here is a list:

- * One
- * Two
- * Three

And a [link to the Django website](https://www.djangoproject.com/).

Publish:

Date: 2022-01-22 Today |

Time: 09:30:04 Now |

Note: You are 2 hours ahead of server time.

Status: Draft

Tags: markdown

A comma-separated list of tags.

Рис. 3.15. Пост с контентом в формате Markdown, прорисовываемом в формат HTML

Пройдите по URL-адресу <http://127.0.0.1:8000/blog/> в своем браузере и посмотрите, как прорисовывается новый пост. Вы должны увидеть следующий ниже результат:

The screenshot shows a blog post titled "My Blog". The title is in a large, bold, black font at the top. Below it is a horizontal line. The main content of the post is titled "Markdown post" in blue text. Underneath this, the word "Tags:" is followed by "markdown" in blue. A small gray text below indicates the post was "Published Jan. 22, 2022, 9:30 a.m. by admin". The main body of the post contains the text "This is a post formatted with markdown" in bold black font. Below this, there is italicized text "This is emphasized and this is more emphasized." and a list of three items: "• One", "• Two", and "• Three". At the bottom, there is a link "And a link to the Django website ...".

Рис. 3.16. Пост с контентом в формате Markdown, прорисованном как HTML

На рис. 3.16 хорошо видно, что конкретно-прикладные шаблонные фильтры очень удобны для адаптации форматирования под конкретно-прикладную задачу. Более подробная информация о конкретно-прикладных фильтрах находится по адресу <https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/#writing-custom-template-filters>.

Добавление карты сайта

Django идет в комплекте с фреймворком карт сайтов, который позволяет динамически создавать карты для своего сайта. Карта сайта – это XML-файл, который сообщает поисковым системам о страницах веб-сайта, их релевантность и частоту их обновления. Использование карты сделает сайт более заметным в рейтинге поисковых систем, поскольку она помогает поисковым роботам индексировать содержимое сайта.

Фреймворк карт сайтов Django зависит от приложения `django.contrib.sites`, которое позволяет ассоциировать объекты с теми или иными веб-сайтами, работающими вместе с вашим проектом. Это удобно, когда вы хотите управлять несколькими сайтами, используя один проект Django. Для установки фреймворка карт сайтов необходимо активировать приложения `sites` и `sitemaps` в своем проекте.

Отредактируйте файл `settings.py` проекта, добавив `django.contrib.sites` и `django.contrib.sitemaps` в настроочный параметр `INSTALLED_APPS`. Кроме того, определите новый настроочный параметр для ИД сайта, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
# ...  
  
SITE_ID = 1  
  
# Определение приложения  
  
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'blog.apps.BlogConfig',  
    'taggit',  
    ''django.contrib.sites'',  
    ''django.contrib.sitemaps'',  
]  
]
```

Теперь запустите следующую ниже команду из командной оболочки, чтобы создать таблицы приложения `Django sites` в базе данных:

```
python manage.py migrate
```

Вы должны увидеть результат, содержащий такие строки:

```
Applying sites.0001_initial... OK  
Applying sites.0002_alter_domain_unique... OK
```

Теперь приложение `sites` синхронизировано с базой данных.

Затем внутри каталога своего приложения `blog` создайте новый файл и назовите его `sitemaps.py`. Откройте файл и добавьте в него следующий ниже исходный код:

```
from django.contrib.sitemaps import Sitemap  
from .models import Post  
  
class PostSitemap(Sitemap):  
    changefreq = 'weekly'  
    priority = 0.9  
  
    def items(self):
```

```
    return Post.published.all()

    def lastmod(self, obj):
        return obj.updated
```

Мы определили конкретно-прикладную карту сайта, унаследовав класс `Sitemap` модуля `sitemaps`. Атрибуты `changefreq` и `priority` указывают частоту изменения страниц постов и их релевантность на веб-сайте (максимальное значение равно 1).

Метод `items()` возвращает набор запросов `QuerySet` объектов, подлежащих включению в эту карту сайта. По умолчанию Django вызывает метод `get_absolute_url()` по каждому объекту, чтобы получить его URL-адрес. Напомним, что мы применили этот метод в главе 2 «Усовершенствование блога за счет продвинутых функциональностей», чтобы формировать канонический URL-адрес постов. Если нужно указать URL-адрес каждого объекта, то в класс `sitemap` можно добавить метод `location`.

Метод `lastmod` получает каждый возвращаемый методом `items()` объект и возвращает время последнего изменения объекта.

Атрибуты `changefreq` и `priority` могут быть либо методами, либо атрибутами. С полным справочным материалом по картам сайтов можно ознакомиться в официальной документации Django на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/sitemaps/>.

Мы создали карту сайта. Теперь необходимо создать для него URL-адрес.

Отредактируйте главный файл `urls.py` проекта `mysite`, добавив карту сайта, как показано ниже. Новые строки выделены жирным шрифтом:

```
from django.urls import path, include
from django.contrib import admin
from django.contrib.sitemaps.views import sitemap
from blog.sitemaps import PostSitemap

sitemaps = [
    'posts': PostSitemap,
]

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls', namespace='blog')),
    path('sitemap.xml', sitemap, {'sitemaps': sitemaps},
         name='django.contrib.sitemaps.views.sitemap')
]
```

В приведенном выше исходном коде были вставлены необходимые инструкции импорта и определен словарь `sitemaps`. На сайте может быть определено несколько карт. Мы определили шаблон URL-адреса, который совпадает с шаблоном `sitemap.xml` и в котором используется встроенное в Django представление `sitemap`. Словарь `sitemaps` передается в представление `sitemap`.

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/sitemap.xml` в своем браузере. Вы увидите результат в формате XML, включающий все опубликованные посты, как показано ниже:

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
  xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <url>
    <loc>http://example.com/blog/2022/1/22/markdown-post/</loc>
    <lastmod>2022-01-22</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/3/notes-on-duke-ellington/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/2/who-was-miles-davis/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/1/who-was-django-reinhardt/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>http://example.com/blog/2022/1/1/another-post/</loc>
    <lastmod>2022-01-03</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.9</priority>
  </url>
</urlset>
```

URL-адрес каждого объекта `Post` формируется путем вызова его метода `get_absolute_url()`.

