

Мигель Гринберг

Разработка веб-приложений с использованием Flask на языке Python

Miguel Grinberg

Flask Web Development

O'REILLY®

Мигель Гринберг

Разработка веб-приложений с использованием Flask на языке Python



Москва, 2014

УДК 004.738.5:004.4Flask
ББК 32.973.26-018.2
Г82

Гринберг М.
Г82 Разработка веб-приложений с использованием Flask на языке Python / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2014. – 272 с.: ил.

ISBN 978-5-97060-138-9

В этой книге вы изучите популярный микрофреймворк Flask на пошаговых примерах создания законченного приложения социального блогинга. Автор книги Мигель Гринберг познакомит вас с основными функциональными возможностями фреймворка и покажет, как расширять приложения дополнительными веб-технологиями, такими как поддержка миграции базы данных и взаимодействия с веб-службами.

Вместо того чтобы навязывать строгие правила, как это делают другие фреймворки, Flask оставляет за вами свободу принятия решений. Если вы имеете опыт программирования на языке Python, данная книга покажет вам, как можно воспользоваться такой свободой творчества!

УДК 004.738.5:004.4Flask
ББК 32.973.26-018.2

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.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-449-37262-0 (англ.)

ISBN 978-5-97060-138-9 (рус.)

Copyright © 2014 Miguel
Grinberg

© Оформление, перевод,
ДМК Пресс, 2014

Алисе посвящается

Содержание

Предисловие	12
Часть I. Введение в Flask	21
Глава 1. Установка	22
Использование виртуальных окружений.....	23
Установка пакетов Python с помощью pip.....	25
Глава 2. Структура простого приложения	26
Инициализация	26
Маршруты и функции представлений	26
Запуск сервера	28
Законченное приложение	29
Цикл запрос–ответ	31
Контексты приложения и запроса.....	31
Обработка запросов	33
Обработчики событий жизненного цикла.....	34
Ответы.....	35
Расширения Flask	37
Поддержка параметров командной строки с помощью Flask-Script.....	37
Глава 3. Шаблоны	40
Механизм шаблонов Jinja2	41
Отображение шаблонов	41
Переменные.....	42
Управляющие структуры.....	43
Интеграция Twitter Bootstrap с помощью Flask-Bootstrap	45
Нестандартные страницы с сообщениями об ошибках	49
Ссылки.....	52
Статические файлы.....	53
Локализация дат и времени с помощью Flask-Moment	54
Глава 4. Веб-формы	57
Защита от подделки межсайтовых запросов	57
Классы форм.....	58
Отображение форм в формат HTML.....	60

Обработка форм в функциях представления.....	62
Переадресация и сеансы.....	65
Всплывающие сообщения.....	67
Глава 5. Базы данных.....	70
Базы данных SQL.....	70
Базы данных NoSQL.....	71
SQL или NoSQL?.....	72
Фреймворки на Python поддержки баз данных.....	72
Интеграция с фреймворком Flask.....	74
Управление базой данных с помощью Flask-SQLAlchemy.....	74
Определение модели.....	75
Отношения.....	78
Операции с базами данных.....	80
Создание таблиц.....	80
Вставка строк.....	80
Изменение строк.....	82
Удаление строк.....	82
Извлечение строк.....	82
Операции с базой данных в функциях представления.....	85
Интеграция с интерактивной оболочкой Python.....	86
Миграция базы данных с помощью Flask-Migrate.....	87
Создание репозитория миграции.....	88
Создание сценария миграции.....	88
Обновление базы данных.....	89
Глава 6. Электронная почта.....	91
Поддержка электронной почты с помощью Flask-Mail.....	91
Отправка электронной почты из интерактивной оболочки Python.....	93
Интеграция поддержки электронной почты в приложение.....	93
Асинхронная отправка электронной почты.....	95
Глава 7. Структура больших приложений.....	97
Структура проекта.....	97
Параметры настройки.....	98
Пакет приложения.....	100
Фабричная функция приложения.....	100
Реализация функциональности приложения в виде макета.....	101
Сценарий запуска.....	104

Файл зависимостей	105
Модульные тесты	106
Настройка базы данных	108

Часть II. Пример: приложение социального

блогинга	109
-----------------------	-----

Глава 8. Аутентификация пользователей

Расширения аутентификации для Flask	110
Защита паролей	111
Хэширование паролей с помощью Werkzeug	111
Создание макета для поддержки аутентификации	114
Аутентификация пользователя с помощью Flask-Login	115
Подготовка модели User для аутентификации	115
Защита маршрутов	117
Добавление формы аутентификации	118
Аутентификация	119
Выход пользователя	121
Тестирование процедуры аутентификации	122
Регистрация нового пользователя	122
Добавление формы регистрации пользователя	123
Регистрация	125
Подтверждение создания учетной записи	126
Создание маркера подтверждения с помощью itsdangerous	126
Отправка электронных писем с инструкциями для подтверждения	128
Управление учетными записями	133

Глава 9. Роли пользователей

Представление ролей в базе данных	135
Присваивание ролей	138
Проверка роли	139

Глава 10. Профили пользователей

Информация для профиля	143
Страница профиля пользователя	144
Редактор профиля	147
Редактор профиля уровня пользователя	147

Редактор профиля уровня администратора.....	149
Аватары пользователей.....	152
Глава 11. Блоггинг	156
Отправка и отображение сообщений.....	156
Сообщения из блогов на страницах профилей.....	159
Постраничный вывод длинных списков сообщений.....	160
Создание фиктивных сообщений	161
Постраничное отображение данных	163
Виджет постраничного отображения	164
Форматирование текста сообщений с помощью Markdown и Flask-PageDown.....	167
Flask-PageDown.....	168
Обработка форматированного текста на сервере	169
Постоянные ссылки на сообщения	171
Редактор сообщений	173
Глава 12. Читающие и читаемые	176
Пересмотр отношений в базе данных	176
Отношение «многие ко многим»	177
Самоссылочные отношения.....	179
Усовершенствованные отношения «многие ко многим»	180
Читающие и читаемые на странице профиля	183
Запрос сообщений читаемых пользователей с помощью операции соединения.....	186
Отображение сообщений читаемых пользователей на главной странице	189
Глава 13. Комментарии пользователей	194
Представление комментариев в базе данных	194
Отправка и отображение комментариев.....	196
Модерирование комментариев	198
Глава 14. Прикладные программные интерфейсы	204
Введение в REST.....	204
Все сущее является ресурсами.....	205
Методы запросов	206
Содержимое запросов и ответов	207
Поддержка версий.....	208

Веб-службы RESTful на основе Flask	209
Создание макета API	209
Обработка ошибок	210
Аутентификация пользователей с помощью Flask-HTTPAuth	212
Аутентификация на основе маркеров	214
Преобразование ресурсов в формат JSON и обратно	217
Реализация конечных точек ресурсов	220
Разбивка больших коллекций ресурсов на страницы	223
Тестирование веб-служб с помощью HTTPie	224
 Часть III. Последняя миля	 226
 Глава 15. Тестирование	 227
Получение отчета о степени охвата кода тестированием	227
Тестовый клиент Flask	231
Тестирование веб-приложений	231
Тестирование веб-служб	235
Сквозное тестирование с помощью Selenium	237
Насколько это необходимо?	241
 Глава 16. Производительность	 243
Регистрация медленных запросов к базе данных	243
Профилирование исходного кода	245
 Глава 17. Развертывание	 247
Порядок развертывания	247
Журналирование ошибок во время эксплуатации	248
Развертывание в облаке	249
Платформа Heroku	250
Подготовка приложения	250
Тестирование с помощью Foreman	256
Включение безопасного протокола HTTP с помощью Flask-SSLify	257
Развертывание командой git push	260
Просмотр журналов	260
Развертывание и обновление	261
Традиционный хостинг	261
Настройка сервера	261

Импортирование переменных окружения	262
Настройка журналирования.....	263
Глава 18. Дополнительные ресурсы	264
Использование интегрированной среды разработки	264
Поиск расширений для Flask.....	265
Участие в разработке Flask	266
Предметный указатель	267
Об авторе	270
Выходные данные	271

Предисловие

Flask отличается от других фреймворков тем, что позволяет разработчику сесть на место водителя и получить полный контроль над его приложением. Возможно, вам уже доводилось слышать фразу: «бороться с фреймворком». Такое происходит с большинством фреймворков при попытке реализовать нестандартное решение. Это может быть попытка использовать другой механизм управления базами данных или иной способ аутентификации пользователей. Отклонение от пути, предусмотренного разработчиками фреймворка, приносит массу неприятностей.

Фреймворк Flask не такой. Хотите использовать реляционную базу данных? Отлично, Flask поддерживает их. Предпочитаете базу данных NoSQL? Нет проблем, Flask способен работать и с ними. Хотите использовать механизм хранения данных собственной разработки или вообще решили обойтись без базы данных? Замечательно. Используя Flask, можно выбирать, какие его компоненты будут применяться в приложении, и даже писать собственные. Все в ваших руках!

Такая свобода объясняется тем, что фреймворк Flask изначально задумывался расширяемым. Он включает надежное ядро, обеспечивающее основные функциональные возможности, необходимые в любых веб-приложениях, и предполагает, что остальное будет реализовано сторонними разработчиками в форме расширений и, конечно же, вами.

В этой книге я расскажу о моих подходах к разработке веб-приложений с применением фреймворка Flask. Я не претендую на истину в последней инстанции, и вы должны рассматривать мои слова как рекомендации, а не как непреложное руководство к действию.

В большинстве книг, посвященных программированию, приводятся короткие фрагменты кода, демонстрирующие разные особенности обсуждаемых технологий по отдельности, оставляя за рамками «связывающий» код, необходимый для объединения этих разных фрагментов в действующее приложение. Я предпочел избрать иной подход. Все примеры, представленные в этой книге, являются частями единого приложения – сначала очень простого, а затем постепенно усложняющегося в каждой последующей главе. В начале пути это приложение состоит всего из нескольких строк кода, а к концу оно превращается в полноценное приложение социальных сетей и блога.

Кому адресована эта книга

Чтобы извлечь выгоду из этой книги, необходимо иметь некоторый опыт программирования на языке Python. Эта книга не предполагает предварительного знакомства с фреймворком Flask, но вы должны быть знакомы с такими понятиями языка Python, как пакеты, модули, функции, декораторы и объектно-ориентированное программирование. Нелишними будут также знакомство с исключениями и умение диагностировать проблемы по трассировке стека.

Следуя за примерами в книге, вы проведете немало времени в командной строке, поэтому вам также потребуются навыки работы в командной строке своей операционной системы.

Современные веб-приложения немыслимы без использования HTML, CSS и JavaScript. Приложение, разрабатываемое на протяжении всей книги, также использует их, но в самом тексте книги не приводятся подробности, касающиеся этих технологий. Знакомство с этими языками совершенно необходимо, если предполагаете писать законченные приложения, не прибегая к помощи разработчика, искушенного в клиентских технологиях.

Исходные тексты приложения, написанного для этой книги, я выложил в открытый доступ на сайте GitHub. Несмотря на то что GitHub позволяет загружать приложения в виде обычных ZIP- или TAR-архивов, я настоятельно рекомендую установить клиента Git и познакомиться с системой управления версиями, хотя бы с основными командами, позволяющими извлекать различные версии приложения непосредственно из репозитория. Короткий список команд, которые вам понадобятся, представлен в разделе «Использование программного кода примеров» ниже. Вы наверняка пожелаете применить систему управления версиями для собственных проектов, поэтому используйте эту книгу как повод изучить Git!

Наконец, не следует рассматривать эту книгу как полное и исчерпывающее руководство по фреймворку Flask. Здесь охватываются многие его особенности, но не упускайте из виду официальную документацию с описанием фреймворка¹.

Структура книги

Эта книга делится на три части.

¹ <http://flask.pocoo.org/>.

В первой части «Введение в Flask» исследуются основы разработки веб-приложений с применением фреймворка Flask и некоторых его расширений:

- глава 1 описывает установку и настройку фреймворка Flask;
- глава 2 погружает читателя в разработку простого приложения с помощью Flask;
- глава 3 знакомит с поддержкой шаблонов в приложениях Flask;
- глава 4 – с поддержкой веб-форм;
- глава 5 – с поддержкой баз данных;
- глава 6 – с поддержкой электронной почты;
- глава 7 описывает типичную структуру крупных и средних приложений.

Во второй части «Пример: приложение социального блогинга» описывается разработка открытого приложения социальных сетей и блогинга на основе фреймворка Flask, которое я создал для этой книги:

- глава 8 описывает реализацию системы аутентификации пользователей;
- глава 9 – реализацию системы ролей и привилегий пользователей;
- глава 10 – реализацию страниц профилей пользователей;
- глава 11 – создание интерфейса для блогинга;
- глава 12 – реализацию поддержки последователей;
- глава 13 – реализацию поддержки комментариев пользователей;
- глава 14 описывает реализацию прикладного программного интерфейса (Application Programming Interface, API).

В третьей части «Последняя миля» раскрываются некоторые важные задачи, не связанные с разработкой приложений непосредственно, которые необходимо решать перед публикацией приложения:

- глава 15 подробно описывает разные стратегии модульного тестирования;
- глава 16 представляет обзор приемов анализа производительности;
- глава 17 описывает варианты развертывания приложений на основе Flask в традиционных облачных окружениях;
- глава 18 перечисляет дополнительные источники информации.

Как работать с примерами программного кода

Примеры программного кода, описываемые в этой книге, доступны на сайте GitHub по адресу: <https://github.com/miguelgrinberg/flasky>.

История изменений в этом репозитории в точности соответствует порядку представления понятий в этой книге. Рекомендуется извлекать код из репозитория, начиная с самых ранних версий, и затем двигаться вперед по списку изменений, по мере чтения книги. Желаящим сайт GitHub предоставляет возможность загружать каждое изменение в виде ZIP- или TAR-архива.

Если кто-то предпочтет извлекать исходный код примеров из репозитория Git, ему придется установить программу-клиента Git, которую можно бесплатно загрузить по адресу: <http://git-scm.com>. Ниже приводится команда, которая загрузит исходный код примеров из репозитория:

```
$ git clone https://github.com/miguelgrinberg/flasky.git
```

Команда `git clone` загрузит исходный код в каталог *flasky*, который будет создан в текущем каталоге. Этот каталог содержит не только исходный код; в него будет скопирован весь репозиторий Git с полной историей изменений приложения.

В первой главе будет предложено *извлечь* (*check out*) первоначальную версию приложения, и в соответствующих местах в книге будет предлагаться переходить дальше по истории изменений. Для извлечения исходного кода и перемещения по истории изменений используется команда `git checkout`, например:

```
$ git checkout 1a
```

Здесь *1a* в команде – это *тег*, именованная точка в истории развития проекта. Репозиторий размечен такими точками в соответствии с главами в книге, то есть тег *1a* соответствует начальной версии файлов приложения, описываемой в главе 1. Большинству глав соответствует более одного тега. Например, в репозитории имеются теги *5a*, *5b* и т. д., соответствующие последовательности версий, представленных в главе 5.

Помимо извлечения файлов из репозитория, может также потребоваться выполнить некоторые настройки. Например, иногда бывает необходимо установить дополнительные пакеты Python или внести

изменения в базу данных. Я буду сообщать об этом в соответствующие моменты.

В процессе чтения вам не придется изменять исходные файлы приложения, но если вы сделаете это, Git не позволит вам извлечь из репозитория следующую версию, чтобы не затереть локальных изменений. В этой ситуации, чтобы перейти к следующей версии, необходимо вернуть файлы в исходное состояние. Проще всего это сделать с помощью команды `git reset`:

```
$ git reset --hard
```

Эта команда затрет все локальные изменения, поэтому, если вы не желаете потерять их, сохраните резервные копии файлов перед запуском этой команды.

Иногда бывает желательно обновлять локальную копию репозитория из внешнего, расположенного на сервере GitHub, куда разработчики могут добавлять улучшения, расширения и исправления ошибок. Сделать это можно следующей последовательностью команд:

```
$ git fetch --all
$ git fetch --tags
$ git reset --hard origin/master
```

Команды `git fetch` обновят историю изменений и список тегов в локальной копии репозитория в соответствии с удаленным репозиторием GitHub, но ни одна из них не оказывает влияния на сами файлы с исходным кодом. Обновление этих файлов осуществляется командой `git reset`. И снова имейте в виду, что команда `git reset` затрет все локальные изменения, которые вы могли выполнить.

Еще одной полезной операцией является получение различий между двумя версиями приложения. Она может пригодиться всем, кто пожелает лучше разобраться в изменениях. Выполняется эта операция командой `git diff`. Например, чтобы увидеть различия между ревизиями 2a и 2b:

```
$ git diff 2a 2b
```

Различия будут сохранены в формате файла «заплаты» (patch). Этот формат не очень удобен для просмотра изменений, особенно если прежде вам не приходилось сталкиваться с подобными файлами. Гораздо удобнее в этом отношении инструменты с графическим интерфейсом, предлагаемые проектом GitHub. Например, различия между ревизиями 2a и 2b можно посмотреть на странице GitHub: <https://github.com/miguelgrinberg/flasky/compare/2a...2b>.

Использование программного кода примеров

Данная книга призвана оказать вам помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить значительные по объему части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Flask Web Development by Miguel Grinberg (O'Reilly). Copyright 2014 Miguel Grinberg, 978-1-449-3726-2».

Если вам кажется, что использование примеров из книги нарушает правила добросовестного применения или условия, сформулированные выше, обращайтесь по адресу permissions@oreilly.com.

Типографские соглашения

В книге приняты следующие соглашения:

Курсив

Применяется для выделения новых терминов, имен файлов и их расширений.

Моноширинный шрифт

Применяется для представления листингов программного кода, а также элементов программ, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

Моноширинный жирный

Используется для выделения команд или других фрагментов текста, которые вводятся пользователем.

Моноширинный курсив

Используется для выделения текста, который должен замещаться значениями, предоставляемыми пользователями, или значениями, определяемыми контекстом.



Так выделяются советы и предложения.



Так выделяются примечания общего характера.



Так выделяются предупреждения и предостережения.

Safari® Books Online

Safari Books Online (www.safaribooksonline.com) – это виртуальная библиотека, содержащая авторитетную информацию в виде книг и видеоматериалов, созданных ведущими специалистами в области технологий и бизнеса.

Профессионалы в области технологий, разработчики программного обеспечения, веб-дизайнеры, а также бизнесмены и творческие работники используют Safari Books Online как основной источник информации для проведения исследований, решения проблем, обучения и подготовки к сертификационным испытаниям.

Библиотека Safari Books Online предлагает широкий выбор продуктов и тарифов для организаций, правительственных учреждений и физических лиц. Подписчики имеют доступ к поисковой базе данных, содержащей информацию о тысячах книг, видеоматериалов и рукописей от таких издателей, как O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, и десятков других. За подробной информацией о Safari Books Online обращайтесь по адресу: <http://www.safaribooksonline.com/>.

Ждем ваших отзывов

Направляйте свои комментарии и вопросы, касающиеся этой книги, по обычной почте:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах Америки или в Канаде)
707-829-0515 (международный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги: <http://www.bit.ly/flask-web-dev>.

Свои пожелания и вопросы технического характера отправляйте по адресу: bookquestions@oreilly.com.

Дополнительную информацию о наших книгах, курсах, конференциях и новостях вы найдете на веб-сайте издательства O'Reilly: <http://www.oreilly.com>.

Ищите нас на Facebook: <http://facebook.com/oreilly>.

Читайте нас в Твиттере: <http://twitter.com/oreillymedia>.

Смотрите нас на YouTube: <http://www.youtube.com/oreillymedia>.

Благодарности

Едва ли я смог бы написать эту книгу в одиночку. Огромную помощь оказали мне моя семья, коллеги, старые друзья и новые, с которыми я познакомился попутно.

Я хотел бы поблагодарить Брендана Кехлера (Brendan Kohler) за подробный технический отзыв и за его помощь в работе над главой о прикладных программных интерфейсах. Я также многим обязан Дэвиду Баумголду (David Baumgold), Тодду Бранхоффу (Todd Brunhoff), Сесиль Рок (Cecil Rock) и Мэтью Хьюгасу (Matthew Hugues), просматривавшим рукопись на разных стадиях готовности и дававшим мне очень полезные советы относительно того, о чем писать и как организовать материал.

Разработка примеров для этой книги потребовала значительных усилий. Я благодарен Даниэлю Хофманну (Daniel Hofmann), который просмотрел весь код приложения и внес множество советов по его улучшению. Я также хочу сказать спасибо моему сыну-подростку Дилану Гринбергу (Dylan Grinberg), который забросил свою игру на несколько выходных и помогал мне тестировать код на разных платформах.

Издательство O'Reilly запустило замечательную программу под названием «Early Release» (ранний выпуск), позволяющую нетерпе-

ливым читателям получить доступ к книге в процессе ее создания. Некоторые из читателей первых выпусков моей книги приняли участие в полезных обсуждениях и обмене опытом, что помогло существенно улучшить ее. Мне хотелось бы поблагодарить Сандипа Гупту (Sandeep Gupta), Дэна Кэрона (Dan Caron), Брайана Уисти (Brian Wisti) и Коди Скотта (Cody Scott) за их помощь в работе над этой книгой.

Сотрудники O'Reilly Media всегда благоволили мне. Прежде всего я хотел бы поблагодарить моего прелестного редактора Меган Бланшетт (Meghan Blanchette) за ее поддержку, советы и помощь с самой первой нашей встречи. Благодаря Мег я получил незабываемый опыт работы над моей первой книгой.

В заключение я хотел бы поблагодарить замечательное сообщество пользователей Flask.

Часть I



Введение в Flask

Установка


Flask¹ – это очень маленький фреймворк, такой маленький, что его часто называют «микрофреймворк». Он настолько мал, что после не продолжительного знакомства с ним вы сможете читать и понимать его исходный код.

Но быть маленьким не означает давать меньше, чем дают другие фреймворки. Flask изначально проектировался как расширяемый фреймворк – он имеет монолитное ядро, реализующее основные службы, а все остальное поддерживается посредством расширений. Поскольку вы можете выбирать только необходимые пакеты, в результате получается достаточно ограниченный комплект программных средств, не поддающийся неконтролируемому разбуханию и в точности соответствующий вашим потребностям.

Фреймворк Flask имеет две основные зависимости. Подсистемы маршрутизации, отладки и интерфейса WSGI (Web Server Gateway Interface) заимствованы из проекта Werkzeug, а поддержка шаблонов – из проекта Jinja2. Проекты Werkzeug и Jinja2 были созданы основными разработчиками Flask.

Flask не имеет встроенной поддержки доступа к базам данных, проверки веб-форм, аутентификации пользователей или других высокоуровневых задач. Существует и множество иных ключевых служб, необходимых большинству веб-приложений и доступных в виде расширений, интегрируемых с основными пакетами. Как разработчик вы можете выбирать расширения, лучше всего подходящие для вашего проекта, или даже писать собственные, если чувствуете, что вам это удастся лучше. Этим Flask отличается от крупных фреймворков, где выбор уже сделан за вас, который очень сложно изменить, если вообще возможно.

В данной главе вы узнаете, как установить Flask. Единственное предварительное условие – наличие компьютера с установленным языком Python.

 Код примеров был протестирован с Python 2.7 и Python 3.3, поэтому рекомендуется использовать одну из этих двух версий.

¹ <http://flask.pocoo.org>.

Использование виртуальных окружений


Самый удобный способ установки Flask – воспользоваться виртуальным окружением. Виртуальное окружение – это отдельная копия интерпретатора Python, в которую можно установить пакеты, не оказывая влияния на глобальный интерпретатор Python, установленный в системе.

Виртуальные окружения удобны тем, что предотвращают конфликты между версиями и захламление системного интерпретатора Python посторонними пакетами. Создание виртуального окружения для каждого приложения гарантирует доступность только необходимых пакетов, при этом системная установка интерпретатора остается чистой и служит всего лишь ресурсом для создания виртуальных окружений. Как дополнительное преимущество виртуальные окружения не требуют наличия у вас прав администратора.

Виртуальные окружения можно создавать с помощью сторонней утилиты `virtualenv`. Чтобы проверить наличие утилиты в системе, введите следующую команду:

```
$ virtualenv --version
```

Если в ответ вы получите сообщение об ошибке, значит, вам придется установить утилиту.

 Python 3.3 включает встроенную поддержку виртуальных окружений в виде модуля `venv` и команды `pyenv`. Команда `pyenv` может использоваться вместо утилиты `virtualenv`, но имейте в виду, что виртуальные окружения, созданные с помощью `pyenv` из установки Python 3.3, не включают утилиту `pip`, которую необходимо установить вручную. Это ограничение устранено в Python 3.4, где `pyenv` можно использовать как полноценную замену `virtualenv`.

Большинство дистрибутивов Linux позволяет установить `virtualenv` в виде отдельного пакета. Например, пользователи Ubuntu могут установить эту утилиту командой:

```
$ sudo apt-get install python-virtualenv
```

В Mac OS X ее можно установить командой `easy_install`:


```
$ sudo easy_install virtualenv
```

Пользователям Microsoft Windows и других операционных систем, официально не поддерживающих пакета `virtualenv`, может потребоваться пройти более сложную процедуру установки.

Откройте в браузере адрес <https://bitbucket.org/pypa/setuptools> – домашнюю страницу дистрибутива `setuptools`. На этой странице най-

дите ссылку для загрузки сценария установки. Этот сценарий имеет имя `ez_setup.py`. Сохраните файл сценария во временной папке на своем компьютере, затем выполните следующие команды в этой папке:

```
$ python ez_setup.py
$ easy_install virtualenv
```

 Предыдущие команды должны выполняться с привилегиями администратора. Для этого в Microsoft Windows запустите командную строку, выбрав пункт **Run as Administrator** (Запуск от имени администратора). В Unix-подобных системах этим двум командам должна предшествовать команда `sudo`, или они должны вызываться с привилегиями пользователя `root`. После установки утилиты `virtualenv` можно вызывать с привилегиями обычного пользователя.

Теперь нужно создать папку, где будут храниться файлы с исходным кодом примеров, загруженные из репозитория на сайте GitHub. Как говорилось в разделе «Как работать с примерами программного кода» выше, проще всего загрузить файлы, извлекая их непосредственно из репозитория с помощью клиента Git. Следующие команды загрузят примеры из GitHub и инициализируют папку для версии «1a» — начальной версии приложения:

```
$ git clone https://github.com/miguelgrinberg/flasky.git
$ cd flasky
$ git checkout 1a
```

Следующий шаг — создание виртуального окружения Python внутри папки *flasky* с использованием команды `virtualenv`. Эта команда имеет единственный обязательный аргумент: имя виртуального окружения. Она создаст в текущем каталоге папку с выбранным именем и скопирует в нее все файлы, необходимые для образования виртуального окружения. Часто виртуальному окружению присваивается имя *venv*:

```
$ virtualenv venv
New python executable in venv/bin/python2.7
Also creating executable in venv/bin/python
Installing setuptools.....done.
Installing pip.....done.
```

Теперь внутри папки *flasky* у вас имеется папка *venv* с совершенно новым виртуальным окружением, содержащим собственный интерпретатор Python. Чтобы начать использовать виртуальное окружение, его необходимо «активировать». Пользующиеся командной оболочкой `bash` (это относится к пользователям Linux и Mac OS X) могут активировать виртуальное окружение командой:

```
$ source venv/bin/activate
```


Пользователи Microsoft Windows могут выполнить команду:

```
$ venv\Scripts\activate
```


После активации виртуального окружения каталог с копией интерпретатора Python будет добавлен в переменную окружения `PATH`, но это изменение носит временный характер; оно продолжает действовать только в текущем сеансе командной оболочки. Для напоминания, что вы активировали виртуальное окружение, команда активации изменяет строку приглашения к вводу в командной оболочке, включая в нее имя окружения:

```
(venv) $
```

По окончании работы с виртуальным окружением, чтобы вернуться к использованию глобального интерпретатора Python, введите команду `deactivate`.

Установка пакетов Python с помощью `pip`

Большинство пакетов Python устанавливается с помощью утилиты `pip`, которую команда `virtualenv` автоматически добавляет во все создаваемые виртуальные окружения. В момент активации виртуального окружения местоположение утилиты `pip` автоматически добавляется в переменную окружения `PATH`.

 Если виртуальное окружение было создано с помощью команды `pyvenv`, входящей в состав Python 3.3, утилиту `pip` придется установить вручную. Инструкции по установке доступны на веб-сайте <http://bit.ly/pip-install>. В версии Python 3.4 команда `pyvenv` устанавливает `pip` автоматически.

Чтобы установить Flask в виртуальное окружение, выполните команду:

```
(venv) $ pip install flask
```

Она установит фреймворк Flask и все его зависимости в виртуальное окружение. Чтобы убедиться в успешности установки, запустите интерпретатор Python и попробуйте импортировать фреймворк:

```
(venv) $ python
>>> import flask
>>>
```

Если на экране не появится сообщение об ошибке, можете поздравить себя: вы готовы перейти к следующей главе, где вы напишете свое первое веб-приложение.

Глава 2

Структура простого приложения

В этой главе вы познакомитесь с разными частями приложений на основе Flask, а также напишете и запустите свое первое веб-приложение.

Инициализация

Любое приложение на основе Flask должно создать *экземпляр приложения*. Веб-сервер будет передавать все запросы, принимаемые от клиентов, этому объекту через протокол, который называется Web Server Gateway Interface (WSGI). Экземпляр приложения – это объект класса Flask, который обычно создается так:

```
from flask import Flask
app = Flask(__name__)
```

Единственным обязательным аргументом конструктора класса Flask является имя главного модуля или пакета приложения. Для большинства приложений на Python это имя хранится в переменной `__name__`.



Аргумент `name`, который передается конструктору Flask приложения, часто является источником недопонимания для начинающих разработчиков. Фреймворк Flask использует этот аргумент для определения пути к корневому каталогу приложения, чтобы позднее с его помощью находить файлы ресурсов.

Далее мы увидим более сложные примеры инициализации, но для простых приложений этого вполне достаточно.


Маршруты и функции представлений

Клиенты, такие как веб-браузеры, отправляют *запросы* веб-серверу, который, в свою очередь, отправляет их экземпляру приложения на основе Flask. Экземпляр приложения должен определить, какой код

должен быть выполнен для обработки обращения к адресу URL в запросе, поэтому он должен хранить отображение адресов URL в функции на языке Python. Ассоциацию между адресом URL и функцией называют *маршрутом (route)*.


Проще всего определить маршрут в приложении на основе Flask с помощью декоратора `app.route`, экспортируемого экземпляром приложения. Этот декоратор регистрирует декорируемую функцию как маршрут. Ниже показан пример объявления маршрута с помощью декоратора:

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

 Декораторы – это стандартная особенность языка Python, они способны изменять поведение функций. Часто декораторы используются для регистрации функций в качестве обработчиков событий.

В предыдущем примере функция `index()` регистрируется как обработчик запросов к корневому адресу URL приложения. Если бы это приложение было развернуто на сервере с доменным именем *www.example.com*, тогда попытка открыть страницу с адресом *http://www.example.com* в браузере привела бы к вызову `index()` на сервере. Возвращаемое значение этой функции называется *ответом (response)* – это то, что получит клиент. Если клиентом является веб-браузер, ответ будет интерпретироваться как документ, который требуется отобразить на экране.

Функции, такие как `index()`, называют *функциями представления (view functions)*. Ответ, возвращаемый функцией представления, может быть простой строкой с разметкой HTML или иметь более сложную форму, как будет показано позднее.

 Встраивание строк ответов непосредственно в программный код на Python усложняет его сопровождение, и в данном случае это было сделано, только чтобы познакомить вас с понятием ответов. Правильный способ создания ответов будет представлен в главе 3.

Если вы внимательно рассмотрите структуру некоторых URL-служб, которыми пользуетесь ежедневно, вы заметите, что многие из них имеют переменные разделы. Например, URL к странице профиля на Facebook имеет вид: **`http://www.facebook.com/<имя-пользователя>`**, то есть имя пользователя является частью URL. Flask поддерживает подобные адреса URL с помощью специального

синтаксиса в декораторе маршрута. Например, ниже определяется маршрут, включающий переменный компонент `name`:

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, %s!</h1>' % name
```

Фрагмент в угловых скобках — это переменная часть, то есть любой URL, совпадающий с постоянной частью, будет отображен в этот маршрут. При вызове функции представления Flask передаст ей динамический компонент в виде аргумента. В предыдущем примере аргумент `name` используется функцией представления для создания персонализированного ответа.

Переменные элементы в маршрутах по умолчанию интерпретируются как строки, но могут также иметь другие типы. Например, маршрут `/user/<int:id>` будет соответствовать адресам URL, содержащим целое число в переменном сегменте `id`. Flask поддерживает в маршрутах типы `int`, `float` и `path`. Тип `path` также является строковым, но при его интерпретации символы косой черты не рассматриваются как разделители и включаются в переменный компонент.

Запуск сервера

Экземпляр приложения имеет метод `run`, который запускает интегрированный веб-сервер, используемый для разработки:

```
if __name__ == '__main__':
    app.run(debug=True)
```

Идиома `__name__ == '__main__'`, широко используемая в языке Python, гарантирует, что веб-сервер для разработки будет запускаться только при непосредственном выполнении сценария. Когда сценарий импортируется другим сценарием, предполагается, что родительский сценарий запустит другой сервер, поэтому вызов `app.run()` пропускается.

После запуска сервер входит в цикл ожидания и обработки запросов. Этот цикл продолжается, пока приложение не будет остановлено, например нажатием комбинации **Ctrl-C**.

Метод `app.run()` имеет несколько необязательных аргументов для настройки режима работы веб-сервера. В процессе разработки довольно удобно бывает включить режим отладки, в котором, кроме всего прочего, активируются *отладчик* и *перезагрузчик*. Для этого следует передать в аргументе `debug` значение `True`.



Веб-сервер, входящий в состав фреймворка Flask, не предназначен для использования в промышленном окружении. С настоящими, промышленными веб-серверами мы познакомимся в главе 17.

Законченное приложение

В предыдущих разделах мы познакомились с разными частями веб-приложения на основе Flask, и теперь пришло время написать такое приложение. Весь сценарий *hello.py* приложения состоит точно из трех частей, описанных выше и объединенных в один файл. Исходный код приложения приводится в примере 2.1.

Пример 2.1 ❖ hello.py: законченное приложение на основе фреймворка Flask

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

if __name__ == '__main__':
    app.run(debug=True)
```



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 2a` и получить эту версию приложения.

Чтобы запустить приложение, активируйте виртуальное окружение, созданное ранее. Убедитесь, что фреймворк Flask установлен. Запустите приложение командой:

```
(venv) $ python hello.py
* Running on http://127.0.0.1:5000/
* Restarting with reloader
```

Откройте веб-браузер и введите адрес **`http://127.0.0.1:5000/`** в адресной строке. На рис. 2.1 показано, как выглядит окно браузера после соединения с приложением.

Если ввести любой другой адрес URL, приложение не будет знать, как его обработать, и вернет код ошибки 404 – широко известный код, который возвращается при попытке перейти к несуществующей странице.

В примере 2.2 приводится расширенная версия приложения, в которой добавлен второй маршрут с переменной частью. При обращении к этому адресу URL приложение выведет персонализированное приветствие.

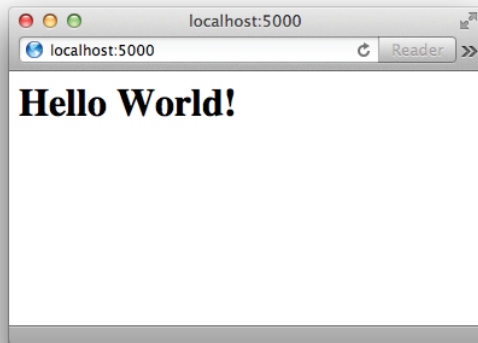


Рис. 2.1 ❖ Flask-приложение hello.py

Пример 2.2 ❖ hello.py: Flask-приложение с динамическим маршрутом

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, %s!</h1>' % name

if __name__ == '__main__':
    app.run(debug=True)
```

💡 Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 2b` и получить эту версию приложения.

Чтобы протестировать работу динамического маршрута, запустите приложение и откройте в браузере страницу с адресом: **http://localhost:5000/user/Dave**. Приложение ответит приветствием, включив в него переменную часть маршрута `name`. Попробуйте подставлять разные имена, чтобы убедиться, что функция представления всегда возвращает ответ, содержащий указанное вами имя. Пример показан на рис. 2.2.

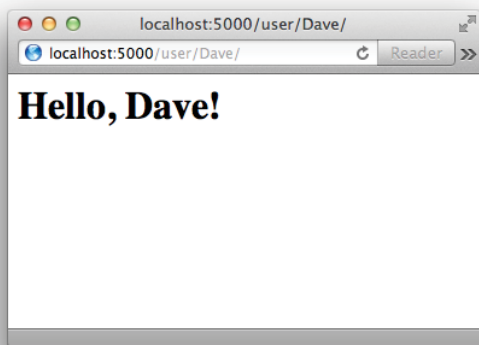


Рис. 2.2 ❖ Динамический маршрут

Цикл запрос–ответ

Теперь, поэкспериментировав с простым приложением на основе Flask, у вас может появиться желание узнать больше о том, как работает Flask. В следующих разделах описываются некоторые архитектурные аспекты фреймворка.

Контексты приложения и запроса

Когда фреймворк Flask принимает запрос от клиента, он должен обеспечить доступ к нескольким объектам из функции представления, которая будет обрабатывать запрос. Хорошим примером может служить *объект запроса*, содержащий HTTP-запрос, отправленный клиентом.

Очевидный способ обеспечить доступ к объекту запроса – передать его в виде аргумента, но для этого потребовалось бы, чтобы каждая функция представления в приложении принимала дополнительный аргумент. Ситуация становится еще более сложной, если принять во внимание, что объект запроса – не единственный объект, доступ к которому должна иметь функция представления, чтобы суметь обработать запрос.


Чтобы избежать захламления функций представления большим числом параметров, которые могут потребоваться или нет, и времен-

но обеспечить глобальный доступ к отдельным объектам, Flask использует *контексты*. Благодаря контекстам имеется возможность писать функции представления, такие как ниже:

```
from flask import request

@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Your browser is %s</p>' % user_agent
```

Обратите внимание, что в этой функции используется объект `request`, как если бы он был глобальной переменной. В действительности `request` не может быть глобальной переменной, если учесть, что многопоточный сервер способен одновременно обрабатывать несколько запросов от разных клиентов, то есть каждый поток должен видеть свой экземпляр объекта запроса. Поддержка контекстов в фреймворке Flask позволяет обеспечить глобальный доступ к некоторым переменным из потока выполнения, не оказывая влияния на другие потоки.

 Поток выполнения – это наименьшая последовательность инструкций, которая может выполняться независимо. Нет ничего необычного, когда в рамках процесса действует сразу несколько активных потоков выполнения, иногда совместно использующих ресурсы, такие как память или дескрипторы файлов. Многопоточные веб-серверы поддерживают пулы потоков и при получении очередного запроса выбирают поток из пула для обработки этого запроса.

В фреймворке Flask имеются два контекста: *контекст приложения* и *контекст запроса*. В табл. 2.1 перечислены переменные, экспортируемые каждым из этих контекстов.

Таблица 2.1. Переменные, экспортируемые контекстами

Имя переменной	Контекст	Описание
<code>current_app</code>	Контекст приложения	Экземпляр активного приложения
<code>g</code>	Контекст приложения	Объект, который может использоваться приложением как временное хранилище в процессе обработки запроса. Эта переменная сбрасывается в исходное состояние для каждого нового запроса
<code>request</code>	Контекст запроса	Объект запроса, включающий содержимое HTTP-запроса, отправленного клиентом
<code>session</code>	Контекст запроса	Сеанс пользователя – словарь, который может использоваться приложением для сохранения значений между запросами

Flask активирует контексты приложения и запроса перед началом обработки запроса и удаляет их после обработки. Когда активируется контекст приложения, потоку выполнения становятся доступны переменные `current_app` и `g`. Аналогично, когда активируется контекст запроса, становятся доступны переменные `request` и `session`. Попытка обратиться к этим переменным за пределами активного контекста приложения или запроса приведет к ошибке. Эти четыре переменные контекстов детально будут рассматриваться в следующих главах, поэтому не волнуйтесь, если пока вам не понятно, где они могут пригодиться.

Следующий сеанс работы в интерактивной оболочке Python демонстрирует, как работают контексты:

```
>>> from hello import app
>>> from flask import current_app
>>> current_app.name
Traceback (most recent call last):
...
RuntimeError: working outside of application context
The Request-Response Cycle | 13
>>> app_ctx = app.app_context()
>>> app_ctx.push()
>>> current_app.name
'hello'
>>> app_ctx.pop()
```

В этом примере первая попытка обратиться к `current_app.name` потерпела неудачу, так как в данный момент отсутствовал активный контекст приложения, но после его активации вторая попытка была выполнена благополучно. Обратите внимание, что контекст приложения можно приобрести вызовом метода `app.app_context()` экземпляра приложения.

Обработка запросов

Получив запрос от клиента, приложение должно определить, какую функцию представления вызвать для его обслуживания. Для этого Flask ищет URL запроса в *карте адресов URL* внутри приложения, определяющей соответствие между адресами URL и функциями представления. Этот ассоциативный массив конструируется фреймворком с помощью декораторов `app.route` или эквивалентных недекорированных вызовов `app.add_url_rule()`.

Чтобы увидеть, на что похожа карта адресов URL в приложении на основе Flask, можно исследовать карту, созданную приложением

hello.py, в интерактивной оболочке Python. Для этого активируйте виртуальное окружение и выполните следующие команды:

```
(venv) $ python
>>> from hello import app
>>> app.url_map
Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,
     <Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
     <Rule 'user/<name>' (HEAD, OPTIONS, GET) -> user>])
```

Маршруты `/` и `/user/<name>` были определены с помощью декоратора `app.route`. Маршрут `/static/<filename>` – это специальный маршрут, добавленный фреймворком Flask, чтобы обеспечить доступ к статическим файлам. Подробнее о статических файлах рассказывается в главе 3.

Элементы `HEAD`, `OPTIONS`, `GET` в карте адресов URL – это HTTP-методы запросов, которые обрабатываются маршрутом. Flask привязывает методы к каждому маршруту, чтобы запросы к тому же URL, но выполненные другими методами, могли обрабатываться другими функциями представления. Методы `HEAD` и `OPTIONS` назначаются фреймворком Flask автоматически, поэтому фактически можно сказать, что в этом приложении с методом `GET` связаны три маршрута. О назначении других HTTP-методов для маршрутов вы узнаете в главе 4.

Обработчики событий жизненного цикла

Иногда бывает желательно выполнить некоторые операции перед обработкой каждого запроса. Например, перед обработкой каждого запроса может быть необходимо создать соединение с базой данных или аутентифицировать пользователя, выполнившего запрос. Вместо дублирования кода в начале каждой функции представления Flask дает возможность зарегистрировать функции для вызова до или после передачи запроса функции представления.

Регистрация обработчиков событий жизненного цикла выполняется с помощью декораторов. Ниже перечислены четыре декоратора, поддерживаемых фреймворком Flask:

- `before_first_request`: регистрирует функцию для вызова перед обработкой первого запроса;
- `before_request`: регистрирует функцию для вызова перед обработкой каждого запроса;
- `after_request`: регистрирует функцию для вызова после обработки каждого запроса, если не возникло необработанных исключений;

- `teardown_request`: регистрирует функцию для вызова после обработки каждого запроса, если возникло необработанное исключение.

Часто, чтобы обеспечить доступность одних и тех же данных в функциях-обработчиках и функциях представления, используется переменная `g` контекста приложения. Например, обработчик `before_request` может загружать информацию о зарегистрированном пользователе из базы данных и сохранять ее в `g.user`. Позднее, когда будет вызвана функция представления, она сможет извлечь необходимую информацию из этого поля.

Примеры обработчиков событий жизненного цикла запросов будут показаны в будущих главах, поэтому не волнуйтесь, если что-то пока остается для вас непонятным.

Ответы

Вызывая функцию представления, фреймворк Flask ожидает, что она вернет ответ на запрос. В большинстве случаев ответ – это простая строка, отправляемая обратно клиенту в виде HTML-страницы.

Но протокол HTTP требует, чтобы в ответ на запрос возвращалась не только строка. Очень важной частью HTTP-ответа является *код состояния*. По умолчанию Flask устанавливает код состояния равным 200, который указывает, что запрос был обработан благополучно.

Когда необходимо вернуть иной код состояния, функция представления может установить его во втором возвращаемом значении, после строки ответа. Например, следующая функция представления возвращает код состояния 400, соответствующий недопустимому запросу:

```
@app.route('/')
def index():
    return '<h1>Bad Request</h1>', 400
```

Ответы, возвращаемые функциями представления, могут также содержать третий аргумент – словарь заголовков, которые должны быть добавлены в HTTP-ответ. Это редко требуется, тем не менее пример возврата заголовков вы увидите в главе 14.

Вместо кортежа с одним, двумя или тремя значениями функции представления могут возвращать объект `Response`. Функция `make_response()` принимает один, два или три аргумента – те же значения, которые может вернуть функция представления, – и возвращает объект `Response`. Иногда полезно выполнять данное преобразование

внутри функции представления и затем использовать методы объекта ответа для дальнейшей его настройки. Следующий пример создает объект ответа и затем устанавливает в нем cookie:

```
from flask import make_response

@app.route('/')
def index():
    response = make_response('<h1>This document carries a cookie!</h1>')
    response.set_cookie('answer', '42')
    return response
```

Существует специальный тип ответа, который называется *перенаправлением* (*redirect*). Этот ответ не включает документ страницы; он просто определяет новый адрес URL для браузера, откуда следует загрузить новую страницу. Прием перенаправления часто используется в веб-формах, с которыми вы познакомитесь в главе 4.

Обычно ответ-перенаправление включает код состояния 302 и адрес URL в заголовке Location. Такой ответ можно сформировать, вернув из функции представления три значения или объект Response, но из-за того, что необходимость перенаправления возникает достаточно часто, в состав фреймворка Flask была включена вспомогательная функция `redirect()`, создающая такой ответ:

```
from flask import redirect

@app.route('/')
def index():
    return redirect('http://www.example.com')
```

Другой специальный ответ можно сформировать с помощью функции `abort`, используемой для обработки ошибок. Следующий пример возвращает код состояния 404, если динамический аргумент `id`, переданный в URL, не представляет известного пользователя:

```
from flask import abort

@app.route('/user/<id>')
def get_user(id):
    user = load_user(id)
    if not user:
        abort(404)
    return '<h1>Hello, %s</h1>' % user.name
```

Обратите внимание, что функция `abort` не возвращает управление вызвавшей ее функции, а передает его веб-серверу, возбуждая исключение.

Расширения Flask

Фреймворк Flask изначально проектировался как расширяемый. В него преднамеренно не включались такие важные функциональные особенности, как поддержка баз данных и аутентификации пользователей, оставляя за вами свободу выбирать пакеты, лучше соответствующие вашим потребностям, или писать свои.

Существует огромное разнообразие *расширений* для самых разных целей, созданных сообществом, а если их окажется недостаточно, можно использовать любые стандартные пакеты или библиотеки для Python. Чтобы дать вам представление о способах интеграции расширений в приложения, в следующем разделе приводится реализация расширения для *hello.py*, которое добавляет в приложение поддержку параметров командной строки.

Поддержка параметров командной строки с помощью Flask-Script

Веб-сервер, встроенный в фреймворк Flask для нужд разработки, поддерживает множество параметров настройки, но единственный способ изменить их – передать в виде аргументов в вызов `app.run()` внутри сценария. Это не очень удобно; в идеале хотелось бы иметь возможность передавать параметры настройки через аргументы командной строки.

Flask-Script – это расширение для Flask, реализующее парсер командной строки. Оно включает поддержку ряда универсальных параметров, а также нескольких нестандартных команд.

Установить расширение можно с помощью утилиты `pip`:

```
(venv) $ pip install flask-script
```

В примере 2.3 демонстрируются изменения, необходимые, чтобы добавить анализ командной строки в приложение *hello.py*.

Пример 2.3 ❖ hello.py: использование расширения Flask-Script

```
from flask.ext.script import Manager
manager = Manager(app)

# ...

if __name__ == '__main__':
    manager.run()
```

Расширения, созданные специально для Flask, доступны в пространстве имен `flask.ext`. Расширение `Flask-Script` экспортирует класс `Manager`, который импортируется из `flask.ext.script`.

Метод инициализации этого расширения является типичным для большинства расширений: экземпляр главного класса инициализируется передачей экземпляра приложения конструктору. Затем созданный объект используется там, где необходимы функциональные возможности расширения. В данном случае для запуска сервера используется метод `manager.run()` объекта расширения, поддерживающего парсинг командной строки.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 2c` и получить эту версию приложения.

Благодаря внесенным изменениям приложение получило поддержку базового набора параметров командной строки. Если теперь попытаться выполнить сценарий *hello.py*, он выведет сообщение с описанием порядка использования:

```
$ python hello.py
usage: hello.py [-h] {shell,runserver} ...

positional arguments:
  {shell,runserver}
    shell              Runs a Python shell inside Flask application context.
    runserver          Runs the Flask development server i.e. app.run()

optional arguments:
  -h, --help          show this help message and exit
```

Команда `shell` применяется для запуска интерактивного сеанса Python в контексте приложения. Этот сеанс можно использовать для выполнения профилактических работ, тестирования или отладки.

Команда `runserver`, как следует из ее имени, запускает веб-сервер. Команда `python hello.py runserver` запустит веб-сервер в отладочном режиме. Кроме команд, поддерживается также множество дополнительных параметров:

```
(venv) $ python hello.py runserver --help
usage: hello.py runserver [-h] [-t HOST] [-p PORT] [--threaded]
                        [--processes PROCESSES] [--passthrough-errors] [-d]
                        [-r]
```

Runs the Flask development server i.e. `app.run()`

optional arguments:

```
-h, --help          show this help message and exit
-t HOST, --host HOST
-p PORT, --port PORT
--threaded
--processes PROCESSES
--passthrough-errors
-d, --no-debug
-r, --no-reload
```

Параметр `--host` может пригодиться, чтобы сообщить веб-серверу сетевой интерфейс, на котором он должен принимать запросы. По умолчанию встроенный веб-сервер принимает запросы с интерфейса `localhost`, то есть исходящие с того же компьютера, где выполняется сам веб-сервер. Следующая команда заставит веб-сервер принимать запросы также с интерфейса, подключенного к общедоступной сети:

```
(venv) $ python hello.py runserver --host 0.0.0.0
* Running on http://0.0.0.0:5000/
* Restarting with reloader
```

Теперь веб-сервер будет доступен с любого компьютера в сети по адресу: **`http://a.b.c.d:5000`**, где «`a.b.c.d`» – внешний IP-адрес компьютера, где действует веб-сервер.

В этой главе было представлено понятие ответа на запрос, но этим тема ответов не исчерпывается. Фреймворк Flask обладает очень мощной поддержкой создания ответов с применением *шаблонов* (*templates*), и этой важной теме посвящена следующая глава.

Ключом к разработке приложений, простых в сопровождении, является чистый и хорошо структурированный код. Примеры, которые вы видели до сих пор, слишком просты, чтобы продемонстрировать это, но проблема в том, что функции представлений преследуют две полностью разные цели, одна из которых скрыта от глаз.

Очевидной целью функций представления является создание ответов на запросы, как было показано в примерах в главе 2. Для простых запросов этого вполне достаточно, но в общем случае запросы вызывают изменение состояния приложения, и функции представления как раз являются тем местом, где производятся эти изменения.

Например, представьте ситуацию регистрации новой учетной записи на веб-сайте. Пользователь вводит адрес электронной почты и пароль в веб-форме и щелкает на кнопке **Submit** (Отправить). Запрос, включающий данные пользователя, передается веб-серверу, и фреймворк Flask передает его функции представления, выполняющей операцию регистрации. Этой функции требуется подключиться к базе данных, чтобы добавить новую учетную запись и затем сгенерировать ответ для отправки обратно браузеру. Эти две задачи формально называются *прикладной логикой* (или *бизнес-логикой*) и *логикой представления* соответственно.

Смешивание прикладной логики с логикой представления делает код сложным для понимания и сопровождения. Вообразите себе код, конструирующий большую HTML-таблицу путем объединения информации, извлеченной из базы данных, с литеральными строками, содержащими разметку HTML. Перемещение логики представления в *шаблоны* помогает упростить сопровождение приложения в будущем.

Шаблон – это файл, содержащий текст ответа с переменными-заполнителями для подстановки динамических данных, которые могут быть получены только в контексте запроса. Процесс подстановки фактических значений на место переменных-заполнителей и возврата получившегося ответа называется *отображением* (*rendering*). Для

решения задачи отображения шаблонов фреймворк Flask использует мощный механизм шаблонов *Jinja2*.

Механизм шаблонов Jinja2

В простейшем виде шаблон Jinja2 представляет собой текстовый файл с ответом. В примере 3.1 демонстрируется шаблон Jinja2, соответствующий ответу, генерируемому функцией представления `index()` из примера 2.1.

Пример 3.1 ❖ `templates/index.html`: шаблон Jinja2

```
<h1>Hello World!</h1>
```

Ответ, возвращаемый функцией представления `user()` из примера 2.2, имеет динамический компонент, представленный *переменной*. В примере 3.2 приводится шаблон, реализующий этот ответ.

Пример 3.2 ❖ `templates/user.html`: шаблон Jinja2

```
<h1>Hello, {{ name }}!</h1>
```

Отображение шаблонов

По умолчанию фреймворк Flask ищет шаблоны в подкаталоге *templates*, находящемся внутри папки приложения. Для следующей версии *hello.py* необходимо сохранить шаблоны из примеров выше в новой папке *templates*, в файлах с именами *index.html* и *user.html*.

Функции представления в приложении требуется изменить так, чтобы они отображали эти шаблоны. Необходимые изменения демонстрируются в примере 3.3.

Пример 3.3 ❖ `hello.py`: отображение шаблонов

```
from flask import Flask, render_template

# ...

@app.route('/index')
def index():
    return render_template('index.html')

@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
```

Функция `render_template`, предоставляемая фреймворком Flask, дает возможность интегрировать механизм шаблонов Jinja2 в прило-

жение. Эта функция принимает имя файла шаблона в первом аргументе. Любые дополнительные аргументы, представленные парами ключ/значение, определяют фактические значения соответствующих переменных в шаблоне. В этом примере второй шаблон принимает переменную `name`.

Именованные аргументы, такие как `name=name` в предыдущем примере, на практике используются очень часто, но тем, кто прежде не использовал их, они могут казаться непривычными и непонятными. Имя «`name`» слева представляет имя аргумента, совпадающее с именем переменной-заполнителя в шаблоне. Имя «`name`» справа – это переменная в текущей области видимости, содержащая значение для аргумента с тем же именем.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 3a` и получить эту версию приложения.

Переменные

Конструкция `{{ name }}` в шаблоне из примера 3.2 ссылается на специальную переменную-заполнитель, сообщающую механизму шаблонов, что ее значение следует получить из данных, переданных во время выполнения операции отображения шаблона.

Механизм Jinja2 распознает переменные любых типов, даже составных, таких как списки, словари и объекты. Ниже приводится еще несколько примеров использования переменных в шаблонах:

```
<p>A value from a dictionary: {{ mydict['key'] }}.</p>
<p>A value from a list: {{ mylist[3] }}.</p>
<p>A value from a list, with a variable index: {{ mylist[myintvar] }}.</p>
<p>A value from an object's method: {{ myobj.somemethod() }}.</p>
```

Переменные можно изменять с помощью *фильтров*, которые добавляются после имени переменной через символ вертикальной черты. Например, ниже показано, как перевести в верхний регистр все символы в тексте, содержащемся в переменной `name`:

```
Hello, {{ name|capitalize }}
```


В табл. 3.1 перечислены некоторые наиболее часто используемые фильтры, поддерживаемые механизмом Jinja2.

Фильтр `safe` стоит того, чтобы сделать несколько замечаний. По умолчанию механизм Jinja2 экранирует содержимое переменных в целях безопасности. Например, если переменная содержит строку `'<h1>Hello</h1>'`, Jinja2 отобразит ее как `'<h1>Hello</h1>'`,

Таблица 3.1. Фильтры переменных, поддерживаемые механизмом Jinja2

Имя фильтра	Описание
safe	Отображает содержимое без экранирования специальных символов
capitalize	Преобразует первый символ в верхний регистр, а остальные – в нижний
lower	Преобразует все символы в нижний регистр
upper	Преобразует все символы в верхний регистр
title	Преобразует первые символы в каждом слове в верхний регистр
trim	Удаляет начальные и завершающие пробельные символы
striptags	Удаляет теги HTML из содержимого перед отображением

в результате чего браузер не будет интерпретировать элемент `h1`, а просто отобразит его. Часто бывает необходимо отобразить разметку HTML в соответствии с правилами, и в этих ситуациях вам поможет фильтр `safe`.

 Никогда не применяйте фильтр `safe` к значениям, полученным из источников, не пользующихся доверием, например введенных пользователем в веб-форме.

Полный список фильтров можно найти в официальной документации с описанием Jinja2¹.

Управляющие структуры

Jinja2 поддерживает несколько управляющих структур, которые можно использовать для изменения потока интерпретации шаблона. В этом разделе мы познакомимся с некоторыми, наиболее полезными структурами.

Ниже показано, как можно использовать условные инструкции в шаблонах:

```
{% if user %}
    Hello, {{ user }}!
{% else %}
    Hello, Stranger!
{% endif %}
```

Часто в шаблонах бывает необходимо отобразить список элементов. Следующий пример показывает, как это можно реализовать с помощью цикла `for`:

¹ <http://bit.ly/jinja-filters>.

```
<ul>
  {% for comment in comments %}
    <li>{{ comment }}</li>
  {% endfor %}
</ul>
```

Jinja2 также поддерживает *макросы*, напоминающие функции в языке Python. Например:

```
{% macro render_comment(comment) %}
  <li>{{ comment }}</li>
{% endmacro %}

<ul>
  {% for comment in comments %}
    {{ render_comment(comment) }}
  {% endfor %}
</ul>
```

Для поддержки многократного использования макросов их можно сохранить в отдельном файле и затем импортировать этот файл во всех шаблонах:

```
{% import 'macros.html' as macros %}
<ul>
  {% for comment in comments %}
    {{ macros.render_comment(comment) }}
  {% endfor %}
</ul>
```

Фрагменты шаблонов, которые повторяются в разных местах, также можно сохранять в отдельных файлах и затем *подключать* их там, где они необходимы:

```
{% include 'common.html' %}
```

Еще одна мощная возможность обеспечить повторное использование — наследование. *Наследование шаблонов* напоминает наследование классов в Python. Например, создадим сначала базовый шаблон с именем *base.html*:

```
<html>
<head>
  {% block head %}
    <title>{{ block title }}{% endblock %} - My Application</title>
  {% endblock %}
</head>
<body>
  {% block body %}
  {% endblock %}
</body>
</html>
```

Здесь теги `block` определяют элементы, которые могут изменяться в производных шаблонах. В данном примере присутствуют блоки с именами `head`, `title` и `body`. Обратите внимание, что блок `title` находится внутри блока `head`. Ниже приводится пример шаблона, следующего базовый шаблон:

```
{% extends «base.html» %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style>
    </style>
{% endblock %}
{% block body %}
<h1>Hello, World!</h1>
{% endblock %}
```

Директива `extends` объявляет, что этот шаблон наследует *base.html*. За ней следуют новые определения трех блоков, объявленных в базовом шаблоне, которые будут вставлены в соответствующие места. Обратите внимание, что новое определение блока `head`, непустого в базовом шаблоне, использует функцию `super()`, чтобы получить оригинальное содержимое.

Практическое использование всех управляющих структур, представленных выше, будет демонстрироваться далее в этой книге, поэтому у вас еще будет возможность увидеть, как они действуют.

Интеграция Twitter Bootstrap с помощью Flask-Bootstrap

Bootstrap¹ – это свободно распространяемый фреймворк, созданный в компании Twitter, который включает компоненты пользовательского интерфейса для создания привлекательных веб-страниц, совместимых со всеми современными веб-браузерами.

Bootstrap – клиентский фреймворк, поэтому он не используется на сервере непосредственно. Сервер должен лишь обеспечить отправку ответов в формате HTML, ссылающихся на таблицы каскадных стилей (Cascading Style Sheets, CSS) из Bootstrap и файлы с кодом на JavaScript, в которых на основе HTML, CSS и JavaScript создаются необходимые компоненты. Идеальное место для всего этого – шаблоны.

¹ <http://getbootstrap.com/>.

Наиболее очевидный путь к интеграции Bootstrap в приложение – внести необходимые изменения в шаблоны. Проще всего это сделать с помощью *расширения* Flask-Bootstrap. Установить это расширение можно с помощью утилиты `pip`:

```
(venv) $ pip install flask-bootstrap
```

Расширения для фреймворка Flask обычно инициализируются одновременно с созданием экземпляра приложения. В примере 3.4 показана инициализация расширения Flask-Bootstrap.

Пример 3.4 ❖ `hello.py`: инициализация Flask-Bootstrap

```
from flask.ext.bootstrap import Bootstrap
# ...
bootstrap = Bootstrap(app)
```

Как и в примере подключения расширения Flask-Script, продемонстрировавшемся в главе 2, сначала Flask-Bootstrap импортируется из пространства имен `flask.ext` и затем инициализируется передачей экземпляра приложения в вызов конструктора.

После инициализации Flask-Bootstrap приложению становится доступен базовый шаблон, включающий все файлы Bootstrap. Благодаря поддержке наследования в механизме Jinja2 приложение может расширять базовый шаблон, определяющий общую структуру страницы и включающий элементы, импортированные из Bootstrap. В примере 3.5 показана новая версия *user.html* в виде производного шаблона.

Пример 3.5 ❖ `templates/user.html`: шаблон, использующий Flask-Bootstrap

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

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle"
        data-toggle="collapse" data-target=".navbar-collapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">Flasky</a>
    </div>
  </div>
</div>
```

```

        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a href="/">Home</a></li>
            </ul>
        </div>
    </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
    <div class="page-header">
        <h1>Hello, {{ name }}!</h1>
    </div>
</div>
{% endblock %}

```

Директива `extends` в Jinja2 реализует наследование путем ссылки на `bootstrap/base.html` из Flask-Bootstrap. Базовый шаблон в расширении Flask-Bootstrap определяет заготовку веб-страницы, подключающую все файлы CSS и JavaScript из Bootstrap.

Базовые шаблоны определяют *блоки*, которые можно переопределять в производных шаблонах. Директивы `block` и `endblock` определяют блоки содержимого, добавляемые в базовый шаблон.

Шаблон `user.html`, представленный выше, определяет три блока с именами `title`, `navbar` и `content`. Все эти блоки экспортируются базовым шаблоном для определения в производных шаблонах. Блок `title` определяет содержимое, которое должно располагаться между тегами `<title>` в заголовке HTML-документа. Блоки `navbar` и `content` зарезервированы для строки навигации и основного содержимого страницы.

Блок `navbar` в этом шаблоне определяет простую строку навигации, использующую компоненты Bootstrap. Блок `content` содержит контейнер `<div>` с названием страницы внутри. Строка с текстом приветствия, составлявшая содержимое страницы в предыдущей версии шаблона, теперь перемещена в название страницы. На рис. 3.1 показано, как выглядит страница приложения после всех этих изменений.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 3b` и получить эту версию приложения. На сайте проекта Bootstrap¹ вы найдете официальную документацию, которая может служить великолепным учебником с массой готовых примеров.

¹ <http://getbootstrap.com/>.

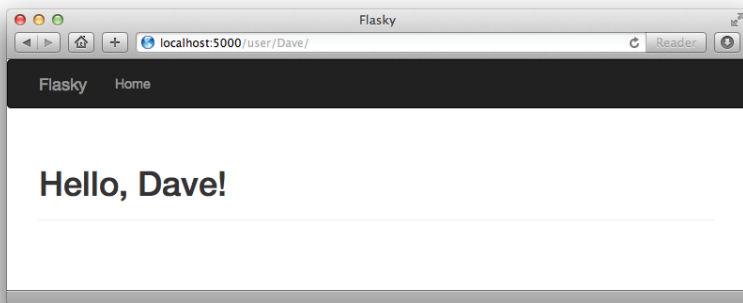


Рис. 3.1 ❖ Шаблоны Twitter Bootstrap

Шаблон *base.html* из Flask-Bootstrap определяет еще несколько дополнительных блоков, которые также можно использовать в производных шаблонах. Полный их список приводится в табл. 3.2.

Таблица 3.2. Блоки базового шаблона из Flask-Bootstrap

Имя блока	Описание
doc	HTML-документ целиком
html_attribs	Атрибуты внутри тега <html>
html	Содержимое тега <html>
head	Содержимое тега <head>
title	Содержимое тега <title>
metas	Список тега <meta>
styles	Определения каскадных таблиц стилей
body_attribs	Атрибуты внутри тега <body>
body	Содержимое тега <body>
navbar	Строка навигации, определяемая пользователем
content	Содержимое страницы, определяемое пользователем
scripts	Объявления JavaScript внизу документа

Многие блоки из табл. 3.2 используются самим расширением Flask-Bootstrap, поэтому их переопределение может вызывать проблемы. Например, блоки `styles` и `scripts` реализуют подключение файлов Bootstrap. Если для нужд приложения потребуется что-то добавить в непустой блок, используйте функцию `super()` из Jinja2. Например, ниже показано, как следует переопределять блок `scripts` в производном шаблоне, чтобы добавить в него новый файл JavaScript:


```
{% block scripts %}
{{ super() }}
<script type="text/javascript" src="my-script.js"></script>
{% endblock %}
```

Нестандартные страницы с сообщениями об ошибках

При вводе недопустимого маршрута в адресную строку браузера сервер возвращает страницу с кодом ошибки 404. Страница по умолчанию выглядит плоско и непривлекательно и выбивается из общего стиля оформления страниц, использующих Bootstrap.

Фреймворк Flask позволяет приложениям определять собственные страницы с сообщениями об ошибках, которые могут основываться на шаблонах, подобно страницам для действующих маршрутов. Наиболее типичными ошибками являются 404 (возникающая при попытке обратиться к несуществующей странице) и 500 (возникающая, когда серверный код генерирует исключение). В примере 3.6 показано, как добавить свои обработчики для этих двух ошибок.

Пример 3.6 ❖ hello.py: нестандартные страницы с сообщениями об ошибках

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

Как и функции представления, обработчики ошибок возвращают ответ. Они также должны возвращать числовой код состояния, соответствующий ошибке.

Шаблоны, используемые в обработчиках ошибок, придется написать самим. Они должны иметь ту же структуру, что и обычные страницы, то есть в данном случае они должны включать строку навигации и название страницы, отражающее сообщение об ошибке.

Самый простой способ написать такие шаблоны – скопировать содержимое *templates/user.html* в *templates/404.html* и *templates/500.html* и затем изменить в этих двух новых файлах элемент с названием страницы. Но при таком подходе возникает масса повторяющегося кода.

Избежать дублирования поможет поддержка наследования шаблонов в Jinja2. По аналогии с расширением Flask-Bootstrap, определяю-

щим базовый шаблон с основной структурой страниц, приложение также может определить собственный базовый шаблон с уточненной структурой страниц, включающий строку навигации и оставляющий определение содержимого страницы за производными шаблонами. В примере 3.7 приводится файл *templates/base.html* – новый шаблон, наследующий шаблон *bootstrap/base.html* и определяющий строку навигации, сам служащий базовым шаблоном для других шаблонов, таких как *templates/user.html*, *templates/404.html* и *templates/500.html*.

Пример 3.7 ❖ *templates/base.html*: базовый шаблон приложения со строкой навигации

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

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle"
        data-toggle="collapse" data-target=".navbar-collapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">Flasky</a>
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li><a href="/">Home</a></li>
      </ul>
    </div>
  </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
  {% block page_content %}{% endblock %}
</div>
{% endblock %}
```

Блок `content` этого шаблона содержит только контейнерный элемент `<div>`, обертывающий новый пустой блок с именем `page_content`, который может быть определен в производном шаблоне.

Шаблоны в приложении теперь будут наследовать этот шаблон вместо базового шаблона из Flask-Bootstrap. В примере 3.8 показано, насколько просто теперь определяется страница с сообщением об ошибке 404 благодаря наследованию шаблона *templates/base.html*.

Пример 3.8 ❖ *templates/404.html*: страница с сообщением об ошибке 404, использующая наследование шаблона

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

{% block title %}Flasky - Page Not Found{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Not Found</h1>
</div>
{% endblock %}
```

На рис. 3.2 показано, как выглядит эта страница в браузере.

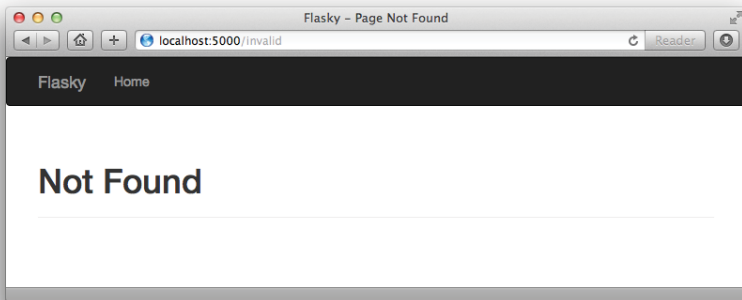


Рис. 3.2 ❖ Нестандартная страница с сообщением об ошибке 404

Теперь можно упростить шаблон *templates/user.html*, унаследовав в нем базовый шаблон приложения, как показано в примере 3.9.

Пример 3.9 ❖ *templates/user.html*: упрощенный шаблон страниц, использующий наследование

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

{% block title %}Flasky{% endblock %}

{% block page_content %}
```

```
<div class="page-header">
    <h1>Hello, {{ name }}!</h1>
</div>
{% endblock %}
```



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 3с` и получить эту версию приложения.

Ссылки

Любое приложение, имеющее более одного маршрута, неизменно должно включать ссылки, связывающие разные страницы, такие как, например, ссылки в строке навигации.

Простое включение адресов URL в виде ссылок непосредственно в шаблон еще возможно для тривиально простых маршрутов, но для динамических маршрутов с переменными компонентами в адресах URL такой прием никуда не годится. Кроме того, явные адреса URL в шаблонах создают нежелательные зависимости между маршрутами. Если впоследствии будет принято решение о реорганизации системы маршрутов, ссылки в шаблонах могут оказаться недействительными.

Чтобы избежать этих проблем, Flask предоставляет вспомогательную функцию `url_for()`, которая генерирует адреса URL из информации, хранящейся в карте адресов URL приложения.

В простейшем случае эта функция принимает имя функции представления (или имя конечной точки маршрута, определенного с помощью `app.add_url_route()`) и возвращает готовый адрес URL. Например, в текущей версии *hello.py* вызов `url_for('index')` вернет `/`. Вызов `url_for('index', _external=True)` вернет абсолютный адрес URL, то есть в данном случае: `http://localhost:5000/`.



Относительные адреса URL используются для организации связей между маршрутами внутри приложения. Абсолютные адреса URL необходимы только для ссылок, которые будут использоваться за пределами приложения. Такие ссылки, например, можно отправлять по электронной почте.

Динамические адреса URL можно генерировать с помощью функции `url_for()`, передавая ей динамические части в виде именованных аргументов. Например, вызов `url_for('user', name='john', _external=True)` вернет `http://localhost:5000/user/john`.

В именованных аргументах функции `url_for()` можно передавать не только динамические части маршрутов. Любые дополнитель-

ные аргументы она добавит в строку запроса. Например, вызов `url_for('index', page=2)` вернет `/?page=2`.

Статические файлы

Веб-приложения состоят не только из программного кода на языке Python и шаблонов. Большинство приложений также использует статические файлы, такие как изображения, сценарии на языке JavaScript и таблицы стилей CSS, ссылки на которые присутствуют в разметке HTML.

Вспомните, как в главе 2 мы неожиданно обнаружили элемент `static` в карте адресов URL приложения *hello.py*. Появление этого элемента обусловлено тем, что ссылки на статические файлы интерпретируются как специальный маршрут `/static/<filename>`. Например, вызов `url_for('static', filename='css/styles.css', _external=True)` вернет `http://localhost:5000/static/css/styles.css`.

С настройками по умолчанию фреймворк Flask ищет статические файлы в подкаталоге *static*, находящемся в корневой папке приложения. При желании файлы можно организовать в подкаталоги внутри этой папки. Когда сервер получит URL из предыдущего примера, он сгенерирует ответ, включающий содержимое файла *static/css/styles.css*.

В примере 3.10 показано, как можно подключить ярлык *favicon.ico* в базовом шаблоне для отображения в адресной строке браузера.

Пример 3.10 ❖ templates/base.html: определения ярлыка сайта (favicon)

```
{% block head %}
{{ super() }}
<link rel="shortcut icon" href="{{ url_for('static', filename = 'favicon.ico') }}"
      type="image/x-icon">
<link rel="icon" href="{{ url_for('static', filename = 'favicon.ico') }}"
      type="image/x-icon">
{% endblock %}
```

Определение ярлыка добавлено в конец блока `head`. Обратите внимание на вызов `super()`, который выполняется для сохранения содержимого, определяемого базовыми шаблонами.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 3d` и получить эту версию приложения.

Локализация дат и времени с помощью Flask-Moment

Обработка дат и времени перестает быть тривиальной, когда приложение оказывается доступным для пользователей из разных уголков мира.

Чтобы не зависеть от географического местоположения каждого пользователя, сервер должен работать с некоторым универсальным представлением времени, поэтому на серверах обычно используется универсальное координированное время (Coordinated Universal Time, UTC). Однако, увидев время UTC, многие пользователи могут быть обескуражены, так как в подавляющем большинстве они предпочитают видеть даты и время в своем часовом поясе, отформатированные в соответствии с национальными правилами.

Наилучшим решением было бы оперировать на сервере временем исключительно в часовом поясе UTC и отправлять его браузеру, где оно преобразовывалось бы в местный часовой пояс и отображалось в соответствии с национальными настройками. Веб-браузеры способны справиться с этой задачей намного лучше, потому что имеют доступ к настройкам часового пояса и форматов отображения дат и времени на локальном компьютере.

И такое решение есть – отличная клиентская библиотека на JavaScript, отображающая даты и время в браузере, – *moment.js*. Для фреймворка Flask имеется расширение Flask-Moment, интегрирующее библиотеку *moment.js* в шаблоны Jinja2. Установить Flask-Moment можно с помощью утилиты `pip`:

```
(venv) $ pip install flask-moment
```

В примере 3.11 показано, как выполняется инициализация расширения.

Пример 3.11 ❖ hello.py: инициализация Flask-Moment

```
from flask.ext.moment import Moment
moment = Moment(app)
```

Помимо *moment.js*, расширение Flask-Moment зависит также от библиотеки *jquery.js*. Эти две библиотеки необходимо подключить где-то в документе HTML – либо непосредственно (в этом случае придется указать версию), либо вызовом вспомогательной функции, экспортируемой расширением, которая возвращает ссылки на проверенные версии этих библиотек в сети доставки содержимого (Content

Delivery Network, CDN). Так как фреймворк Bootstrap уже включает библиотеку *jquery.js*, в данном случае достаточно подключить только библиотеку *moment.js*. В примере 3.12 показано, как сделать это в блоке `scripts` базового шаблона.

Пример 3.12 ❖ `templates/base.html`: импортирование библиотеки *moment.js*

```
{% block scripts %}
{{ super() }}
{{ moment.include_moment() }}
{% endblock %}
```

Для работы с отметками времени (timestamps) в расширении Flask-Moment имеется класс *moment*, доступный для использования в шаблонах. В примере 3.13 выполняется передача переменной `current_time` в шаблон для отображения.

Пример 3.13 ❖ `hello.py`: добавление переменной с датой и временем

```
from datetime import datetime

@app.route('/')
def index():
    return render_template('index.html',
                           current_time=datetime.utcnow())
```

А в примере 3.14 показано, как осуществляется отображение `current_time` в шаблоне.

Пример 3.14 ❖ `templates/index.html`: отображение отметки времени с помощью Flask-Moment

```
<p>The local date and time is {{ moment(current_time).format('LLL') }}.</p>
<p>That was {{ moment(current_time).fromNow(refresh=True) }}</p>
```

Вызов `format('LLL')` отобразит дату и время в соответствии с локальным часовым поясом и национальными настройками на клиентском компьютере. Аргумент определяет формат отображения, от 'L' до 'LLLL', в зависимости от требуемой детальности. Функция `format()` может также принимать нестандартные спецификаторы формата.

Вызов `fromNow()` во второй строке отобразит время в секундах относительно указанного в переменной и автоматически будет обновлять его с течением времени. Первоначально это время будет отображено как строка «a few seconds ago» (несколько секунд тому назад). Но так как параметр `refresh` вынуждает постоянно обновлять прошедшее время, то если оставить страницу открытой на несколько минут, мож-

но увидеть последовательно сменяющие друг друга надписи «a minute ago» (минуту тому назад), затем «2 minutes ago» (2 минуты тому назад) и так далее.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 3e` и получить эту версию приложения.

Расширение Flask-Moment экспортирует также методы `format()`, `fromNow()`, `fromTime()`, `calendar()`, `valueOf()` и `unix()` из *moment.js*. За дополнительной информацией о возможностях управления форматированием обращайтесь к документации: <http://momentjs.com/docs/#/displaying/>.



Расширение Flask-Moment предполагает, что отметки времени, поступающие со стороны сервера, являются простыми объектами `datetime` и соответствуют часовому поясу UTC. За дополнительной информацией об объектах `datetime` и их возможностях обращайтесь к документации с описанием пакета `datetime` в стандартной библиотеке Python: <http://bit.ly/datepack>.

Строки, отображаемые расширением Flask-Moment, можно локализовать на разные языки. Язык можно выбрать в шаблоне, передав код языка в вызов функции `lang()`:

```
{{ moment.lang('es') }}
```

Вооруженные приемами, обсуждавшимися в этой главе, вы сможете создавать для своих веб-приложений современные веб-страницы, дружелюбные по отношению к пользователям. В следующей главе мы рассмотрим еще один аспект шаблонов, не затрагивавшийся в этой главе: взаимодействие с пользователем посредством веб-форм.

Глава 4

Веб-формы

Объект запроса, с которым мы познакомились в главе 2, хранит всю информацию, отправленную клиентом вместе с запросом. В частности, свойство `request.form` обеспечивает доступ данным формы, отправленным в запросе POST.

Несмотря на то что возможностей объекта `request` вполне достаточно для обработки веб-форм, существует множество задач, которые быстро могут стать утомительными. Двумя примерами таких задач могут служить создание разметки HTML для форм и проверка данных, полученных с формой.

Расширение `Flask-WTF`¹ делает работу с формами намного более приятной. По сути, это расширение является оберткой вокруг пакета `WTForms`².

Установить расширение `Flask-WTF` вместе со всеми его зависимостями можно с помощью утилиты `pip`:

```
(venv) $ pip install flask-wtf
```

Защита от подделки межсайтовых запросов

По умолчанию `Flask-WTF` защищает все формы от атак, основанных на подделке межсайтовых запросов (Cross-Site Request Forgery, CSRF). Суть атаки CSRF заключается в том, что злонамеренный веб-сайт отправляет запросы другому веб-сайту, на котором авторизована жертва³.

Для реализации защиты от атак вида CSRF `Flask-WTF` требует от приложения настройки ключа шифрования. С помощью этого ключа `Flask-WTF` генерирует зашифрованные блоки, которые используются

¹ <http://pythonhosted.org/Flask-WTF/>.

² <http://wtforms.simplecodes.com/>.

³ http://ru.wikipedia.org/wiki/Межсайтовая_подделка_запроса. — Прим. перев.

для проверки аутентичности запросов, содержащие данные форм. В примере 4.1 показано, как настроить ключ шифрования.

Пример 4.1 ❖ hello.py: настройка Flask-WTF

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'hard to guess string'
```

Словарь `app.config` — универсальное хранилище, используемое для хранения настроек фреймворком, расширениями или самим приложением. Добавлять настройки в объект `app.config` можно с помощью привычного синтаксиса обращения к словарям. Объект с настройками также имеет методы для импортирования настроек из файлов или из окружения.

Для хранения универсального ключа шифрования фреймворк Flask и некоторые сторонние расширения используют параметр настройки `SECRET_KEY`. Как следует из имени, криптостойкость шифрования зависит от степени секретности значения этого параметра. Выбирайте для своих приложений разные секретные ключи и храните их в тайне от других.



Для большей безопасности секретный ключ следует хранить в переменной окружения, а не в программном коде. Этот прием описывается в главе 7.

Классы форм

При использовании расширения Flask-WTF каждая веб-форма представлена классом, наследующим класс `Form`. Класс определяет список полей формы, каждое из которых представлено объектом. К каждому объекту поля может быть подключен один или более *валидаторов* (*validators*), где под валидаторами подразумеваются функции, осуществляющие допустимость данных, отправленных пользователем.

В примере 4.2 показана простая веб-форма, состоящая из текстового поля и кнопки отправки.


Пример 4.2 ❖ hello.py: определение класса формы

```
from flask.ext.wtf import Form
from wtforms import StringField, SubmitField
from wtforms.validators import Required

class NameForm(Form):
    name = StringField('What is your name?', validators=[Required()])
    submit = SubmitField('Submit')
```

Поля в форме определяются как переменные класса, и каждой переменной класса присваивается объект поля того или иного типа. В предыдущем примере определяется форма `NameForm`, содержащая текстовое поле `name` и кнопку отправки формы `submit`. Класс `StringField` представляет элемент `<input>` с атрибутом `type="text"`. Класс `SubmitField` представляет элемент `<input>` с атрибутом `type="submit"`. Первый аргумент конструктора поля – текст для метки, которая будет отображаться рядом с полем в HTML-форме.

Необязательный аргумент `validators`, который можно видеть в вызове конструктора `StringField` в данном примере, определяет список функций проверки данных, отправленных пользователем. Валидатор `Required()` гарантирует, что пользователь не сможет отправить пустое поле.

 Базовый класс `Form` определяется расширением `Flask-WTF`, поэтому его необходимо импортировать из `flask.ext.wtf`. Однако классы полей и валидаторы импортируются непосредственно из пакета `WTForms`.