Атрибут `lastmod` соответствует полю даты `updated` поста, как было указано в карте сайта, а атрибуты `changefreq` и `priority` также взяты из класса `PostSitemap`.

Для формирования URL-адресов используется домен `example.com`. Этот домен получен из хранящегося в базе данных объекта `Site`. Указанный объект был создан по умолчанию, когда вы синхронизировали фреймворк сайтов со своей базой данных. Подробнее о фреймворке сайтов можно почитать на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/sites/>.

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/sites/site/` в своем браузере. Вы должны увидеть что-то вроде этого:

The screenshot shows a list of sites in the Django admin interface. At the top, there's a search bar and an 'ADD SITE +' button. Below that, a table lists one site: 'example.com'. The table has two columns: 'DOMAIN NAME' and 'DISPLAY NAME'. The 'example.com' entry has a checked checkbox next to it. The 'DISPLAY NAME' column shows 'example.com'. At the bottom left, it says '1 site'.

DOMAIN NAME	DISPLAY NAME
<input checked="" type="checkbox"/> example.com	example.com

Рис. 3.17. Представление списка на сайте администрирования для модели `Site` фреймворка сайтов

На рис. 3.17 показано представление списка на сайте администрирования для фреймворка сайтов. Здесь можно задать домен или хост, который будет использоваться фреймворком сайтов и зависящими от него приложениями. Для того чтобы сгенерировать URL-адреса, существующие в локальной среде, измените доменное имя на `localhost:8000`, как показано на рис. 3.18, и сохраните его:

The screenshot shows the 'Change site' page for the 'example.com' object. It has fields for 'Domain name:' (localhost:8000) and 'Display name:' (localhost:8000). At the bottom, there are buttons for 'Delete', 'Save and add another', 'Save and continue editing', and a large blue 'SAVE' button.

Domain name:	localhost:8000
Display name:	localhost:8000

Рис. 3.18. Представление редактирования на сайте администрирования для модели `Site` фреймворка сайтов

Снова пройдите по URL-адресу `http://127.0.0.1:8000/sitemap.xml` в своем браузере. Теперь в отображаемых в ленте URL-адресах будет использовать-

ся новое хост-имя, и они будут выглядеть вот так: `http://localhost:8000/blog/2022/1/22/markdown-post/`. Теперь ссылки доступны в локальной среде. В производственной среде, чтобы иметь возможность генерировать абсолютные URL-адреса, придется использовать домен своего веб-сайта.

Создание новостных лент для постов блога

В Django есть встроенный фреймворк синдицированных новостных лент, который можно использовать для динамического генерирования новостных RSS- или Atom-лент, по форме похожих на создание карт сайта с использованием фреймворка сайтов. Новостная веб-лента – это формат данных (обычно XML), который предоставляет пользователям самый последний обновленный контент. Пользователи могут подписываться на новостную ленту с помощью агрегатора новостных лент, то есть программного обеспечения, которое используется для чтения новостных лент и получения уведомлений о новом контенте.

Внутри каталога своего приложения `blog` создайте новый файл и назовите его `feeds.py`. Добавьте в него следующие ниже строки:

```
import markdown
from django.contrib.syndication.views import Feed
from django.template.defaultfilters import truncatewords_html
from django.urls import reverse_lazy
from .models import Post

class LatestPostsFeed(Feed):
    title = 'My blog'
    link = reverse_lazy('blog:post_list')
    description = 'New posts of my blog.'

    def items(self):
        return Post.published.all()[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return truncatewords_html(markdown.markdown(item.body), 30)

    def item_pubdate(self, item):
        return item.publish
```

В приведенном выше исходном коде мы определили новостную ленту, создав подкласс класса Feed фреймворка синдицированных новостных лент. Атрибуты `title`, `link` и `description` соответствуют элементам RSS `<title>`, `<link>` и `<description>` в указанном порядке.

Функция-утилита `reverse_lazy()` используется для того, чтобы генерировать URL-адрес для атрибута `link`. Метод `reverse()` позволяет формировать URL-адреса по их имени и передавать optionalные параметры. Мы использовали `reverse()` в главе 2 «Усовершенствование блога за счет продвинутых функциональностей».

Функция-утилита `reverse_lazy()` представляет собой лениво вычисляемую версию `reverse()`. Она позволяет использовать обратный URL-адрес до того, как конфигурация URL-адреса проекта будет загружена.

Метод `items()` извлекает включаемые в новостную ленту объекты. Мы извлекаем последние пять опубликованных постов, которые затем будут включены в новостную ленту.

Методы `item_title()`, `item_description()` и `item_pubdate()` будут получать каждый возвращаемый методом `items()` объект и возвращать заголовок, описание и дату публикации по каждому элементу.

В методе `item_description()` используется функция `markdown()`, чтобы конвертировать контент в формате Markdown в формат HTML, и функция шаблонного фильтра `truncatewords_html()`, чтобы сокращать описание постов после 30 слов, избегая незакрытых HTML-тегов.

Теперь отредактируйте файл `blog/urls.py`, импортировав класс `LatestPostsFeed` и создав экземпляр новостной ленты в новом шаблоне URL-адреса, как показано ниже. Новые строки выделены жирным шрифтом:

```
from django.urls import path
from . import views
from .feeds import LatestPostsFeed

app_name = 'blog'

urlpatterns = [
    # представления поста
    path('', views.post_list, name='post_list'),
    # path('', views.PostListView.as_view(), name='post_list'),
    path('tag/<slug:tag_slug>/',
        views.post_list, name='post_list_by_tag'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>',
        views.post_detail,
        name='post_detail'),
    path('<int:post_id>/share/',
        views.post_share, name='post_share'),
    path('<int:post_id>/comment/',
        views.post_comment, name='post_comment'),
```

```
    path('feed/', LatestPostsFeed(), name='post_feed'),  
]
```