В табл. 4.1 перечислены все классы полей форм, поддерживаемые пакетом `WTForms`.

Таблица 4.1. Поля форм, поддерживаемые пакетом `WTForms`

Тип поля	Описание
<code>StringField</code>	Текстовое поле
<code>TextAreaField</code>	Многострочное текстовое поле
<code>PasswordField</code>	Текстовое поле для ввода пароля
<code>HiddenField</code>	Скрытое текстовое поле
<code>DateField</code>	Текстовое поле, принимающее значение <code>datetime.date</code> в заданном формате
<code>DateTimeField</code>	Текстовое поле, принимающее значение <code>datetime.datetime</code> в заданном формате
<code>IntegerField</code>	Текстовое поле, принимающее целочисленное значение
<code>DecimalField</code>	Текстовое поле, принимающее значение <code>decimal.Decimal</code>
<code>FloatField</code>	Текстовое поле, принимающее вещественное значение
<code>BooleanField</code>	Флажок, имеющий два возможных значения: <code>True</code> и <code>False</code>
<code>RadioField</code>	Список радиокнопок
<code>SelectField</code>	Раскрывающийся список вариантов
<code>SelectMultipleField</code>	Раскрывающийся список вариантов с возможностью выбора сразу нескольких из них
<code>FileField</code>	Поле загрузки файла
<code>SubmitField</code>	Кнопка отправки формы
<code>FormField</code>	Встроенная форма как поле вмещающей формы
<code>FieldList</code>	Список полей указанного типа

В табл. 4.2 приводится список встроенных валидаторов из пакета WTForms.

Таблица 4.2. Валидаторы из пакета WTForms

Валидатор	Описание
Email	Проверяет адрес электронной почты
EqualTo	Сравнивает значения двух полей; может пригодиться для сравнения паролей при вводе их дважды
IPAddress	Проверяет сетевой адрес IPv4
Length	Проверяет длину введенной строки
NumberRange	Проверяет вхождение введенного значения в определенный диапазон
Optional	Допускает отсутствие введенных данных в поле, пропускает дополнительные валидаторы
Required	Требует наличия введенного значения в поле
Regex	Проверяет введенное значение на соответствие регулярному значению
URL	Проверяет адрес URL
AnyOf	Проверяет введенное значение на совпадение с любым из списка допустимых значений
NoneOf	Проверяет введенное значение на отсутствие совпадения с любым из списка недопустимых значений

Отображение форм в формат HTML

Поля форм в шаблонах принимают вид вызываемых методов, при вызове которых происходит отображение полей в разметку HTML. Если предположить, что функция представления передает экземпляр NameForm в шаблон в виде аргумента с именем form, шаблон может сгенерировать простую HTML-форму, как показано ниже:

```
<form method="POST">
    {{ form.name.label }} {{ form.name() }}
    {{ form.submit() }}
</form>
```

Конечно, получившаяся форма будет выглядеть довольно унылой. Чтобы улучшить внешний вид формы, методам можно передавать дополнительные аргументы, которые будут преобразованы в HTML-атрибуты. Например, для полей можно указать атрибут id или class и определить стили CSS:

```
<form method="POST">
    {{ form.name.label }} {{ form.name(id='my-text-field') }}
    {{ form.submit() }}
</form>
```

Но, даже имея возможность определять HTML-атрибуты, для приведения формы к желаемому внешнему виду требуется приложить существенные усилия, поэтому лучше использовать собственные стили форм из фреймворка Bootstrap везде, где только возможно. Расширение Flask-Bootstrap предоставляет высокоуровневую вспомогательную функцию для отображения формы Flask-WTF с использованием стилей Bootstrap единственным вызовом. Так, предыдущую форму можно отобразить следующим образом:

```
{% import "bootstrap/wtf.html" as wtf %}
{{ wtf.quick_form(form) }}
```

Директива `import` действует подобно одноименной директиве языка Python и позволяет импортировать и использовать элементы шаблонов в других шаблонах. Импортируемый здесь файл `bootstrap/wtf.html` определяет вспомогательные функции для отображения форм Flask-WTF с использованием стилей Bootstrap. Функция `wtf.quick_form()` принимает объект формы Flask-WTF и отображает его с использованием стилей по умолчанию Bootstrap. Полный шаблон для приложения *hello.py* приводится в примере 4.3.

Пример 4.3 ❖ `templates/index.html`: использование расширений Flask-WTF и Flask-Bootstrap для отображения формы

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky{% endblock %}
{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
</div>
{{ wtf.quick_form(form) }}
{% endblock %}
```

Область содержимого в шаблоне теперь делится на два раздела. Первый раздел – заголовок страницы, где отображается приветствие. Здесь используется условная директива. Условные директивы в Jinja2 имеют формат `{% if variable %}...{% else %}...{% endif %}`. Если условие оценивается как `True`, отображаться будет то, что находится между директивами `if` и `else`. Если условие оценивается как `False`, отображаться будет то, что находится между директивами `else` и `endif`. Шаблон в примере отобразит строку «Hello, Stranger!» (Привет, Незнакомец!), если при обращении к шаблону аргумент `name` не будет определен. Во втором разделе с помощью функции `wtf.quick_form()` отображается объект `NameForm`.

Обработка форм в функциях представления

В новой версии *hello.py* функция представления `index()` отобразит форму и получит ее данные. Обновленная версия функции `index()` приводится в примере 4.4.

Пример 4.4 ❖ *hello.py*: методы маршрутов

```
@app.route('/', methods=['GET', 'POST'])
def index():
    name = None
    form = NameForm()
    if form.validate_on_submit():
        name = form.name.data
        form.name.data = ''
    return render_template('index.html', form=form, name=name)
```

Аргумент `methods`, добавленный в декоратор `app.route`, сообщает фреймворку Flask, что он должен зарегистрировать функцию представления как обработчик запросов GET и POST. Когда аргумент `methods` отсутствует, функция представления регистрируется только для обработки запросов GET.

Добавление метода POST в список необходимо потому, что отправку форм удобнее производить методом POST. Формы можно также отправлять методом GET, но GET-запросы не имеют тела, и данные передаются в виде параметров строки запроса в URL, которая видна в адресной строке браузера. По этой и некоторым другим причинам отправку форм предпочтительнее выполнять методом POST.

Локальная переменная `name` будет хранить имя, введенное пользователем. Если пользователь оставит поле пустым, переменная будет инициализирована значением `None`. Функция представления создаст экземпляр класса `NameForm`, приводившегося выше, для представления формы. Метод `validate_on_submit()` формы вернет `True`, если она была отправлена пользователем и данные прошли проверку всеми валидаторами полей. В любых других случаях `validate_on_submit()` вернет `False`. Значение, возвращаемое этим методом, может служить основой для принятия решения о необходимости отображения или обработки формы.

При первом обращении пользователя к приложению сервер получит запрос GET без данных формы, поэтому `validate_on_submit()` вернет `False`. Тело инструкции `if` будет пропущено, и в ответ на запрос будет выполнено отображение шаблона, который получит объект формы и

переменную `name` со значением `None` в виде аргументов. В результате пользователь увидит форму в окне браузера.

Когда пользователь заполнит и отправит форму, сервер получит запрос `POST` с данными. Метод `validate_on_submit()` вызовет валидатор `Required()`, подключенный к полю `name`. Если поле непустое, оно будет принято валидатором и `validate_on_submit()` вернет `True`. После этого имя, введенное пользователем, станет доступно через атрибут `data` поля. В теле инструкции `if` это имя будет присвоено локальной переменной `name`, и затем поле формы будет очищено путем присваивания пустой строки атрибуту `data`. Вызов метода `render_template()` в последней строке отобразит шаблон, но на этот раз аргумент `name` будет содержать имя из формы, и приветствие получится персонализированным.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 4a` и получить эту версию приложения.

На рис. 4.1 показано, как выглядит форма в окне браузера, когда пользователь впервые заходит на сайт. В ответ на отправку формы приложение выведет персонализированное приветствие. При этом форма все еще будет отображаться внизу, поэтому пользователь сможет ввести и отправить другое имя. Приложение в этом состоянии изображено на рис. 4.2.

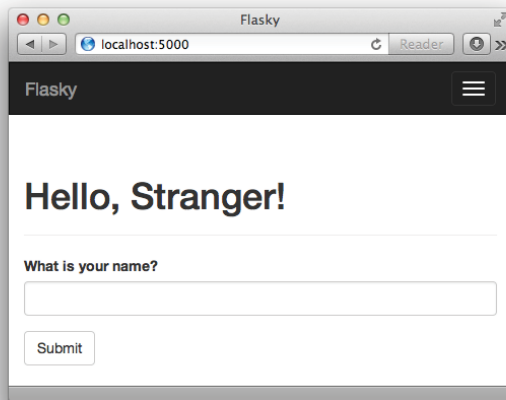


Рис. 4.1 ❖ Веб-форма Flask-WTF

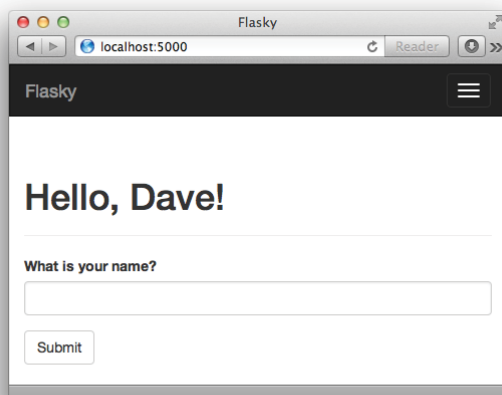


Рис. 4.2 ❖ Веб-форма после отправки

Если пользователь отправит форму с незаполненным полем, валидатор `Required()` обнаружит ошибку, как показано на рис. 4.3. Обратите внимание, как много функциональности вы получаете автоматически. Это отличный пример, показывающий, как много могут дать приложения-

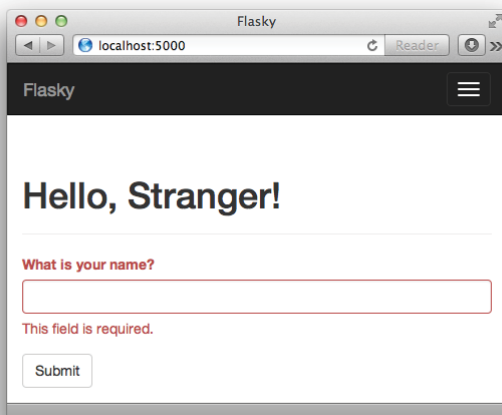


Рис. 4.3 ❖ Веб-форма, в которой валидатор обнаружил ошибку

ям тщательно спроектированные расширения, такие как Flask-WTF и Flask-Bootstrap.

Переадресация и сеансы


Последняя версия *hello.py* имеет проблему, связанную с удобством использования. Если ввести и отправить имя, а затем щелкнуть на кнопке **Refresh** (Обновить) браузера, браузер выведет обескураживающее предупреждение, предлагая подтвердить повторную отправку формы. Это происходит потому, что в ответ на требование пользователя обновить страницу браузер пытается выполнить последний запрос. Если последним был POST-запрос с данными формы, операция обновления страницы может вызвать повторную отправку формы, что далеко не всегда является желательным.

Многие пользователи не понимают смысла предупреждения. По этой причине считается хорошей практикой никогда не оставлять POST-запрос последним.

Добиться этого можно, отвечая на POST-запросы кодом *переадресации*. Переадресация – это ответ специального типа, включающий адрес URL вместо строки с разметкой HTML. Когда браузер примет такой ответ, он выполнит запрос GET для перехода по указанному адресу URL и отобразит указанную страницу. Для отображения страницы может потребоваться на несколько миллисекунд больше из-за необходимости отправить второй запрос, но пользователь едва ли заметит такую задержку. После этого последним выполненным запросом будет GET-запрос, соответственно обновление страницы будет выполняться без всяких предупреждений со стороны браузера. Этот трюк известен как «Шаблон Post/Redirect/Get».

Но подобный прием влечет за собой другую проблему. При обработке POST-запроса приложение имеет доступ к имени, введенному пользователем в `form.name.data`, но как только обработка запроса завершится, данные формы будут утеряны. Поскольку POST-запрос обрабатывается с переадресацией, приложению требуется сохранить имя, чтобы обработчик переадресованного запроса смог получить его и сконструировать текст приветствия.

Приложения могут хранить данные между запросами в пределах *сеанса пользователя* (*user session*) – в хранилище, доступном для каждого подключившегося клиента. Понятие сеанса пользователя было введено в главе 2 и представляет переменную, связанную с контекстом запроса. Она называется `session` и действует подобно стандартному словарию Python.

 По умолчанию информация о сеансах сохраняется в cookies, которые подписываются ключом `SECRET_KEY`. Любое постороннее вмешательство в содержимое cookie делает подпись недействительной и закрывает сеанс. В примере 4.5 приводится новая версия функции представления `index()`, реализующая переадресацию и поддержку сеансов пользователей.

Пример 4.5 ❖ hello.py: переадресация и сеансы пользователей

```
from flask import Flask, render_template, session, redirect, url_for

@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'))
```

В предыдущей версии приложения для хранения имени, введенного пользователем, применялась локальная переменная `name`. Теперь это имя хранится в сеансе пользователя в виде `session['name']`, благодаря чему оно запоминается между запросами.

Обработка запросов, имеющих в своем составе допустимые данные из формы, теперь завершается вызовом `redirect()`, вспомогательной функции, генерирующей HTTP-ответ переадресации. Функция `redirect()` принимает аргумент с адресом URL для переадресации. В данном случае переадресация выполняется на корневой URL, поэтому ответ можно было бы сформировать проще, как `redirect('/')`, то есть без применения функции `url_for()`. Однако предпочтительнее все же использовать функцию `url_for()`, потому что она генерирует адреса URL на основе карты адресов, благодаря чему гарантируется совместимость с определениями маршрутов, любые изменения в которых будут автоматически учтены.

Первый и единственный обязательный аргумент функции `url_for()` – это имя *конечной точки*, или внутреннее имя маршрута. По умолчанию именем конечной точки маршрута является имя соответствующей функции представления. В данном примере корневому адресу URL соответствует функция представления `index()`, поэтому функции `url_for()` передается имя `index`.

Последнее изменение находится в вызове функции `render_template()`. Теперь значение для аргумента `name` извлекается непосредственно из сеанса: `session.get('name')`. Как и при работе с обычными словарями, применение метода `get()` помогает избежать исключения при обращении к несуществующему ключу, потому что для несуществующих ключей он возвращает значение по умолчанию `None`.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 4b` и получить эту версию приложения.

Эта новая версия приложения показывает ожидаемое поведение при попытке обновить страницу.

Всплывающие сообщения

Иногда бывает полезно сообщить пользователю о результатах обработки запроса. Это может быть подтверждающее сообщение, предупреждение или сообщение об ошибке. Типичным примером может служить сообщение о вводе неправильного имени пользователя или пароля после неудачной попытки аутентификации на сайте.

Фреймворк Flask поддерживает такую возможность как базовую. В примере 4.6 демонстрируется, как можно использовать функцию `flash()` с этой целью.

Пример 4.6 ❖ hello.py: всплывающие сообщения

```
from flask import Flask, render_template, session, redirect, url_for, flash
```

```
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        old_name = session.get('name')
        if old_name is not None and old_name != form.name.data:
            flash('Looks like you have changed your name!')
            session['name'] = form.name.data
            form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html', form = form,
                           name = session.get('name'))
```

Эта версия сравнивает имя, отправленное пользователем, с именем, полученным прежде и хранящимся в сеансе. Если имена отличаются, вызывается функция `flash()` с сообщением для отправки клиенту в ответе.

Однако одного только вызова `flash()` недостаточно – отображение сообщений должно быть предусмотрено в шаблонах. Лучшим местом для этого является базовый шаблон, потому что это обеспечит поддержку всплывающих сообщений во всех страницах. Фреймворк Flask позволяет использовать в шаблонах функцию `get_flashed_messages()`, которая извлекает сообщение и отображает его, как показано в примере 4.7.

Пример 4.7 ❖ templates/base.html: отображение всплывающего сообщения

```
{% block content %}
<div class="container">
    {% for message in get_flashed_messages() %}
    <div class="alert alert-warning">
        <button type="button" class="close" data-dismiss="alert">&times;</button>
        {{ message }}
    </div>
    {% endfor %}

    {% block page_content %}{% endblock %}
</div>
{% endblock %}
```

В этом примере сообщения отображаются с применением CSS-стилей оформления предупреждений из Bootstrap (одно такое сообщение изображено на рис. 4.4).

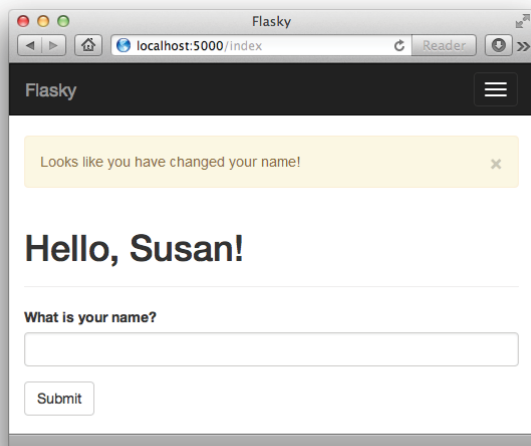


Рис. 4.4 ❖ Всплывающее сообщение

Инструкция цикла используется здесь по той простой причине, что в очереди может иметься несколько сообщений для отображения, по одному на каждый вызов `flash()`, произведенный в цикле обработки запроса. Функция `get_flashed_messages()` не возвращает одно и то же

сообщение дважды, поэтому всплывающие сообщения отображаются только один раз и затем удаляются.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 4c` и получить эту версию приложения.

Возможность принимать данные, введенные пользователем в веб-формах, необходима в большинстве приложений. Не менее востребованной является также возможность хранения данных в постоянном хранилище. Поэтому в следующей главе мы познакомимся с приемами использования баз данных в приложениях на основе Flask.

Глава 5

Базы данных

Базы данных используются приложениями для хранения в организованном виде. Приложение может выполнять *запросы* к базе данных для извлечения необходимых порций данных. Наиболее часто в веб-приложениях используются базы данных, основанные на *реляционной* модели, которые также называются базами данных SQL (Structured Query Language – язык структурированных запросов). Но в последние годы начала расти популярность *документоориентированных* баз данных и баз данных, хранящих информацию в виде пар *ключ/значение*, которые неформально называют базами данных NoSQL.

Базы данных SQL

Реляционные базы данных хранят информацию в *таблицах*, моделирующих различные сущности в терминах предметной модели приложения. Например, база данных для приложения управления заказами наверняка будет иметь таблицы customers (заказчики), products (товары) и orders (заказы).

Таблица имеет фиксированное число *столбцов* и переменное число *строк*. Столбцы определяют атрибуты сущностей, представляемых таблицей. Например, таблица customers может включать такие столбцы, как name (имя), address (адрес), phone (телефон) и т. д. Каждая строка в таблице определяет фактические значения для каждого из столбцов.

Таблицы имеют специальные столбцы, которые называются *первичными ключами* и хранят уникальные идентификаторы строк в таблице. Таблицы могут также иметь столбцы, называемые *внешними ключами*, ссылающиеся на первичные ключи других строк в той же или в другой таблице. Эти ссылки между строками называются *отношениями* и составляют основу реляционной модели базы данных.

На рис. 5.1 показана схема простой базы данных с двумя таблицами, хранящими информацию о пользователях и их ролях. Линия, соединяющая две таблицы, представляет отношение между ними.

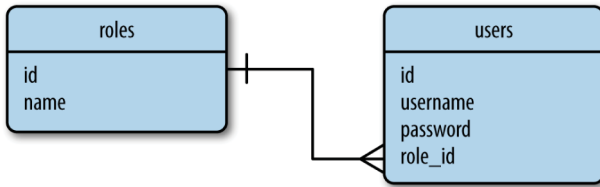


Рис. 5.1 ❖ Пример реляционной базы данных

Таблица **roles** в этой базе данных хранит список всех возможных ролей пользователей, каждая из которых идентифицируется уникальным значением в столбце **id** – первичном ключе таблицы. Таблица **users** хранит список пользователей, каждый из которых также идентифицируется уникальным значением в поле **id**. Помимо столбца **id** первичного ключа, в таблице **roles** имеется столбец **name**, а в таблице **users** имеются столбцы **username** и **password**. Столбец **role_id** в таблице **users** является внешним ключом, ссылающимся на столбец **id** в таблице **roles**. Этот столбец определяет роль для каждого пользователя.

Как видно из этого примера, реляционные базы данных позволяют эффективно хранить данные и избегать их дублирования. Изменить имя роли в этой базе данных не составляет труда, так как оно хранится только в одном месте. Как только название роли изменится в таблице **roles**, все пользователи, для которых столбец **role_id** ссылается на измененную роль, сразу же обнаружат это изменение.

С другой стороны, хранение взаимосвязанных данных во множестве разных таблиц может стать источником сложностей. Получение списка пользователей с их ролями уже представляет проблему, потому что для этого необходимо прежде выполнить *соединение* таблиц. Впрочем, системы управления реляционными базами данных поддерживают встроенную операцию соединения между таблицами.

Базы данных NoSQL

Базы данных, которые не следуют реляционной модели, описанной выше, обычно называют базами данных NoSQL. Одной общей чертой организации всех баз данных NoSQL является использование *коллекций* вместо таблиц и *документов* вместо записей (строк). Архитектуры баз данных NoSQL плохо поддерживают операцию соединения, поэтому большинство из них вообще не поддерживают ее. Чтобы получить список пользователей с их ролями из базы данных NoSQL со

структурой, подобной той, что изображена на рис. 5.1, приложение должно само выполнить операцию соединения, читая поле `role_id` для каждого пользователя и выполняя поиск в таблице `roles`.

Более удобная организация базы данных NoSQL показана на рис. 5.2. Она является результатом применения операции *денормализации*, уменьшающей число таблиц за счет дублирования данных.

База данных с такой структурой явно хранит название роли для каждого пользователя. Изменение названия роли в этом случае может превратиться в весьма ресурсоемкую операцию, требующую изменить большое число документов.

Но не все так плохо в базах данных NoSQL. Дублирование данных ускоряет выполнение запросов. Получить список пользователей с их ролями не представляет никаких сложностей, так как отпадает необходимость в операции соединения.

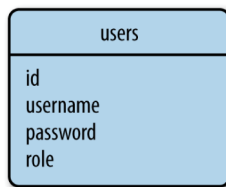


Рис. 5.2 ❖ Пример базы данных NoSQL

SQL или NoSQL?

Базы данных SQL превосходно подходят для хранения структурированных данных в эффективной и компактной форме. Эти базы данных делают очень многое для поддержки непротиворечивости хранимых данных. Базы данных NoSQL ослабляют требования к непротиворечивости и, как результат, способны работать с более высокой скоростью.

Полный анализ и сравнение характеристик типов баз данных выходит далеко за рамки этой книги. Для малых и средних приложений прекрасно подойдут базы данных любого типа, SQL или NoSQL, показывая при этом практически одинаковую производительность.

Фреймворки на Python поддержки баз данных

Для Python имеются пакеты поддержки большинства известных баз данных, как открытые, так и коммерческие. Фреймворк Flask не накладывает никаких ограничений на использование пакетов баз данных, благодаря чему вы сможете использовать MySQL, Postgres, SQLite, Redis, MongoDB или CouchDB.

Если этого разнообразия окажется недостаточно, есть возможность использовать многочисленные пакеты абстракции доступа к базам данных, такие как SQLAlchemy или MongoEngine, позволяющие работать на более высоком уровне – с обычными объектами Python – вместо использования сущностей базы данных, таких как таблицы, документы или языки запросов.

При выборе фреймворка поддержки баз данных необходимо принять во внимание ряд факторов:

- простота в использовании – при непосредственном сравнении низкоуровневых механизмов доступа к базам данных с высокоуровневыми абстракциями вторая группа выглядит намного привлекательнее. Абстракции, которые также называются объектно-реляционными отображениями (Object-Relational Mappers, ORM) или объектно-документными отображениями (Object-Document Mappers, ODM), обеспечивают прозрачное преобразование высокоуровневых объектно-ориентированных операций в низкоуровневые инструкции баз данных;
- производительность – преобразования прикладных объектов в сущности баз данных, выполняемые механизмами ORM и ODM, требуют дополнительных вычислительных ресурсов. В большинстве случаев накладные расходы оказываются не особенно велики, но так бывает не всегда. В общем случае удобство механизмов ORM и ODM значительно перевешивает небольшие потери производительности, поэтому производительность не всегда может рассматриваться как веский аргумент для отказа от ORM и ODM. Однако есть смысл выбирать такие абстракции баз данных, которые поддерживают низкоуровневый доступ, на случай, если потребуется оптимизировать некоторые операции, реализуя их с применением инструкций базы данных;
- переносимость – необходимо учитывать доступность базы данных для платформ, где ведется разработка, и платформ, где будет действовать приложение. Например, если планируется, что приложение будет действовать на облачной платформе, тогда следует выбирать из баз данных, предоставляющих такую возможность.

Другой аспект переносимости касается механизмов ORM и ODM. Некоторые из них поддерживают единственную базу данных, однако другие предоставляют порой весьма широкий выбор, поддерживая один и тот же объектно-ориентированный интерфейс. Отличным

примером может служить фреймворк SQLAlchemy, поддерживающий множество реляционных баз данных, включая популярные MySQL, Postgres и SQLite.

Интеграция с фреймворком Flask

Строго говоря, совершенно необязательно выбирать фреймворк, поддерживающий интеграцию с Flask, но такая поддержка позволит вам не писать код интеграции вручную. Подобная интеграция может упростить настройку и использование, поэтому предпочтение следует отдавать пакетам, реализованным как расширения для Flask.

Учитывая все вышесказанное, для использования в примерах к этой книге было выбрано расширение Flask-SQLAlchemy, являющееся оберткой вокруг SQLAlchemy.

Управление базой данных с помощью Flask-SQLAlchemy

Flask-SQLAlchemy – это расширение для Flask, упрощающее использование SQLAlchemy внутри приложений на основе Flask. SQLAlchemy – мощный фреймворк для работы с реляционными базами данных, поддерживающий множество разных баз данных. Он предлагает высокоуровневые функции объектно-реляционного отображения и низкоуровневый доступ на языке SQL.

Подобно большинству других расширений, Flask-SQLAlchemy можно установить с помощью утилиты `pip`:


```
(venv) $ pip install flask-sqlalchemy
```

В расширении Flask-SQLAlchemy база данных определяется адресом URL. В табл. 5.1 перечислены форматы URL для трех наиболее популярных баз данных.

Таблица 5.1. Форматы URL-доступа к базам данных в Flask-SQLAlchemy

База данных	URL
MySQL	<code>mysql://username:password@hostname/database</code>
Postgres	<code>postgresql://username:password@hostname/database</code>
SQLite (Unix)	<code>sqlite:///absolute/path/to/database</code>
SQLite (Windows)	<code>sqlite:///c:/absolute/path/to/database</code>

Компонент *hostname* в этих URL определяет сервер, где располагается база данных MySQL, которая может быть именем локального (*localhost*) или удаленного сервера. На сервере одновременно может находиться несколько баз данных, поэтому компонент *database* определяет имя используемой базы данных. Для баз данных, требующих аутентификации, в URL включаются компоненты *username* и *password*.

 Для базы данных SQLite не требуется указывать имя сервера, поэтому компоненты «hostname», «username» и «password» отсутствуют в URL, а в компоненте «database» указывается имя файла базы данных.

Адрес URL базы данных должен быть определен как ключ `SQLALCHEMY_DATABASE_URI` в объекте с настройками приложения. Еще одним интересным параметром настройки является ключ `SQLALCHEMY_COMMIT_ON_TEARDOWN`, который можно установить в значение `True`, чтобы разрешить автоматическое подтверждение изменений в базе данных в конце обработки каждого запроса. За информацией о других параметрах настройки расширения Flask-SQLAlchemy обращайтесь к документации с описанием этого расширения. В примере 5.1 демонстрируется, как инициализировать и настроить простую базу данных SQLite.

Пример 5.1 ❖ hello.py: настройка базы данных

```
from flask.ext.sqlalchemy import SQLAlchemy

basedir = os.path.abspath(os.path.dirname(__file__))

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = \
    'sqlite:/// ' + os.path.join(basedir, 'data.sqlite')
app.config['SQLALCHEMY_COMMIT_ON_TEARDOWN'] = True

db = SQLAlchemy(app)
```

В переменной `db` сохраняется экземпляр класса `SQLAlchemy`, представляющий базу данных и обеспечивающий доступ ко всем функциональным возможностям Flask-SQLAlchemy.

Определение модели

Термин *модель* используется для ссылки на хранимые сущности, которые используются приложением. В контексте ORM моделью обычно является класс на языке Python с атрибутами, соответствующими столбцам в таблице.

Экземпляр базы данных из расширения Flask-SQLAlchemy предоставляет базовый класс для определения моделей, а также ряд вспомогательных классов и функций, которые можно использовать для определения структуры данных. Таблицы `roles` и `users`, изображенные на рис. 5.1, можно определить как модели `Role` и `User` в примере 5.2.

Пример 5.2 ❖ `hello.py`: определения моделей `Role` и `User`

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)

    def __repr__(self):
        return '<Role %r>' % self.name

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, index=True)

    def __repr__(self):
        return '<User %r>' % self.username
```

Переменная класса `__tablename__` определяет имя таблицы в базе данных. Расширение Flask-SQLAlchemy может автоматически присвоить переменной `__tablename__` имя таблицы по умолчанию, но при выборе имен по умолчанию расширение не следует соглашению об использовании множественного числа, поэтому лучше указывать имена таблиц явно. Остальные переменные класса, объявленные как экземпляры класса `db.Column`, — это атрибуты модели.

В первом аргументе конструктору `db.Column` передается тип столбца и атрибута модели. В табл. 5.2 перечислены некоторые из поддерживаемых типов столбцов и соответствующие им типы в языке Python, используемые в модели.

Таблица 5.2. Часто используемые типы столбцов в SQLAlchemy

Имя типа	Тип в Python	Описание
<code>Integer</code>	<code>int</code>	Обычное целое число, обычно 32-битное
<code>SmallInteger</code>	<code>int</code>	Короткое целое, обычно 16-битное
<code>BigInteger</code>	<code>int</code> или <code>long</code>	Целое число неограниченной точности
<code>Float</code>	<code>float</code>	Вещественное число
<code>Numeric</code>	<code>decimal.Decimal</code>	Число с фиксированной точкой
<code>String</code>	<code>str</code>	Строка переменной длины


Таблица 5.2 (окончание)

Имя типа	Тип в Python	Описание
Text	str	Строка переменной длины, оптимизированная для хранения больших строк
Unicode	unicode	Строка переменной длины с символами Юникода
UnicodeText	unicode	Строка переменной длины с символами Юникода, оптимизированная для хранения больших строк
Boolean	bool	Логическое значение
Date	datetime.date	Дата
Time	datetime.time	Время
DateTime	datetime.datetime	Дата и время
Interval	datetime.timedelta	Интервал времени
Enum	str	Список строк
PickleType	Любой объект на языке Python	Автоматически сериализуется с помощью модуля Pickle
LargeBinary	str	Двоичный объект

Остальные аргументы `db.Column` определяют дополнительные настройки атрибутов. В табл. 5.3 перечислены некоторые из поддерживаемых настроек.

Таблица 5.3. Часто используемые настройки для столбцов в SQLAlchemy

Имя настройки	Описание
primary_key	Если установлена в <code>True</code> , столбец таблицы будет использоваться как первичный ключ
unique	Если установлена в <code>True</code> , для этого столбца будет запрещено сохранять не уникальные значения
index	Если установлена в <code>True</code> , для столбца таблицы будет создан индекс, что обеспечит более эффективное выполнение запросов
nullable	Если установлена в <code>True</code> , в этом столбце будет разрешено сохранять пустые значения. Если установлена в <code>False</code> , в этом столбце будет запрещено сохранять пустые значения
default	Определяет значение по умолчанию для столбца

 Расширение Flask-SQLAlchemy требует, чтобы все модели определяли первичные ключи, для которых обычно выбирается столбец с именем `id`.

Обе модели в примере 5.2 определяют методы `__repr__()`, возвращающие экземпляры модели в удобочитаемом строковом представ-

лении. Определение этого метода не является обязательным условием, но он может пригодиться во время отладки и тестирования.

Отношения

Связи между строками из разных таблиц в реляционных базах данных устанавливаются с помощью отношений. Реляционная схема на рис. 5.1 отражает простое отношение между пользователями и их ролями. Это отношение называется «один ко многим», потому что одна и та же роль может быть назначена многим пользователям, а каждому конкретному пользователю может быть присвоена только одна роль.

Пример 5.3 показывает, как определить отношение «один ко многим» в классах модели.

Пример 5.3 ❖ hello.py: отношения

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role')

class User(db.Model):
    # ...
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Как показано на рис. 5.1, отношение связывает две строки посредством внешнего ключа в таблице `users`. Здесь столбец `role_id`, добавленный в модель `User`, определяется как внешний ключ, устанавливающий отношение. Аргумент `'roles.id'` в вызове `db.ForeignKey()` указывает, что значение данного столбца должно интерпретироваться как значение столбца `id` строк из таблицы `roles`.

Атрибут `users`, добавленный в модель `Role`, – это объектно-ориентированное представление отношения. В данном случае при обращении к атрибуту `users` будет возвращаться список пользователей, которым присвоена данная роль. Первый аргумент в вызове `db.relationship()` определяет модель на другом конце отношения. Имя модели можно определить в виде строки, если класс к этому моменту еще не определен.

Аргумент `backref` в вызове `db.relationship()` определяет обратную ссылку отношения – атрибут `role` в модели `User`. Данный атрибут можно использовать вместо `role_id` для доступа к модели `Role` как к объекту.

В большинстве случаев метод `db.relationship()` может сам найти внешний ключ отношения, но иногда ему не удастся определить,

какой столбец используется в качестве внешнего ключа. Например, если в модели `User` определить два или более внешних ключей, ссылающихся на модель `Role`, фреймворк `SQLAlchemy` не сможет определить, какой из них следует использовать. Всякий раз, когда возникает подобная неоднозначность, в вызов `db.relationship()` следует передавать дополнительные аргументы. В табл. 5.4 перечислены некоторые из аргументов, часто используемых для определения отношения.

Таблица 5.4. Часто используемые параметры настройки отношений в SQLAlchemy

Имя настройки	Описание
<code>backref</code>	Добавляет обратную ссылку на другую модель, участвующую в отношении
<code>primaryjoin</code>	Явно определяет условие соединения двух моделей. Эта настройка необходима только в случае неоднозначности
<code>lazy</code>	Определяет, как должны извлекаться элементы, связанные отношением. Возможными значениями являются: <code>select</code> (элементы извлекаются при первой попытке обращения к ним), <code>immediate</code> (элементы извлекаются в момент извлечения исходного объекта), <code>joined</code> (элементы извлекаются немедленно, но как соединение (<code>join</code>)), <code>subquery</code> (элементы извлекаются немедленно, но как подзапрос), <code>noload</code> (элементы никогда не извлекаются) и <code>dynamic</code> (вместо извлечения элементов возвращается запрос, который может их извлечь)
<code>uselist</code>	Если установлена в значение <code>False</code> , используется скаляр вместо списка
<code>order_by</code>	Определяет порядок сортировки элементов, связанных отношением
<code>secondary</code>	Определяет имя ассоциативной таблицы при использовании отношения «многие ко многим»
<code>secondaryjoin</code>	Определяет вторичное условие соединения при использовании отношения «многие ко многим», когда <code>SQLAlchemy</code> оказывается не в состоянии определить его самостоятельно



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 5a` и получить эту версию приложения.

Помимо отношения «один ко многим», существуют и другие типы отношений. Отношение «один к одному» выражается точно так же, как «один ко многим», но в параметре `uselist` методу `db.relationship()` передается значение `False`. Отношение «многие к одному» тоже можно выразить как «один ко многим», если поменять таблицы местами или создать внешний ключ и определить отношение `db.relationship()`

на стороне «многие». Самый сложный тип отношений – «многие ко многим». Для определения отношения этого типа требуется дополнительная таблица, которую называют *ассоциативной*. Пoblйже с отношениями «многие ко многим» вы познакомитесь в главе 12.

Операции с базами данных

Теперь модели полностью соответствуют схеме на рис. 5.1 и готовы к использованию. Для изучения приемов работы с моделями лучше всего воспользоваться интерактивной оболочкой Python. В следующих разделах рассказывается о наиболее типичных операциях с базами данных.

Создание таблиц

Самое первое, что следует сделать, – дать команду расширению Flask-SQLAlchemy создать базу данных на основе классов моделей. Эту операцию выполняет функция `db.create_all()`:

```
(venv) $ python hello.py shell
>>> from hello import db
>>> db.create_all()
```

Если теперь проверить каталог приложения, можно увидеть новый файл с именем *data.sqlite*, которое было указано в настройках базы данных SQLite. Функция `db.create_all()` не будет пытаться повторно создать или обновить таблицы, если они уже существуют в базе данных. Это не совсем удобно на этапе разработки, когда модели могут изменяться довольно часто и желательно, чтобы эти изменения отражались на базе данных. Самый простой и грубый способ обновить существующие таблицы – предварительно удалить их:

```
>>> db.drop_all()
>>> db.create_all()
```

К сожалению, этот прием имеет нежелательный побочный эффект – он уничтожает все данные в старой базе данных. Более удачное решение этой проблемы будет представлено ближе к концу главы.

Вставка строк

Следующий пример создает несколько ролей и пользователей:

```
>>> from hello import Role, User
>>> admin_role = Role(name='Admin')
>>> mod_role = Role(name='Moderator')
>>> user_role = Role(name='User')
```



```
>>> user_john = User(username='john', role=admin_role)
>>> user_susan = User(username='susan', role=user_role)
>>> user_david = User(username='david', role=user_role)
```

Конструкторы моделей принимают начальные значения атрибутов в виде именованных аргументов. Обратите внимание, что можно использовать и атрибут `role`, даже при том, что он является не столбцом в базе данных, а высокоуровневым представлением отношения «один ко многим». Атрибут `id` в новых объектах не требуется устанавливать явно: управление первичными ключами осуществляется самим расширением Flask-SQLAlchemy. Объекты существуют пока только в программе на языке Python; они еще не были записаны в базу данных. Как следствие их атрибуты `id` пока не имеют значений:

```
>>> print(admin_role.id)
None
>>> print(mod_role.id)
None
>>> print(user_role.id)
None
```

Изменения в базу данных вносятся с помощью *сеанса* базы данных, который в расширении Flask-SQLAlchemy доступен в виде объекта `db.session`. Чтобы подготовить объекты к записи в базу данных, их следует добавить в сеанс:

```
>>> db.session.add(admin_role)
>>> db.session.add(mod_role)
>>> db.session.add(user_role)
>>> db.session.add(user_john)
>>> db.session.add(user_susan)
>>> db.session.add(user_david)
```

То же самое можно записать более кратко:


```
>>> db.session.add_all([admin_role, mod_role, user_role,
...     user_john, user_susan, user_david])
```

Чтобы сохранить объекты в базе данных, нужно подтвердить изменения в сеансе вызовом метода `commit()`:


```
>>> db.session.commit()
```

Проверим, были ли установлены атрибуты `id` на этот раз:

```
>>> print(admin_role.id)
1
>>> print(mod_role.id)
2
>>> print(user_role.id)
3
```

 Объект сеанса базы данных `db.session` никак не связан с объектом сеанса `session` в ядре фреймворка Flask, обсуждавшимся в главе 4. Сеансы баз данных также называют транзакциями.

Сеансы баз данных оказывают неоценимую помощь в поддержании непротиворечивости данных. Операция подтверждения (`commit`) выполняет запись всех объектов, которые были добавлены в сеанс, атомарно. Если в процессе записи произойдет ошибка, все изменения целиком будут отменены. Если вы всегда будете подтверждать взаимосвязанные изменения как одно целое, вы сможете избежать нарушения целостности данных, которое может возникать при подтверждении изменений по частям, когда часть изменений удалось подтвердить, а часть – нет.

 Сеанс базы данных можно также откатить (отменить). Если вызвать метод `db.session.rollback()`, состояние любых объектов, добавленных в сеанс базы данных, будет приведено в соответствие с текущим их состоянием в базе данных.

Изменение строк

Метод `add()` сеанса базы данных можно также использовать для обновления моделей. Продолжая тот же сеанс интерактивной оболочки, следующий пример переименовывает роль "Admin" в "Administrator":

```
>>> admin_role.name = 'Administrator'
>>> db.session.add(admin_role)
>>> db.session.commit()
```

Удаление строк

Сеанс базы данных имеет также метод `delete()`. Следующий пример удаляет роль "Moderator" из базы данных:

```
>>> db.session.delete(mod_role)
>>> db.session.commit()
```

Обратите внимание, что фактическое удаление, так же как вставка и обновление, выполняется в момент подтверждения сеанса.

Извлечение строк

Для каждого класса-модели расширение Flask-SQLAlchemy создает объект запроса `query`. В самом простом случае объект запроса возвращает все содержимое соответствующей таблицы:

```
>>> Role.query.all()
[<Role u'Administrator'>, <Role u'User'>]
>>> User.query.all()
[<User u'john'>, <User u'susan'>, <User u'david'>]
```

Объект запроса можно настроить на выполнение более специализированного поиска с применением фильтров. Следующий пример выполняет поиск всех пользователей, которым присвоена роль "User":

```
>>> User.query.filter_by(role=user_role).all()
[<User u'susan'>, <User u'david'>]
```

Также возможно получить SQL-код, который фреймворк SQLAlchemy генерирует для данного запроса, преобразовав объект запроса в строку:

```
>>> str(User.query.filter_by(role=user_role))
'SELECT users.id AS users_id, users.username AS users_username,
users.role_id AS users_role_id FROM users WHERE :param_1 = users.role_id'
```

Если теперь покинуть интерактивную оболочку, объекты, созданные в предыдущих примерах, перестанут существовать как программные объекты, но продолжат существовать как записи в соответствующих таблицах базы данных. Если потом запустить новый сеанс интерактивной оболочки Python, можно будет воссоздать объекты Python из записей в базе данных. Следующий пример выполняет запрос, загружающий информацию о пользователях с ролью "User":

```
>>> user_role = Role.query.filter_by(name='User').first()
```

Фильтры, такие как `filter_by()`, возвращают новый запрос. Поддерживается возможность применять целые последовательности фильтров для получения требуемого запроса.

В табл. 5.5 перечислены некоторые часто используемые фильтры. Полный их список можно найти в документации к фреймворку SQLAlchemy: http://docs.sqlalchemy.org/en/rel_0_9/.

Таблица 5.5. Часто используемые фильтры запросов SQLAlchemy

Фильтр	Описание
<code>filter()</code>	Возвращает новый объект запроса, добавляющий в оригинальный запрос новый фильтр
<code>filter_by()</code>	Возвращает новый объект запроса, добавляющий новый фильтр проверки на равенство
<code>limit()</code>	Возвращает новый объект запроса, ограничивающий число результатов заданным количеством
<code>offset()</code>	Возвращает новый объект запроса, осуществляющий смещение относительно начала результатов, возвращаемых оригинальным запросом
<code>order_by()</code>	Возвращает новый объект запроса, выполняющий сортировку результатов в соответствии с указанными критериями
<code>group_by()</code>	Возвращает новый объект запроса, группирующий результаты оригинального запроса в соответствии с указанными критериями

После применения необходимых фильтров к запросу можно вызвать метод `all()`, чтобы выполнить запрос и получить результаты в виде списка. Но существуют и другие способы выполнения запросов. В табл. 5.6 перечислены иные методы, выполняющие запрос.

Таблица 5.6. Часто используемые методы выполнения запросов из SQLAlchemy

Метод	Описание
<code>all()</code>	Возвращает все результаты в виде списка
<code>first()</code>	Возвращает первый результат или <code>None</code> , если результаты отсутствуют
<code>first_or_404()</code>	Возвращает первый результат. Если результаты отсутствуют, прерывает обработку HTTP-запроса, возвращая код ошибки 404
<code>get()</code>	Возвращает строку, соответствующую указанному значению первичного ключа, или <code>None</code> , если результаты отсутствуют
<code>get_or_404()</code>	Возвращает строку, соответствующую указанному значению первичного ключа. Если результаты отсутствуют, прерывает обработку HTTP-запроса, возвращая код ошибки 404
<code>count()</code>	Возвращает число результатов
<code>paginate()</code>	Возвращает объект <code>Pagination</code> , содержащий результаты в указанном диапазоне

Отношения действуют подобно запросам. Следующий пример обращается к отношению «один ко многим», связывающему таблицы `roles` и `users`:

```
>>> users = user_role.users
>>> users
[<User u'susan'>, <User u'david'>]
>>> users[0].role
<Role u'User'>
```

Выражение `user_role.users` имеет небольшую проблему. При обращении к выражению `user_role.users` запускается неявный запрос, который выполняется методом `all()`. В результате возвращается список пользователей. Так как объект запроса скрыт, к нему нельзя применить дополнительные фильтры. В данном конкретном примере было бы удобно, если бы возвращался список пользователей, отсортированный по алфавиту. В примере 5.4 демонстрируется изменение поведения отношения передачей аргумента `lazy = 'dynamic'`, в результате чего запрос не будет выполняться автоматически.

Пример 5.4 ❖ `app/models.py`: динамические отношения

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role', lazy='dynamic')
    # ...
```

После такой настройки отношения `user_role.users` будет возвращать еще не выполненный запрос, к которому можно применить необходимые фильтры:

```
>>> user_role.users.order_by(User.username).all()
[<User u'david'>, <User u'susan'>]
>>> user_role.users.count()
2
```

Операции с базой данных в функциях представления

Операции с базами данных, описанные в предыдущих разделах, можно выполнять непосредственно внутри функций представления. В примере 5.5 демонстрируется новая версия маршрута главной страницы, которая записывает в базу данных имена, введенные пользователями.

Пример 5.5 ❖ hello.py: выполнение операций с базой данных в функциях представления

```
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.name.data).first()
        if user is None:
            user = User(username = form.name.data)
            db.session.add(user)
            session['known'] = False
        else:
            session['known'] = True
        session['name'] = form.name.data
        form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html',
        form = form, name = session.get('name'),
        known = session.get('known', False))
```

Эта измененная версия приложения проверяет присутствие введенного пользователем имени в базе данных с помощью фильтра `filter_by()` запроса и устанавливает переменную `known` в сеансе пользователя, чтобы после переадресации эту информацию можно было передать в шаблон, где она используется при конструировании текста приветствия. Обратите внимание, что для нормальной работы приложения необходимо заранее создать таблицы в базе данных, как это было показано выше.

Новая версия соответствующего шаблона представлена в примере 5.6. По значению аргумента `known` этот шаблон определяет, какой текст вставить во вторую строку приветствия.

Пример 5.6 ❖ `templates/index.html`

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
    {% if not known %}
    <p>Pleased to meet you!</p>
    {% else %}
    <p>Happy to see you again!</p>
    {% endif %}
</div>
{{ wtf.quick_form(form) }}
{% endblock %}
```



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 5b` и получить эту версию приложения.

Интеграция с интерактивной оболочкой Python

Необходимость импортирования экземпляра базы данных и моделей при каждом запуске интерактивного сеанса кому-то может показаться утомительной. Чтобы не вводить снова и снова инструкции импорта, можно с помощью расширения `Flask-Script` настроить автоматическое импортирование некоторых объектов в интерактивную оболочку.

Чтобы добавить объекты в список импорта команды `shell`, их необходимо зарегистрировать с помощью функции `make_context`, как показано в примере 5.7.

Пример 5.7 ❖ `hello.py`: добавление контекста команды `shell`

```
from flask.ext.script import Shell

def make_shell_context():
    return dict(app=app, db=db, User=User, Role=Role)
manager.add_command("shell", Shell(make_context=make_shell_context))
```

Функция `make_shell_context()` регистрирует приложение, экземпляр базы данных и модели, благодаря чему они автоматически будут импортироваться в интерактивную оболочку:

```
$ python hello.py shell
>>> app
<Flask 'app'>
>>> db
<SQLAlchemy engine='sqlite:///home/flask/flasky/data.sqlite'>
>>> User
<class 'app.User'>
```



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 5b` и получить эту версию приложения.

Миграция базы данных с помощью Flask-Migrate

В процессе разработки приложения часто возникает необходимость изменять модели базы данных, и когда такое случается, требуется обновить структуру самой базы данных.

Расширение `Flask-SQLAlchemy` создает таблицы в базе данных, только когда они отсутствуют, поэтому единственный путь обновить структуру таблиц – предварительно уничтожить старые таблицы, но этот путь ведет к полной потере всех данных.

Более удачное решение заключается в использовании фреймворка *миграции базы данных (database migration)*. Так же, как система управления версиями исходного кода следит за изменениями в файлах с исходными текстами, фреймворк миграции базы данных следит за изменениями в схеме базы данных и может последовательно применять их.

Ведущий разработчик `SQLAlchemy` написал фреймворк миграции базы данных `Alembic`¹, однако приложения на основе `Flask` могут использовать расширение `Flask-Migrate`² – легковесную обертку вокруг фреймворка `Alembic`, которая интегрируется с расширением `Flask-Script` и позволяет выполнять все операции через команды `Flask-Script`.

¹ <http://bit.ly/alembic-doc>.

² <http://bit.ly/fl-migrate>.

Создание репозитория миграции

Для начала расширение Flask-Migrate необходимо установить в виртуальное окружение:

```
(venv) $ pip install flask-migrate
```

В примере 5.8 показано, как инициализировать это расширение.

Пример 5.8 ❖ hello.py: настройка расширения Flask-Migrate

```
from flask.ext.migrate import Migrate, MigrateCommand
```

```
# ...
```

```
migrate = Migrate(app, db)
manager.add_command('db', MigrateCommand)
```

Для экспортирования команд миграции расширение Flask-Migrate предоставляет класс `MigrateCommand`, подключаемый к объекту `manager` из расширения Flask-Script. В этом примере выполняется подключение команды `db`.

Перед тем как можно будет выполнить миграцию базы данных, необходимо с помощью подкоманды `init` создать репозиторий миграции:

```
(venv) $ python hello.py db init
Creating directory /home/flask/flasky/migrations...done
Creating directory /home/flask/flasky/migrations/versions...done
Generating /home/flask/flasky/migrations/alembic.ini...done
Generating /home/flask/flasky/migrations/env.py...done
Generating /home/flask/flasky/migrations/env.pyc...done
Generating /home/flask/flasky/migrations/README...done
Generating /home/flask/flasky/migrations/script.py.mako...done
Please edit configuration/connection/logging settings in
'/home/flask/flasky/migrations/alembic.ini' before proceeding.
```

Эта команда создаст папку *migrations*, где будут храниться все сценарии миграции.



Файлы в репозитории миграции всегда должны сохраняться в системе управления версиями, вместе с остальными исходными файлами приложения.

Создание сценария миграции

В фреймворке Alembic миграция базы данных реализуется в виде *сценария миграции (migration script)*. Этот сценарий имеет две функции: `upgrade()` и `downgrade()`. Функция `upgrade()` применяет изменения в базе данных, являющиеся частью миграции, а функция `downgrade()` отменяет их. Благодаря возможности применять и отменять измене-

ния с помощью Alembic можно привести базу данных к состоянию на любой момент в истории изменений.

Сценарии миграции для Alembic можно создавать вручную или автоматически, используя команды `revision` и `migrate` соответственно. При работе в ручном режиме фреймворк создаст заготовку сценария с пустыми функциями `upgrade()` и `downgrade()`, которые разработчик должен реализовать самостоятельно, с использованием директив, экспортируемых объектом `Operations`. При работе в автоматическом режиме фреймворк сам сгенерирует код для функций `upgrade()` и `downgrade()`, определяя различия между определениями моделей и текущей структурой базы данных.



Сценарии миграции, созданные автоматически, не всегда точны и могут пропускать некоторые детали. Поэтому их всегда следует просматривать перед применением.

Автоматическое создание сценария миграции выполняется подкомандой `migrate`:

```
(venv) $ python hello.py db migrate -m "initial migration"
INFO [alembic.migration] Context impl SQLiteImpl.
INFO [alembic.migration] Will assume non-transactional DDL.
INFO [alembic.autogenerate] Detected added table 'roles'
INFO [alembic.autogenerate] Detected added table 'users'
INFO [alembic.autogenerate.compare] Detected added index
'ix_users_username' on '['username']'
Generating /home/flask/flasky/migrations/versions/1bc
594146bb5_initial_migration.py...done
```



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 5d` и получить эту версию приложения. Обратите внимание, что для этой версии приложения не требуется генерировать сценарий миграции, потому что все сценарии миграции включены в репозиторий с исходными текстами.

Обновление базы данных

После того как вы просмотрите сценарий миграции и исправите все недостатки, его можно применить к базе данных командой `db upgrade`:

```
(venv) $ python hello.py db upgrade
INFO [alembic.migration] Context impl SQLiteImpl.
INFO [alembic.migration] Will assume non-transactional DDL.
INFO [alembic.migration] Running upgrade None -> 1bc594146bb5, initial migration
```

Для первой миграции эта команда фактически эквивалентна вызову `db.create_all()`, но в последующих миграциях команда `upgrade` будет применять изменения, не уничтожая содержимого таблиц.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, удалите файл `data.sqlite` базы данных и затем выполните команду `upgrade`, чтобы сгенерировать базу данных с помощью фреймворка миграции.

Тема проектирования и использования баз данных очень важна; ей посвящены целые книги. Вы должны рассматривать эту главу лишь как краткий обзор; далее в книге мы обсудим более сложные темы. Следующая глава посвящена приемам отправки электронной почты.

Глава 6

Электронная почта

Во многих приложениях необходима возможность извещать пользователей об определенных событиях, и обычно для этого используется электронная почта. Несмотря на то что для рассылки электронных писем можно использовать пакет *smtplib* из стандартной библиотеки Python, в приложениях на основе Flask можно использовать более удачное решение – расширение Flask-Mail, являющееся оберткой вокруг пакета *smtplib*.

Поддержка электронной почты с помощью Flask-Mail

Расширение Flask-Mail можно установить с помощью утилиты `pip`:

```
(venv) $ pip install flask-mail
```

Это расширение обеспечивает возможность соединения с SMTP-сервером (Simple Mail Transfer Protocol – простой протокол передачи электронной почты) и передачи ему электронных писем для рассылки. По умолчанию Flask-Mail пытается соединиться с портом 25 сервера *localhost* и отправить почту без аутентификации. В табл. 6.1 приводится список ключей параметров настройки соединения с SMTP-сервером.

Таблица 6.1. Параметры настройки соединения с SMTP-сервером для расширения Flask-Mail

Ключ	По умолчанию	Описание
MAIL_HOSTNAME	<i>localhost</i>	Имя хоста или IP-адрес сервера электронной почты
MAIL_PORT	25	Порт сервера электронной почты
MAIL_USE_TLS	False	Управляет использованием протокола безопасности на транспортном уровне TLS (Transport Layer Security)

Таблица 6.1 (окончание)

Ключ	По умолчанию	Описание
MAIL_USE_SSL	False	Управляет использованием протокола защищенных сокетов SSL (Secure Sockets Layer)
MAIL_USERNAME	None	Имя пользователя на сервере электронной почты
MAIL_PASSWORD	None	Пароль пользователя на сервере электронной почты

В процессе разработки может быть более удобным соединяться с внешним SMTP-сервером. В примере 6.1 показано, как настроить приложение для отправки электронной почты с использованием учетной записи в Google Gmail.

Пример 6.1 ❖ `hello.py`: настройка Flask-Mail для использования Gmail

```
import os
# ...
app.config['MAIL_SERVER'] = 'smtp.googlemail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME')
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD')
```



Никогда не указывайте явно имя пользователя и пароль в программном коде, особенно если планируете открыть исходные тексты своего приложения. Чтобы защитить информацию о своей учетной записи, импортируйте ее из переменных окружения.

В примере 6.2 показано, как инициализировать Flask-Mail.

Пример 6.2 ❖ `hello.py`: инициализация Flask-Mail

```
from flask.ext.mail import Mail
mail = Mail(app)
```

В окружении необходимо определить две переменные для хранения имени пользователя и пароля для соединения с почтовым сервером. Если вы пользуетесь ОС Linux или Mac OS X и командной оболочкой `bash`, определить эти переменные можно так:

```
(venv) $ export MAIL_USERNAME=<Gmail username>
(venv) $ export MAIL_PASSWORD=<Gmail password>
```

А в Microsoft Windows так:

```
(venv) $ set MAIL_USERNAME=<Gmail username>
(venv) $ set MAIL_PASSWORD=<Gmail password>
```

Отправка электронной почты из интерактивной оболочки Python

Чтобы проверить настройки, запустите сеанс командной оболочки и отправьте тестовое сообщение:

```
(venv) $ python hello.py shell
>>> from flask.ext.mail import Message
>>> from hello import mail
>>> msg = Message('test subject', sender='you@example.com',
...     recipients=['you@example.com'])
>>> msg.body = 'text body'
>>> msg.html = '<b>HTML</b> body'
>>> with app.app_context():
...     mail.send(msg)
...
```

Обратите внимание, что функция `send()` из расширения Flask-Mail использует `current_app`, поэтому ее следует вызывать только после активации контекста приложения.

Интеграция поддержки электронной почты в приложение

Чтобы избежать необходимости каждый раз создавать электронные письма вручную, было бы неплохо объединить все этапы подготовки и отправки сообщений в единую функцию. Как дополнительное преимущество эта функция могла бы отображать тело электронного письма с применением шаблонов Jinja2. Реализация такой функции показана в примере 6.3.

Пример 6.3 ❖ hello.py: поддержка электронной почты

```
from flask.ext.mail import Message
app.config['FLASKY_MAIL_SUBJECT_PREFIX'] = '[Flasky]'
app.config['FLASKY_MAIL_SENDER'] = 'Flasky Admin <flasky@example.com>'

def send_email(to, subject, template, **kwargs):
    msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
                  sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    mail.send(msg)
```

Функция `send_email` опирается на два параметра настройки в приложении, определяющие префикс для строки темы письма и адрес отправителя. Она принимает адрес получателя, строку темы, шаблон

для отображения тела письма и список именованных аргументов. Имя файла шаблона должно быть указано без расширения, благодаря чему можно использовать два шаблона – для простой текстовой версии тела письма и для версии в формате HTML. Именованные аргументы, полученные от вызывающего кода, передаются в вызовы `render_template()` и могут использоваться шаблонами для создания тела письма.

Функцию представления `index()` легко можно дополнить возможностью отправки электронного письма администратору при получении с формой нового имени. Эти дополнения приводятся в примере 6.4.

Пример 6.4 ❖ `hello.py`: пример использования электронной почты

```
# ...
app.config['FLASKY_ADMIN'] = os.environ.get('FLASKY_ADMIN')
# ...
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.name.data).first()
        if user is None:
            user = User(username=form.name.data)
            db.session.add(user)
            session['known'] = False
            if app.config['FLASKY_ADMIN']:
                send_email(app.config['FLASKY_ADMIN'], 'New User',
                           'mail/new_user', user=user)
        else:
            session['known'] = True
            session['name'] = form.name.data
            form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'),
                           known=session.get('known', False))
```

Адрес получателя определяется из переменной окружения `FLASKY_ADMIN`, значение которой копируется в параметр настройки с тем же именем на этапе запуска приложения. Необходимо также создать два файла шаблонов для текстовой и HTML-версии письма. Предполагается, что эти файлы будут храниться в папке *mail*, вложенной в папку *templates*. Сделано это с целью отделить их от обычных шаблонов. Шаблоны электронных писем ожидают получить имя пользователя в виде аргумента, поэтому в вызов `send_email()` передается именованный аргумент.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 6a` и получить эту версию приложения.

Помимо переменных окружения `MAIL_USERNAME` и `MAIL_PASSWORD`, описанных выше, для этой версии приложения необходимо определить переменную окружения `FLASKY_ADMIN`. В Linux и Mac OS X это можно сделать так:

```
(venv) $ export FLASKY_ADMIN=<your-email-address>
```

А в Microsoft Windows так:

```
(venv) $ set FLASKY_ADMIN=<Gmail username>
```

После создания этих переменных окружения можно проверить, действительно ли каждый раз при вводе нового имени приложение посылает электронное письмо.

Асинхронная отправка электронной почты

Отправив несколько электронных писем, можно заметить, что функция `mail.send()` блокирует работу приложения на несколько секунд, пока письмо не будет отправлено, что создает ощущение зависания браузера. Чтобы избежать нежелательных задержек в процессе обработки запроса, функцию отправки электронной почты можно выполнять в фоновом потоке. Необходимые для этого изменения представлены в примере 6.5.

Пример 6.5 ❖ hello.py: асинхронная отправка электронной почты

```
from threading import Thread

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(to, subject, template, **kwargs):
    msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
                  sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    thr = Thread(target=send_async_email, args=[app, msg])
    thr.start()
    return thr
```

Эта реализация подчеркивает одну интересную проблему. Многим расширениям фреймворка Flask для работы необходим доступ к контекстам приложения и запроса. Функция `send()` из расшире-

ния Flask-Mail использует `current_app`, поэтому ей также необходим доступ к контексту приложения. Но когда `mail.send()` выполняется в отдельном потоке, для нее можно создать искусственный контекст приложения вызовом `app.app_context()`.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 6b` и получить эту версию приложения.

Если теперь запустить приложение, можно заметить, что задержки исчезли, но имейте в виду, что для массовой рассылки электронной почты желательно создавать отдельное задание, вместо того чтобы запускать отдельный поток выполнения для отправки каждого письма. Например, выполнение функции `send_async_email()` можно передавать в очередь заданий Celery¹.

Эта глава завершает обзор особенностей, которые необходимы практически во всех веб-приложениях. Наша проблема теперь состоит в том, что сценарий *hello.py* постепенно становится все больше и больше, что усложняет работу с ним. В следующей главе вы узнаете, как структурировать крупные приложения.

¹ <http://www.celeryproject.org/>.

Глава 7

Структура больших приложений

Хранить исходный код небольшого веб-приложения в одном файле может быть и удобно, но такой подход точно не годится для больших приложений. С ростом сложности приложения работать со все увеличивающимся единственным файлом будет все труднее и труднее.

В отличие от большинства других веб-фреймворков, Flask не выдвигает никаких требований к организации крупных проектов; определение структуры приложения полностью возлагается на разработчика. В этой главе мы рассмотрим один из возможных способов организации больших приложений в пакеты и модули. Эта структура будет использоваться во всех остальных примерах.

Структура проекта

В примере 7.1 показана базовая структура типичного приложения на основе Flask.

Пример 7.1 ❖ Базовая структура приложения на основе Flask

```
| -flasky
|   |-app/
|       |-templates/
|       |-static/
|       |-main/
|           |-__init__.py
|           |-errors.py
|           |-forms.py
|           |-views.py
|       |-__init__.py
|       |-email.py
|       |-models.py
|   |-migrations/
|   |-tests/
|       |-__init__.py
|       |-test*.py
```

```
| -venv/  
| -requirements.txt  
| -config.py  
| -manage.py
```

Эта структура имеет четыре уровня вложенности папок:

- само приложение обычно располагается в пакете с именем *app*;
- папка *migrations* содержит сценарии миграции базы данных, как описывалось выше;
- модульные тесты хранятся в пакете *tests*;
- папка *venv* содержит виртуальное окружение Python, о котором рассказывалось выше.

Здесь также появилось несколько новых файлов:

- *requirements.txt* – список зависимостей пакетов, чтобы упростить воссоздание идентичного виртуального окружения на другом компьютере;
- *config.py* – хранит настройки приложения;
- *manage.py* – запускает приложение и другие прикладные задачи.

Чтобы помочь вам лучше понять эту структуру, в следующих разделах описывается процесс преобразования в нее приложения *hello.py*.

Параметры настройки

Приложения часто нуждаются во множестве параметров настройки. Отличным примером этого могут служить разные настройки базы данных для этапов разработки, тестирования и эксплуатации, чтобы исключить возможность взаимовлияния.

Вместо простой структуры, подобной словарю, как в приложении *hello.py*, для хранения настроек можно использовать иерархию конфигурационных классов. В примере 7.2 приводится содержимое файла *config.py*.

Пример 7.2 ❖ config.py: конфигурация приложения

```
import os  
basedir = os.path.abspath(os.path.dirname(__file__))  
  
class Config:  
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'hard to guess string'  
    SQLALCHEMY_COMMIT_ON_TEARDOWN = True  
    FLASKY_MAIL_SUBJECT_PREFIX = '[Flasky]'  
    FLASKY_MAIL_SENDER = 'Flasky Admin <flasky@example.com>'  
    FLASKY_ADMIN = os.environ.get('FLASKY_ADMIN')  
  
    @staticmethod
```

```

def init_app(app):
    pass

class DevelopmentConfig(Config):
    DEBUG = True
    MAIL_SERVER = 'smtp.googlemail.com'
    MAIL_PORT = 587
    MAIL_USE_TLS = True
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') or \
        'sqlite:/// ' + os.path.join(basedir, 'data-dev.sqlite')

class TestingConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') or \
        'sqlite:/// ' + os.path.join(basedir, 'data-test.sqlite')

class ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:/// ' + os.path.join(basedir, 'data.sqlite')

config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,

    'default': DevelopmentConfig
}

```

Базовый класс `Config` содержит настройки, общие для всех конфигураций, а разные подклассы определяют настройки для отдельных конфигураций. По мере необходимости можно добавлять дополнительные конфигурации.

Для повышения гибкости и безопасности конфигураций некоторые настройки можно импортировать из переменных окружения. Например, значение `SECRET_KEY` можно устанавливать в окружении и предусматривать значение по умолчанию на случай отсутствия данной настройки в окружении.

Переменной `SQLALCHEMY_DATABASE_URI` присваиваются разные значения в каждой из трех конфигураций. Это позволяет приложению использовать разные базы данных в разных конфигурациях.

Классы конфигураций можно определять в методе класса `init_app()`, принимающем экземпляр приложения в виде аргумента. Здесь же можно выполнять операции по инициализации в зависимости от выбранной конфигурации. Пока базовый класс `Config` реализует пустой метод `init_app()`.

Внизу сценария *config.py* выполняется регистрация разных конфигураций в словаре `config`. Одна из конфигураций (в данном случае – для разработки) регистрируется как конфигурация по умолчанию.

Пакет приложения

Пакет приложения хранит весь прикладной код, шаблоны и статические файлы. В данном случае он имеет простое имя *app*, однако при желании ему можно дать любое другое имя. Папки *templates* и *static* являются частью пакета приложения, поэтому эти две папки помещены внутрь папки *app*. Модели базы данных и функции поддержки электронной почты также помещены внутрь этого пакета в виде отдельных модулей *app/models.py* и *app/email.py*.

Фабричная функция приложения

Способ создания приложения в версии с единственным файлом достаточно удобен, но он имеет один большой недостаток. Так как приложение создается в глобальной области видимости, нет никакой возможности динамически изменить его конфигурацию: к моменту запуска сценария экземпляр приложения уже будет создан, поэтому менять что-то в конфигурации уже слишком поздно. Этот недостаток особенно сильно мешает на этапе модульного тестирования, когда бывает желательно опробовать приложение с разными настройками.

Решение проблемы заключается в том, чтобы переместить создание экземпляра приложения в фабричную функцию, которую можно явно вызывать из сценария. Это не только даст сценарию время на выполнение настроек, но также позволит создавать несколько экземпляров приложения – иногда это очень может пригодиться при тестировании. Фабричная функция приложения, представленная в примере 7.3, определяется в конструкторе пакета *app*.

Этот конструктор импортирует большую часть используемых расширений фреймворка Flask, но, так как к этому моменту экземпляр приложения пока отсутствует, он создает их неинициализированными, вызывая конструкторы расширений без аргументов. Функция `create_app()` является фабричной функцией приложения и принимает аргумент с именем конфигурации. Параметры настройки, хранящиеся в одном из классов, объявленных в *config.py*, можно импортировать непосредственно в приложение с помощью метода `from_object()` объекта `app.config`. Объект с настройками выбирается по имени из словаря `config`. Как только приложение будет создано и настроено, можно

выполнить инициализацию созданных расширений вызовом их методов `init_app()`.

Пример 7.3 ❖ `app/__init__.py`: конструктор пакета приложения

```
from flask import Flask, render_template
from flask.ext.bootstrap import Bootstrap
from flask.ext.mail import Mail
from flask.ext.moment import Moment
from flask.ext.sqlalchemy import SQLAlchemy
from config import config

bootstrap = Bootstrap()
mail = Mail()
moment = Moment()
db = SQLAlchemy()

def create_app(config_name):
    app = Flask(__name__)
    app.config.from_object(config[config_name])
    config[config_name].init_app(app)

    bootstrap.init_app(app)
    mail.init_app(app)
    moment.init_app(app)
    db.init_app(app)

    # здесь выполняется подключение маршрутов и
    # нестандартных страниц с сообщениями об ошибках

    return app
```

Фабричная функция возвращает созданный экземпляр приложения, но имейте в виду, что приложения, создаваемые фабричной функцией, в этот момент инициализированы не полностью – в них отсутствуют маршруты и обработчики страниц с сообщениями об ошибках. Но об этом мы поговорим в следующем разделе.

Реализация функциональности приложения в виде макета

Переход на использование фабрики приложения создает дополнительные сложности для определения маршрутов. В приложениях, состоящих из единственного сценария, экземпляр приложения существует в глобальной области видимости, поэтому маршруты можно определять с помощью декоратора `app.route`. Но теперь, когда экземпляр приложения создается во время выполнения, декоратор `app.route` начинает существовать только после вызова `create_app()`, то есть

слишком поздно. Аналогичная проблема проявляется и с определением нестандартных обработчиков страниц с сообщениями об ошибках, которые определяются с применением декоратора `app.errorhandler`.

К счастью, фреймворк Flask предлагает отличное решение – *макеты* (*blueprints*). Макет похож на приложение тем, что так же, как приложение, позволяет определять маршруты. Разница лишь в том, что маршруты, ассоциированные с макетом, находятся в пассивном состоянии до момента регистрации в приложении, когда они станут его частью. Используя макет, объявленный в глобальной области видимости, вы можете определять маршруты практически так же, как в приложении, состоящем из единственного сценария.

Подобно приложениям, все макеты можно объявить в одном файле или избрать более структурированный подход, создав несколько модулей в пакете приложения. Для большей гибкости мы создадим вложенный пакет в пакете приложения, где и определим макет. В примере 7.4 приводится реализация конструктора пакета, создающая макет.

Пример 7.4 ❖ `app/main/__init__.py`: создание макета

```
from flask import Blueprint

main = Blueprint('main', __name__)

from . import views, errors
```

Макеты создаются как экземпляры класса `Blueprint`. Конструктор этого класса принимает два аргумента: имя макета и модуль или пакет, где находится макет. Как и в приложениях, в большинстве случаев во втором аргументе можно передавать переменную `__name__`.

Маршруты приложения хранятся в модуле `app/main/views.py` внутри пакета, а обработчики ошибок – в модуле `app/main/errors.py`. Импортирование этих модулей связывает маршруты и обработчики с макетом. Важно отметить, что модули импортируются внизу сценария `app/__init__.py`, чтобы избежать циклических зависимостей, так как `views.py` и `errors.py` должны импортировать главный макет.

Регистрация макета осуществляется внутри фабричной функции приложения `create_app()`, как показано в примере 7.5.

Пример 7.5 ❖ `app/__init__.py`: регистрация макета

```
def create_app(config_name):
    # ...
    from main import main as main_blueprint
    app.register_blueprint(main_blueprint)

    return app
```

В примере 7.6 демонстрируются обработчики ошибок.

Пример 7.6 ❖ `app/main/errors.py`: макет с обработчиками ошибок

```
from flask import render_template
from . import main

@main.app_errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@main.app_errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

Отличие определений обработчиков ошибок внутри макета заключается в используемом декораторе. Если бы использовался декоратор `errorhandler`, обработчики вызывались бы только для обработки ошибок, возникающих внутри макета. Для определения глобальных обработчиков следует использовать декоратор `app_errorhandler`.

В примере 7.7 приводится реализация маршрутов приложения внутри макета.

Пример 7.7 ❖ `app/main/views.py`: макет с маршрутами для приложения

```
from datetime import datetime
from flask import render_template, session, redirect, url_for
from . import main
from .forms import NameForm
from .. import db
from ..models import User

@main.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        # ...
        return redirect(url_for('index'))
    return render_template('index.html',
                           form=form, name=session.get('name'),
                           known=session.get('known', False),
                           current_time=datetime.utcnow())
```

Порядок определения функций представления внутри макета имеет две основные отличительные особенности. Во-первых, как и в случае с обработчиками ошибок в примере выше, используется декоратор маршрута из макета. Второе отличие заключено в использовании функции `url_for()`. Как вы наверняка помните, первый аргумент этой функции определяет имя конечной точки маршрута, каковым в прос-

тых приложениях по умолчанию является имя функции представления. Например, в приложении, реализованном в единственном файле, адрес URL для функции представления `index()` можно получить вызовом `url_for('index')`.

Особенность макетов в том, что ко всем конечным точкам, объявленным в макете, Flask применяет пространство имен, благодаря чему можно создать несколько макетов с функциями представления, имеющими одинаковые имена конечных точек, не создавая конфликтов имен между ними. Пространством имен служит имя макета (первый аргумент в вызове конструктора `Blueprint`), поэтому функция представления `index()` регистрируется с именем конечной точки `main.index`, а соответствующий ей адрес URL можно получить вызовом `url_for('main.index')`.

Кроме того, функция `url_for()` поддерживает сокращенный формат записи имен конечных точек в макете, в соответствии с которым разрешается опускать имя макета, как, например, в вызове `url_for('.index')`. Данная форма записи означает, что для обработки текущего запроса используется макет. Это фактически означает, что для переадресации внутри того же макета можно использовать краткую форму, а для переадресации между макетами следует использовать полностью квалифицированные имена конечных точек.

Для полноты изменений объекты форм также были перенесены внутрь макета, в модуль `app/main/forms.py`.

Сценарий запуска

Файл `manage.py` в корневой папке приложения используется для запуска приложения. Содержимое этого сценария представлено в примере 7.8.

Пример 7.8 ❖ `manage.py`: сценарий запуска

```
#!/usr/bin/env python
import os
from app import create_app, db
from app.models import User, Role
from flask.ext.script import Manager, Shell
from flask.ext.migrate import Migrate, MigrateCommand

app = create_app(os.getenv('FLASK_CONFIG') or 'default')
manager = Manager(app)
migrate = Migrate(app, db)

def make_shell_context():
```



```

    return dict(app=app, db=db, User=User, Role=Role)
manager.add_command("shell", Shell(make_context=make_shell_context))
manager.add_command('db', MigrateCommand)

if __name__ == '__main__':
    manager.run()

```

Сценарий начинается с создания приложения. Конфигурация приложения определяется переменной окружения `FLASK_CONFIG`, если она определена; в противном случае используется конфигурация по умолчанию. Затем производится инициализация расширений Flask-Script, Flask-Migrate и контекста для интерактивной оболочки Python.

Первая строка в сценарии, начинающаяся с пары символов `#!`, добавлена для удобства, чтобы в Unix-подобных операционных системах сценарий мог запускаться командой `./manage.py` вместо более полной команды `python manage.py`.

Файл зависимостей

Приложения должны включать файл *requirements.txt*, в котором перечислены все зависимости с точными номерами версий. Это может пригодиться, если понадобится воссоздать точно такое же виртуальное окружение на другой машине, например там, где приложение будет развернуто для эксплуатации. Этот файл можно сгенерировать автоматически, с помощью утилиты `pip`:

```
(venv) $ pip freeze >requirements.txt
```

Желательно обновлять этот файл после каждой установки или обновления какого-либо пакета. Ниже приводится пример содержимого файла *requirements.txt*:

```

Flask==0.10.1
Flask-Bootstrap==3.0.3.1
Flask-Mail==0.9.0
Flask-Migrate==1.1.0
Flask-Moment==0.2.0
Flask-SQLAlchemy==1.0
Flask-Script==0.6.6
Flask-WTF==0.9.4
Jinja2==2.7.1
Mako==0.9.1
MarkupSafe==0.18
SQLAlchemy==0.8.4
WTForms==1.0.5

```

```
Werkzeug==0.9.4
alembic==0.6.2
blinker==1.3
itsdangerous==0.23
```

Когда понадобится сконструировать точную копию виртуального окружения, можно создать новое виртуальное окружение и выполнить в нем следующую команду:

```
(venv) $ pip install -r requirements.txt
```

Номера версий в примере файла *requirements.txt*, что приводился выше, наверняка окажутся не самыми свежими к моменту, когда вы будете читать эти строки. При желании можно попробовать использовать более новые версии пакетов, а при появлении каких-либо проблем – вернуться к версиям, указанным в файле *requirements.txt*, которые, как известно, совместимы с приложением.

Модульные тесты

Это приложение слишком мало, и в нем почти нечего тестировать, тем не менее в примере 7.9 ниже приводится пара простых тестов.

Пример 7.9 ❖ tests/test_basics.py: модульные тесты

```
import unittest
from flask import current_app
from app import create_app, db

class BasicsTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()


    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_app_exists(self):
        self.assertFalse(current_app is None)

    def test_app_is_testing(self):
        self.assertTrue(current_app.config['TESTING'])
```


В тестах используется пакет `unittest` из стандартной библиотеки Python. Методы `setUp()` и `tearDown()` выполняются до и после каждого

теста, а любые методы с именами, начинающимися с префикса `test_`, выполняются как тесты.

 Если хотите узнать больше о создании модульных тестов с применением пакета `unittest`, обращайтесь к официальной документации: <http://bit.ly/py-unittest>.

Метод `setUp()` пытается создать окружение для теста, близко напоминающее действующее приложение. Сначала он создает экземпляр приложения, настроенный для тестирования, и активирует его контекст. Этот шаг гарантирует доступность `current_app` для теста. Затем он создает новую базу данных, которая может использоваться тестами. База данных и контекст удаляются в методе `tearDown()`.

Первый тест проверяет наличие экземпляра приложения. Второй убеждается, что приложение выполняется с настройками для тестирования. Чтобы превратить папку `tests` в полноценный пакет, в нее необходимо добавить файл `tests/__init__.py`, но он может оставаться пустым, потому что пакет `unittest` может автоматически выполнить сканирование всех модулей в поисках тестов.

 Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 7a` и получить эту версию приложения. Убедитесь также, что установили все зависимости, выполнив команду `pip install -r requirements.txt`.

Чтобы получить возможность запускать тестирование, нужно добавить команду в сценарий `manage.py`, как показано в примере 7.10, добавляющем команду `test`.

Пример 7.10 ❖ `manage.py`: команда запуска модульного тестирования

@manager.command

```
def test():
    """Запускает модульные тесты."""
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
```

Декоратор `manager.command` упрощает создание новых команд. Имя декорируемой функции будет использоваться как имя команды, а строка документирования¹ – отображаться в справочном сообщении. Функция `test()` вызывает функцию `TextTestRunner` из пакета `unittest`.

Теперь запустить модульное тестирование можно следующей командой:

¹ Строка в тройных кавычках, в самом начале функции. – Прим. перев.

```
(venv) $ python manage.py test
test_app_exists (test_basics.BasicsTestCase) ... ok
test_app_is_testing (test_basics.BasicsTestCase) ... ok
```

```
-----
```

```
Ran 2 tests in 0.001s
```

```
OK
```

Настройка базы данных

Реструктурированная версия приложения использует иную базу данных, чем приложение в виде единственного сценария.

Адрес URL базы данных извлекается из переменной окружения, а если она не установлена, по умолчанию используется база данных SQLite. В каждой из трех конфигураций используются отличные от других имена переменных окружения и имена файлов базы данных SQLite. Например, в конфигурации для разработки URL извлекается из переменной окружения `DEV_DATABASE_URL`, а если она не определена, используется база данных SQLite с именем *data-dev.sqlite*.

Независимо от того, откуда был получен URL базы данных, для новой базы данных необходимо создать таблицы. Если вы пользуетесь расширением Flask-Migrate для отслеживания изменений в структуре базы данных, таблицы можно создать или обновить единственной командой:

```
(venv) $ python manage.py db upgrade
```

Итак, мы с вами достигли конца первой части книги. Теперь вы знаете все, что необходимо для создания веб-приложений на основе фреймворка Flask, но, возможно, вы пока слабо представляете, как сложить из этих элементов настоящее приложение. Цель следующей части как раз и состоит в том, чтобы помочь вам в этом — она проведет вас через процесс разработки законченного приложения.

Часть II



Пример: приложение социального блогинга

Глава 8

Аутентификация пользователей

Большинству приложений необходимо постоянно контролировать своих пользователей. Когда пользователь подключается к приложению, он должен пройти *аутентификацию* – процедуру установки его идентичности. После этого приложение сможет предложить ему персонализированный интерфейс и определенный круг возможностей.

Чаще всего для аутентификации от пользователя требуют ввести некоторый идентификатор (адрес электронной почты или имя пользователя) и секретный пароль. В этой главе мы создадим как раз такую законченную систему аутентификации для приложения Flasky.

Расширения аутентификации для Flask

Для Python существует множество неплохих пакетов аутентификации, но ни один из них не соответствует полностью нашим потребностям. Решение аутентификации пользователя, представленное в этой главе, основано на нескольких пакетах и обеспечивает их совместную работу. Список этих пакетов приводится ниже:

- Flask-Login: управляет сеансами пользователей, прошедших аутентификацию;
- Werkzeug: осуществляет хэширование и проверку паролей;
- itsdangerous: создает зашифрованные маркеры и осуществляет их проверку.

В дополнение к пакетам, связанным с аутентификацией, используются также расширения общего назначения:

- Flask-Mail: для отправки электронных писем, связанных с аутентификацией;
- Flask-Bootstrap: для поддержки HTML-шаблонов;
- Flask-WTF: для поддержки веб-форм.

Защита паролей

Защищенности пользовательской информации, хранящейся в базах данных, часто уделяется слишком мало внимания на этапе проектирования веб-приложений. Оставляя лазейку, которой сможет воспользоваться злоумышленник, чтобы взломать ваш сервер и получить доступ к базе данных, вы рискуете безопасностью ваших пользователей, и этот риск намного больше, чем вы думаете. Ни для кого не секрет, что многие пользователи используют один и тот же пароль для разных сайтов. Даже если в вашей базе данных не хранится никакой конфиденциальной информации, доступ к паролям, хранящимся в вашей базе данных, может дать злоумышленнику ключ к учетным записям пользователей на других сайтах.

Чтобы повысить безопасность хранения паролей пользователей в базе данных, следует хранить не сами пароли, а их хэши. Функция хэширования должна принимать пароль и применять к нему одно или более криптографических преобразований. Результатом такой функции должна быть новая последовательность символов, ничем не напоминающая оригинальный пароль. Хэши паролей можно использовать для проверки вместо настоящих паролей, потому что функции хэширования повторимы: для одной и той же исходной строки они всегда возвращают один и тот же результат.



Хэширование паролей – весьма сложная задача, которую многие понимают недостаточно четко. Поэтому рекомендую не пытаться реализовать собственное решение, а использовать библиотеки, надежность которых была проверена сообществом. Если вам интересно узнать, что подразумевается под хэшированием паролей, прочитайте статью «Salted Password Hashing – Doing it Right»¹.

Хэширование паролей с помощью Werkzeug

Модуль `security` из пакета `Werkzeug` реализует надежное хэширование паролей. Эта функциональность экспортируется в виде двух функций, которые можно использовать для регистрации и аутентификации соответственно:

- `generate_password_hash(password, method=pbkdf2:sha1, salt_length=8)`: принимает пароль в простом текстовом виде и возвращает его хэш в виде строки, которую можно сохранить в базе данных. Значения по умолчанию для аргументов `method`

¹ <http://bit.ly/saltedpass>.

и `salt_length` обеспечивают достаточно высокую криптостойкость для большинства применений;

- `check_password_hash(hash, password)`: принимает хэш, извлеченный из базы данных, и пароль в текстовом виде, введенный пользователем. Возвращает значение `True`, если пароль соответствует хэшу.

В примере 8.1 показаны изменения в модели `User`, созданной в главе 5, учитывающие хэширование паролей.

Пример 8.1 ❖ `app/models.py`: хэширование паролей в модели `User`

```
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model):
    # ...
    password_hash = db.Column(db.String(128))

    @property
    def password(self):
        raise AttributeError('password is not a readable attribute')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

Функция хэширования пароля реализована в виде свойства с именем `password`, доступного только для записи. При попытке присвоить значение этому свойству его метод записи (`setter`) вызовет функцию `generate_password_hash()` из пакета `Werkzeug` и запишет результат в поле `password_hash`. При попытке прочитать свойство `password` будет возвращена ошибка, что вполне оправдано, так как на основе хэша нельзя восстановить оригинальный пароль.