Пройдите по URL-адресу <http://127.0.0.1:8000/blog/feed/> в своем браузере. Теперь вы должны увидеть новостную RSS-ленту, содержащую последние пять постов блога:

```
<?xml version="1.0" encoding="utf-8"?>  
<rss xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">  
  <channel>  
    <title>My blog</title>  
    <link>http://localhost:8000/blog/</link>  
    <description>New posts of my blog.</description>  
    <atom:link href="http://localhost:8000/blog/feed/" rel="self"/>  
    <language>en-us</language>  
    <lastBuildDate>Fri, 2 Jan 2020 09:56:40 +0000</lastBuildDate>  
    <item>  
      <title>Who was Django Reinhardt?</title>  
      <link>http://localhost:8000/blog/2020/1/2/who-was-django-  
reinhardt/</link>  
      <description>Who was Django Reinhardt.</description>  
      <guid>http://localhost:8000/blog/2020/1/2/who-was-django-  
reinhardt/</guid>  
    </item>  
    ...  
  </channel>  
</rss>
```

Если вы используете Chrome, то увидите исходный код XML. Если же вы используете Safari, то он попросит вас установить программу чтения новостных RSS-лент.

Давайте установим RSS-клиента для рабочего стола, чтобы просматривать новостную RSS-ленту с удобным интерфейсом. Мы будем использовать многоплатформенный RSS-ридер Fluent Reader.

Скачайте Fluent Reader для Linux, macOS или Windows с <https://github.com/yang991178/fluent-reader/releases>.

Установите Fluent Reader и откройте его. Вы увидите следующий ниже экран:



Рис. 3.19. *Fluent Reader* без источников новостных RSS-лент

В правом верхнем углу окна кликните по значку настроек. Вы увидите экран добавления источников новостных RSS-лент, подобный следующему ниже:

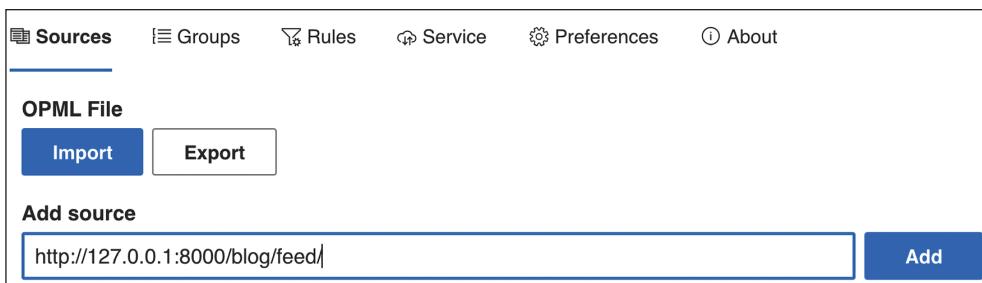


Рис. 3.20. Добавление новостной RSS-ленты в *Fluent Reader*

Введите `http://127.0.0.1:8000/blog/feed/` в поле **Add source** (Добавить источник) и кликните по кнопке **Add** (Добавить).

В таблице под формой вы увидите новую запись с новостной RSS-лентой блога, примерно в таком виде:

The screenshot shows the 'Sources' tab in the Fluent Reader application. At the top, there are tabs for 'Sources', 'Groups', 'Rules', 'Service', 'Preferences', and 'About'. Below the tabs, there's a section titled 'OPML File' with 'Import' and 'Export' buttons. A 'Add source' button is present, along with a text input field containing the URL 'http://127.0.0.1:8000/blog/feed/'. A table below lists the added source with columns for 'Name' and 'URL'. The table row for 'My blog' shows 'Name' as 'My blog' and 'URL' as 'http://127.0.0.1:8000/blog/feed/'.

Name	URL
My blog	http://127.0.0.1:8000/blog/feed/

Рис. 3.21. Источники новостных RSS-лент в Fluent Reader

Теперь вернитесь к главному экрану Fluent Reader. Вы должны увидеть посты, включенные в новостную RSS-ленту блога, как показано ниже:

The screenshot shows the main screen of Fluent Reader displaying the blog's RSS feed. At the top, there are several small icons and a 'All articles' button. Below, five posts are listed in cards:

- Markdown post**: This is a post formatted with markdown. This is emphasized and this is more emphasized. Here is a list: One Two Three And a link to the Django website ...
- Notes on Duke Ellington**: Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...
- Who was Miles Davis?**: Miles Davis was an American trumpeter, bandleader, and composer. He is among the most influential and acclaimed figures in the history of jazz and 20th-century music.
- Who was Django Reinhardt?**: Jean Reinhardt, known to all by his Romani nickname Django, was a Belgian-born Romani-French jazz guitarist and composer. He was the first major jazz talent to emerge from Europe and ...
- Another post**: Post body.

Рис. 3.22. Новостная RSS-лента блога в Fluent Reader

Кликните по посту, чтобы увидеть описание:

The screenshot shows a mobile-style interface for a blog post. At the top, it says "My blog". Below that is a title "Notes on Duke Ellington". Under the title is the date and time "1/3/2022, 2:19:33 PM". A truncated description follows: "Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half ...". To the right of the truncated text is a three-dot ellipsis icon.

Рис. 3.23. Описание поста в Fluent Reader

Кликните по третьему значку в правом верхнем углу окна, чтобы загрузить полный контент страницы поста:

The screenshot shows the full content of the blog post. It includes the title "Notes on Duke Ellington", the date "1/3/2022, 2:19:33 PM", and the author information "Published Jan. 3, 2022, 1:19 p.m. by admin". The main content is: "Edward Kennedy "Duke" Ellington was an American composer, pianist, and leader of a jazz orchestra, which he led from 1923 until his death over a career spanning more than half a century." Below the content is a "Share this post" link. A section titled "Similar posts" lists two other posts: "Who was Miles Davis?" and "Who was Django Reinhardt?". At the bottom, there is a "1 comment" section and a "Add a new comment" button.

Рис. 3.24. Полный контент поста в Fluent Reader

Последним шагом будет добавление ссылки на подписку на новостную RSS-ленту в боковую панель блога.