Метод `verify_password` принимает пароль и передает его функции `check_password_hash()` из пакета `Werkzeug` для сравнения с хэшированной версией, хранящейся в модели `User`. Если этот метод вернет `True`, можно считать, что введен верный пароль.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта [GitHub](https://github.com), вы можете выполнить команду `git checkout 8a` и получить эту версию приложения.

Теперь у нас имеется вся функциональность, необходимая для хэширования паролей, и ее можно проверить в интерактивной оболочке:


```
(venv) $ python manage.py shell
>>> u = User()
>>> u.password = 'cat'
>>> u.password_hash
'pbkdf2:sha1:1000$duxMk00F$4735b293e397d6eeaf650aaf490fd9091f928bed'
>>> u.verify_password('cat')
True
>>> u.verify_password('dog')
False
>>> u2 = User()
>>> u2.password = 'cat'
>>> u2.password_hash
'pbkdf2:sha1:1000$UjvnGeTP$875e28eb0874f44101d6b332442218f66975ee89'
```

Обратите внимание, что для пользователей `u` и `u2` были созданы разные хэши, даже при том, что оба выбрали один и тот же пароль. Чтобы гарантировать работоспособность этих функций в будущем, проверки, выполненные выше, можно оформить в виде модульных тестов, которые легко будет выполнить в любой момент. В примере 8.2 представлен новый модуль из пакета *tests*, содержащий три новых теста, проверяющих последние изменения в модели `User`.

Пример 8.2 ❖ tests/test_user_model.py: тесты для проверки функций хэширования паролей

```
import unittest
from app.models import User

class UserModelTestCase(unittest.TestCase):
    def test_password_setter(self):
        u = User(password = 'cat')
        self.assertTrue(u.password_hash is not None)

    def test_no_password_getter(self):
        u = User(password = 'cat')
        with self.assertRaises(AttributeError):
            u.password

    def test_password_verification(self):
        u = User(password = 'cat')
        self.assertTrue(u.verify_password('cat'))
        self.assertFalse(u.verify_password('dog'))

    def test_password_salts_are_random(self):
        u = User(password='cat')
        u2 = User(password='cat')
        self.assertTrue(u.password_hash != u2.password_hash)
```

Создание макета для поддержки аутентификации

Мы познакомились с макетами в главе 7, где они использовались для определения маршрутов в глобальной области видимости после перемещения создания экземпляра приложения в фабричную функцию. Маршруты, связанные с системой аутентификации, можно добавить в макет `auth`. Использование разных макетов для разных функциональных особенностей приложения – это отличный способ организации кода.

Макет `auth` будет находиться в пакете Python с тем же именем. Конструктор пакета создает объект макета и импортирует маршруты из модуля `views.py`. Реализация конструктора приводится в примере 8.3.

Пример 8.3 ❖ `app/auth/__init__.py`: создание макета

```
from flask import Blueprint

auth = Blueprint('auth', __name__)

from . import views
```

Модуль `app/auth/views.py`, представленный в примере 8.4, импортирует макет и определяет маршруты, связанные с аутентификацией, используя его декоратор `route`. Вновь добавленный маршрут `/login` отображает страницу с помощью шаблона с тем же именем.

Пример 8.4 ❖ `app/auth/views.py`: маршруты макета и функции представления

```
from flask import render_template
from . import auth

@auth.route('/login')
def login():
    return render_template('auth/login.html')
```

Обратите внимание, что файл шаблона, имя которого передается функции `render_template()`, хранится в папке `auth`. Эту папку необходимо создать внутри `app/templates`, так как Flask предполагает, что относительные пути к шаблонам задаются относительно папки `template`. Хранение шаблонов макета в отдельной папке снижает риск конфликтов имен с макетом `main` или любыми другими макетами, которые могут быть добавлены в будущем.



Макеты можно также настроить на хранение шаблонов в их собственных, независимых папках. Когда имеется несколько папок с шаблонами, функция `render_template()` сначала выполнит поиск в папках с шаблонами для приложения, а затем в папках, настроенных в макетах.

Макет `auth` необходимо подключить к приложению в фабричной функции `create_app()`, как показано в примере 8.5.

Пример 8.5 ❖ `app/__init__.py`: подключение макета

```
def create_app(config_name):
    # ...
    from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint, url_prefix='/auth')

    return app
```

Аргумент `url_prefix` в функции регистрации макета является необязательным. Если он указан, все маршруты, объявленные в макете, будут зарегистрированы с заданным префиксом, в данном случае с префиксом `/auth`. Например, маршрут `/login` будет зарегистрирован как `/auth/login`, а полностью квалифицированный адрес URL на веб-сервере разработки будет иметь вид: `http://localhost:5000/auth/login`.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 8b` и получить эту версию приложения.

Аутентификация пользователя с помощью Flask-Login

После прохождения процедуры аутентификации пользователем ее результат сохраняется, чтобы дать пользователю возможность посещать другие страницы приложения. `Flask-Login` – маленькое, но очень полезное расширение, специализирующееся на управлении данным аспектом системы аутентификации, не будучи тесно связанным с каким-либо конкретным механизмом аутентификации.

Для начала необходимо установить расширение в виртуальное окружение:

```
(venv) $ pip install flask-login
```

Подготовка модели `User` для аутентификации

Чтобы дать расширению `Flask-Login` возможность работать с моделью `User`, в ней требуется реализовать несколько методов. Необходимые методы перечислены в табл. 8.1.

Таблица 8.1. Методы, возвращающие информацию о пользователе для Flask-Login

Метод	Описание
<code>is_authenticated()</code>	Должен возвращать <code>True</code> , если прошел процедуру аутентификации, и <code>False</code> – в противном случае
<code>is_active()</code>	Должен возвращать <code>True</code> , если пользователь имеет право на аутентификацию, и <code>False</code> – в противном случае. Возвращаемое значение <code>False</code> можно использовать как признак необходимости отключения учетной записи
<code>is_anonymous()</code>	Для обычных пользователей всегда должен возвращать <code>False</code>
<code>get_id()</code>	Должен возвращать уникальный идентификатор пользователя в виде строки Юникода

Эти четыре метода можно реализовать непосредственно в классе модели, но имеется более простое решение в виде класса `UserMixin` из расширения `Flask-Login`, в котором определены реализации этих методов по умолчанию, пригодные для использования в большинстве случаев. Измененная модель `User` показана в примере 8.6.

Пример 8.6 ❖ `app/models.py`: модель `User`, дополненная поддержкой аутентификации

```
from flask.ext.login import UserMixin

class User(UserMixin, db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key = True)
    email = db.Column(db.String(64), unique=True, index=True)
    username = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Обратите внимание на появление нового поля `email`. В данном приложении пользователь будет аутентифицироваться по адресу электронной почты, так как обычно пользователи реже забывают свои электронные адреса, чем придуманные имена.

Инициализация расширения `Flask-Login` производится в фабричной функции приложения, как показано в примере 8.7.

Пример 8.7 ❖ `app/__init__.py`: инициализация расширения `Flask-Login`

```
from flask.ext.login import LoginManager

login_manager = LoginManager()
login_manager.session_protection = 'strong'
login_manager.login_view = 'auth.login'
```

```
def create_app(config_name):
    # ...
    login_manager.init_app(app)
    # ...
```

Атрибуту `session_protection` объекта `LoginManager` можно присваивать значения `None`, `'basic'` или `'strong'`, соответствующие разным уровням защищенности пользовательских сеансов от постороннего вмешательства. Если установить значение `'strong'`, Flask-Login будет следить за IP-адресом клиента и агентом браузера и завершать сеанс принудительно при обнаружении изменений. Атрибуту `login_view` присваивается имя конечной точки, соответствующей странице аутентификации. Напомню: так как маршрут `login` находится внутри макета, в его начало необходимо добавить имя макета.

Наконец, Flask-Login требует, чтобы приложение определило функцию обратного вызова для загрузки информации о пользователе по заданному идентификатору. Эта функция показана в примере 8.8.

Пример 8.8 ❖ `app/models.py`: функция загрузки информации о пользователе

```
from . import login_manager

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

Функция `load_user` принимает идентификатор пользователя в виде строки Юникода и, если указанный идентификатор существует, возвращает объект, представляющий пользователя, в противном случае возвращается `None`.

Защита маршрутов

Для защиты маршрутов, чтобы обращаться к ним могли только аутентифицированные пользователи, расширение Flask-Login предоставляет декоратор `login_required`. Ниже приводится пример его использования:

```
from flask.ext.login import login_required

@app.route('/secret')
@login_required
def secret():
    return 'Only authenticated users are allowed!'
```

Если к маршруту попытается обратиться неаутентифицированный пользователь, Flask-Login перехватит запрос и отправит этому пользователю страницу аутентификации.

Добавление формы аутентификации

Форма аутентификации имеет текстовое поле для ввода адреса электронной почты, поле для ввода пароля, флажок «запомнить меня» и кнопку отправки формы. Класс формы, реализованный на основе расширения Flask-WTF, представлен в примере 8.9.

Пример 8.9 ❖ app/auth/forms.py: форма аутентификации

```
from flask.ext.wtf import Form
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import Required, Email

class LoginForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64),
                                           Email()])
    password = PasswordField('Password', validators=[Required()])
    remember_me = BooleanField('Keep me logged in')
    submit = SubmitField('Log In')
```

Для проверки содержимого поля ввода адреса электронной почты используются валидаторы `Length()` и `Email()` из `WTForms`. Класс `PasswordField` представляет элемент `<input>` с атрибутом `type="password"`. Класс `BooleanField` представляет флажок.

Шаблон, используемый для отображения страницы аутентификации, хранится в файле *auth/login.html*. Этот шаблон необходим, только чтобы обеспечить отображение формы с применением макроса `wtf.quick_form()` из `Flask-Bootstrap`. На рис. 8.1 показано, как выглядит страница аутентификации в окне браузера.

Для формирования строки навигации в шаблоне *base.html* используется условная директива из `Jinja2`, с помощью которой отображается ссылка «Sign In» (Войти) или «Sign Out» (Выйти), в зависимости от состояния аутентификации текущего пользователя. Порядок использования условной директивы можно увидеть в примере 8.10.

Пример 8.10 ❖ app/templates/base.html: ссылки «Sign In» и «Sign Out» в строке навигации

```
<ul class="nav navbar-nav navbar-right">
    {% if current_user.is_authenticated() %}
    <li><a href="{{ url_for('auth.logout') }}">Sign Out</a></li>
    {% else %}
```

```

<li><a href="{{ url_for('auth.login') }}">Sign In</a></li>
{% endif %}
</ul>

```

Переменная `current_user`, используемая в условной директиве, определяется расширением Flask-Login и автоматически доступна в шаблонах и функциях представления. Эта переменная содержит объект, представляющий аутентифицированного пользователя, или объект-заглушку, если пользователь еще не аутентифицирован. Объекты, представляющие анонимных пользователей, отвечают на вызов метода `is_authenticated()` значением `False`, что позволяет легко определить, был ли пользователь аутентифицирован или нет.

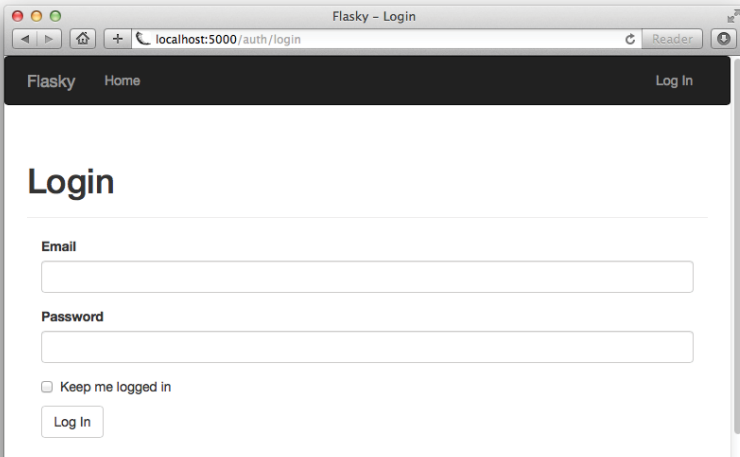


Рис. 8.1 ❖ Форма аутентификации

Аутентификация

В примере 8.11 приводится реализация функции представления `login()`.

Пример 8.11 ❖ `app/auth/views.py`: маршрут входа

```

from flask import render_template, redirect, request, url_for, flash
from flask.ext.login import login_user
from . import auth
from ..models import User

```

```
from .forms import LoginForm


@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is not None and user.verify_password(form.password.data):
            login_user(user, form.remember_me.data)
            return redirect(request.args.get('next') or url_for('main.index'))
        flash('Invalid username or password.')
    return render_template('auth/login.html', form=form)
```

Эта функция создает объект `LoginForm` и использует его как простую форму, подобно тому, как демонстрировалось в главе 4. Когда приложение получает запрос типа `GET`, функция представления просто отображает шаблон, возвращая клиенту форму. Когда приложение получает форму, отправленную в запросе `POST`, вызывается функция `validate_on_submit()` из расширения `Flask-WTF`, проверяющая переменные формы, и затем выполняется попытка аутентифицировать пользователя.

Для проведения аутентификации функция сначала извлекает информацию о пользователе из базы данных по полученному из формы адресу электронной почты. Если пользователь с указанным электронным адресом существует, вызывается его метод `verify_password()`, которому передается пароль, так же полученный из формы. Если пароль верный, вызывается функция `login_user()` из расширения `Flask-Login` для запоминания аутентифицированного пользователя. Она принимает объект, представляющий пользователя, и необязательный логический флаг «запомнить меня», тоже полученный с формой. Значение `False` в этом аргументе приводит к закрытию сеанса пользователя сразу после закрытия окна пользователя, вследствие чего при следующем посещении приложения ему вновь придется пройти процедуру аутентификации. Значение `True` вызовет создание cookie с длительным сроком хранения и отправку его браузеру пользователя, с помощью которого можно будет восстановить прерванный сеанс.

В соответствии с шаблоном `Post/Redirect/Get`, обсуждавшимся в главе 4, обработка запроса `POST`, с которым на сервер отправляются данные, необходимые для аутентификации, завершается переадресацией, но в данном случае есть два возможных адреса URL. Если форма аутентификации была отправлена в ответ на попытку доступа к защищенному URL, тогда выполняется переадресация по этому URL, сохраненному расширением `Flask-Login` в аргументе `next` стро-

ки запроса и доступному в словаре `request.args`. Если аргумент `next` отсутствует, выполняется переадресация на главную страницу. Если комбинация адреса электронной почты и пароля оказалась недопустимой, определяется всплывающее сообщение, и форма отображается еще раз, позволяя повторить попытку входа.

 На действующем сервере доступ к маршруту `login` должен быть организован через защищенную версию протокола HTTP, чтобы данные формы передавались на сервер в зашифрованном виде. В противном случае данные, необходимые для аутентификации, могут быть перехвачены во время передачи, и все усилия по обеспечению безопасности на стороне сервера окажутся потраченными впустую.

Далее необходимо изменить шаблон формы. Эти изменения приводятся в примере 8.12.

Пример 8.12 ❖ `app/templates/auth/login.html`: шаблон отображения формы аутентификации

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky - Login{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Login</h1>
</div>
<div class="col-md-4">
  {{ wtf.quick_form(form) }}
</div>
{% endblock %}
```

Выход пользователя

В примере 8.13 показана реализация маршрута `logout`.

Пример 8.13 ❖ `app/auth/views.py`: маршрут выхода

```
from flask.ext.login import logout_user, login_required

@auth.route('/logout')
@login_required
def logout():
    logout_user()
    flash('You have been logged out.')
    return redirect(url_for('main.index'))
```

Когда пользователь совершает выход, вызывается функция `logout_user()` из расширения `Flask-Login`, удаляющая сеанс пользователя.

Выход завершается выводом всплывающего сообщения, подтверждающего действие, и переадресацией на главную страницу.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 8c` и получить эту версию приложения. Это обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория. Чтобы убедиться, что все зависимости установлены, выполните также команду `run pip install -r requirements.txt`.

Тестирование процедуры аутентификации

Чтобы проверить работу системы аутентификации, можно добавить на главную страницу приветствие с именем пользователя. Раздел шаблона, генерирующий текст приветствия, приводится в примере 8.14.

Пример 8.14 ❖ `app/templates/index.html`: приветствие пользователя, прошедшего аутентификацию

```
Hello,
{% if current_user.is_authenticated() %}
    {{ current_user.username }}
{% else %}
    Stranger
{% endif %}!
```

В этом шаблоне для проверки состояния пользователя используется метод `current_user.is_authenticated()`.

Так как регистрация пользователей пока не поддерживается, вы можете зарегистрировать нового пользователя в интерактивной оболочке:

```
(venv) $ python manage.py shell
>>> u = User(email='john@example.com', username='john', password='cat')
>>> db.session.add(u)
>>> db.session.commit()
```

Теперь можно попробовать выполнить вход. На рис. 8.2 показана главная страница приложения после аутентификации пользователя.

Регистрация нового пользователя

Когда новый пользователь пожелает воспользоваться услугами приложения, он должен зарегистрироваться. Ссылка на странице аутентификации поможет ему перейти на страницу регистрации, где он сможет ввести адрес электронной почты, имя пользователя и пароль.

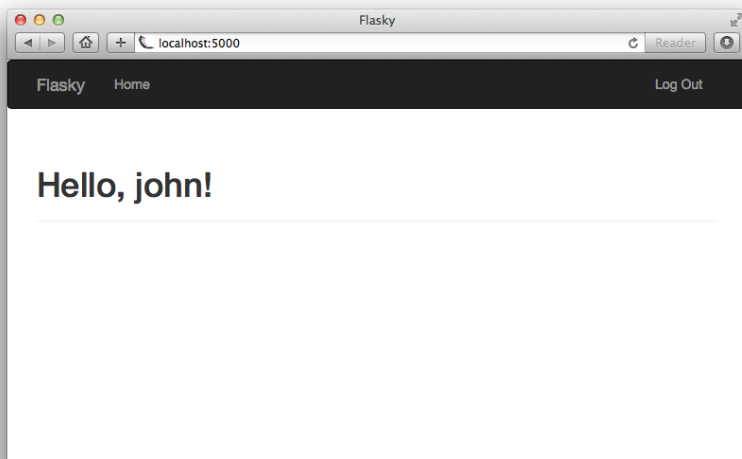


Рис. 8.2 ❖ Главная страница после успешной аутентификации

Добавление формы регистрации пользователя

Форма, которая будет использоваться в странице регистрации, предлагает пользователю ввести адрес электронной почты, имя пользователя и пароль. Эта форма представлена в примере 8.15.

Пример 8.15 ❖ `app/auth/forms.py`: форма регистрации пользователя

```
from flask.ext.wtf import Form
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import Required, Length, Email, Regexp, EqualTo
from wtforms import ValidationError
from ..models import User

class RegistrationForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64),
                                             Email()])
    username = StringField('Username', validators=[
        Required(), Length(1, 64), Regexp('^[A-Za-z][A-Za-z0-9_]*$', 0,
            'Usernames must have only letters, '
            'numbers, dots or underscores')])
    password = PasswordField('Password', validators=[
        Required(), EqualTo('password2', message='Passwords must match.')])
    password2 = PasswordField('Confirm password', validators=[Required()])
    submit = SubmitField('Register')
```

```
def validate_email(self, field):
    if User.query.filter_by(email=field.data).first():
        raise ValidationError('Email already registered.')

def validate_username(self, field):
    if User.query.filter_by(username=field.data).first():
        raise ValidationError('Username already in use.')
```

Чтобы убедиться, что поле ввода имени пользователя содержит только буквы, цифры, точки и символы подчеркивания, эта форма использует валидатор `Regexr` из расширения `WTForms`. Помимо регулярного выражения, этот валидатор принимает флаги регулярного выражения и текст сообщения об ошибке, который отображается при обнаружении недопустимых символов.

Для большей надежности пароль предлагается ввести дважды, при этом требуется, чтобы содержимое двух полей совпадало полностью. Совпадение проверяется с помощью валидатора `EqualTo`. Этот валидатор подключается к одному из полей ввода, и в качестве аргумента ему передается имя другого поля.

Эта форма также включает два нестандартных валидатора, реализованных как методы. Если форма определяет метод с именем, начинающимся с префикса `validate_`, за которым следует имя поля, этот метод будет вызываться вместе с любыми другими валидаторами. В данном случае нестандартные валидаторы для полей `email` и `username` проверяют, не дублируются ли значения этих полей. В случае ошибки нестандартные валидаторы должны возбуждать исключение `ValidationError` с текстом сообщения об ошибке в качестве аргумента.

Данный шаблон хранится в файле `/templates/auth/register.html`. Как и шаблон формы аутентификации, он отображается вызовом `wtf.quick_form()`. Внешний вид страницы регистрации представлен на рис. 8.3.

Страница регистрации должна быть связана ссылкой с формой аутентификации, чтобы пользователь, не имеющий учетной записи, мог легко найти эту страницу. Соответствующие изменения в шаблоне страницы аутентификации приводятся в примере 8.16.

Пример 8.16 ❖ `app/templates/auth/login.html`: ссылка на страницу регистрации

```
<p>
    New user?
    <a href="{{ url_for('auth.register') }}">
        Click here to register
    </a>
</p>
```

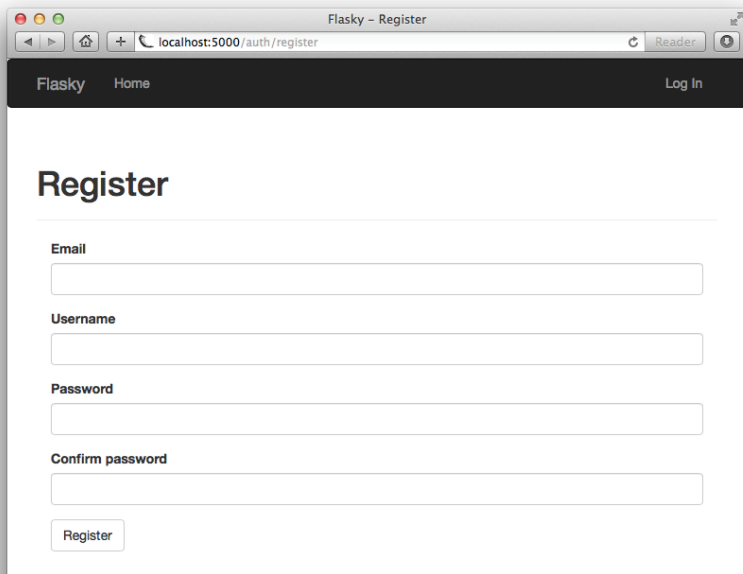


Рис. 8.3 ❖ Форма регистрации нового пользователя

Регистрация

Реализация процедуры регистрации не представляет особых сложностей. После отправки и проверки формы валидаторами информация о новом пользователе добавляется в базу данных. Функция представления, которая решает эту задачу, показана в примере 8.17.

Пример 8.17 ❖ app/auth/views.py: маршрут регистрации пользователя

```
@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(email=form.email.data,
                    username=form.username.data,
                    password=form.password.data)
        db.session.add(user)
        flash('You can now login.')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)
```



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 8d` и получить эту версию приложения.

Подтверждение создания учетной записи

Для некоторых типов приложений очень важно убедиться, что информация, представленная пользователем, не содержит ошибок. Например, типичным требованием является достигаемость пользователя через электронную почту.

С целью подтвердить верность адреса электронной почты приложение посылает пользователю электронное письмо сразу после регистрации. Вновь созданная учетная запись остается отмеченной как неподтвержденная, пока пользователем не будут выполнены инструкции, изложенные в электронном письме, подтверждающие достигаемость пользователя. Процедура подтверждения учетной записи обычно заключается в том, чтобы щелкнуть на ссылке, содержащей специально сконструированный URL, включающий маркер подтверждения.

Создание маркера подтверждения с помощью *itsdangerous*

Ссылка для подтверждения создания учетной записи могла бы быть адресом URL в формате `http://www.example.com/auth/confirm/<id>`, где *id* – числовой идентификатор, присвоенный пользователю в базе данных. Когда пользователь щелкнет по этой ссылке, функция представления, обрабатывающая данный маршрут, примет аргумент *id* и изменит статус подтверждения учетной записи.

Но, как нетрудно догадаться, это небезопасная реализация, так как любой пользователь, знающий формат ссылки подтверждения, сможет подтвердить любую учетную запись, просто послав случайное число в URL. Идея состоит в том, чтобы заменить *id* в URL маркером, содержащим ту же информацию в зашифрованном виде.

Как вы наверняка помните из обсуждения в главе 4, для защиты пользовательских сеансов от внешнего вмешательства Flask использует небольшие блоки данных (cookies) с криптографической подписью. Такие блоки подписываются с помощью пакета *itsdangerous*. Ту же идею можно использовать и для создания маркеров подтверждения.

Ниже приводится порядок выполнения операций в интерактивной оболочке, демонстрирующий, как с помощью *itsdangerous* генериро-

вать зашифрованные маркеры, содержащие числовые идентификаторы пользователей:

```
(venv) $ python manage.py shell
>>> from manage import app
>>> from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
>>> s = Serializer(app.config['SECRET_KEY'], expires_in = 3600)
>>> token = s.dumps({'confirm': 23 })
>>> token
'eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4MTcxODU1OCwiaWF0IjoxMzgxNzE0OTU4fQ.eyJ ...'
>>> data = s.loads(token)
>>> data
{'confirm': 23}
```

Пакет *itsdangerous* содержит несколько генераторов маркеров разных типов. Среди них имеется класс *TimedJSONWebSignatureSerializer*, генерирующий веб-сигнатуры JSON (JSON Web Signatures, JWS) с определенным сроком действия. Конструктор этого класса принимает ключ шифрования, который в приложениях на основе Flask может быть определен как параметр *SECRET_KEY*.

Метод *dumps()* генерирует цифровую подпись для данных, переданных в аргументе, и затем сериализует данные с подписью в строковый маркер. Аргумент *expires_in* определяет срок действия маркера в секундах.

Расшифровка маркера выполняется методом *loads()* объекта *Serializer*, который принимает маркер в виде единственного аргумента. Функция проверяет сигнатуру и срок хранения и, если все в порядке, возвращает исходные данные. Когда метод *loads()* получает недопустимый маркер или определяет, что срок хранения истек, он возбуждает исключение.

Создание маркера и его проверку можно добавить в модель *User*. Соответствующие изменения представлены в примере 8.18.

Пример 8.18 ❖ app/models.py: подтверждение учетной записи пользователя

```
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
from flask import current_app
from . import db

class User(UserMixin, db.Model):
    # ...
    confirmed = db.Column(db.Boolean, default=False)

    def generate_confirmation_token(self, expiration=3600):
        s = Serializer(current_app.config['SECRET_KEY'], expiration)
```

```


return s.dumps({'confirm': self.id})

def confirm(self, token):
    s = Serializer(current_app.config['SECRET_KEY'])
    try:
        data = s.loads(token)
    except:
        return False
    if data.get('confirm') != self.id:
        return False
    self.confirmed = True
    db.session.add(self)
    return True

```

Метод `generate_confirmation_token()` генерирует маркер и по умолчанию устанавливает срок хранения один час. Метод `confirm()` проверяет маркер и, если ошибок не обнаружено, записывает в новый атрибут `confirmed` значение `True`.

Помимо проверки маркера, функция `confirm()` проверяет также соответствие `id` из маркера с числовым идентификатором аутентифицировавшегося пользователя, хранящимся в переменной `current_user`. Это гарантирует, что даже если злоумышленник узнает, как генерируются маркеры, он не сможет подтвердить чужую учетную запись.

 Так как в модель был добавлен новый столбец, хранящий признак подтверждения каждой учетной записи, необходимо создать новый сценарий миграции базы данных и применить его.

Два новых метода, добавленных в модель `User`, легко проверить с помощью модульных тестов. Соответствующие тесты вы найдете в репозитории приложения на сайте GitHub.

Отправка электронных писем с инструкциями для подтверждения

После добавления нового пользователя в базу данных текущий маршрут `/register` выполняет переадресацию на маршрут `/index`. Однако теперь, перед переадресацией, этот маршрут должен отправить электронное письмо с инструкциями для подтверждения. Соответствующие изменения показаны в примере 8.19.

Пример 8.19 ❖ `app/auth/views.py`: маршрут регистрации с отправкой электронного письма

```

from ..email import send_email

@auth.route('/register', methods = ['GET', 'POST'])

```



```
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        # ...
        db.session.add(user)
        db.session.commit()
        token = user.generate_confirmation_token()
        send_email(user.email, 'Confirm Your Account',
                   'auth/email/confirm', user=user, token=token)
        flash('A confirmation email has been sent to you by email.')
        return redirect(url_for('main.index'))
    return render_template('auth/register.html', form=form)
```

Обратите внимание, что сюда был добавлен вызов `db.session.commit()`, несмотря на то что приложение настроено на автоматическое подтверждение изменений в базе данных по окончании обработки запроса. Это объясняется тем, что числовой идентификатор присваивается новым пользователям только после подтверждения изменений в базе данных, а так как этот идентификатор необходим для формирования маркера, операцию подтверждения записи в базу данных нельзя отложить на потом.

Шаблоны электронных писем, необходимые для макета с реализацией аутентификации, будут добавляться в папку *templates/auth/email*, чтобы отделить их от шаблонов HTML-страниц. Как уже говорилось в главе 6, для каждого электронного письма необходимо создать две версии шаблона – текстовую и с разметкой HTML. В примере 8.20 приводится версия шаблона тела письма в простом текстовом формате, а эквивалентную версию в формате HTML вы сможете найти в репозитории на GitHub.

Пример 8.20 ❖ `app/auth/templates/auth/email/confirm.txt`:
тело письма в простом текстовом формате

```
Dear {{ user.username }},
```

```
Welcome to Flasky!
```

```
To confirm your account please click on the following link:
```

```
{{ url_for('auth.confirm', token=token, _external=True) }}
```

```
Sincerely,
```

```
The Flasky Team
```

```
Note: replies to this email address are not monitored.
```

По умолчанию функция `url_for()` генерирует относительные адреса URL, то есть вызов `url_for('auth.confirm', token='abc')`, например, вернет строку `"/auth/confirm/abc"`. Разумеется, этот URL нельзя послать в письме. Относительные адреса URL могут использоваться только в контексте веб-страницы, потому что браузер автоматически преобразует их в абсолютные, добавляя имя хоста и номер порта, соответствующие текущей странице, но когда в электронном письме такой контекст отсутствует. Чтобы сформировать полностью квалифицированный URL, включающий схему (`http://` или `https://`), имя хоста и порт, в вызов функции `url_for()` следует передать аргумент `_external=True`.

В примере 8.21 приводится функция представления, осуществляющая подтверждение учетных записей.

Пример 8.21 ❖ `app/auth/views.py`: подтверждение учетных записей пользователей

```
from flask.ext.login import current_user

@auth.route('/confirm/<token>')
@login_required
def confirm(token):
    if current_user.confirmed:
        return redirect(url_for('main.index'))
    if current_user.confirm(token):
        flash('You have confirmed your account. Thanks!')
    else:
        flash('The confirmation link is invalid or has expired.')
    return redirect(url_for('main.index'))
```

Этот маршрут защищен декоратором `login_required` из расширения Flask-Login, поэтому когда пользователь щелкнет по ссылке из электронного письма, ему будет предложено аутентифицироваться, прежде чем он попадет в эту функцию представления.

Функция сначала проверяет, подтверждал ли прежде этот пользователь свою учетную запись, и если подтверждал – переадресует его на главную страницу, так как очевидно, что никаких дополнительных операций выполнять в этом случае не требуется. Это убережет приложение от лишней работы, если пользователь по ошибке щелкнет по ссылке несколько раз.

Так как фактическое подтверждение реализовано в модели `User`, функция представления должна просто вызвать метод `confirm()` и затем вывести сообщение, соответствующее результату. В случае успешного подтверждения атрибут `confirmed` в модели `User` изменится, и это изменение будет подтверждено по окончании обработки запроса.

В каждом приложении можно определить свой круг операций, доступных пользователю до момента, пока его учетная запись не будет подтверждена. Например, такому пользователю можно разрешить выполнить операцию аутентификации, а затем переадресовывать его на страницу, предлагающую подтвердить учетную запись.

Добиться этого можно с помощью обработчика события `before_request`, о котором упоминалось в главе 2. При использовании обработчика `before_request` в макете он будет вызываться только для запросов, принадлежащих макету. Чтобы в макете определить обработчик для всего приложения, вместо `before_request` необходимо использовать декоратор `before_app_request`. В примере 8.22 показано, как можно реализовать этот обработчик.

Пример 8.22 ❖ `app/auth/views.py`: фильтрация неподтвержденных учетных записей в обработчике `before_app_request`


```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated() \
        and not current_user.confirmed \
        and request.endpoint[:5] != 'auth.':
        return redirect(url_for('auth.unconfirmed'))

@auth.route('/unconfirmed')
def unconfirmed():
    if current_user.is_anonymous() or current_user.confirmed:
        return redirect('main.index')
    return render_template('auth/unconfirmed.html')
```

Обработчик `before_app_request` перехватывает запросы, только если выполняются следующие три условия:

1. Пользователь был аутентифицирован (вызов `current_user.is_authenticated()` должен вернуть `True`).
2. Учетная запись не подтверждена.
3. Конечная точка запроса (доступна как `request.endpoint`) находится за пределами макета аутентификации. Доступ к маршруту аутентификации должен оставаться открытым, так как эти маршруты позволяют подтверждать и выполнять другие операции с учетными записями.

Если все три условия выполняются, производится переадресация на новый маршрут `/auth/unconfirmed`, для которого отображается страница с предложением подтвердить учетную запись.

 Когда обработчик `before_request` или `before_app_request` возвращает ответ или осуществляет переадресацию, Flask отправляет ответ клиенту, не вызы-

вая функцию представления, соответствующую оригинальному запросу. Это позволяет обработчикам эффективно перехватывать запросы при необходимости.

Страница, появляющаяся перед пользователем, который не подтвердил свою учетную запись (см. рис. 8.4), содержит инструкции по подтверждению и ссылку, щелкнув на которой, пользователь может инициировать повторную отправку письма со ссылкой для подтверждения, на случай если первоначальное письмо было утеряно. В примере 8.23 приводится реализация маршрута, осуществляющая повторную отправку письма.

Пример 8.23 ❖ `app/auth/views.py`: повторная отправка письма со ссылкой для подтверждения

```
@auth.route('/confirm')
@login_required
def resend_confirmation():
    token = current_user.generate_confirmation_token()
    send_email('auth/email/confirm',
               'Confirm Your Account', user, token=token)
    flash('A new confirmation email has been sent to you by email.')
    return redirect(url_for('main.index'))
```

Этот маршрут выполняет те же действия, что и маршрут регистрации, используя `current_user` для получения информации о текущем

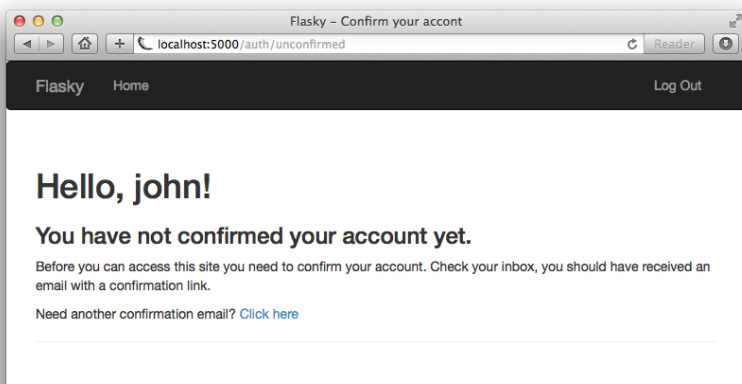


Рис. 8.4 ❖ Страница с сообщением о неподтвержденной учетной записи

пользователе. Он защищен декоратором `login_required`, гарантирующим его доступность только для аутентифицированных пользователей.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 8e` и получить данную версию приложения. Это обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория.

Управление учетными записями

Пользователям, имеющим учетные записи в приложении, иногда может потребоваться внести какие-либо изменения. Ниже перечислены операции, которые можно было бы добавить в макет аутентификации, используя приемы, о которых рассказывалось в данной главе:

- Изменение пароля – осторожные пользователи могут желать периодически изменять свой пароль. Реализовать это достаточно просто, потому что для этого нужно лишь отправить аутентифицированному пользователю форму, запрашивающую старый и новый пароли. (Эта функция реализована в версии 8f приложения, в репозитории GitHub.)
- Сброс пароля – чтобы не заблокировать доступ к приложению для пользователя, который забыл свой пароль, можно предусмотреть операцию сброса пароля. Для ее безопасной реализации необходимо использовать маркеры, похожие на те, что используются для подтверждения учетной записи. Пользователю, запросившему сброс пароля, можно отправлять электронное письмо со ссылкой, содержащей маркер сброса пароля. После того как пользователь щелкнет по этой ссылке и маркер будет проверен приложением, ему можно отправить форму для ввода нового пароля. (Эта функция реализована в версии 8g приложения, в репозитории GitHub.)
- Изменение адреса электронной почты – пользователям можно дать возможность изменять зарегистрированный адрес электронной почты, но, прежде чем принимать его, следует убедиться в его верности отправкой электронного письма со ссылкой для подтверждения. Чтобы выполнить эту операцию, пользователь должен ввести в форме новый адрес электронной почты. Затем на этот адрес должно быть отправлено письмо со ссылкой для подтверждения. Когда приложение получит мар-

кер обратно, оно сможет изменить адрес. Пока сервер ожидает подтверждения, он может хранить новый адрес в новом поле (в базе данных), зарезервированном для адреса, ожидающего подтверждения, или включить его в маркер вместе с числовым идентификатором. (Эта функция реализована в версии 8h приложения, в репозитории GitHub.)

В следующей главе мы дополним поддержкой ролей подсистему учетных записей пользователей в приложении Flasky.

Глава 9

Роли пользователей

Не все пользователи веб-приложений равны в своих правах. В большинстве веб-приложений есть малая доля особенно доверенных пользователей, наделенных дополнительными полномочиями, которые помогают обеспечивать бесперебойную его работу. Отличным примером могут служить администраторы, но во многих приложениях существуют также пользователи, обладающие промежуточными полномочиями, такие как модераторы.

Существует множество способов реализации ролей в приложениях. Выбор того или иного способа в значительной степени зависит от количества ролей и их сложности. Например, в простом приложении может оказаться достаточно всего двух ролей, одна для обычных пользователей и одна для администраторов. В этом случае вполне может хватить логического поля `is_administrator` в модели `User`. В более сложных приложениях могут потребоваться дополнительные роли, с разными уровнями привилегий, от обычного пользователя до администратора. В некоторых приложениях вообще может оказаться бессмысленным говорить о дискретных ролях; в таких приложениях более соответствующей может оказаться модель, основанная на *ком-плексе привилегий*.

Реализация ролей пользователей, представленная в этой главе, является собой гибрид дискретных ролей и набора привилегий. Пользователям присваиваются дискретные роли, но сами роли определяются в терминах привилегий.

Представление ролей в базе данных

В главе 5 была создана простая таблица `roles` с целью демонстрации отношения «один ко многим». В примере 9.1 приводится усовершенствованная модель `Role`.

Пример 9.1 ❖ `app/models.py`: привилегии ролей

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    default = db.Column(db.Boolean, default=False, index=True)
    permissions = db.Column(db.Integer)
    users = db.relationship('User', backref='role', lazy='dynamic')
```

Поле `default` должно иметь значение `True` только для одной роли, для остальных ролей оно должно быть установлено в значение `False`. Роль по умолчанию (со значением `True` в поле `default`) – это роль, которая присваивается всем новым пользователям в процессе регистрации.

Второе дополнение в этой модели – поле `permissions`, хранящее целое число, которое интерпретируется как битовая маска привилегий. Каждому биту в этом числе будет соответствовать определенная привилегия, а каждой роли – свой набор привилегий (битов, установленных в 1).

Перечень привилегий, очевидно, зависит от специфики приложения. Список привилегий для нашего приложения Flasky приводится в табл. 9.1.

Таблица 9.1. Привилегии в приложении

Привилегия	Значения битов	Описание
Следование за пользователями	0b00000001 (0x01)	Разрешается следовать за другими пользователями
Комментирование чужих сообщений	0b00000010 (0x02)	Разрешается комментировать статьи, написанные другими пользователями
Создание статей	0b00000100 (0x04)	Разрешается писать собственные статьи
Модерирование чужих комментариев	0b00001000 (0x08)	Разрешается подавлять оскорбительные комментарии, оставляемые другими пользователями
Административный доступ	0b10000000 (0x80)	Административный доступ к сайту

Обратите внимание, что для привилегий отведено только восемь битов, и к настоящему моменту определено пять привилегий. Остальные три бита зарезервированы для возможного расширения списка привилегий в будущем.

Представление привилегий из табл. 9.1 в программном коде приводится в примере 9.2.

Пример 9.2 ❖ app/models.py: константы привилегий

```
class Permission:
    FOLLOW = 0x01
    COMMENT = 0x02
    WRITE_ARTICLES = 0x04
    MODERATE_COMMENTS = 0x08
    ADMINISTER = 0x80
```

В табл. 9.2 приводится список ролей пользователей, которые будут поддерживаться, а также соответствующие им биты привилегий.

Таблица 9.2. Роли пользователей

Роль	Привилегии	Описание
Anonymous	0b00000000 (0x00)	Неаутентифицированный пользователь. Доступ к приложению только для чтения
User	0b00000111 (0x07)	Базовый комплект привилегий. Разрешается писать статьи и комментарии, следовать за другими пользователями. Этот набор привилегий присваивается новым пользователям по умолчанию
Moderator	0b00001111 (0x0f)	Добавляется право подавлять чужие комментарии, которые могут расцениваться как оскорбительные или не связанные с обсуждаемой темой
Administrator	0b11111111 (0xff)	Полный доступ, включая право на изменение ролей других пользователей

Возможность организации ролей на основе комбинаций привилегий позволяет добавлять новые роли в будущем.

Добавление вручную ролей в базу данных – довольно утомительное занятие и чревато ошибками. Поэтому добавим для этой цели метод класса в класс Role, как показано в примере 9.3.

Пример 9.3 ❖ app/models.py: создание ролей в базе данных

```
class Role(db.Model):
    # ...
    @staticmethod
    def insert_roles():
        roles = {
            'User': (Permission.FOLLOW |
                    Permission.COMMENT |
                    Permission.WRITE_ARTICLES, True),
            'Moderator': (Permission.FOLLOW |
```

```

        Permission.COMMENT |
        Permission.WRITE_ARTICLES |
        Permission.MODERATE_COMMENTS, False),
    'Administrator': (0xff, False)
}
for r in roles:
    role = Role.query.filter_by(name=r).first()
    if role is None:
        role = Role(name=r)
    role.permissions = roles[r][0]
    role.default = roles[r][1]
    db.session.add(role)
db.session.commit()

```

Функция `insert_roles()` не создает новых объектов ролей непосредственно. Вместо этого она пытается найти имеющиеся роли по именам и изменить их. Новый объект роли создается, только когда роль с заданным именем отсутствует в базе данных. Это сделано для того, чтобы в будущем иметь возможность дополнять список ролей. Чтобы добавить новую роль или изменить набор привилегий для имеющейся роли, внесите необходимые изменения в массив `roles` и вызовите функцию. Обратите внимание, что роль «Anonymous» не требует представления в базе данных, так как соответствует пользователям, не имеющим учетных записей в базе данных.

Чтобы сохранить эти роли в базе данных, можно воспользоваться интерактивной оболочкой:

```

(venv) $ python manage.py shell
>>> Role.insert_roles()
>>> Role.query.all()
[<Role u'Administrator'>, <Role u'User'>, <Role u'Moderator'>]

```

Присваивание ролей

Когда пользователь регистрирует учетную запись в приложении, ему должна быть присвоена надлежащая роль. Большинству пользователей на этапе регистрации присваивается роль «User», которая в базе данных отмечена как роль по умолчанию. Единственное исключение сделано для администратора, которому должна быть присвоена роль «Administrator». Этот пользователь идентифицируется адресом электронной почты, хранящимся в переменной `FLASKY_ADMIN`, поэтому, когда этот адрес появится в запросе регистрации, новому пользователю будет присвоена соответствующая роль. В примере 9.4 показано, как это реализовано в конструкторе модели `User`.

Пример 9.4 ❖ app/models.py: определение роли по умолчанию для пользователей

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        super(User, self).__init__(**kwargs)
        if self.role is None:
            if self.email == current_app.config['FLASKY_ADMIN']:
                self.role = Role.query.filter_by(permissions=0xff).first()
            if self.role is None:
                self.role = Role.query.filter_by(default=True).first()
    # ...
```

Конструктор класса User сначала вызывает конструктор базового класса, и если после этого объекту не была присвоена роль, ему присваивается роль администратора или роль по умолчанию, в зависимости от адреса электронной почты.

Проверка роли

Чтобы упростить реализацию ролей и привилегий, в модель User можно добавить вспомогательный метод, проверяющий наличие указанной привилегии, как показано в примере 9.5.

Пример 9.5 ❖ app/models.py: определяет наличие указанной привилегии у пользователя

```
from flask.ext.login import UserMixin, AnonymousUserMixin

class User(UserMixin, db.Model):
    # ...

    def can(self, permissions):
        return self.role is not None and \
            (self.role.permissions & permissions) == permissions

    def is_administrator(self):
        return self.can(Permission.ADMINISTER)

class AnonymousUser(AnonymousUserMixin):
    def can(self, permissions):
        return False

    def is_administrator(self):
        return False

login_manager.anonymous_user = AnonymousUser
```

Метод `can()`, добавленный в модель `User`, выполняет операцию *по-разрядного И* между проверяемыми привилегиями и привилегиями, присвоенными роли. Если все проверяемые биты в роли установлены в значение 1, метод возвращает `True`. Это означает, что пользователь обладает всеми требуемыми привилегиями. Проверка наличия административных привилегий выполняется настолько часто, что она реализована как отдельный метод `is_administrator()`.

Для непротиворечивости создан также класс `AnonymousUser` с методами `can()` и `is_administrator()`. Он наследует класс `AnonymousUserMixin` из `Flask-Login` и зарегистрирован как класс объектов, которые могут присваиваться переменной `current_user`, когда пользователь не аутентифицирован. Это позволит приложению безопасно вызывать методы `current_user.can()` и `current_user.is_administrator()` без необходимости предварительно проверять факт аутентификации пользователя.

К функциям представления, которые должны быть доступны только пользователям с определенными привилегиями, можно применять собственный декоратор. В примере 9.6 приводится реализация двух декораторов, один обеспечивает универсальную проверку привилегий, а другой проверяет наличие привилегий администратора.

Пример 9.6 ❖ `app/decorators.py`: декораторы для проверки привилегий пользователей

```
from functools import wraps
from flask import abort
from flask.ext.login import current_user

def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not current_user.can(permission):
                abort(403)
            return f(*args, **kwargs)
        return decorated_function
    return decorator

def admin_required(f):
    return permission_required(Permission.ADMINISTER)(f)
```

Эти декораторы реализованы с помощью пакета *functools* из стандартной библиотеки Python и возвращают HTTP-код ошибки 403, «Forbidden», если текущий пользователь не обладает необходимыми привилегиями. В главе 3 мы определили собственные страницы для

отображения ошибок 404 и 500, и теперь нам точно так же нужно добавить страницу для отображения ошибки 403.

Ниже приводятся два примера использования этих декораторов:

```
from decorators import admin_required, permission_required

@main.route('/admin')
@login_required
@admin_required
def for_admins_only():
    return "For administrators!"

@main.route('/moderator')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def for_moderators_only():
    return "For comment moderators!"
```

В шаблонах тоже может потребоваться проверять привилегии, поэтому класс `Permission` необходимо сделать доступным и в шаблонах. Чтобы избежать необходимости добавлять дополнительный аргумент в каждый вызов `render_template()`, можно задействовать *процессор контекста*. Процессор контекста делает переменную доступной глобально во всех шаблонах. Необходимые для этого изменения приводятся в примере 9.7.

Пример 9.7 ❖ `app/main/__init__.py`: добавление класса `Permission` в контекст шаблона

```
@main.app_context_processor
def inject_permissions():
    return dict(Permission=Permission)
```

Для нужд тестирования в модульных тестах можно определять новые роли и привилегии. В примере 9.8 показаны два простых теста, которые могут также служить демонстрацией этого приема.

Пример 9.8 ❖ `tests/test_user_model.py`: модульные тесты для ролей и привилегий

```
class UserModelTestCase(unittest.TestCase):
    # ...

    def test_roles_and_permissions(self):
        Role.insert_roles()
        u = User(email='john@example.com', password='cat')
        self.assertTrue(u.can(Permission.WRITE_ARTICLES))
        self.assertFalse(u.can(Permission.MODERATE_COMMENTS))
```

```
def test_anonymous_user(self):  
    u = AnonymousUser()  
    self.assertFalse(u.can(Permission.FOLLOW))
```



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 9a` и получить эту версию приложения. Это обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория.

Прежде чем перейти к следующей главе создайте заново или обновите базу данных, используемую для разработки, чтобы всем учетным записям пользователей, созданным до реализации поддержки ролей и привилегий, были присвоены роли.

Теперь система поддержки пользователей содержит все необходимое. В следующей главе мы будем использовать ее для создания страниц профилей пользователей.

Глава 10

Профили пользователей

В этой главе мы реализуем профили пользователей для приложения Flasky. Все сайты социальных сетей поддерживают страницы профилей для своих пользователей, представляющие сводную информацию о них. Пользователи могут обмениваться ссылками на свои профили, поэтому важно, чтобы адреса URL были как можно более короткими и запоминающимися.

Информация для профиля

Чтобы сделать страницы профилей более интересными, можно организовать хранение дополнительной информации о пользователях. В примере 10.1 приводится расширение модели `User` несколькими новыми полями.

Пример 10.1 ❖ `app/models.py`: дополнительные поля с информацией о пользователе

```
class User(UserMixin, db.Model):
    # ...
    name = db.Column(db.String(64))
    location = db.Column(db.String(64))
    about_me = db.Column(db.Text())
    member_since = db.Column(db.DateTime(), default=datetime.utcnow)
    last_seen = db.Column(db.DateTime(), default=datetime.utcnow)
```

В новых полях хранятся настоящее имя пользователя, местоположение, описание, написанное самим пользователем, дата регистрации и дата последнего посещения. Обратите внимание, что полю `about_me` присвоен тип `db.Text()`. Тип `db.Text` отличается от `db.String` тем, что не имеет ограничения максимальной длины.

Два поля с отметками времени по умолчанию получают значение текущего времени. Обратите внимание на отсутствие круглых скобок

() после `datetime.utcnow`. Это объясняется тем, что в аргументе `default` конструктору `db.Column()` может передаваться функция, поэтому всякий раз, когда потребуется получить значение по умолчанию, будет вызываться указанная функция. Это значение по умолчанию – все, что нужно для управления полем `member_since`.

Поле `last_seen` также инициализируется текущим временем, соответствующим моменту создания, но его нужно обновлять всякий раз, когда пользователь обращается к сайту. В класс `User` можно добавить метод, который будет обновлять это поле. Реализация такого метода приводится в примере 10.2.

Пример 10.2 ❖ `app/models.py`: обновление времени последнего посещения

```
class User(UserMixin, db.Model):
    # ...

    def ping(self):
        self.last_seen = datetime.utcnow()
        db.session.add(self)
```

Метод `ping()` должен вызываться при приеме любых запросов от пользователя. Так как обработчик `before_app_request` из макета `auth` вызывается перед обработкой всех запросов, его с успехом можно использовать для вызова метода `ping()`, как показано в примере 10.3.

Пример 10.3 ❖ `app/auth/views.py`: обновление времени последнего посещения пользователем

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated():
        current_user.ping()
        if not current_user.confirmed \
            and request.endpoint[:5] != 'auth.':
            return redirect(url_for('auth.unconfirmed'))
```

Страница профиля пользователя

Создание страницы профиля для каждого пользователя не несет каких-то новых сложностей. В примере 10.4 приводится определение маршрута.

Пример 10.4 ❖ `app/main/views.py`: маршрут для страницы профиля

```
@main.route('/user/<username>')
def user(username):
```



```

user = User.query.filter_by(username=username).first()
if user is None:
    abort(404)
return render_template('user.html', user=user)

```

Этот маршрут должен быть добавлен в макет `main`. Адрес URL страницы профиля для пользователя `john` будет иметь вид `http://localhost:5000/user/john`. Если имя пользователя, указанное в URL, будет найдено в базе данных, шаблон `user.html` отобразит страницу профиля для этого пользователя. Неверное имя пользователя вызовет возврат ошибки 404. Шаблон `user.html` должен отобразить информацию, хранящуюся в объекте `user`. Первая версия этого шаблона представлена в примере 10.5.

Пример 10.5 ❖ `app/templates/user.html`: шаблон страницы профиля пользователя

```

{% block page_content %}
<div class="page-header">
  <h1>{{ user.username }}</h1>
  {% if user.name or user.location %}
  <p>
    {% if user.name %}{{ user.name }}{% endif %}
    {% if user.location %}
      From <a href="http://maps.google.com/?q={{ user.location }}">
        {{ user.location }}
      </a>
    {% endif %}
  </p>
  {% endif %}
  {% if current_user.is_administrator() %}
  <p><a href="mailto:{{ user.email }}">{{ user.email }}</a></p>
  {% endif %}
  {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
<p>
  Member since {{ moment(user.member_since).format('L') }}.
  Last seen {{ moment(user.last_seen).fromNow() }}.
</p>
</div>
{% endblock %}

```

Этот шаблон имеет несколько интересных особенностей:

- поля `name` и `location` отображаются внутри одного элемента `<p>`, который создается, только если хотя бы одно из полей определено;
- поле `location` местоположения пользователя отображается как ссылка на Google Maps;

- если аутентифицированный пользователь обладает привилегиями администратора, его адрес электронной почты отображается как ссылка *mailto*.

Большинство пользователей предпочло бы иметь как можно более простой доступ к своим страницам профилей. С этой целью можно добавить соответствующую ссылку в строку навигации. Необходимые для этого изменения в шаблоне *base.html* приводятся в примере 10.6.

Пример 10.6 ❖ app/templates/base.html

```
{% if current_user.is_authenticated() %}
<li>
  <a href="{{ url_for('main.user', username=current_user.username) }}">
    Profile
  </a>
</li>
{% endif %}
```

Условная директива необходима, чтобы обеспечить отображение ссылки на профиль только для аутентифицированных пользователей, потому что строка навигации отображается для всех пользователей, в том числе и для неаутентифицированных, не имеющих профиля.

На рис. 10.1 показано, как выглядит страница профиля в браузере, здесь также можно видеть ссылку на профиль.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 10a` и получить эту версию приложения. Данное обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория.

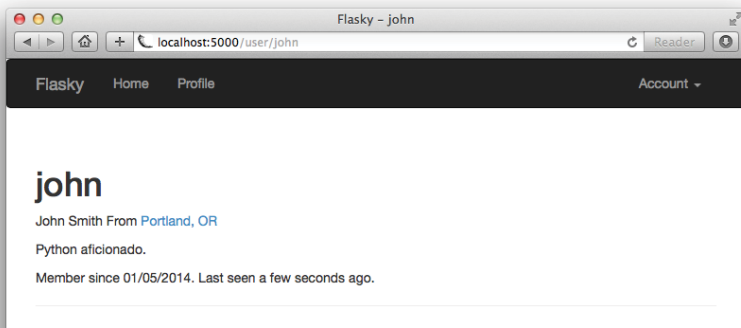


Рис. 10.1 ❖ Страница профиля пользователя

Редактор профиля

Существуют два режима редактирования профиля пользователя. Самый очевидный – режим редактирования самим пользователем, пожелавшим изменить информацию, представленную на его личной странице профиля. Менее очевидный – режим редактирования администратором страницы профиля любого пользователя, не только полей с личной информацией, но и других полей модели `User`, к которым у пользователя нет прямого доступа, таких как роль. Поскольку требования к этим двум режимам существенно отличаются, мы создадим две формы редактирования.

Редактор профиля уровня пользователя

Форма редактирования для использования обычным пользователем показана в примере 10.7.

Пример 10.7 ❖ `app/main/forms.py`: форма редактирования профиля

```
class EditProfileForm(Form):
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')
```

Обратите внимание, что все поля в этой форме объявлены необязательными – валидатор длины `Length` определен так, что допускает нулевую длину. Определение маршрута для этой формы приводится в примере 10.8.

Пример 10.8 ❖ `app/main/views.py`: маршрут редактирования профиля

```
@main.route('/edit-profile', methods=['GET', 'POST'])
@login_required
def edit_profile():
    form = EditProfileForm()
    if form.validate_on_submit():
        current_user.name = form.name.data
        current_user.location = form.location.data
        current_user.about_me = form.about_me.data
        db.session.add(user)
        flash('Your profile has been updated.')
        return redirect(url_for('user', username=current_user.username))
    form.name.data = current_user.name
    form.location.data = current_user.location
    form.about_me.data = current_user.about_me
    return render_template('edit_profile.html', form=form)
```

Эта функция представления устанавливает начальные значения во всех полях перед отображением формы. Для всех полей эта операция выполняется путем присваивания значения переменной `form.<имя-поля>.data`. Когда вызов `form.validate_on_submit()` возвращает `False`, три поля этой формы инициализируются значениями соответствующих полей из `current_user`. Затем, когда форма будет отправлена на сервер, атрибуты `data` полей формы будут содержать измененные значения, поэтому они копируются обратно в поля объекта `User`, и сам объект добавляется в сеанс базы данных. На рис. 10.2 показано, как выглядит страница редактирования профиля.

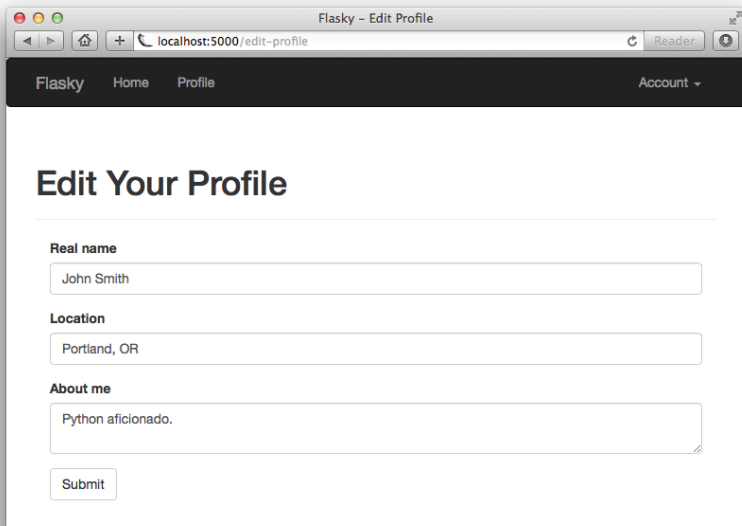


Рис. 10.2 ❖ Редактор профиля

Чтобы упростить пользователю переход к этой странице, в профиль можно добавить прямую ссылку, как показано в примере 10.9.

Пример 10.9 ❖ `app/templates/user.html`: ссылка на страницу редактирования профиля

```
{% if user == current_user %}
<a class="btn btn-default" href="{{ url_for('.edit_profile') }}">
    Edit Profile
</a>
{% endif %}
```

Условная директива, окружающая ссылку, обеспечит ее появление, только когда пользователь будет просматривать свой профиль.

Редактор профиля уровня администратора

Форма редактирования профиля для администраторов сложнее, чем аналогичная форма для обычных пользователей. Помимо трех полей, что использовались выше, эта форма позволяет администраторам изменять адрес электронной почты пользователя, имя пользователя, признак подтверждения регистрации и роль. Ее реализация приводится в примере 10.10.

Пример 10.10 ❖ app/main/forms.py: форма редактирования профиля для администраторов

```
class EditProfileAdminForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64),
                                             Email()])
    username = StringField('Username', validators=[
        Required(), Length(1, 64), Regexp('^[A-Za-z][A-Za-z0-9_]*$', 0,
        'Usernames must have only letters, '
        'numbers, dots or underscores')])
    confirmed = BooleanField('Confirmed')
    role = SelectField('Role', coerce=int)
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')

    def __init__(self, user, *args, **kwargs):
        super(EditProfileAdminForm, self).__init__(*args, **kwargs)
        self.role.choices = [(role.id, role.name)
                              for role in Role.query.order_by(Role.name).all()]
        self.user = user

    def validate_email(self, field):
        if field.data != self.user.email and \
            User.query.filter_by(email=field.data).first():
            raise ValidationError('Email already registered.')

    def validate_username(self, field):
        if field.data != self.user.username and \
            User.query.filter_by(username=field.data).first():
            raise ValidationError('Username already in use.')
```

Класс `SelectField` из `WTForm` является оберткой для HTML-элемента `<select>`, который отображается как раскрывающийся список и используется в этой форме для выбора роли пользователя. В экзмп-

ляре `SelectField` следует определить элементы списка посредством атрибута `choices`. Они должны быть заданы списком кортежей, каждый из которых содержит два элемента: строковый идентификатор элемента списка и текст для отображения в элементе управления. Список `choices` определяется в конструкторе формы, исходя из значений, полученных из модели `Role` с помощью запроса, сортирующего роли по их именам. В качестве значения идентификатора в каждом кортеже используется значение поля `id` роли, а так как идентификатор является целым числом, в вызов конструктора `SelectField` добавлен аргумент `coerce=int`, чтобы значение поля хранилось в виде целого числа, а не строки (по умолчанию).

Поля `email` и `username` определяются так же, как в форме аутентификации, но к их проверке следует подойти более внимательно. Для обоих полей сначала необходимо проверить, изменились ли они, и только если поля изменились, нужно проверить, не совпадают ли они с данными другого пользователя. Если эти поля не изменялись, проверка должна считаться пройденной. Чтобы реализовать эту логику, конструктор формы принимает объект `User` и сохраняет его в переменной-члене, которая затем используется в собственных методах-валидаторах.

Определение маршрута для административного редактора профиля приводится в примере 10.11.

Пример 10.11 ❖ `app/main/views.py`: маршрут редактирования профиля для администраторов

```
@main.route('/edit-profile/<int:id>', methods=['GET', 'POST'])
@login_required
@admin_required
def edit_profile_admin(id):
    user = User.query.get_or_404(id)
    form = EditProfileAdminForm(user=user)
    if form.validate_on_submit():
        user.email = form.email.data
        user.username = form.username.data
        user.confirmed = form.confirmed.data
        user.role = Role.query.get(form.role.data)
        user.name = form.name.data
        user.location = form.location.data
        user.about_me = form.about_me.data
        db.session.add(user)
        flash('The profile has been updated.')
        return redirect(url_for('.user', username=user.username))
    form.email.data = user.email
```

```

form.username.data = user.username
form.confirmed.data = user.confirmed
form.role.data = user.role_id
form.name.data = user.name
form.location.data = user.location
form.about_me.data = user.about_me
return render_template('edit_profile.html', form=form, user=user)

```

Этот маршрут имеет практически ту же структуру, что и более простой маршрут для обычных пользователей. В данной функции представления пользователь определяется по его числовому идентификатору `id`, поэтому для удобства можно использовать функцию `get_or_404()` из `Flask-SQLAlchemy`, на случай если задано ошибочное значение `id`.

Отдельного упоминания заслуживает также конструктор `SelectField`, используемый для определения поля с ролью пользователя. Когда устанавливается начальное значение поля, атрибуту `field.role.data` присваивается значение `role_id`, потому что в списке кортежей в атрибуте `choices` для ссылки на каждый элемент списка используется числовой идентификатор. После отправки формы на сервер значение `id` извлекается из атрибута `data` поля формы и используется в запросе для загрузки объекта роли по идентификатору `id`. Аргумент `coerce=int` в вызове `SelectField` гарантирует, что атрибут `data` этого поля будет иметь целочисленный тип.

Для доступа к этому редактору в страницу профиля добавлена еще одна кнопка, как показано в примере 10.12.

Пример 10.12 ❖ `app/templates/user.html`: ссылка на форму редактирования профиля для администраторов

```

{% if current_user.is_administrator() %}
<a class="btn btn-danger"
    href="{{ url_for('.edit_profile_admin', id=user.id) }}">
    Edit Profile [Admin]
</a>
{% endif %}

```

Эта кнопка отображается с применением иного стиля из `Bootstrap`, чтобы придать ей особый внешний вид. Условная директива в данном случае обеспечивает появление кнопки только на страницах профилей пользователей, обладающих административными привилегиями.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта [GitHub](#), вы можете выполнить команду `git checkout 10b` и получить эту версию приложения.

Аватары пользователей

Внешний вид страницы профиля можно улучшить, добавив в нее изображение аватара. В этом разделе вы узнаете, как добавить поддержку аватаров пользователей, зарегистрированных на сайте Gravatar, ведущей службе аватаров. Служба Gravatar связывает изображения аватаров с адресами электронной почты. Пользователь должен создать учетную запись на сайте <http://gravatar.com> и выгрузить туда свои изображения. Чтобы сгенерировать URL аватара для данного адреса электронной почты, вычисляется хэш MD5:

```
(venv) $ python
>>> import hashlib
>>> hashlib.md5('john@example.com'.encode('utf-8')).hexdigest()
'd4c74594d841139328695756648b6bd6'
```

Затем генерируется URL аватара, добавлением хэша MD5 в конец URL <http://www.gravatar.com/avatar/> или <https://secure.gravatar.com/avatar/>. Например, можно ввести <http://www.gravatar.com/avatar/d4c74594d841139328695756648b6bd6> в адресную строку браузера, чтобы получить изображение аватара для адреса электронной почты john@example.com или изображение по умолчанию, если указанный адрес электронной почты не зарегистрирован. Строка запроса в URL может включать дополнительные аргументы, определяющие характеристики изображения. Эти аргументы перечислены в табл. 10.1.

Таблица 10.1. Аргументы строки запроса для сайта Gravatar

Имя аргумента	Описание
s	Размер изображения в пикселях
r	Рейтинг изображения. Возможные значения: "g", "pg", "r" и "x"
d	Изображение по умолчанию для пользователей, не имеющих зарегистрированных аватаров в службе Gravatar. Возможные значения: "404" – возвращает ошибку 404; URL, ссылающийся на изображение по умолчанию; или один из следующих генераторов изображений: "mm", "identicon", "monsterid", "wavatar", "retro" или "blank"
fd	Принудительно использовать аватары по умолчанию

Реализацию для получения адреса URL аватара на сайте Gravatar можно добавить в модель User, как показано в примере 10.13.

Пример 10.13 ❖ `app/models.py`: создание адреса URL аватара на сайте Gravatar

```
import hashlib
from flask import request

class User(UserMixin, db.Model):
    # ...
    def gravatar(self, size=100, default='identicon', rating='g'):
        if request.is_secure:
            url = 'https://secure.gravatar.com/avatar/'
        else:
            url = 'http://www.gravatar.com/avatar/'
        hash = hashlib.md5(self.email.encode('utf-8')).hexdigest()
        return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
            url=url, hash=hash, size=size, default=default, rating=rating)
```

Эта реализация выбирает схему `http://` или `https://` в зависимости от схемы в запросе пользователя. Адрес URL создается путем объединения базового URL, хэша адреса электронной почты пользователя и аргументов, каждому из которых предусмотрено значение по умолчанию. С помощью этой реализации легко можно сгенерировать URL аватара прямо в интерактивной оболочке Python:

```
(venv) $ python manage.py shell
>>> u = User(email='john@example.com')
>>> u.gravatar()
'http://www.gravatar.com/avatar/d4c74594d84113932869575bd6?s=100&d=identicon&r=g'
>>> u.gravatar(size=256)
'http://www.gravatar.com/avatar/d4c74594d84113932869575bd6?s=256&d=identicon&r=g'
```

Метод `gravatar()` можно также вызывать из шаблонов Jinja2. В примере 10.14 демонстрируется, как в страницу профиля добавить аватар размером 256×256 пикселей.

Пример 10.14 ❖ `app/templatess/user.html`: аватар в странице профиля

```
...

...
```

Используя аналогичный подход, можно добавить в базовый шаблон отображение маленькой миниатюры аватара аутентифицированного пользователя в строку навигации. Для улучшения внешнего вида изображений аватаров можно использовать классы CSS из файла *styles.css*, хранящегося в папке приложения со статическими

файлами и подключаемого базовым шаблоном *base.html*. На рис. 10.3 показана страница профиля пользователя с аватаром.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 10c` и получить эту версию приложения.

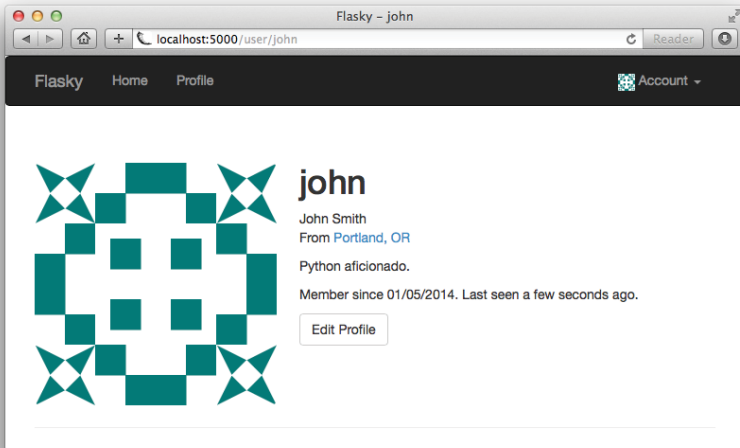


Рис. 10.3 ❖ Страница профиля пользователя с аватаром

Для создания ссылки на аватар требуется генерировать хэш MD5, что является довольно дорогостоящей вычислительной операцией. Если потребуется отобразить на странице большое число аватаров, вычислительная нагрузка может оказаться внушительной. Поскольку хэш MD5 для каждого адреса электронной почты не изменяется, его можно сохранить в модели `User`. Изменения в модели `User`, необходимые для сохранения хэшей MD5 в базе данных, приводятся в примере 10.15.

Пример 10.15 ❖ `app/models.py`: создание адреса URL аватара на сайте Gravatar с кэшированием хэшей MD5

```
class User(UserMixin, db.Model):
    # ...
    avatar_hash = db.Column(db.String(32))

    def __init__(self, **kwargs):
```

```

# ...
if self.email is not None and self.avatar_hash is None:
    self.avatar_hash = hashlib.md5(
        self.email.encode('utf-8')).hexdigest()

def change_email(self, token):
    # ...
    self.email = new_email
    self.avatar_hash = hashlib.md5(
        self.email.encode('utf-8')).hexdigest()
    db.session.add(self)
    return True

def gravatar(self, size=100, default='identicon', rating='g'):
    if request.is_secure:
        url = 'https://secure.gravatar.com/avatar'
    else:
        url = 'http://www.gravatar.com/avatar'
    hash = self.avatar_hash or hashlib.md5(
        self.email.encode('utf-8')).hexdigest()
    return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
        url=url, hash=hash, size=size, default=default, rating=rating)

```

Хэш MD5 адреса электронной почты вычисляется и сохраняется в момент инициализации модели и при изменении адреса электронной почты. Метод `gravatar()` использует хэш из модели, если он имеется; в противном случае хэш, как и прежде, вычисляется из адреса электронной почты.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 10d` и получить данную версию приложения. Это обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория.

В следующей главе будет создан механизм блогинга, составляющий основу функциональности данного приложения.

Глава 11

БЛОГИНГ

Эта глава посвящена реализации основной функции приложения Flasky, позволяющей пользователям читать и писать сообщения. Здесь вы познакомитесь с несколькими новыми приемами повторного использования шаблонов, построения вывода длинных списков элементов и работой с форматированным текстом.

Отправка и отображение сообщений

Для поддержки возможности писать и читать сообщения необходимо определить новую модель для базы данных, представляющую эти сообщения. Объявление модели приводится в примере 11.1.

Пример 11.1 ❖ app/models.py: модель Post

```
class Post(db.Model):
    __tablename__ = 'posts'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))

class User(UserMixin, db.Model):
    # ...
    posts = db.relationship('Post', backref='author', lazy='dynamic')
```

Все сообщения в блоге имеют тело (body), временную метку (timestamp) и связаны отношением «один ко многим» с моделью User. Поле body определено как db.Text, то есть не имеет ограничений по длине.

Форма, которая будет отображаться на главной странице приложения, дает пользователям возможность писать сообщения. Эта форма очень проста; она содержит лишь текстовую область для ввода сообщения и кнопку отправки. Определение формы приводится в примере 11.2.

Пример 11.2 ❖ `app/main/forms.py`: форма ввода сообщения

```
class PostForm(Form):
    body = TextAreaField("What's on your mind?", validators=[Required()])
    submit = SubmitField('Submit')
```

Функция представления `index()` обрабатывает форму и передает список старых сообщений в шаблон, как показано в примере 11.3.

Пример 11.3 ❖ `app/main/views.py`: маршрут главной страницы с формой ввода сообщения

```
@main.route('/', methods=['GET', 'POST'])
def index():
    form = PostForm()
    if current_user.can(Permission.WRITE_ARTICLES) and \
        form.validate_on_submit():
        post = Post(body=form.body.data,
                    author=current_user._get_current_object())
        db.session.add(post)
        return redirect(url_for('.index'))
    posts = Post.query.order_by(Post.timestamp.desc()).all()
    return render_template('index.html', form=form, posts=posts)
```

Эта функция передает форму и полный список сообщений в блоге. Список отсортирован по времени создания сообщений в порядке убывания (то есть от новых к старым). Форма создания нового сообщения обрабатывается как обычно: если текстовое поле прошло проверку, создается новый экземпляр `Post`. Однако перед созданием выполняется проверка привилегий пользователя на наличие права создавать новые сообщения.

Обратите внимание, что атрибуту `author` нового объекта `Post` присваивается значение выражения `current_user._get_current_object()`. Переменная `current_user` из `Flask-Login`, подобно всем переменным контекста, реализована как локальный для потока промежуточный объект (прокси-объект). Этот объект действует подобно объекту `User`, но в действительности это тонкая обертка вокруг фактического объекта `User`. В базу можно записать только настоящий объект `User`, для получения которого следует вызвать метод `_get_current_object()`.

Вслед за формой, отображаемой под приветствием в шаблоне `index.html`, располагаются сообщения, созданные ранее. Список сообщений – это первая попытка отобразить в блоге ход времени, когда все сообщения упорядочиваются сверху вниз в хронологическом порядке, от новых к старым. Изменения в шаблоне представлены в примере 11.4.

Пример 11.4 ❖ `app/templates/index.html`: шаблон главной страницы с сообщениями из блога

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
...
<div>
    {% if current_user.can(Permission.WRITE_ARTICLES) %}
    {{ wtf.quick_form(form) }}
    {% endif %}
</div>
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        <div class="profile-thumbnail">
            <a href="{{ url_for('.user', username=post.author.username) }}">
                
            </a>
        </div>
        <div class="post-date">{{ moment(post.timestamp).fromNow() }}</div>
        <div class="post-author">
            <a href="{{ url_for('.user', username=post.author.username) }}">
                {{ post.author.username }}
            </a>
        </div>
        <div class="post-body">{{ post.body }}</div>
    </li>
    {% endfor %}
</ul>
...
```

Обратите внимание, как с помощью метода `User.can()` пропускается создание формы ввода нового сообщения для пользователей, не имеющих привилегии `WRITE_ARTICLES` в роли. Список сообщений реализован как маркированный список HTML, с применением классов CSS для форматирования. Слева выводится уменьшенный аватар автора, при этом аватар и имя пользователя автора отображаются как ссылка на страницу профиля. Используемые здесь стили CSS хранятся в файле *styles.css*, в папке *static* приложения. Вы можете извлечь этот файл из репозитория на сайте GitHub. На рис. 11.1 показано, как выглядит главная страница с формой ввода и списком сообщений.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 11a` и получить данную версию приложения. Это обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория.

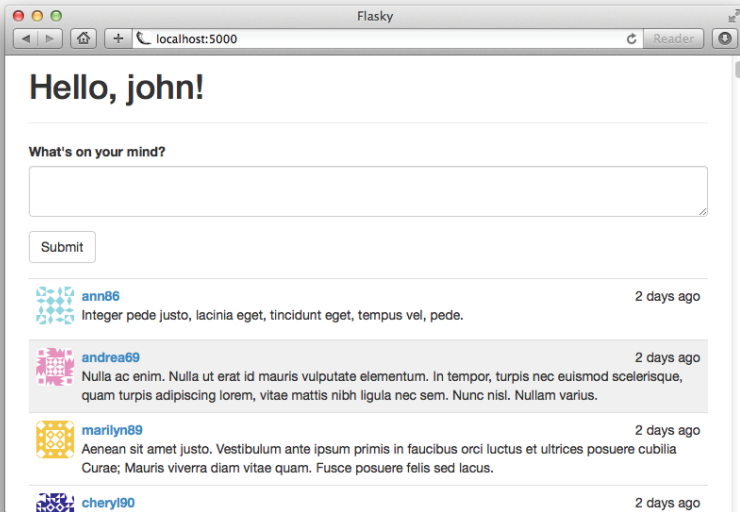


Рис. 11.1 ❖ Главная страница
с формой ввода и списком сообщений

Сообщения из блогов на страницах профилей

Страницу профиля пользователя можно улучшить, добавив в нее список сообщений, отправленных самим пользователем. В примере 11.5 показаны изменения в функции представления, необходимые для этого.

Пример 11.5 ❖ `app/main/views.py`: маршрут для страницы профиля со списком сообщений

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        abort(404)
    posts = user.posts.order_by(Post.timestamp.desc()).all()
    return render_template('user.html', user=user, posts=posts)
```

Список сообщений, принадлежащих пользователю, извлекается с помощью отношения `User.posts`, которое является объектом запроса, поддерживающим возможность применения фильтров, таких как `order_by()`.

В шаблон *user.html* необходимо также добавить дерево HTML-элементов `` для отображения списка сообщений, подобно тому как это сделано в шаблоне *index.html*. Однако наличие двух идентичных фрагментов разметки HTML выглядит избыточным, поэтому для подобных случаев в Jinja2 имеется директива `include()`. Шаблон *user.html* подключает список из внешнего файла, как показано в примере 11.6.

Пример 11.6 ❖ `app/templates/user.html`: шаблон страницы профиля со списком сообщений

```
...
<h3>Posts by {{ user.username }}</h3>
{% include '_posts.html' %}
...
```

В завершение этой реорганизации дерево элементов `` следует перенести из *index.html* в новый шаблон *_posts.html* и заменить директивой `include()`. Обратите внимание, что использование символа подчеркивания в качестве префикса в имени шаблона *_posts.html* не является обязательным требованием; это всего лишь соглашение, помогающее отличать обычные шаблоны от фрагментов шаблонов.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 11b` и получить эту версию приложения.

Постраничный вывод длинных списков сообщений

С ростом сайта будет увеличиваться и число сообщений. В какой-то момент станет совершенно непрактичным отображать весь список сообщений на главной странице и в странице профиля. Большие страницы требуют больше времени на их создание, загрузку и отображение веб-браузером, поэтому впечатления пользователя будут ухудшаться с увеличением размеров страниц. Решение этой проблемы заключается в разбиении длинных списков на страницы и отображении их по частям.