Откройте шаблон `blog/base.html` и добавьте в него следующий ниже исходный код, выделенный жирным шрифтом:

```
{% load blog_tags %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
    <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
    <div id="content">
        {% block content %}
        {% endblock %}
    </div>
    <div id="sidebar">
        <h2>My blog</h2>
        <p>
            This is my blog.
            I've written {% total_posts %} posts so far.
        </p>
        <p>
            <a href="{% url "blog:post_feed" %}">
                Subscribe to my RSS feed
            </a>
        </p>
        <h3>Latest posts</h3>
        {% show_latest_posts 3 %}
        <h3>Most commented posts</h3>
        {% get_most_commented_posts as most_commented_posts %}
        <ul>
            {% for post in most_commented_posts %}
            <li>
                <a href="{{ post.get_absolute_url }}>{{ post.title }}</a>
            </li>
            {% endfor %}
        </ul>
    </div>
</body>
</html>
```

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/blog/` в своем браузере и взгляните на боковую панель. Новая ссылка приведет пользователей к новостной ленте блога:

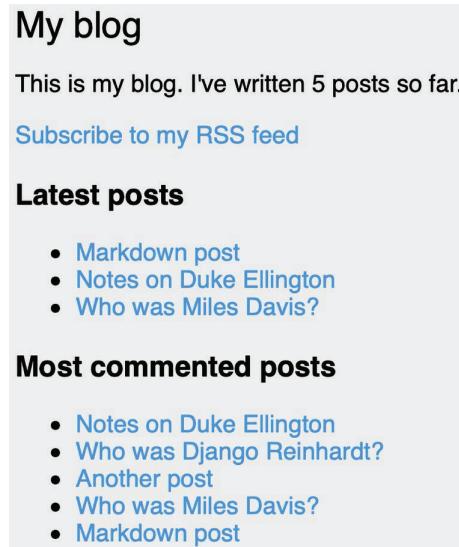


Рис. 3.25. Ссылка на подписку на новостную RSS-ленту, добавленная на боковую панель

Подробнее о встроенному в Django фреймворке синдицированных новостных лент можно почитать на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/syndication/>.

Добавление полнотекстового поиска в блог

Далее мы добавим в блог возможности поиска. Поиск в базе данных на основе вводимых пользователем данных находит широкое применение в веб-приложениях. Встроенный в Django ORM-преобразователь позволяет выполнять простые операции сопоставления, используя, например, фильтр `contains` (или его нечувствительную к регистру версию `icontains`). Следующий ниже запрос можно использовать, чтобы найти посты, содержащие слово `framework` в их теле:

```
from blog.models import Post
Post.objects.filter(body__contains='framework')
```

Однако если вы хотите выполнять сложные операции поиска, извлекая результаты по сходству или взвешивая термины на основе частоты их встречаемости в тексте или степени важности различных полей (например, релевантности термина, появляющегося в заголовке по сравнению с его появлением

теле), то необходимо задействовать полнотекстовый поисковый механизм. При рассмотрении больших блоков текста формирование запросов с операциями над цепочками символов становится недостаточным. Полнотекстовый поиск сравнивает фактические слова с сохраненным контентом, пытаясь удовлетворить критериям поиска.

Django предоставляет мощную функциональность поиска, построенную поверх функциональных возможностей реляционной базы данных PostgreSQL по полнотекстовому поиску. Модуль `django.contrib.postgres` предоставляет функциональности, предлагаемые базой данных PostgreSQL, которые не являются общими для других поддерживаемых веб-фреймворком Django баз данных. О поддержке принятого в PostgreSQL полнотекстового поиска можно узнать на странице <https://www.postgresql.org/docs/14/textsearch.html>.



Хотя веб-фреймворк Django не зависит от базы данных, он предоставляет модуль, который поддерживает часть богатого набора функциональных возможностей, предлагаемого базой данных PostgreSQL, которого нет в других поддерживаемых Django базах данных.

Установка базы данных PostgreSQL

В настоящее время в проекте `mysite` используется база данных SQLite. Поддержка полнотекстового поиска SQLite ограничена, и Django не поддерживает его прямо «из коробки». Однако PostgreSQL подходит для полнотекстового поиска гораздо лучше, и мы можем воспользоваться модулем `django.contrib.postgres`, чтобы задействовать возможности полнотекстового поиска PostgreSQL. Мы выполним миграции данных из SQLite в PostgreSQL, чтобы применить ее функциональные возможности полнотекстового поиска.



База данных SQLite приемлема для целей разработки. Однако для производственной среды вам понадобится более мощная база данных, такая как PostgreSQL, MariaDB, MySQL или Oracle.

Скачайте установщик PostgreSQL для macOS или Windows на странице <https://www.postgresql.org/download/>.

На той же странице вы найдете инструкции по установке PostgreSQL в различных дистрибутивах Linux. Следуйте инструкциям на веб-сайте по установке и запуску PostgreSQL.

Если вы используете macOS и решили установить PostgreSQL с помощью `Postgres.app`, то вам нужно будет настроить переменную `$PATH`, чтобы применить инструменты командной строки, как описано на странице <https://postgresapp.com/documentation/cli-tools.html>.

Вам также необходимо установить PostgreSQL-адаптер `psycopg2` для Python. Выполните следующую ниже команду в командной оболочке, чтобы его установить:

```
pip install psycopg2-binary==2.9.3
```

Создание базы данных PostgreSQL

Давайте создадим пользователя базы данных PostgreSQL. Мы будем использовать интерфейс PostgreSQL на основе терминала `psql`. Войдите в терминал PostgreSQL, выполнив следующую ниже команду в командной оболочке:

```
psql
```

Вы увидите следующий ниже результат:

```
psql (14.2)
Type "help" for help.
```

Введите следующую ниже команду, чтобы создать пользователя, который может создавать базы данных:

```
CREATE USER blog WITH PASSWORD 'xxxxxx';
```

Замените `xxxxxx` на желаемый пароль и выполните команду. Вы увидите такой результат:

```
CREATE ROLE
```

Пользователь был создан. Теперь давайте создадим базу данных `blog` и передадим права на владение этой базой данных только что созданному пользователю.

Исполните следующую ниже команду:

```
CREATE DATABASE blog OWNER blog ENCODING 'UTF8';
```

Этой командой мы сообщаем PostgreSQL, что нужно создать базу данных с именем `blog`, мы передаем право на владение базой данных `blog` ее пользователю, которого мы создали ранее, и указываем, что для новой базы данных должна использоваться кодировка UTF8. Вы увидите следующий ниже результат:

```
CREATE DATABASE
```

Мы успешно создали пользователя PostgreSQL и базу данных.

Выгрузка существующих данных

Перед переключением на другую базу данных необходимо выгрузить существующие в проекте Django данные из базы данных SQLite. Мы экспортируем данные, переключаем базу данных проекта на PostgreSQL и импортируем данные в новую базу данных.

Django предлагает простой способ загрузки и выгрузки данных из базы данных в файлы, которые называются **фикастурами**. Django поддерживает фикстуры в форматах JSON, XML или YAML. Мы собираемся создать фикстуру со всеми данными, содержащимися в базе данных.

Команда `dumpdata` выгружает данные из базы данных в стандартный вывод, по умолчанию сериализованный в формате JSON. Результирующая структура данных включает информацию о модели и ее полях, позволяя Django загружать ее в базу данных.

Выгружаемые данные можно ограничивать моделями приложения, предоставив команде имена приложений либо указав отдельные модели для вывода данных, используя формат `app.Model`. Также можно указывать формат, используя флаг `--format`. По умолчанию `dumpdata` выводит сериализованные данные в стандартный вывод. Однако, используя флаг `--output`, можно указать выходной файл. Флаг `--indent` позволяет указывать отступ. Дополнительную информацию об опциях команды `dumpdata` можно получить, выполнив команду `python manage.py dumpdata --help`.

Исполните следующую ниже команду из командной оболочки:

```
python manage.py dumpdata --indent=2 --output=mysite_data.json
```

Вы увидите результат, аналогичный приведенному ниже:

```
[.....]
```

Все существующие данные были экспортированы в формат JSON в новый файл с именем `mysite_data.json`. Содержимое файла можно просмотреть, чтобы увидеть структуру JSON, которая включает в себя разнообразные объекты данных различных моделей установленных вами приложений. Если при выполнении команды вы получаете ошибку кодировки, то включите флаг `-Xutf8`, как показано ниже, чтобы активировать режим Python UTF-8:

```
python -Xutf8 manage.py dumpdata --indent=2 --output=mysite_data.json
```

Теперь мы переключим базу данных в проекте Django, а затем импортируем данные в новую базу данных.

Переключение базы данных в проекте

Отредактируйте файл `settings.py` проекта, видоизменив настроечный параметр `DATABASES` и придав ему следующий вид. Новый исходный код выделен жирным шрифтом:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'blog',
        'USER': 'blog',
        'PASSWORD': 'xxxxxx',
    }
}
```

Замените `xxxxxx` паролем, который вы использовали при создании пользователя PostgreSQL. Новая база данных пуста.

Выполните следующую ниже команду, чтобы применить все миграции к новой базе данных PostgreSQL:

```
python manage.py migrate
```

Вы увидите результат, включая все миграции, которые были применены, как показано ниже:

```
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions, sites,
taggit
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002.Alter_permission_name_max_length... OK
  Applying auth.0003_Alter_user_email_max_length... OK
  Applying auth.0004_Alter_user_username_opts... OK
  Applying auth.0005_Alter_user_last_login_null... OK
  Applying auth.0006_Require_contenttypes_0002... OK
  Applying auth.0007_Alter_validators_add_error_messages... OK
  Applying auth.0008_Alter_user_username_max_length... OK
  Applying auth.0009_Alter_user_last_name_max_length... OK
  Applying auth.0010_Alter_group_name_max_length... OK
  Applying auth.0011_Update_proxy_permissions... OK
  Applying auth.0012_Alter_user_first_name_max_length... OK
  Applying taggit.0001_initial... OK
  Applying taggit.0002_auto_20150616_2121... OK
  Applying taggit.0003_taggeditem_add_unique_index... OK
  Applying blog.0001_initial... OK
  Applying blog.0002_Alter_post_slug... OK
  Applying blog.0003_comment... OK
  Applying blog.0004_post_tags... OK
  Applying sessions.0001_initial... OK
```

```
Applying sites.0001_initial... OK
Applying sites.0002_alter_domain_unique... OK
Applying taggit.0004_alter_taggeditem_content_type_alter_taggeditem_tag... OK
Applying taggit.0005_auto_20220424_2025... OK
```

Загрузка данных в новую базу данных

Выполните следующую ниже команду, чтобы загрузить данные в базу данных PostgreSQL:

```
python manage.py loaddata mysite_data.json
```

Вы увидите следующий ниже результат:

```
Installed 104 object(s) from 1 fixture(s)
```

Число объектов может отличаться в зависимости от пользователей, постов, комментариев и других объектов, которые были созданы в базе данных ранее.

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver
```

Пройдите по URL-адресу <http://127.0.0.1:8000/admin/blog/post/> в своем браузере, чтобы убедиться, что все записи были загружены в новую базу данных. Вы должны увидеть все посты, как показано ниже:

Select post to change					
<input type="text"/> <input type="button" value="Search"/>					
« 2022 January 1 January 2 January 3 January 22 »					
Action:	<input type="button" value="-----"/>	<input type="button" value="Go"/>	0 of 5 selected		
<input type="checkbox"/>	TITLE	SLUG	AUTHOR	PUBLISH	2 ▲ STATUS 1 ▲
<input type="checkbox"/>	Another post	another-post	admin	Jan. 1, 2022, 11:57 p.m.	Published
<input type="checkbox"/>	Who was Django Reinhardt?	who-was-django-reinhardt	admin	Jan. 1, 2022, 11:59 p.m.	Published
<input type="checkbox"/>	Who was Miles Davis?	who-was-miles-davis	admin	Jan. 2, 2022, 1:18 p.m.	Published
<input type="checkbox"/>	Notes on Duke Ellington	notes-on-duke-ellington	admin	Jan. 3, 2022, 1:19 p.m.	Published
<input type="checkbox"/>	Markdown post	markdown-post	admin	Jan. 22, 2022, 9:30 a.m.	Published