Создание фиктивных сообщений

Для отладки реализации страничного отображения списка необходимо наполнить тестовую базу данных большим числом сообщений. Можно, конечно, добавить сообщения вручную, но это весьма утомительно и требует значительного времени. Для выполнения данной работы лучше подошло бы автоматизированное решение. Для создания фиктивных данных можно использовать разные пакеты Python. Наиболее полным из них является *ForgeryPy*, который можно установить с помощью утилиты `pip`:

```
(venv) $ pip install forgerypy
```

Строго говоря, пакет *ForgeryPy* не является зависимостью приложения, потому что он нужен только на этапе разработки. Чтобы отделить зависимости готовой версии приложения от зависимостей этапа разработки, файл *requirements.txt* можно заменить папкой *requirements*, в которой хранить два набора зависимостей. Файл *dev.txt* внутри новой папки может хранить список зависимостей, необходимых для разработки, а файл *prod.txt* – список зависимостей, необходимых готовой версии приложения. Так как в обоих файлах будет присутствовать большое число зависимостей, общих для обоих файлов, их можно хранить в файле *common.txt* и подключать его в файлах *dev.txt* и *prod.txt* с помощью директивы `-r`. В примере 11.7 представлено содержимое файла *dev.txt*.

Пример 11.7 ❖ *requirements/dev.txt*: файл зависимостей для этапа разработки

```
-r common.txt
ForgeryPy==0.1
```

В примере 11.8 показаны дополнительные методы класса в моделях *User* и *Post*, генерирующие фиктивные данные.

Пример 11.8 ❖ *app/models.py*: создание активных пользователей и сообщений

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def generate_fake(count=100):
        from sqlalchemy.exc import IntegrityError
        from random import seed
        import forgerypy

        seed()
```

```
for i in range(count):
    u = User(email=forgergy_py.internet.email_address(),
             username=forgergy_py.internet.user_name(True),
             password=forgergy_py.lorem_ipsum.word(),
             confirmed=True,
             name=forgergy_py.name.full_name(),
             location=forgergy_py.address.city(),
             about_me=forgergy_py.lorem_ipsum.sentence(),
             member_since=forgergy_py.date.date(True))
    db.session.add(u)
    try:
        db.session.commit()
    except IntegrityError:
        db.session.rollback()

class Post(db.Model):
    # ...
    @staticmethod
    def generate_fake(count=100):
        from random import seed, randint
        import forgergy_py

        seed()
        user_count = User.query.count()
        for i in range(count):
            u = User.query.offset(randint(0, user_count - 1)).first()
            p = Post(body=forgergy_py.lorem_ipsum.sentences(randint(1, 3)),
                    timestamp=forgergy_py.date.date(True),
                    author=u)
            db.session.add(p)
        db.session.commit()
```

Значения для атрибутов этих фиктивных объектов генерируются с помощью генераторов случайной информации из пакета `ForgergyPy`, которые могут создавать имена, электронные адреса, предложения и многое другое, напоминающие настоящие.

Адреса электронной почты и имена пользователей должны быть уникальными, но, так как генераторы из `ForgergyPy` действуют достаточно случайно, есть вероятность создания дубликатов. В этих редких ситуациях попытка подтвердить сеанс базы данных будет возбуждать исключение `IntegrityError`. Обработчик этого исключения просто откатывает сеанс перед продолжением. Итерации цикла, сгенерировавшие повторяющиеся значения, не сохраняют информацию о пользователе в базу данных, поэтому общее число созданных фиктивных пользователей может оказаться меньше запрошенного.

Метод, генерирующий сообщения, должен присвоить каждому сообщению случайно выбранного пользователя. Для этого исполь-

зуется фильтр `offset()` запроса. Этот фильтр отбрасывает указанное число результатов. Определяя случайное смещение и вызывая затем метод `first()`, каждый раз можно получать ссылку на случайного пользователя.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 10c` и получить эту версию приложения. Чтобы гарантировать установку всех необходимых расширений, выполните также команду `pip install -r requirements/dev.txt`.

С помощью новых методов легко можно создать большое число фиктивных пользователей и сообщений прямо в интерактивной оболочке Python:

```
(venv) $ python manage.py shell
>>> User.generate_fake(100)
>>> Post.generate_fake(100)
```

Если теперь запустить приложение, на главной странице можно увидеть длинный список фиктивных сообщений.

Постраничное отображение данных

В примере 11.9 приводятся изменения в маршруте для главной страницы, необходимые для поддержки постраничного отображения.

Пример 11.9 ❖ `app/main/views.py`: постраничное отображение списка сообщений

```
@main.route('/', methods=['GET', 'POST'])
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.order_by(Post.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
        pagination=pagination)
```

Номер страницы определяется из строки запроса, доступной как `request.args`. Когда номер страницы не указан явно, по умолчанию используется число 1 (первая страница). Аргумент `type=int` гарантирует возврат значения по умолчанию, если аргумент не сможет быть преобразован в целое число.

Чтобы загрузить единственную страницу записей, вместо метода `all()` вызывается метод `paginate()`. Этот метод принимает но-

мер страницы в первом и единственном обязательном аргументе. Чтобы определить число элементов на странице, можно передать необязательный аргумент `per_page`. Если этот аргумент не указан явно, по умолчанию он принимает значение 20. Еще один необязательный аргумент `error_out` позволяет генерировать ошибку 404 при попытке запросить страницу за пределами допустимого диапазона, если передать в нем значение `True` (по умолчанию). Если передать в `error_out` значение `False`, для страниц с номерами вне допустимого диапазона будет возвращаться пустой список элементов. Чтобы сделать размер страниц настраиваемым, значение аргумента `per_page` извлекается из параметра настройки приложения с именем `FLASKY_POSTS_PER_PAGE`.

После добавления этих изменений список на главной странице будет содержать ограниченное число сообщений. Чтобы увидеть вторую страницу сообщений, добавьте строку запроса `?page=2` в конец URL в адресной строке браузера.

Виджет постраничного отображения

Метод `paginate()` возвращает объект класса `Pagination`, объявленного в расширении `Flask-SQLAlchemy`. Этот объект содержит несколько свойств, которые удобно использовать для создания ссылок на страницы в шаблоне, поэтому он передается шаблону в виде аргумента. Список атрибутов объекта `Pagination` приводится в табл. 11.1.

Таблица 11.1. Атрибуты объекта *Pagination* из расширения *Flask-SQLAlchemy*

Атрибут	Описание
<code>items</code>	Записи в текущей странице
<code>query</code>	Исходный запрос, подвергшийся разбиению на страницы
<code>page</code>	Номер текущей страницы
<code>prev_num</code>	Номер предыдущей страницы
<code>next_num</code>	Номер следующей страницы
<code>has_next</code>	<code>True</code> , если имеется следующая страница
<code>has_prev</code>	<code>True</code> , если имеется предыдущая страница
<code>pages</code>	Общее число страниц в запросе
<code>per_page</code>	Число элементов в странице
<code>total</code>	Общее число элементов, возвращаемых запросом

Объект `Pagination` имеет также несколько удобных методов, которые перечислены в табл. 11.2.

Таблица 11.2. Методы объекта *Pagination* из расширения *Flask-SQLAlchemy*

Метод	Описание
<code>iter_pages(left_edge=2, left_current=2, right_current=5, right_edge=2)</code>	Итератор, возвращающий последовательность номеров страниц для отображения в виджете. Аргумент <code>left_edge</code> определяет число страниц с левого края, <code>left_current</code> – число страниц слева от текущей страницы, <code>right_current</code> – справа от текущей страницы, и <code>right_edge</code> – число страниц с правого края. Например, для страницы с номером 50 при общем числе страниц 100 данный итератор с аргументами по умолчанию вернет следующие страницы: 1, 2, None, 48, 49, 50, 51, 52, 53, 54, 55, None, 99, 100. Значение <code>None</code> в последовательности указывает на наличие промежутка
<code>prev()</code>	Объект <i>Pagination</i> для предыдущей страницы
<code>next()</code>	Объект <i>Pagination</i> для следующей страницы

Вооружившись эти мощным объектом и классами CSS из *Bootstrap*, легко можно организовать постраничный вывод сообщений в шаблоне. Реализация, представленная в примере 11.10, оформлена в виде макроса `Jinja2`.

Пример 11.10 ❖ `app/templates/_macros.html`: макрос постраничного ввода списков для использования в шаблонах

```
{% macro pagination_widget(pagination, endpoint) %}
<ul class="pagination">
  <li{% if not pagination.has_prev %} class="disabled"{% endif %}>
    <a href="{% if pagination.has_prev %}{ url_for(endpoint,
      page = pagination.page - 1, **kwargs) }}{% else %}#{% endif %}">
      &laquo;
    </a>
  </li>
  {% for p in pagination.iter_pages() %}
    {% if p %}
      {% if p == pagination.page %}
        <li class="active">
          <a href="{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
        </li>
      {% else %}
        <li>
          <a href="{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
        </li>
      {% endif %}
    {% else %}
      <li class="disabled"><a href="#">&hellip;</a></li>
    {% endif %}
  {% endfor %}
</ul>
```

```

{% endfor %}
<li{% if not pagination.has_next %} class="disabled"{% endif %}>
  <a href="{% if pagination.has_next %}{% url_for(endpoint,
    page = pagination.page + 1, **kwargs) %}{% else %}#{% endif %}">
    &raquo;
  </a>
</li>
</ul>
{% endmacro %}

```

Макрос создает элемент постраничного отображения, оформленный как неупорядоченный список. Он определяет следующие ссылки:

- ссылку на «предыдущую страницу», которая оформляется с применением класса `disabled`, если текущая страница является первой;
- ссылки на все страницы, возвращаемые итератором `iter_pages()` объекта. Эти страницы отображаются как ссылки, полученные вызовами `url_for()` с явными номерами страниц в качестве аргументов. Ссылка на текущую страницу отображается с применением класса CSS `active`. Промежутки в последовательности страниц отображаются как символ многоточия;
- ссылку на «следующую страницу», которая оформляется с применением класса `disabled`, если текущая страница является последней.

Макросы Jinja2 всегда принимают именованные аргументы без необходимости включать `**kwargs` в список аргументов. Все именованные аргументы макрос постраничного отображения списка передает в вызов `url_for()`, генерирующий ссылки на страницы. Этот прием можно использовать с маршрутами, имеющими динамическую часть, такими как страница профиля.

Вызов макроса `pagination_widget` можно добавить после подключения шаблона `_posts.html`, в шаблонах `index.html` и `user.html`. В примере 11.11 показано, как это сделать в шаблоне главной страницы приложения.

Пример 11.11 ❖ `app/templates/index.html`: постраничный вывод списка сообщений

```

{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% import "_macros.html" as macros %}
...
{% include '_posts.html' %}

```

```
<div class="pagination">
  {{ macros.pagination_widget(pagination, '.index') }}
</div>
{% endif %}
```

На рис. 11.2 показано, как выглядят ссылки на страницы.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 11d` и получить эту версию приложения.

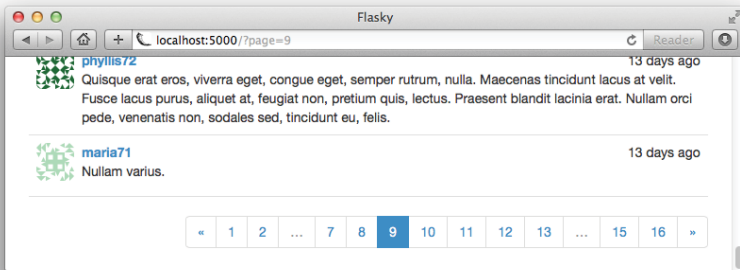


Рис. 11.2 ❖ Постраничный вывод сообщений

Форматирование текста сообщений с помощью Markdown и Flask-PageDown

Простого неформатированного текста вполне достаточно для коротких сообщений, но пользователи, желающие писать большие статьи, определенно ощутят нехватку поддержки форматирования. В этом разделе мы дополним текстовое поле ввода поддержкой синтаксиса Markdown и реализуем предварительный просмотр сообщения в форматированном виде.

Для реализации этих особенностей потребуется несколько новых пакетов:

- PageDown, инструмент преобразования разметки Markdown в разметку HTML на стороне клиента, реализованный на JavaScript;
- Flask-PageDown, обертка вокруг фреймворка PageDown для Flask, позволяющая интегрировать PageDown в формы Flask-WTF;

- Markdown, инструмент преобразования разметки Markdown в разметку HTML на стороне сервера, реализованный на Python;
- Bleach, инструмент предварительной обработки HTML, реализованный на Python.

Пакеты на Python можно установить с помощью утилиты pip:

```
(venv) $ pip install flask-pagedown markdown bleach
```

Flask-PageDown

Расширение Flask-PageDown определяет класс PageDownField, имеющий тот же интерфейс, что и класс TextAreaField из WTForms. Прежде чем поле этого типа можно будет использовать, расширение должно быть инициализировано, как показано в примере 11.12.

Пример 11.12 ❖ app/__init__.py: инициализация расширения Flask-PageDown

```
from flask.ext.pagedown import PageDown
# ...

pagedown = PageDown()
# ...
def create_app(config_name):
    # ...
    pagedown.init_app(app)
    # ...
```

Чтобы преобразовать элемент ввода текста в редактор с поддержкой форматирования Markdown, поле body в форме PostForm должно быть создано вызовом конструктора PageDownField, как показано в примере 11.13.

Пример 11.13 ❖ app/main/forms.py: форма создания сообщений с поддержкой форматирования Markdown

```
from flask.ext.pagedown.fields import PageDownField

class PostForm(Form):
    body = PageDownField("What's on your mind?", validators=[Required()])
    submit = SubmitField('Submit')
```

Область предварительного просмотра разметки Markdown создается с помощью библиотек PageDown, поэтому их следует добавить в шаблон. Расширение Flask-PageDown упрощает эту задачу, предоставляя макрос, подключающий необходимые файлы из CDN, как показано в примере 11.14.

Пример 11.14 ❖ `app/index.html`: объявление шаблона Flask-PageDown

```
{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 10e` и получить эту версию приложения. Чтобы гарантировать установку всех необходимых расширений, выполните также команду `pip install -r requirements/dev.txt`.

После внесения этих изменений разметка Markdown, введенная в текстовую область, будет немедленно отображаться в формате HTML в области предварительного просмотра ниже. На рис. 11.3 показано, как выглядит форма создания сообщения с отформатированным текстом.

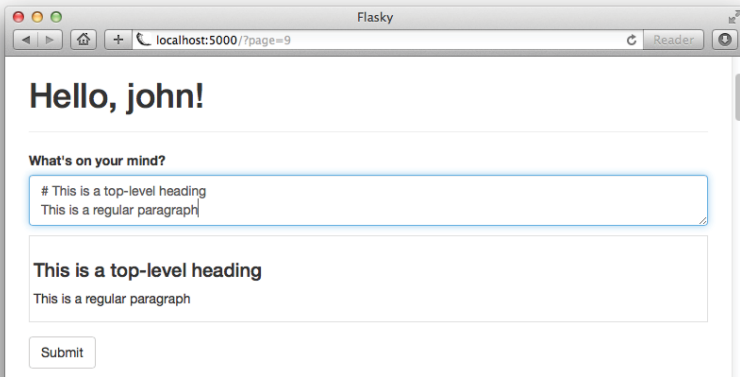


Рис. 11.3 ❖ Форма ввода форматированного текста

Обработка форматированного текста на сервере

При отправке формы на сервер в запрос POST включается только разметка Markdown; разметка HTML из области предварительно просмотра просто уничтожается. Отправка разметки HTML, сгенерированной для области предварительного просмотра, представляет угрозу безопасности, так как злоумышленник относительно легко сможет сконструировать разметку HTML, не соответствующую разметке Markdown, и отправить ее. Чтобы устранить любой риск, от-

правляться будет только разметка Markdown, а на сервере она вновь будет преобразовываться в разметку HTML с использованием пакета *Markdown*, инструмента преобразования разметки Markdown в разметку HTML на языке Python. Получившаяся разметка HTML будет обработана с помощью инструмента *Bleach*, сужающего круг допустимых тегов HTML.

Преобразование разметки Markdown в разметку HTML можно организовать в шаблоне *_posts.html*, но это неэффективно, так как сообщения придется преобразовывать при каждом отображении страницы. Чтобы избежать многократных преобразований, преобразованную версию можно сохранять вместе с самим сообщением в момент его создания. Разметку HTML для отображения можно хранить в новом поле модели *Post*, непосредственно доступном из шаблонов. Оригинальный текст сообщений в формате Markdown также можно сохранять в базе данных, на случай, если его потребуется отредактировать. Все необходимые изменения в модели *Post* приводятся в примере 11.15.

Пример 11.15 ❖ app/models/post.py: обработка разметки Markdown в модели *Post*

```
from markdown import markdown
import bleach

class Post(db.Model):
    # ...
    body_html = db.Column(db.Text)
    # ...
    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'blockquote', 'code',
                        'em', 'i', 'li', 'ol', 'pre', 'strong', 'ul',
                        'h1', 'h2', 'h3', 'p']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Post.body, 'set', Post.on_changed_body)
```

Функция *on_changed_body* регистрируется как обработчик события *set* для поля *body*. Это означает, что она автоматически будет вызываться при изменении поля *body* в любом экземпляре класса. Функция создает HTML-версию сообщения и сохраняет ее в поле *body_html*, обеспечивая тем самым автоматическое преобразование разметки Markdown в разметку HTML.

Фактическое преобразование выполняется в три этапа. Сначала с помощью функции `markdown()` выполняется начальное преобразование текста в разметку HTML. Затем результат со списком допустимых тегов HTML передается функции `clean()`. Функция `clean()` удаляет из разметки все теги, отсутствующие в «белом списке». В заключение вызывается функция `linkify()`, предоставляемая фреймворком Bleach, которая преобразует адреса URL, присутствующие в тексте, в правильно оформленные ссылки `<a>`. Этот последний этап необходим, потому что автоматическое создание ссылок не предусматривается спецификацией Markdown. PageDown поддерживает ее как расширение, поэтому `linkify()` используется для совместимости.

Последнее изменение заключается в замене `post.body` на `post.body_html` в шаблоне, как показано в примере 11.16.

Пример 11.16 ❖ `app/templates/_posts.html`: использование HTML-версий сообщений в шаблоне

```
...
<div class="post-body">
    {% if post.body_html %}
        {{ post.body_html | safe }}
    {% else %}
        {{ post.body }}
    {% endif %}
</div>
...
```

Окончание `| safe` при отображении тела в формате HTML сообщает фреймворку Jinja2, что он не должен экранировать элементы HTML. По умолчанию Jinja2 экранирует все переменные шаблонов для безопасности. Так как разметка HTML сгенерирована на стороне сервера, ее можно смело отображать, не прибегая к защитным мерам.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 11f` и получить эту версию приложения. Это обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория. Чтобы гарантировать установку всех необходимых расширений, выполните также команду `pip install -r requirements/dev.txt`.

Постоянные ссылки на сообщения


У пользователей может появиться желание обмениваться ссылками на конкретные сообщения со своими друзьями в социальных сетях. Для этой цели каждое сообщение может быть связано с отдельной

страницей, имеющей уникальный URL. Маршрут и функция представления для поддержки постоянных ссылок показаны в примере 11.17.

Пример 11.17 ❖ `app/main/views.py`: постоянные ссылки на сообщения

```
@main.route('/post/<int:id>')
def post(id):
    post = Post.query.get_or_404(id)
    return render_template('post.html', posts=[post])
```

Адреса URL, присваиваемые сообщениям, конструируются с применением значения поля `id`, которое присваивается сообщению при добавлении в базу данных.

 Для некоторых приложений может оказаться предпочтительнее создавать постоянные ссылки с читаемыми адресами URL вместо числовых идентификаторов. Альтернативой числовым идентификаторам может служить «ключ» — уникальная строка, связанная с сообщением.

Обратите внимание, что шаблон *post.html* принимает список, содержащий единственное сообщение. Отправка списка позволяет использовать здесь шаблон *_posts.html*, который уже используется в *index.html* и *user.html*.

Постоянные ссылки добавляются в конец каждого сообщения внутри универсального шаблона *_posts.html*, как показано в примере 11.18.

Пример 11.18 ❖ `app/templates/_posts.html`: постоянные ссылки в сообщениях

```
<ul class="posts">
  {% for post in posts %}
  <li class="post">
    ...
    <div class="post-content">
      ...
      <div class="post-footer">
        <a href="{{ url_for('.post', id=post.id) }}">
          <span class="label label-default">Permalink</span>
        </a>
      </div>
    </div>
  </li>
  {% endfor %}
</ul>
```

Новый шаблон *post.html*, отображающий страницу по постоянной ссылке, показан в примере 11.19. Он подключает шаблон *_posts.html*.

Пример 11.19 ❖ `app/templates/post.html`: шаблон страницы, на которую указывает постоянная ссылка

```
{% extends «base.html» %}

{% block title %}Flasky - Post{% endblock %}

{% block page_content %}
{% include '_posts.html' %}
{% endblock %}
```



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 11g` и получить эту версию приложения.

Редактор сообщений

Последняя особенность, связанная с сообщениями в блоге, которую нам предстоит реализовать, — это редактор сообщений, позволяющий пользователям исправлять свои сообщения. Редактор сообщений будет реализован в виде отдельной страницы. В верхней части страницы будет отображаться текущая версия сообщения для справки, а ниже — редактор разметки Markdown. Редактор будет основан на расширении `Flask-PageDown`, поэтому в самом низу страницы будет отображаться исправленная версия сообщения. Шаблон `edit_post.html` этой страницы показан в примере 11.20.

Пример 11.20 ❖ `app/templates/edit_post.html`: шаблон редактора сообщений

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky - Edit Post{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Edit Post</h1>
</div>
<div>
  {{ wtf.quick_form(form) }}
</div>
{% endblock %}

{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

Маршрут, поддерживающий редактор сообщений, показан в примере 11.21.

Пример 11.21 ❖ `app/main/views.py`: маршрут редактирования сообщений

```
@main.route('/edit/<int:id>', methods=['GET', 'POST'])
@login_required
def edit(id):
    post = Post.query.get_or_404(id)
    if current_user != post.author and \
        not current_user.can(Permission.ADMINISTER):
        abort(403)
    form = PostForm()
    if form.validate_on_submit():
        post.body = form.body.data
        db.session.add(post)
        flash('The post has been updated.')
        return redirect(url_for('post', id=post.id))
    form.body.data = post.body
    return render_template('edit_post.html', form=form)
```

Эта функция представления позволяет редактировать сообщения только их авторам и администраторам. Если пользователь попытается отредактировать сообщение, написанное другим пользователем, функция представления вернет код ошибки 403. Здесь, как и в главной странице, используется класс `PostForm` веб-формы.

Для полноты поддержки новой особенности ссылку на редактор сообщений можно добавить в конец каждого сообщения, рядом с постоянной ссылкой, как показано в примере 11.22.

Пример 11.22 ❖ `app/templates/_posts.html`: ссылки на редактор сообщений

```
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        ...
        <div class="post-content">
            ...
            <div class="post-footer">
                ...
                {% if current_user == post.author %}
                <a href="{{ url_for('.edit', id=post.id) }}">
                    <span class="label label-primary">Edit</span>
                </a>
                {% elif current_user.is_administrator() %}
                <a href="{{ url_for('.edit', id=post.id) }}">
                    <span class="label label-danger">Edit [Admin]</span>
```

```

        </a>
        {% endif %}
    </div>
</div>
</li>
{% endfor %}
</ul>

```

В результате этих изменений к сообщениям в блоге, принадлежащим текущему пользователю, будет добавлена ссылка «Edit» (Изменить). Если текущим пользователем является администратор, ссылка будет добавлена ко всем сообщениям. Кроме того, ссылки для администраторов будут оформлены немного иначе, чтобы лишний раз подчеркнуть, что эта функция будет выполняться от лица администратора. На рис. 11.4 показано, как выглядят постоянные ссылки и ссылки на редактор в окне браузера.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 11h` и получить эту версию приложения.



Рис. 11.4 ❖ Постоянные ссылки и ссылки на редактор в сообщениях

Глава 12

Читающие и читаемые

Социальные веб-приложения дают своим пользователям возможность следить за событиями, происходящими у других пользователей. В приложениях подобные отношения называют по-разному: «читающие» (followers)¹, «друзья» (friends), «контакты» (contacts), «связи» (connections) или «приятели» (buddies), но суть их не меняется от названия – во всех случаях создаются прямые ссылки между парами пользователей, и эти ссылки применяются в запросах к базе данных.

В этой главе вы узнаете, как реализовать поддержку «читающих» в приложении Flasky. Она дает возможность одним пользователям «читать» других и настраивать фильтры сообщений на главной странице так, чтобы отображались только сообщения, оставленные пользователями, которых они читают.

Пересмотр отношений в базе данных

Как обсуждалось в главе 5, взаимосвязи, устанавливаемые между записями в базе данных, называют отношениями. Наиболее типичным представителем является отношение «один ко многим», оно соответствует ситуации, когда запись оказывается связана со списком родственных записей. Для реализации отношения этого типа элементы со стороны «ко многим» должны иметь внешний ключ, ссылающийся на связанный элемент со стороны «один». В примере приложения, в его текущем состоянии, имеются два отношения «один ко многим»: одно связывает роли со списками пользователей, а другое связывает пользователей с их сообщениями.

¹ Здесь и далее используется терминология, заимствованная на сайте Twitter.com. – *Прим. перев.*

Большинство других типов отношений можно вывести из отношения «один ко многим». Отношение «многие к одному» выглядит как «один ко многим» со стороны «многие». Отношение «один к одному» можно рассматривать как упрощенный вариант отношения «один ко многим», где сторона «ко многим» ограничена единственным элементом. И только отношение «многие ко многим» не может быть реализовано как упрощенный вариант отношения «один ко многим», так как представляет списки элементов с обеих сторон. Этот тип отношений подробно описывается в следующем разделе.

Отношение «многие ко многим»

В отношениях «один ко многим», «многие к одному» и «один к одному» имеется хотя бы одна сторона с единственным элементом, поэтому ссылки между взаимосвязанными записями могут быть реализованы с помощью внешних ключей, указывающих на единственный элемент. Но как реализовать отношение, имеющее множество элементов с обеих сторон?

Рассмотрим классический пример отношения «многие ко многим»: база данных студентов и предметов, которые они изучают. Очевидно, что нельзя добавить внешний ключ на предмет в таблицу студентов, потому что всякий студент изучает множество предметов – единственного внешнего ключа оказывается недостаточно. Аналогично нельзя добавить внешний ключ на студента в таблицу предметов, потому что один предмет изучает множество студентов. С обеих сторон необходим список внешних ключей.

Решение заключается в том, чтобы добавить в базу данных третью таблицу, которую обычно называют *ассоциативной таблицей*. Теперь отношение «многие ко многим» можно разложить на два отношения «один ко многим» каждой из двух таблиц с ассоциативной таблицей. На рис. 12.1 показано, как строится отношение «многие ко многим» между студентами и предметами.

Роль ассоциативной таблицы в этом примере играет таблица *registrations*. Каждая строка в этой таблице представляет подписку студента на изучение предмета.

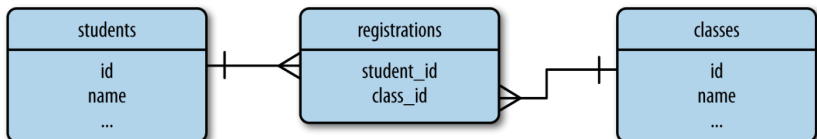


Рис. 12.1 ❖ Пример отношения «многие ко многим»

Запрос к отношению «многие ко многим» выполняется в два этапа. Чтобы получить список предметов, изучаемых одним студентом, используется отношение «один ко многим» между таблицами `students` и `registrations`, в результате чего получается список регистрационных записей для указанного студента. Затем в работу включается отношение «один ко многим» между таблицами `classes` и `registrations` и выполняется переход в направлении от многих к одному, чтобы получить все предметы, связанные с регистрационными записями для данного студента. Аналогично выполняется поиск всех студентов, изучающих данный предмет: сначала извлекается список всех регистрационных записей для данного предмета, а затем – список студентов, связанных с этими регистрационными записями.

Следование по двум отношениям, чтобы получить требуемый запрос, может показаться сложной задачей, однако для достаточно простых отношений, как в данном примере, почти всю работу фреймворк `SQLAlchemy` выполняет самостоятельно. Ниже приводится код, реализующий отношение «многие ко многим», изображенное на рис. 12.1:

```
registrations = db.Table('registrations',
    db.Column('student_id', db.Integer, db.ForeignKey('students.id')),
    db.Column('class_id', db.Integer, db.ForeignKey('classes.id'))
)

class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    classes = db.relationship('Class',
                             secondary=registrations,
                             backref=db.backref('students', lazy='dynamic'),
                             lazy='dynamic')

class Class(db.Model):
    id = db.Column(db.Integer, primary_key = True)
    name = db.Column(db.String)
```

Отношение определяется с помощью все той же конструкции `db.relationship()`, которая использовалась для определения отношений «один ко многим», но при определении отношений «многие ко многим» необходимо добавлять аргумент `secondary` с объектом ассоциативной таблицы. Отношение можно определить в любом из двух классов с аргументом `backref`, экспортирующим отношение с одной стороны в другую. Ассоциативная таблица определяется как простая таблица, а не как модель, потому что этой таблицей будет управлять сам фреймворк `SQLAlchemy`.

Отношение `classes` использует семантику списка, что делает работу с отношениями «многие ко многим», созданными таким способом, чрезвычайно простой. Например, для объекта `s`, представляющего студента, и объекта `c`, представляющего предмет, регистрацию студента для изучения предмета можно реализовать так:

```
>>> s.classes.append(c)
>>> db.session.add(s)
```

Получить список предметов, изучаемых студентом `s`, и список студентов, изучающих предмет `c`, можно получить так:

```
>>> s.classes.all()
>>> c.students.all()
```

Отношение `students`, доступное в модели `Class`, определяется аргументом `backref`. Обратите внимание, что в этом примере аргумент `db.backref()` дополнен атрибутом `lazy = 'dynamic'`, благодаря чему с обеих сторон возвращается объект запроса, к которому можно применять дополнительные фильтры.

Если позднее студент решит отказаться от изучения предмета `c`, обновить информацию в базе данных можно будет, как показано ниже:

```
>>> s.classes.remove(c)
```

Самоссылочные отношения

Отношение «многие ко многим» можно использовать для моделирования пользователей, читающих других пользователей, но здесь есть одна проблема. В примере со студентами и предметами имелись две сущности, связанные посредством ассоциативной таблицы. Однако, ведя речь о чтении одними пользователями других, мы имеем только пользователей – вторая сущность отсутствует.

Отношения, в которых с обеих сторон находится одна и та же таблица, называют *самоссылочными* (*self-referential*). В данном случае сущностями слева являются пользователи, которые могут называться «читающими» («followers»). Сущностями справа также являются пользователи, но они уже называются «читаемыми» («followed»). Концептуально самоссылочные отношения ничем не отличаются от обычных отношений, но они труднее в понимании. На рис. 12.2 изображена диаграмма самоссылочного отношения в базе данных, представляющая принцип чтения одними пользователями других.

Ассоциативная таблица в данном случае называется `follows`. Каждая строка в этой таблице представляет пользователя, читающе-

го другого пользователя. Отношение «один ко многим», изображенное слева, связывает пользователей со списком строк в таблице `follows`, в которых они интерпретируются как читающие. Отношение «один ко многим», изображенное справа, связывает пользователей со списком строк в таблице `follows`, в которых они интерпретируются как читаемые.

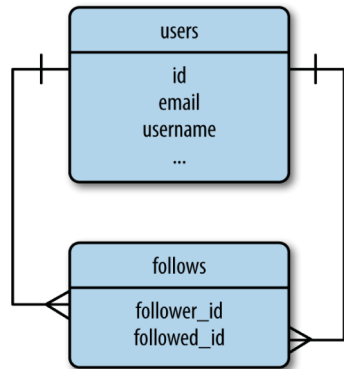


Рис. 12.2 ❖ Читающие, отношение «многие ко многим»

Усовершенствованные отношения «многие ко многим»

После настройки самоссыльных отношений, как показано в предыдущем примере, база данных сможет представлять читающих, но с одним ограничением. Обычно при работе с отношениями «многие ко многим» требуется хранить дополнительные данные, так или иначе связанные с отношениями между двумя сущностями. Так, для отношений между читающими и читаемыми может пригодиться дата, когда один пользователь стал читающим другого, что позволит сортировать списки читающих в хронологическом порядке. Единственное место, где может храниться такая информация, – ассоциативная таблица, но в реализации, подобной той, что была показана выше, в примере со студентами и предметами, ассоциативная таблица является внутренней, полностью управляемой фреймворком SQLAlchemy.

Чтобы получить возможность работать с дополнительными данными в отношениях, ассоциативную таблицу следует реализовать как полноценную модель, доступную для приложения. В примере 12.1 приводится определение новой модели `Follow`, представляющей ассоциативную таблицу.

Пример 12.1 ❖ `app/models/user.py`: ассоциативная таблица `follows` как модель

```
class Follow(db.Model):
    __tablename__ = 'follows'
    follower_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                           primary_key=True)
    followed_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                           primary_key=True)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

Фреймворк SQLAlchemy не сможет использовать эту ассоциативную таблицу прозрачно, иначе приложение не получит доступа к дополнительным полям в ней. Поэтому отношение «многие ко многим» следует разложить на два простых отношения «один ко многим», для левой и правой сторон, и определить их как стандартные отношения. Как это сделать, показано в примере 12.2.

Пример 12.2 ❖ `app/models/user.py`: реализация отношения «многие ко многим» в виде двух отношений «один ко многим»

```
class User(UserMixin, db.Model):
    # ...
    followed = db.relationship('Follow',
                               foreign_keys=[Follow.follower_id],
                               backref=db.backref('follower', lazy='joined'),
                               lazy='dynamic',
                               cascade='all, delete-orphan')
    followers = db.relationship('Follow',
                                foreign_keys=[Follow.followed_id],
                                backref=db.backref('followed', lazy='joined'),
                                lazy='dynamic',
                                cascade='all, delete-orphan')
```


Здесь отношения `followed` и `followers` определены как отдельные отношения «один ко многим». Обратите внимание, что с целью устранения неоднозначности для каждого отношения потребовалось явно указать, какой внешний ключ используется, добавив необязательный именованный аргумент `foreign_keys`. Аргументы в вызове `db.backref()` в этих отношениях применяются не друг к другу, а к модели `Follow`.

Аргумент `lazy` в вызовах `db.backref()` определяет, как выполняется соединение. В режиме `lazy='joined'` связанные объекты извлекаются немедленно из запроса соединения. Например, если пользователь читает сотню других пользователей, вызов `user.followed.all()` вернет список, содержащий 100 экземпляров `Follow`, каждый из которых будет иметь свойства `follower` и `followed`, ссылающиеся на соответствующих пользователей. Режим `lazy='joined'` позволяет выполнять все необходимые операции в единственном запросе к базе данных. Если в аргументе `lazy` передать значение по умолчанию `select`, выборка читающих и читаемых будет отложена до первого обращения, а для установки каждого атрибута потребуется выполнить отдельный запрос, то есть для получения полного списка читаемых пользователей понадобится выполнить 100 дополнительных запросов к базе данных.

Аргументы `lazy` в вызовах `db.relationship()` в обоих отношениях преследуют иные цели. Они соответствуют стороне «один» и воз-

вращают списки со стороны «ко многим»; в режиме `dynamic` при обращении к атрибутам отношений возвращаются объекты запросов, а не сами данные, благодаря чему открывается возможность применить дополнительные фильтры перед выполнением запроса.

Аргумент `cascade` определяет, как операции с родительским объектом будут влиять на связанные с ним объекты. Одним из примеров вариантов каскадирования может служить правило, требующее при добавлении объекта в сеанс базы данных добавлять также объекты, связанные с ним отношениями. Настройки каскадирования по умолчанию пригодны для большинства ситуаций, но в отношениях «многие ко многим» они не пригодны. С настройками по умолчанию, когда объект удаляется, соответствующему внешнему ключу во всех связанных с ним объектах присваивается пустое значение. Но для ассоциативной таблицы правильнее было бы удалить строки, ссылающиеся на удаленную запись. Именно это и обеспечивает значение `delete-orphan` в параметре `cascade`.

 В параметре `cascade` передается список значений, разделенных запятыми. Кому-то может показаться странным, но значение `all` представляет все (all) возможные настройки каскадирования, кроме `delete-orphan`. Список `'all, delete-orphan'` оставляет включенными все настройки по умолчанию и добавляет удаление «осиротевших» (`delete orphans`) записей.

Теперь необходимо добавить в приложение поддержку двух отношений «один ко многим», чтобы реализовать отношение «многие ко многим». Так, операции с этим отношением будут повторяться достаточно часто, их лучше оформить в виде вспомогательных методов в модели `User`. Всего нам понадобятся четыре метода, которые представлены в примере 12.3.

Пример 12.3 ❖ `app/models/user.py`: вспомогательные методы для реализации читающих

```
class User(db.Model):
    # ...
    def follow(self, user):
        if not self.is_following(user):
            f = Follow(follower=self, followed=user)
            db.session.add(f)

    def unfollow(self, user):
        f = self.followed.filter_by(followed_id=user.id).first()
        if f:
            db.session.delete(f)

    def is_following(self, user):
```

```

return self.followed.filter_by(
    followed_id=user.id).first() is not None

def is_followed_by(self, user):
    return self.followers.filter_by(
        follower_id=user.id).first() is not None

```

Метод `follow()` вручную вставляет в ассоциативную таблицу экземпляр `Follow`, связывающий читающего с читаемым и дающий приложению возможность устанавливать дополнительные поля. Два связываемых пользователя вручную передаются конструктору экземпляра `Follow`, а затем полученный объект добавляется в сеанс базы данных, как обычно. Обратите внимание, что в данном случае нет необходимости вручную устанавливать поле `timestamp`, потому что оно настроено так, что по умолчанию получает значение текущего времени. Метод `unfollow()` использует отношение `followed` для поиска экземпляра `Follow`, ссылающегося на читаемого пользователя, связь с которым следует разорвать. Чтобы разорвать связь между двумя пользователями, достаточно просто удалить объект `Follow`. Методы `is_following()` и `is_followed_by()` выполняют поиск левой и правой сторон в отношениях «один ко многим», соответственно, для указанного пользователя и возвращают `True`, если искомый пользователь найден.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 12a` и получить данную версию приложения. Это обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория.

Теперь база данных полностью готова к реализации поддержки описываемой в этой главе особенности. Модульные тесты, проверяющие отношение в базе данных, можно найти в репозитории на сайте GitHub.

Читающие и читаемые на странице профиля

Было бы неплохо добавить на страницу профиля кнопку «Follow» (Читать), если просматривающий ее пользователь еще не является читающим владельца профиля, или кнопку «Unfollow» (Отменить), если пользователь является читающим. Также отличным дополнением было бы отображение счетчиков читающих и читаемых, возможность вывода списков читающих и читаемых и отображение подписи

«Follows You» (Читают) там, где это имеет смысл. Соответствующие изменения в шаблоне профиля показаны в примере 12.4, а на рис. 12.3 можно посмотреть, как выглядят описываемые дополнения.

Пример 12.4 ❖ `app/templates/user.html`: вывод дополнительной информации о читающих и читаемых на странице профиля

```
{% if current_user.can(Permission.FOLLOW) and user != current_user %}
    {% if not current_user.is_following(user) %}
        <a href="{{ url_for('.follow', username=user.username) }}"
            class="btn btn-primary">Follow</a>
    {% else %}
        <a href="{{ url_for('.unfollow', username=user.username) }}"
            class="btn btn-default">Unfollow</a>
    {% endif %}
{% endif %}
<a href="{{ url_for('.followers', username=user.username) }}">
    Followers: <span class="badge">{{ user.followers.count() }}</span>
</a>
<a href="{{ url_for('.followed_by', username=user.username) }}">
    Following: <span class="badge">{{ user.followed.count() }}</span>
</a>
{% if current_user.is_authenticated() and user != current_user and
    user.is_following(current_user) %}
| <span class="label label-default">Follows you</span>
{% endif %}
```

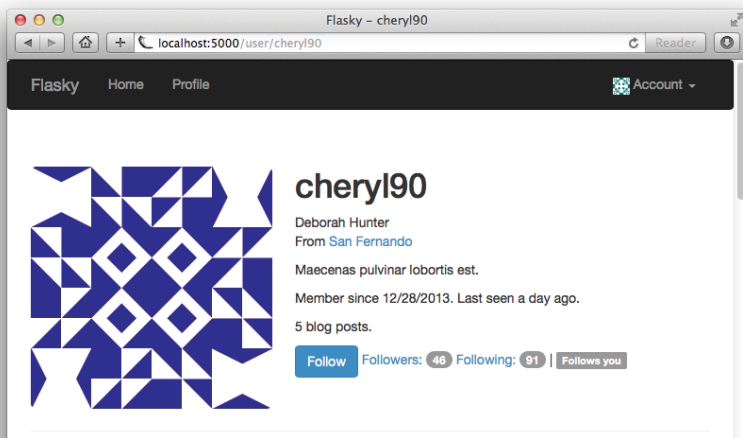


Рис. 12.3 ❖ Информация о читающих и читаемых на странице профиля

В результате изменений в шаблоне профиля появились четыре новые конечные точки. Маршрут `/follow/<имя_пользователя>` вызывается, когда пользователь щелкает на кнопке «Follow» (Читать) в странице профиля другого пользователя. Его реализация приводится в примере 12.5.

Пример 12.5 ❖ `app/main/views.py`: маршрут «follow» и функция представления