5 posts

Рис. 3.26. Список постов на сайте администрирования

Простые операции поиска

Отредактируйте файл `settings.py` проекта, добавив `django.contrib.postgres` в настроечный параметр `INSTALLED_APPS`, как показано ниже:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'blog.apps.BLogConfig',  
    'taggit',  
    'django.contrib.sites',  
    'django.contrib.sitemaps',  
    'django.contrib.postgres',  
]
```

Откройте оболочку Django, выполнив следующую ниже команду в системной командной оболочке:

```
python manage.py shell
```

Теперь можно выполнять поиск по одному полю, используя операцию `search` набора запросов `QuerySet`.

Выполните следующий ниже исходный код в оболочке Python:

```
>>> from blog.models import Post  
>>> Post.objects.filter(title__search='django')  
<QuerySet [<Post: Who was Django Reinhardt?>]>
```

В этом запросе PostgreSQL используется для того, чтобы создать поисковый вектор для поля `body` и поисковый запрос из термина `django`. Результаты получаются путем сопоставления запроса с вектором.

Поиск по нескольким полям

Возможно, вам захочется выполнить поиск по нескольким полям. В этом случае необходимо определить объект `SearchVector`. Давайте сформируем вектор, который позволит выполнять поиск по полям `title` и `body` модели `Post`.

Выполните следующий ниже исходный код в оболочке Python:

```
>>> from django.contrib.postgres.search import SearchVector  
>>> from blog.models import Post  
>>>
```

```
>>> Post.objects.annotate(
...     search=SearchVector('title', 'body'),
... ).filter(search='django')
<QuerySet [ <Post: Markdown post>, <Post: Who was Django Reinhardt?> ]>
```

Используя `annotate` и определяя `SearchVector` с обоими полями, предоставляется функциональность сопоставления запроса как с заголовком, так и с телом постов.



Полнотекстовый поиск – это интенсивный процесс. Если вы выполняете поиск среди нескольких сотен строк, то вам следует определить функциональный индекс, который сопоставляется с используемым поисковым индексом. Django предоставляет поле `SearchVectorField` для ваших моделей. Подробнее об этом можно почитать по адресу <https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/#performance>.

Разработка представления поиска

Теперь вы создадите конкретно-прикладное представление, позволяющее пользователям выполнять поиск постов. Прежде всего понадобится форма для поиска. Отредактируйте файл `forms.py` приложения `blog`, добавив следующую ниже форму:

```
class SearchForm(forms.Form):
    query = forms.CharField()
```

Поле запроса будет использоваться для того, чтобы давать пользователям возможность вводить поисковые запросы. Отредактируйте файл `views.py` приложения `blog`, добавив следующий ниже исходный код:

```
# ...
from django.contrib.postgres.search import SearchVector
from .forms import EmailPostForm, CommentForm, SearchForm

# ...

def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
            results = Post.objects.annotate(
                search=SearchVector('title', 'body'),
            ).filter(search=query)
```

```
query = form.cleaned_data['query']
results = Post.published.annotate(
    search=SearchVector('title', 'body'),
).filter(search=query)

return render(request,
    'blog/post/search.html',
    {'form': form,
     'query': query,
     'results': results})
```

В приведенном выше представлении сначала создается экземпляр формы `SearchForm`. Для проверки того, что форма была передана на обработку, в словаре `request.GET` отыскивается параметр `query`. Форма отправляется методом GET, а не методом POST, чтобы результирующий URL-адрес содержал параметр `query` и им было легко делиться. После передачи формы на обработку создается ее экземпляр, используя переданные данные GET, и проверяется валидность данных формы. Если форма валидна, то с помощью конкретно-прикладного экземпляра `SearchVector`, сформированного с использованием полей `title` и `body`, выполняется поиск опубликованных постов.

Теперь представление поиска готово и необходимо создать шаблон отображения формы и результатов при выполнении пользователем поиска.

Внутри каталога `templates/blog/post/` создайте новый файл, назовите его `search.html` и добавьте в него следующий ниже исходный код:

```
{% extends "blog/base.html" %}
{% load blog_tags %}

{% block title %}Search{% endblock %}

{% block content %}
{% if query %}
<h1>Posts containing "{{ query }}"</h1>
<h3>
    {% with results.count as total_results %}
        Found {{ total_results }} result{{ total_results|pluralize }}
    {% endwith %}
</h3>
{% for post in results %}
    <h4>
        <a href="{{ post.get_absolute_url }}">
            {{ post.title }}
        </a>
    </h4>
    {{ post.body|markdown|truncatewords_html:12 }}
{% empty %}
    <p>There are no results for your query.</p>
{% endfor %}
{% endif %}
```

```
{% endfor %}  
<p><a href="{% url "blog:post_search" %}">Search again</a></p>  
{% else %}  
    <h1>Search for posts</h1>  
    <form method="get">  
        {{ form.as_p }}  
        <input type="submit" value="Search">  
    </form>  
{% endif %}  
{% endblock %}
```

Как и в представлении поиска, по наличию параметра `query` определяется, что форма была передана на обработку. Перед передачей запроса мы отображаем форму и кнопку передачи формы. После передачи формы поиска на обработку отображается выполненный запрос, общее число результатов и список постов, совпадающих с поисковым запросом.

Наконец, отредактируйте файл `urls.py` приложения `blog`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [  
    # представления поста  
    path('', views.post_list, name='post_list'),  
    # path('', views.PostListView.as_view(), name='post_list'),  
    path('tag/<slug:tag_slug>',  
        views.post_list, name='post_list_by_tag'),  
    path('<int:year>/<int:month>/<int:day>/<slug:post>',  
        views.post_detail,  
        name='post_detail'),  
    path('<int:post_id>/share/',  
        views.post_share, name='post_share'),  
    path('<int:post_id>/comment/',  
        views.post_comment, name='post_comment'),  
    path('feed/', LatestPostsFeed(), name='post_feed'),  
    path('search/', views.post_search, name='post_search'),  
]
```

Далее пройдите по URL-адресу <http://127.0.0.1:8000/blog/search/> в своем браузере. Вы должны увидеть следующую ниже форму для поиска:

The screenshot shows a search interface. On the left, there is a text input field labeled "Query:" with a placeholder "Search posts..." and a blue "SEARCH" button below it. On the right, there is a sidebar with the title "My blog" followed by the text "This is my blog. I've written 5 posts so far." and a link "Subscribe to my RSS feed". Below this, under the heading "Latest posts", is a list of three items: "Markdown post", "Notes on Duke Ellington", and "Who was Miles Davis?". Under the heading "Most commented posts", is a list of five items: "Notes on Duke Ellington", "Who was Django Reinhardt?", "Another post", "Who was Miles Davis?", and "Markdown post".

Рис. 3.27. Форма с полем запроса для поиска постов

Ведите запрос и кликните по кнопке **SEARCH** (Найти). Вы увидите результаты поискового запроса, как показано ниже:

The screenshot shows the search results for the query "jazz". The main area displays the heading "Posts containing \"jazz\"". Below it, it says "Found 3 results" and lists three results: "Notes on Duke Ellington", "Who was Miles Davis?", and "Who was Django Reinhardt?". To the right, there is a sidebar with the title "My blog" followed by the text "This is my blog. I've written 5 posts so far." and a link "Subscribe to my RSS feed". Below this, under the heading "Latest posts", is a list of three items: "Markdown post", "Notes on Duke Ellington", and "Who was Miles Davis?". Under the heading "Most commented posts", is a list of five items: "Notes on Duke Ellington", "Who was Django Reinhardt?", "Another post", "Who was Miles Davis?", and "Markdown post".

Рис. 3.28. Результаты поиска термина «jazz»

Поздравляю! Вы создали базовый поисковый механизм для своего блога.

Выделение основ слов и ранжирование результатов

Выделение основы слова, или стемминг, – это процесс приведения слов к их словообразовательной базе, основанию или корневой форме. Стемминг используется поисковыми системами для редуцирования индексированных слов до их основы и для того, чтобы иметь возможность сопоставлять изменяемые или производные слова. Например, слова *music*, *musical* и *musicality* поисковая система может считать похожими словами. В процессе выделения основы каждый поисковый токен нормализуется в лексему, или единицу лексического значения, которая лежит в основе набора слов, связанных посредством флексии. При создании поискового запроса слова *music*, *musical* и *musicality* будут преобразованы в *music*.

Django предоставляет класс `SearchQuery`, чтобы транслировать термины в объект поискового запроса. По умолчанию термины пропускаются через базовые алгоритмы выделения основ слов, что помогает получать более оптимальные совпадения.

Поисковый механизм PostgreSQL также удаляет стоп-слова, такие как *a*, *the*, *on* и *of*. Стоп-слова – это коллекция часто используемых слов в языке. Они удаляются при создании поискового запроса, поскольку появляются слишком часто, чтобы быть релевантными для поисковых запросов. Список стоп-слов, используемых PostgreSQL для английского языка, находится на странице <https://github.com/postgres/postgres/blob/master/src/backend/snowball/stopwords/english.stop>.

Мы также хотим упорядочивать результаты по релевантности. PostgreSQL предоставляет функцию ранжирования, которая упорядочивает результаты на основе частоты появления терминов запроса и степени их близости друг к другу.

Отредактируйте файл `views.py` приложения `blog`, добавив следующую ниже инструкцию импорта:

```
from django.contrib.postgres.search import SearchVector, \
    SearchQuery, SearchRank
```

Затем отредактируйте представление `post_search`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
```

```
search_vector = SearchVector('title', 'body')
search_query = SearchQuery(query)
results = Post.published.annotate(
    search=search_vector,
    rank=SearchRank(search_vector, search_query)
).filter(search=search_query).order_by('-rank')

return render(request,
    'blog/post/search.html',
    {'form': form,
     'query': query,
     'results': results})
```

В приведенном выше исходном коде создается объект `SearchQuery`, по нему фильтруются результаты, и для упорядочивания результатов по релевантности используется `SearchRank`.

Теперь можно перейти по адресу `http://127.0.0.1:8000/blog/search/` в своем браузере и протестировать различные поисковые запросы, чтобы проверить выделение основ слов и ранжирование. Ниже приведен пример ранжирования по числу появлений слова `django` в заголовке и теле постов:

The screenshot shows a search results page for the term "django". The main content area displays two posts: one about Jean Reinhardt and another post formatted with Markdown. The sidebar includes links to the blog's RSS feed and a list of latest posts.

Posts containing "django"

Found 2 results

Who was Django Reinhardt?
Jean Reinhardt, known to all by his Romani nickname Django, was a ...

Markdown post
This is a post formatted with markdown
This is emphasized and this ...

Search again

My blog
This is my blog. I've written 5 posts so far.
[Subscribe to my RSS feed](#)

Latest posts

- [Markdown post](#)
- [Notes on Duke Ellington](#)
- [Who was Miles Davis?](#)

Most commented posts

- [Notes on Duke Ellington](#)
- [Who was Django Reinhardt?](#)
- [Another post](#)
- [Who was Miles Davis?](#)
- [Markdown post](#)

Рис. 3.29. Результаты поиска по термину «django»

Выделение основ слов и удаление стоп-слов на разных языках

Объекты `SearchVector` и `SearchQuery` можно настроить под исполнение процедур выделения основ слов и удаления стоп-слов на любом языке. Для того чтобы использовать другую конфигурацию поиска, в `SearchVector` и `SearchQuery` передается атрибут `config`. Такой подход позволяет использовать раз-

ные языковые парсеры и словари. В следующем ниже примере выделяются основы слов и удаляются стоп-слова на испанском языке:

```
search_vector = SearchVector('title', 'body', config='spanish')
search_query = SearchQuery(query, config='spanish')
results = Post.published.annotate(
    search=search_vector,
    rank=SearchRank(search_vector, search_query)
).filter(search=search_query).order_by('-rank')
```

Используемый в PostgreSQL словарь испанских стоп-слов находится на странице <https://github.com/postgres/postgres/blob/master/src/backend/snowball/stopwords/spanish.stop>.

Взвешивание запросов

Влияние конкретных векторов можно усиливать таким образом, чтобы им придавался больший вес при упорядочивании результатов по релевантности. Например, взвешивание можно использовать, чтобы придавать большую релевантность постам, которые сочетаются по заголовку, а не по содержимому.

Отредактируйте файл `views.py` приложения `blog`, видоизменив представление `post_search`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
            query = form.cleaned_data['query']
            search_vector = SearchVector('title', weight='A') + \
                SearchVector('body', weight='B')
            search_query = SearchQuery(query)
            results = Post.published.annotate(
                search=search_vector,
                rank=SearchRank(search_vector, search_query)
            ).filter(rank__gte=0.3).order_by('-rank')

    return render(request,
                  'blog/post/search.html',
                  {'form': form,
                   'query': query,
                   'results': results})
```

В приведенном выше исходном коде к векторам поиска, сформированным с использованием полей `title` и `body`, применяются разные веса. По умолчанию веса таковы: D, C, B и A, и они относятся соответственно к числам 0.1, 0.2, 0.4 и 1.0. Вес 1.0 применяется к вектору поиска `title` (A), и вес 0.4 – к вектору `body` (B). Совпадения с заголовком будут преобладать над совпадениями с содержимым тела поста. Результаты фильтруются, чтобы отображать только те, у которых ранг выше 0.3.

Поиск по триграммному сходству

Еще одним подходом к поиску является триграммное сходство. Триграмма – это группа из трех следующих друг за другом символов. Сходство двух строковых литералов можно измерять, подсчитывая число общих для них триграмм. Во многих языках данный подход оказывается очень эффективным при измерении сходства слов.

Для того чтобы использовать триграммы в PostgreSQL, сначала необходимо установить расширение `pg_trgm`. Исполните следующую ниже команду в командной оболочке, чтобы подсоединиться к своей базе данных:

```
psql blog
```

Затем исполните следующую ниже команду, чтобы установить расширение `pg_trgm`:

```
CREATE EXTENSION pg_trgm;
```

Вы получите такой результат:

```
CREATE EXTENSION
```

Давайте отредактируем представление и видоизменим его под триграммный поиск.

Отредактируйте файл `views.py` приложения `blog`, добавив следующую ниже инструкцию импорта:

```
from django.contrib.postgres.search import TrigramSimilarity
```

Затем видоизмените представление `post_search`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
def post_search(request):
    form = SearchForm()
    query = None
    results = []

    if 'query' in request.GET:
        form = SearchForm(request.GET)
```

```

if form.is_valid():
    query = form.cleaned_data['query']
    results = Post.published.annotate(
        similarity=TrigramSimilarity('title', query),
    ).filter(similarity__gt=0.1).order_by('-similarity')

return render(request,
              'blog/post/search.html',
              {'form': form,
               'query': query,
               'results': results})

```

Пройдите по URL-адресу <http://127.0.0.1:8000/blog/search/> в своем браузере и протестируйте различные варианты триграммного поиска. В следующем ниже примере показана гипотетическая опечатка в термине django, показаны результаты поиска термина yango:

The screenshot shows a search results page for the term "yango". The main content area displays a single result: "Who was Django Reinhardt?". Below the result is a link to "Search again". To the right, there is a sidebar with sections for "My blog", "Latest posts", and "Most commented posts", each listing several blog post titles.

Section	Content
My blog	This is my blog. I've written 5 posts so far.
Latest posts	<ul style="list-style-type: none"> Markdown post Notes on Duke Ellington Who was Miles Davis?
Most commented posts	<ul style="list-style-type: none"> Notes on Duke Ellington Who was Django Reinhardt? Another post Who was Miles Davis? Markdown post

Рис. 3.30. Результаты поиска термина «yango»

В приложение для ведения блога был добавлен мощный поисковый механизм.

Более подробная информация о полнотекстовом поиске находится на странице <https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/>.

Дополнительные ресурсы

Следующие ниже ресурсы предоставляют дополнительную информацию, относящуюся к темам, затронутым в этой главе.

- Исходный код к этой главе: <https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter03>.

- Приложение Django-taggit: <https://github.com/jazzband/django-taggit>.
- ORM-менеджеры приложения Django-taggit: <https://django-taggit.readthedocs.io/en/latest/api.html>.
- Взаимосвязи многие-ко-многим: https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/.
- Встроенные в Django функции агрегирования: <https://docs.djangoproject.com/en/4.1/topics/db/aggregation/>.
- Встроенные шаблонные теги и фильтры: <https://docs.djangoproject.com/en/4.1/ref/templates/builtins/>.
- Написание конкретно-прикладных шаблонных тегов: <https://docs.djangoproject.com/en/4.1/howto/custom-template-tags/>.
- Справочный материал по формату Markdown: <https://daringfireball.net/projects/markdown/basics>.
- Встроенный в Django фреймворк карт сайтов: <https://docs.djangoproject.com/en/4.1/ref/contrib/sitemaps/>.
- Встроенный в Django фреймворк сайтов Django: <https://docs.djangoproject.com/en/4.1/ref/contrib/sites/>.
- Встроенный в Django фреймворк синдицированных новостных лент: <https://docs.djangoproject.com/en/4.1/ref/contrib/syndication/>.
- Скачиваемые файлы PostgreSQL: <https://www.postgresql.org/download/>.
- Возможности полнотекстового поиска в PostgreSQL: <https://www.postgresql.org/docs/14/textsearch.html>.
- Поддержка со стороны Django полнотекстового поиска PostgreSQL: <https://docs.djangoproject.com/en/4.1/ref/contrib/postgres/search/>.

Резюме

В этой главе вы реализовали систему тегирования, интегрировав стороннее приложение в свой проект. Вы генерировали рекомендуемые посты, используя сложные наборы запросов QuerySet. Вы также научились создавать конкретно-прикладные шаблонные теги и фильтры Django, чтобы обеспечивать шаблонам конкретно-прикладные функциональности. Создали карту сайта, чтобы поисковые системы имели возможность сканировать ваш сайт, и новостную RSS-ленту, дабы пользователи могли подписываться на ваш блог. Затем вы разработали в своем блоге поисковый механизм, используя полнотекстовый поисковый механизм PostgreSQL.

В следующей главе вы научитесь разрабатывать социальный веб-сайт с использованием встроенного в Django фреймворка аутентификации и реализовывать функциональности с применением учетных записей пользователей и конкретно-прикладные профили пользователей.