```
@main.route('/follow/<username>')
@login_required
@permission_required(Permission.FOLLOW)
def follow(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('Invalid user.')
        return redirect(url_for('.index'))
    if current_user.is_following(user):
        flash('You are already following this user.')
        return redirect(url_for('.user', username=username))
    current_user.follow(user)
    flash('You are now following %s.' % username)
    return redirect(url_for('.user', username=username))
```

Эта функция представления извлекает информацию об указанном пользователе, проверяет ее допустимость и, если текущий пользователь еще не читает указанного пользователя, вызывает вспомогательный метод `follow()` модели `User` для создания связи. Маршрут `/unfollow/<имя_пользователя>` реализован аналогично.

Маршрут `/followers/<имя_пользователя>` вызывается, когда пользователь щелкает на счетчике читающих в странице профиля другого пользователя. Его реализация приводится в примере 12.6.

Пример 12.6 ❖ `app/main/views.py`: маршрут «followers» и функция представления

```
@main.route('/followers/<username>')
def followers(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('Invalid user.')
        return redirect(url_for('.index'))
    page = request.args.get('page', 1, type=int)
    pagination = user.followers.paginate(
        page, per_page=current_app.config['FLASKY_FOLLOWERS_PER_PAGE'],
        error_out=False)
    follows = [{'user': item.follower, 'timestamp': item.timestamp}
               for item in pagination.items]
```

```
return render_template('followers.html', user=user, title="Followers of",
                       endpoint='.followers', pagination=pagination,
                       follows=follows)
```

Эта функция извлекает информацию об указанном пользователе и проверяет ее допустимость, затем выполняет постраничное отображение списка читающих его пользователей, используя прием, описанный в главе 11. Так как обращение к отношению `followers` возвращает список экземпляров `Follow`, он преобразуется в другой список, содержащий поля `user` и `timestamp`, с целью упростить отображение.

Шаблон, отображающий список читающих, можно сделать достаточно универсальным, чтобы его можно было также использовать для отображения списка читаемых. Шаблон принимает пользователя, заголовок для страницы, конечную точку для использования в ссылках постраничного просмотра, объект `Pagination` и список результатов.

Конечная точка `followed_by` почти идентична. Единственное отличие заключается в том, что список пользователей извлекается из отношения `user.followed` и аргументы шаблона устанавливаются соответственно.

Шаблон *followers.html* реализован как таблица с двумя столбцами, в которой слева отображаются имена пользователей и их аватары, а справа – дата и время. Вы можете исследовать реализацию, загрузив исходный код из репозитория на сайте GitHub.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 12b` и получить эту версию приложения.

Запрос сообщений читаемых пользователей с помощью операции соединения

Главная страница приложения в настоящее время отображает все сообщения, хранящиеся в базе данных, в обратном хронологическом порядке. Теперь, когда реализация поддержки читающих/читаемых закончена, можно дать пользователям возможность просматривать сообщения только читаемых ими пользователей.

Наиболее очевидный способ загрузить все сообщения, написанные читаемыми пользователями, состоит в том, чтобы сначала извлечь список читаемых пользователей, затем получить написанные ими сообщения, объединить их в общий список и отсортировать. Однако

такое решение плохо масштабируется; с ростом объема базы данных придется прилагать все больше усилий для получения такого объединенного списка, и многие операции, такие как постраничный вывод, не смогут выполняться достаточно эффективно. Решить проблему производительности можно, если удастся реализовать извлечение сообщений с помощью единственного запроса.

Сделать это можно с помощью операции, которая называется *соединением* (*join*). Операция соединения принимает две таблицы или более и находит все комбинации строк, удовлетворяющие заданному условию. Получившиеся объединенные строки вставляются во временную таблицу, которая является результатом соединения. Проще и понятнее объяснить действие операции соединения на примере.

В табл. 12.1 представлено содержимое таблицы *users* с тремя пользователями.

Таблица 12.1. Таблица *users*

id	username
1	john
2	susan
3	david

В табл. 12.2 представлено содержимое соответствующей таблицы *posts* с несколькими сообщениями.

Таблица 12.2. Таблица *posts*

id	author_id	body
1	2	Сообщение пользователя <i>susan</i>
2	1	Сообщение пользователя <i>john</i>
3	3	Сообщение пользователя <i>david</i>
4	1	Второе сообщение пользователя <i>john</i>

Наконец, в табл. 12.3 показано, кто кого читает. Здесь можно видеть, что пользователь *john* читает пользователя *david*, пользователь *susan* читает пользователя *john*, а пользователь *david* не читает никого.

Таблица 12.3. Таблица *follows*

follower_id	followed_id
1	3
2	1
2	3

Чтобы получить список сообщений пользователей, которых читает *susan*, необходимо объединить таблицы *posts* и *follows*. Сначала нужно отфильтровать таблицу *follows*, чтобы оставить только строки, где пользователь *susan* выступает в роли читающего. В данном примере это две последние строки. Затем создать временную таблицу со всеми возможными комбинациями строк из таблицы *posts* и отфильтрованной таблицы *follows*, где *author_id* совпадает с *followed_id*, в результате чего получится список сообщений всех пользователей, которых читает *susan*. В табл. 12.4 показан результат операции соединения. Столбцы, по которым выполнялось соединение, отмечены символом ***.

Таблица 12.4. Таблица соединения

id	author_id*	body	follower_id	followed_id*
2	1	Сообщение пользователя john	2	1
3	3	Сообщение пользователя david	2	3
4	1	Второе сообщение пользователя john	2	1

Эта таблица содержит полный список сообщений, написанных пользователями, которых читает *susan*. Фактический запрос для Flask-SQLAlchemy, выполняющий операцию соединения, выглядит достаточно сложно:

```
return db.session.query(Post).select_from(Follow).\
    filter_by(follower_id=self.id).\
    join(Post, Follow.followed_id == Post.author_id)
```

Все запросы, которые вы видели до сих пор, начинаются с обращения к атрибуту *query* запрашиваемой модели. Такой формат не очень хорошо подходит для данного случая, потому что запрос должен вернуть строки сообщений, тогда как первой операцией, которую следует выполнить, является фильтрация таблицы *follows*. По этой причине здесь используется более каноническая форма запроса. Чтобы вы могли понять, как работает этот запрос, разберем его по частям:

- `db.session.query(Post)` определяет, что запрос возвращает объекты *Post*;
- `select_from(Follow)` сообщает, что запрос начинается с модели *Follow*;
- `filter_by(follower_id=self.id)` выполняет фильтрацию таблицы *follows* по пользователю-читателю;
- `join(Post, Follow.followed_id == Post.author_id)` соединяет результаты `filter_by()` с объектами *Post*.

Запрос можно упростить, поменяв порядок операций фильтрации и соединения:

```
return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
    .filter(Follow.follower_id == self.id)
```

Выполняя операцию соединения первой, запрос можно начать с `Post.query`, благодаря чему остается применить только два фильтра: `join()` и `filter()`. Но являются ли эти два решения идентичными? Может показаться, что перемена операций местами ведет к увеличению объема работы, которую придется выполнить, но в действительности это не так. Фреймворк `SQLAlchemy` сначала соберет все фильтры и затем сгенерирует наиболее эффективный запрос. Код SQL для этих двух запросов идентичен. Добавим окончательную версию этого запроса в модель `Post`, как показано в примере 12.7.

Пример 12.7 ❖ `app/models/user.py`: получение сообщений, написанных читаемыми пользователями

```
class User(db.Model):
    # ...
    @property
    def followed_posts(self):
        return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
            .filter(Follow.follower_id == self.id)
```

Обратите внимание, что метод `followed_posts()` определен как свойство (с помощью декоратора `@property`), благодаря чему отпадает необходимость указывать скобки `()` при обращении к нему, и все отношения получают единообразный синтаксис.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта [GitHub](https://github.com), вы можете выполнить команду `git checkout 12c` и получить эту версию приложения.

Соединения являются весьма непростыми операциями; вам может потребоваться поэкспериментировать с примерами в интерактивной оболочке, прежде чем вы поймете все тонкости.

Отображение сообщений читаемых пользователей на главной странице

Теперь главная страница дает пользователям возможность выбирать между отображением всех сообщений в блоге или только написанных читаемыми пользователями. В примере 12.8 показано, как реализован этот выбор.

Пример 12.8 ❖ `app/main/views.py`: выбор между отображением всех сообщений или только принадлежащих читаемым пользователям

```
@app.route('/', methods = ['GET', 'POST'])
def index():
    # ...
    show_followed = False
    if current_user.is_authenticated():
        show_followed = bool(request.cookies.get('show_followed', ''))
    if show_followed:
        query = current_user.followed_posts
    else:
        query = Post.query
    pagination = query.order_by(Post.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
                           show_followed=show_followed, pagination=pagination)
```

Выбор запоминается в cookie с именем `show_followed` – если его значением является непустая строка, отображаться будут только сообщения читаемых пользователей. Значения cookies доступны в объекте запроса в виде словаря `request.cookies`. Строковое значение cookie преобразуется в логическое, и, исходя из результата, локальной переменной `query` присваивается запрос, извлекающий полный или отфильтрованный список сообщений. Для вывода всех сообщений используется запрос `Post.query`, а для вывода отфильтрованных сообщений – недавно созданное свойство `User.followed_posts`. Запрос, хранящийся в локальной переменной `query`, затем разбивается на страницы, и результаты передаются шаблону, как и прежде.

Установка cookie `show_followed` выполняется двумя новыми маршрутами, представленными в примере 12.9.

Пример 12.9 ❖ `app/main/views.py`: выбор отображения всех сообщений или только принадлежащих читаемым пользователям

```
@main.route('/all')
@login_required
def show_all():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '1', max_age=30*24*60*60)
    return resp

@main.route('/followed')
@login_required
def show_followed():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '1', max_age=30*24*60*60)
    return resp
```

Ссылки на эти маршруты добавлены в шаблон главной страницы. Когда они вызываются, `cookie show_followed` получает соответствующее значение, после чего производится переадресация обратно на главную страницу.

Настройки `cookies` можно выполнять лишь в объекте ответа, поэтому эти маршруты должны создавать объекты ответов явно, вызовом `make_response()`, вместо того чтобы доверить эту работу фреймворку Flask.

Функция `set_cookie()` принимает имя `cookie` и значение в первых двух аргументах. В необязательном аргументе `max_age` устанавливается срок хранения `cookie` в секундах. Если опустить этот аргумент, `cookie` будет храниться, пока пользователь не закроет окно браузера. В данном случае устанавливается срок хранения 30 суток, поэтому данная настройка сохранится, даже если пользователь не будет заходить в приложение несколько дней.

Изменения в шаблоне добавляют две вкладки в верхней части страницы, которые вызывают маршруты `/all` и `/followed`. Вы можете исследовать изменения, загрузив исходный код из репозитория на сайте GitHub. На рис. 12.4 показано, как выглядит измененная главная страница.

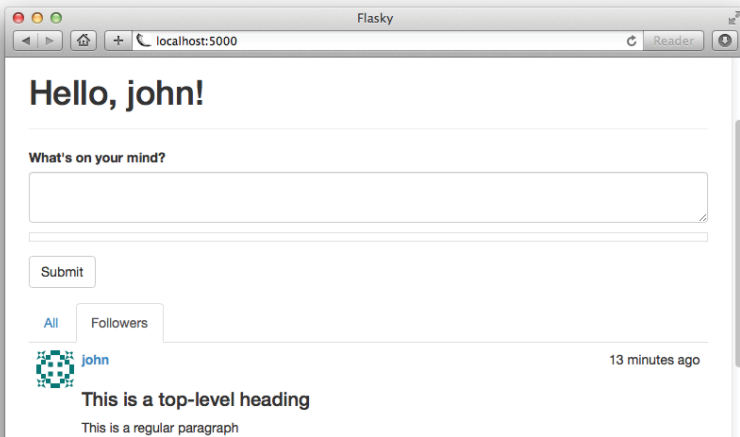


Рис. 12.4 ❖ Сообщения читаемых пользователей на главной странице



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 12a` и получить эту версию приложения.

Если сейчас попробовать открыть приложение и переключиться на отображение сообщений только читаемых пользователей, вы заметите, что ваши собственные сообщения исчезли из списка. В этом нет ничего необычного, потому что пользователи не могут читать себя сами.

Однако, несмотря на то что с логической точки зрения запрос действует правильно, большинство пользователей все же предпочли бы видеть свои собственные сообщения. Решить эту проблему проще всего, регистрируя всех пользователей в момент создания как читающих самих себя. Этот трюк показан в примере 12.10.

Пример 12.10 ❖ `app/models/user.py`: регистрация пользователей как читающих самих себя при создании

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        # ...
        self.follow(self)
```

К сожалению, у вас в базе данных уже может иметься несколько пользователей, которые не были зарегистрированы как читающие сами себя. Если база данных еще достаточно маленькая, можно просто удалить старые учетные записи и создать их снова, но если такой способ не годится, можно добавить функцию, которая выполнит необходимые действия для устранения проблемы. Такая функция представлена в примере 12.11.

Пример 12.11 ❖ `app/models/user.py`: регистрация существующих пользователей как читающих самих себя

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def add_self_follows():
        for user in User.query.all():
            if not user.is_following(user):
                user.follow(user)
                db.session.add(user)
                db.session.commit()
    # ...
```

Теперь можно обновить базу данных, вызвав функцию из предыдущего примера в интерактивной оболочке:


```
(venv) $ python manage.py shell  
>>> User.add_self_follows()
```

Создание функций, которые вносят исправления в базу данных, – распространенный прием, потому что использование таких подготовленных автоматизированных обновлений менее чревато ошибками, чем выполнение операций с базой данных вручную. В главе 17 вы увидите, как эту и другие подобные ей функции можно встроить в сценарий развертывания приложения.

Регистрация пользователей как читающих самих себя делает приложение более удобным, но при этом возникает несколько сложностей. Счетчики читающих и читаемых пользователей на странице профиля теперь оказываются увеличенными на единицу из-за присутствия ссылки на самого себя. Их необходимо уменьшить на единицу для большей точности, что легко сделать непосредственно в шаблоне, отображая `{{ user.followers.count() - 1 }}` и `{{ user.followed.count() - 1 }}`. Списки читающих и читаемых пользователей также нужно скорректировать, чтобы исключить из них самого себя, что тоже легко сделать прямо в шаблоне с помощью условной директивы. Наконец, любые модульные тесты, проверяющие счетчики читающих/читаемых, также следует изменить, чтобы учесть влияние ссылки на самого себя.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 12e` и получить эту версию приложения.

В следующей главе мы займемся реализацией подсистемы комментариев – еще одной очень важной функции приложений социальных сетей.

Глава 13

Комментарии пользователей

Поддержка взаимодействий между пользователями является ключом к успеху платформы социального блогинга. В этой главе вы узнаете, как дать пользователям возможность оставлять комментарии. Приемы, представленные здесь, достаточно универсальны, чтобы их с успехом можно было применять в самых разных социальных приложениях.

Представление комментариев в базе данных

Комментарии не особенно отличаются от сообщений. И те, и другие имеют тело, автора и время создания, и в данной конкретной реализации допускают применение разметки Markdown. На рис. 13.1 показаны организация таблицы `comments` и ее отношения с другими таблицами в базе данных.

Комментарии пишутся к определенным сообщениям в блоге, соответственно, таблица `posts` связана с таблицей `comments` отношением «один ко многим». Это отношение можно использовать для получения списка комментариев, связанных с определенным сообщением.

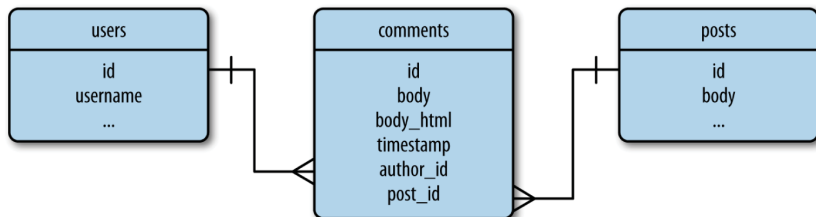


Рис. 13.1 ❖ Представление комментариев в базе данных

Таблица `comments` также связана отношением «один ко многим» с таблицей `users`. Это отношение дает доступ ко всем комментариям, написанным определенным пользователем, и косвенно позволяет получить число комментариев, принадлежащих пользователю, которое можно вывести на странице профиля. В примере 13.1 приводится определение модели `Comment`.

Пример 13.1 ❖ `app/models.py`: модель `Comment`

```
class Comment(db.Model):
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    body_html = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    disabled = db.Column(db.Boolean)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))
    post_id = db.Column(db.Integer, db.ForeignKey('posts.id'))

    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'code', 'em', 'i',
                        'strong']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Comment.body, 'set', Comment.on_changed_body)
```

Модель `Comment` имеет почти тот же набор атрибутов, что и модель `Post`. Единственное дополнение — логическое поле `disabled`, которое будет использоваться модераторами для подавления комментариев, носящих оскорбительный характер или не связанных с темой сообщения. Как и для сообщений, для комментариев тоже определяется обработчик события, которое генерируется при изменении поля `body`, чтобы автоматизировать отображение разметки `Markdown` в разметку `HTML`. Процесс преобразования идентичен тому, что был показан в главе 11, но, так как комментарии обычно имеют небольшую длину, список допустимых тегов `HTML` также более ограничен — из него были удалены теги, имеющие отношение к оформлению абзацев, и оставлены только теги, отвечающие за оформление символов.

Наконец, необходимо добавить в модели `User` и `Post` определение отношений «один ко многим» с таблицей `comments`, как показано в примере 13.2.

Пример 13.2 ❖ app/models/user.py: отношения «один ко многим» таблиц users и posts с таблицей comments

```
class User(db.Model):
    # ...
    comments = db.relationship('Comment', backref='author', lazy='dynamic')

class Post(db.Model):
    # ...
    comments = db.relationship('Comment', backref='post', lazy='dynamic')
```

Отправка и отображение комментариев

В этом приложении комментарии выводятся на отдельных страницах сообщений, которые были добавлены в виде постоянных ссылок в главе 11. В шаблон страницы включена и форма создания комментария. В примере 13.3 представлена веб-форма для ввода комментария – очень простая форма, состоящая из текстового поля ввода и кнопки отправки.

Пример 13.3 ❖ app/main/forms.py: форма ввода комментария

```
class CommentForm(Form):
    body = StringField('', validators=[Required()])
    submit = SubmitField('Submit')
```

В примере 13.4 показан измененный маршрут /post/<int:id> с поддержкой комментариев.

Пример 13.4 ❖ app/main/views.py: поддержка комментариев в страницах отдельных сообщений

```
@main.route('/post/<int:id>', methods=['GET', 'POST'])
def post(id):
    post = Post.query.get_or_404(id)
    form = CommentForm()
    if form.validate_on_submit():
        comment = Comment(body=form.body.data,
                          post=post,
                          author=current_user._get_current_object())
        db.session.add(comment)
        flash('Your comment has been published.')
        return redirect(url_for('.post', id=post.id, page=-1))
    page = request.args.get('page', 1, type=int)
    if page == -1:
        page = (post.comments.count() - 1) // \
            current_app.config['FLASKY_COMMENTS_PER_PAGE'] + 1
    pagination = post.comments.order_by(Comment.timestamp.asc()).paginate(
        page, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],
```

```

        error_out=False)
    comments = pagination.items
    return render_template('post.html', posts=[post], form=form,
                           comments=comments, pagination=pagination)

```

Эта функция представления создает экземпляр формы и передает ее в шаблон *post.html* для отображения. Логика вставки нового комментария после отправки формы напоминает логику обработки сообщений. Как и в случае с сообщениями, в поле `author` экземпляра комментария нельзя записать значение переменной `current_user` непосредственно, потому что эта переменная контекста фактически является промежуточным прокси-объектом. Фактический объект `User` можно получить с помощью выражения `current_user._get_current_object()`.

Комментарии сортируются в хронологическом порядке, поэтому новые комментарии всегда добавляются в конец списка. После ввода нового комментария выполняется операция переадресации на тот же адрес URL, только функции `url_for()` в параметре `page` передается значение `-1`, специальное число, соответствующее последней странице комментариев. Сделано это для того, чтобы последний введенный комментарий тут же отображался на экране. Когда номер страницы, извлеченный из строки запроса, оказывается равным `-1`, определяются общее число комментариев и размер страницы, после чего вычисляется фактический номер последней страницы.

Список комментариев, связанных с сообщением, извлекается с помощью отношения «один ко многим» `post.comments`, сортируется по времени создания и разбивается на страницы с использованием тех же приемов, что применялись для отображения списков сообщений. Комментарии и объект `Pagination` передаются шаблону для отображения. В *config.py* добавлена переменная `FLASKY_COMMENTS_PER_PAGE`, определяющая размер одной страницы с комментариями.

Отображение каждого комментария осуществляется с помощью нового шаблона *_comments.html*, похожего на шаблон *_posts.html*, но использующего иной набор классов CSS. Этот шаблон подключается в шаблоне *_post.html* под телом сообщения и сопровождается вызовом макроса постраничного вывода. Изменения в шаблонах можно увидеть, если загрузить исходные тексты приложения из репозитория на сайте GitHub.

Чтобы закончить реализацию этой особенности, в сообщениях, что отображаются на главной странице и на странице профиля, необходимо добавить ссылки на страницы с комментариями. Как это сделать, показано в примере 13.5.

Пример 13.5 ❖ `_app/templates/_posts.html`: ссылка на страницу с комментариями к сообщению

```
<a href="{{ url_for('.post', id=post.id) }}#comments">
  <span class="label label-primary">
    {{ post.comments.count() }} Comments
  </span>
</a>
```

Обратите внимание, что текст ссылки включает число комментариев, которое легко получить из отношения «один ко многим» между таблицами `posts` и `comments` с помощью фильтра `count()`.

Интересно также отметить структуру ссылки на страницу с комментариями, которая повторяет структуру постоянной ссылки на страницу с сообщением, но имеет дополнительное окончание `#comments`. Это окончание называется *фрагментом URL (URL fragment)* и используется с целью обеспечить прокрутку страницы до нужной позиции. Веб-браузер находит элемент с атрибутом `id`, имеющим значение, совпадающее с фрагментом URL, и прокручивает страницу так, чтобы элемент оказался вверху окна. В данном случае начальная позиция устанавливается в заголовке списка комментариев `<h4 id="comments">Comments</h4>`. На рис. 13.2 показано, как выглядят комментарии на странице.

Дополнительные изменения были внесены в макрос постраничного отображения. В ссылки на страницы с комментариями также был добавлен фрагмент `#comments`, и в макрос была добавлена поддержка аргумента `fragment`, который теперь передается в вызов макроса из шаблона `post.html`.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 13a` и получить данную версию приложения. Это обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория.

Модерирование комментариев

В главе 9 было определено несколько ролей пользователей, каждая из которых имеет свой набор привилегий. Одна из привилегий — `Permission.MODERATE_COMMENTS` — дает пользователям право модерировать комментарии, написанные другими пользователями.

Доступ к этой функции можно организовать в виде ссылки в строке навигации, доступной только пользователям, обладающим правом

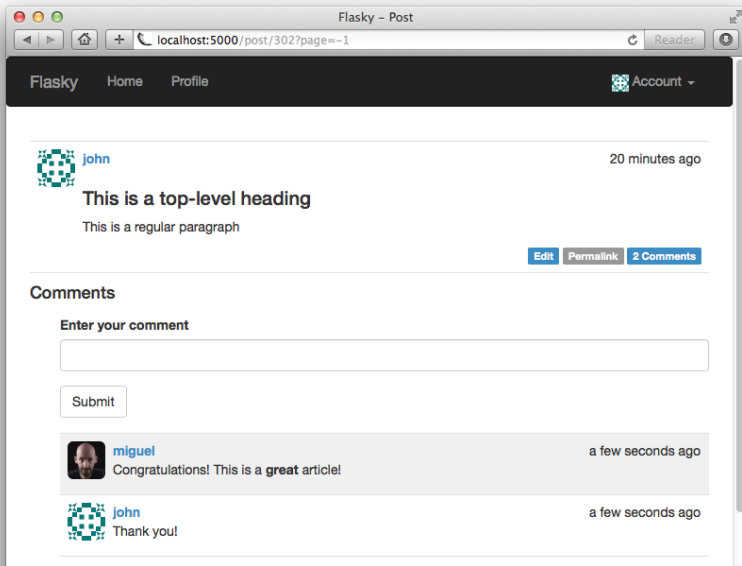


Рис. 13.2 ❖ Комментарии к сообщению в блоге

использовать ее. Реализация ссылки находится в шаблоне *base.html* и использует условную директиву, как показано в примере 13.6.

Пример 13.6 ❖ *app/templates/base.html*: ссылка в строке навигации для перехода к модерированию комментариев

```
...
{% if current_user.can(Permission.MODERATE_COMMENTS) %}
<li><a href="{{ url_for('main.moderate') }}">Moderate Comments</a></li>
{% endif %}
...
```

Страница модерирования отображает комментарии ко всем сообщениям в одном общем списке, причем самые свежие комментарии отображаются первыми. Под каждым комментарием имеется кнопка, управляющая атрибутом *disabled*. Маршрут */moderate* показан в примере 13.7.

Пример 13.7 ❖ `app/main/views.py`: маршрут для модерирования комментариев

```
@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def moderate():
    page = request.args.get('page', 1, type=int)
    pagination = Comment.query.order_by(Comment.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],
        error_out=False)
    comments = pagination.items
    return render_template('moderate.html', comments=comments,
                           pagination=pagination, page=page)
```

Это очень простая функция. Она извлекает комментарии из базы данных и передает их в шаблон для отображения. Вместе с комментариями шаблон получает объект постраничного отображения и номер текущей страницы.

Шаблон *moderate.html*, представленный в примере 13.8, также довольно прост. Для отображения комментариев он использует шаблон *_comments.html*, созданный ранее.

Пример 13.8 ❖ `app/templates/moderate.html`: шаблон страницы модерирования комментариев

```
{% extends "base.html" %}
{% import "_macros.html" as macros %}

{% block title %}Flasky - Comment Moderation{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Comment Moderation</h1>
</div>
{% set moderate = True %}
{% include '_comments.html' %}
{% if pagination %}
<div class="pagination">
    {{ macros.pagination_widget(pagination, '.moderate') }}
</div>
{% endif %}
{% endblock %}
```

Этот шаблон возлагает отображение комментариев на шаблон *_comments.html*, но, прежде чем передать ему управление, он устанавливает переменную `moderate` в значение `True` с помощью директивы `set`. На основе этой переменной шаблон *_comments.html* определяет, следует ли включать отображение инструментов модерирования.

В часть шаблона `_comments.html`, отвечающую за отображение тела каждого комментария, необходимо внести два изменения. Для обычных пользователей (когда переменная `moderate` не установлена) комментарии, отмеченные как запрещенные, не должны выводиться. Для модераторов (когда переменная `moderate` установлена в значение `True`) комментарии должны выводиться независимо от состояния атрибута `disabled`, а под комментарием должна отображаться кнопка, позволяющая переключать его состояние. Необходимые изменения приводятся в примере 13.9.

Пример 13.9 ❖ `app/templates/_comments.html`: отображение комментариев

```
...
<div class="comment-body">
    {% if comment.disabled %}
    <p></p><i>This comment has been disabled by a moderator.</i></p>
    {% endif %}
    {% if moderate or not comment.disabled %}
        {% if comment.body_html %}
            {{ comment.body_html | safe }}
        {% else %}
            {{ comment.body }}
        {% endif %}
    {% endif %}
</div>
{% if moderate %}
    <br>
    {% if comment.disabled %}
    <a class="btn btn-default btn-xs" href="{{ url_for('.moderate_enable',
        id=comment.id, page=page) }}">Enable</a>
    {% else %}
    <a class="btn btn-danger btn-xs" href="{{ url_for('.moderate_disable',
        id=comment.id, page=page) }}">Disable</a>
    {% endif %}
{% endif %}
...
```

После внесения этих изменений вместо запрещенных комментариев пользователи будут видеть короткие примечания. Модераторы будут видеть и примечания, и текст запрещенных комментариев. Для модераторов под каждым комментарием также будет отображаться кнопка, переключающая состояние комментария. Кнопка вызывает один из двух новых маршрутов в зависимости от состояния комментария. Определения этих маршрутов приводятся в примере 13.10.

Пример 13.10 ❖ `app/main/views.py`: маршруты переключения состояния комментария

```

@main.route('/moderate/enable/<int:id>')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def moderate_enable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = False
    db.session.add(comment)
    return redirect(url_for('.moderate',
                             page=request.args.get('page', 1, type=int)))

@main.route('/moderate/disable/<int:id>')
@login_required
@permission_required(Permission.MODERATE_COMMENTS)
def moderate_disable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = True
    db.session.add(comment)
    return redirect(url_for('.moderate',
                             page=request.args.get('page', 1, type=int)))

```

Оба маршрута извлекают соответствующий объект комментария из базы данных, записывают в поле `disabled` соответствующее значение и записывают объект обратно в базу. В конце осуществляется переадресация на страницу модерирования комментариев (см. рис. 13.3). Если при этом в строке запроса был задан аргумент `page`, он включается в ответ переадресации. Кнопки в шаблоне `_comments.html` включают аргумент `page`, поэтому переадресация выполняется на ту же страницу, где находился модератор перед щелчком на кнопке.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 13b` и получить эту версию приложения.

Эта глава завершает обзор социальных особенностей приложения. В следующей главе вы узнаете, как экспортировать функциональность приложения в виде API, чтобы им могли пользоваться клиенты, отличные от веб-браузеров.

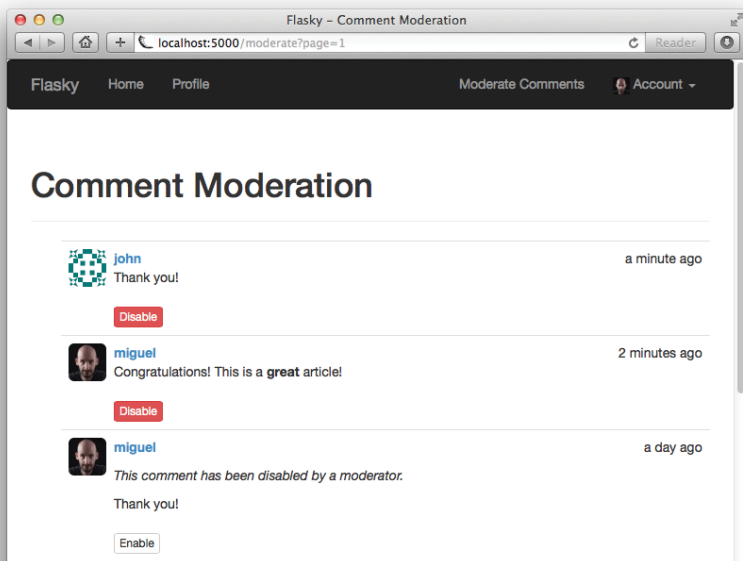


Рис. 13.3 ❖ Страница модерирования комментариев

Глава 14

Прикладные программные интерфейсы

В последние годы наблюдается тенденция переноса все большей и большей доли прикладной логики на сторону клиента в соответствии с архитектурой полнофункциональных интернет-приложений (Rich Internet Application, RIA). Согласно RIA, основной (а иногда и единственной) функцией сервера является передача клиентскому приложению данных, извлекаемых из хранилища. В этой модели сервер превращается в *веб-службу*, или *прикладной программный интерфейс* (*Application Programming Interface, API*).

Существует несколько протоколов, с помощью которых полнофункциональные интернет-приложения могут взаимодействовать с веб-службами. Протоколы вызова удаленных процедур (Remote Procedure Call, RPC), такой как XML-RPC и производный от него упрощенный протокол доступа к объектам (Simplified Object Access Protocol, SOAP), были популярны несколько лет тому назад. Позднее появилась архитектура передачи репрезентативного состояния (Representational State Transfer, REST), ставшая более предпочтительной для веб-приложений из-за того, что основана на знакомой модели World Wide Web.

Фреймворк Flask идеально подходит для создания веб-служб RESTful благодаря своей легковесной природе. В этой главе вы узнаете, как на основе Flask реализовать RESTful API.

Введение в REST

В своей диссертации¹ доктор Рой Филдинг (Roy Fielding) ввел понятие архитектурного стиля REST для веб-служб и перечислил шесть определяющих характеристик:

¹ <http://bit.ly/REST-fielding>.

- клиент-сервер – между клиентом и сервером должно быть ясное разделение;
- отсутствие поддержки состояния – запрос, отправляемый клиентом, должен содержать всю информацию, необходимую для его обработки. Сервер не должен хранить какую-либо информацию о состоянии клиента между запросами;
- кэширование – ответы, возвращаемые сервером, могут отмечаться как кэшируемые или некаэшируемые, чтобы клиенты (или промежуточные программные компоненты между клиентом и сервером) могли использовать кэширование для оптимизации;
- однородный интерфейс – протокол, посредством которого клиенты будут обращаться к ресурсам сервера, должен быть хорошо определен и стандартизован. Часто в роли такого протокола используется протокол HTTP;
- многоуровневая система – по мере необходимости, для повышения производительности, надежности и масштабируемости, между клиентами и серверами могут добавляться прокси-серверы, кэши или шлюзы;
- код по требованию – клиенты могут загружать с сервера дополнительный код для выполнения в своих контекстах.

Все сущее является ресурсами


Понятие *ресурс* является основополагающим в архитектурном стиле REST. В этом контексте под ресурсом подразумевается элемент предметной области приложения. Например, пользователи, сообщения и комментарии – все это в приложении блогинга является ресурсами.

Каждый ресурс должен иметь уникальный URL, представляющий его. Так, в приложении блогинга сообщение может быть представлено адресом URL `/api/posts/12345`, где `12345` – уникальный идентификатор сообщения, такой как значение первичного ключа в базе данных. Формат или содержимое URL не имеет большого значения; значение имеет лишь уникальная идентификация каждого ресурса.

Коллекция всех ресурсов одного класса также должна идентифицироваться адресом URL. Например, URL коллекции сообщений мог бы иметь вид `/api/posts/`, а URL коллекции всех комментариев мог бы выглядеть как `/api/comments/`.

API может также определять коллекции адресов URL, представляющих логические подмножества ресурсов одного класса. Например, коллекцию всех комментариев к сообщению `12345` мог бы

представлять адрес URL `/api/posts/12345/comments/`. Определение адресов URL, представляющих коллекции ресурсов, которые завершаются символом косой черты, является распространенной практикой, так как это придает им вид «папки».

 Помните, что Flask по-особому интерпретирует маршруты, оканчивающиеся символом косой черты. Если клиент обратится по адресу URL без завершающего символа косой черты и единственным соответствующим маршрутом на сервере будет маршрут с косой чертой в конце, тогда Flask автоматически выполнит переадресацию на этот URL. В обратном случае переадресация не выполняется.

Методы запросов

Посылая серверу запрос, с помощью URL клиентское приложение определяет требуемый ресурс, а с помощью метода – желаемую операцию. Чтобы получить список доступных сообщений в блоге, клиент должен послать GET-запрос по адресу `http://www.example.com/api/posts/`, а чтобы добавить новое сообщение – послать POST-запрос по тому же адресу с содержимым сообщения в теле запроса. Чтобы получить сообщение 12345, клиент должен послать GET-запрос по адресу `http://www.example.com/api/posts/12345`. В табл. 14.1 перечислены методы запросов и их назначение, которые часто используются в RESTful API.

Таблица 14.1. HTTP-методы запросов в RESTful API

Метод запроса	Цель	Описание	Код состояния HTTP
GET	Отдельный ресурс	Получение ресурса	200
GET	Коллекция ресурсов	Получение коллекции ресурсов (или одной страницы, если сервер реализует постраничный доступ)	200
POST	Коллекция ресурсов	Создание нового ресурса и добавление его в коллекцию. Сервер определяет URL для вновь созданного ресурса и возвращает его в заголовке <code>Location</code>	201
PUT	Отдельный ресурс	Изменение существующего ресурса. При желании этот метод можно также использовать для создания новых ресурсов, когда клиенту разрешено определять URL вновь созданного ресурса	200
DELETE	Отдельный ресурс	Удаление ресурса	200
DELETE	Коллекция ресурсов	Удаление всех ресурсов в коллекции	200



Архитектура REST не требует реализации всех методов для выполнения операций с ресурсами. Если клиент задействует метод, не поддерживаемый для данного ресурса, он получит в ответ код состояния 405 «Method Not Allowed» (метод не поддерживается). Фреймворк Flask обработает эту ошибку автоматически.

Содержимое запросов и ответов

Ресурсы передаются между клиентом и сервером в теле запроса или ответа, но REST не определяет формат представления ресурсов. Для передачи информации о формате представления ресурса в запросе или ответе используется заголовок Content-Type. Чтобы договориться о формате, поддерживаемом обеими сторонами, клиентом и сервером, нередко используется стандартный механизм сообщения предпочтений в протоколе HTTP.

Наиболее часто для взаимодействий с веб-службами RESTful используются два формата: формат записи объектов JavaScript (JavaScript Object Notation, JSON) и расширяемый язык разметки (Extensible Markup Language, XML). Для полнофункциональных веб-приложений наиболее привлекательным выглядит формат JSON, потому что он тесно связан с JavaScript, языком клиентских сценариев в веб-браузерах. В нашем примере реализации API для приложения блоггинга сообщение в формате JSON можно было бы представить, как показано ниже:

```
{
  "url": "http://www.example.com/api/posts/12345",
  "title": "Writing RESTful APIs in Python",
  "author": "http://www.example.com/api/users/2",
  "body": "... здесь находится текст статьи ...",
  "comments": "http://www.example.com/api/posts/12345/comments"
}
```

Обратите внимание, что поля url, author и comments в сообщении выше представлены полными адресами URL соответствующих ресурсов. Это важно, потому что с помощью этих URL клиент сможет получить новые ресурсы.

В хорошо спроектированных RESTful API клиенту достаточно знать лишь несколько URL ресурсов верхнего уровня и получать все остальное из ссылок, включаемых в ответы, подобно тому, как человек может открывать все новые страницы, щелкая по ссылкам, имеющимся на уже известных ему страницах.

Поддержка версий

В традиционных сервер-центричных веб-приложениях сервер имеет полный контроль над приложением. Когда разработчики вносят какие-либо изменения, установки новой версии достаточно, чтобы обновить приложение у всех пользователей, потому что даже та часть приложения, которая выполняется в веб-браузерах пользователей, все равно загружается с сервера.

В ситуации с полнофункциональными веб-приложениями и веб-службами дела обстоят несколько сложнее, потому что часто клиенты разрабатываются независимо от серверной части, возможно даже другими людьми. Представьте ситуацию, когда веб-служба RESTful используется различными клиентами, включая веб-браузеры и приложения для смартфонов. Если сценарий, выполняемый в веб-браузере, можно обновить на сервере в любой момент, то приложение в смартфоне нельзя обновить принудительно – владелец смартфона должен, как минимум, разрешить такое обновление. Кроме того, даже если владелец смартфона сам пожелает обновить приложение, невозможно обеспечить появление обновленного приложения для смартфона во всех магазинах и центрах распространения приложений одновременно с развертыванием новой версии на сервере.

По этим причинам веб-службы должны быть более толерантными, чем обычные веб-приложения, и поддерживать старые версии своих клиентов. Нередко эта проблема решается путем включения номера версии в адреса URL веб-службы. Например, первая версия веб-службы блогинга могла бы обеспечивать доступ к списку сообщений по адресу `/api/v1.0/posts/`.

Включение номера версии в URL веб-службы поможет представлять новые возможности новым клиентам и продолжать поддерживать старых клиентов. Обновление службы блогинга могло бы привести к изменению представления сообщений в формате JSON. В этом случае веб-служба могла бы экспортировать сообщения в новом формате по адресу `/api/v1.1/posts/`, сохранив доступ к сообщениям в более старом формате JSON по адресу `/api/v1.0/posts/`. С этого момента сервер будет обслуживать все свои URL для версий `v1.1` и `v1.0`.

Хотя поддержка множества версий увеличивает сложность сопровождения, иногда это единственный способ обеспечить дальнейшее развитие приложения, не вызывая проблем на стороне клиентов.

Веб-службы RESTful на основе Flask

Фреймворк Flask существенно упрощает создание веб-служб RESTful. Для объявления маршрутов, экспортируемых веб-службой, можно использовать уже знакомый декоратор `route()` с его необязательным аргументом `methods`. Работа с данными в формате JSON также не представляет сложностей, поскольку такие данные, получаемые вместе с запросом, автоматически становятся доступны в виде словаря `request.json`, а чтобы преобразовать словарь с данными для ответа в формат JSON, достаточно лишь воспользоваться вспомогательной функцией `jsonify()`.

В следующих разделах будет показано, как можно добавить в приложение Flasky веб-службу RESTful, которая даст клиентам доступ к сообщениям в блоге и другим сопутствующим ресурсам.

Создание макета API

Маршруты, связанные с RESTful API, образуют самостоятельное подмножество, поэтому поместим их в собственный макет, чтобы логически отделить их реализацию от остальной части приложения. Общая структура макета API внутри приложения представлена в примере 14.1.

Пример 14.1 ❖ Структура макета API

```
| -flasky
|   | -app/
|   |   | -api_1_0
|   |   | | -__init__.py
|   |   | | -user.py
|   |   | | -post.py
|   |   | | -comment.py
|   |   | | -authentication.py
|   |   | | -errors.py
|   |   | | -decorators.py
```

Обратите внимание, что в имя пакета, используемого для организации API, включен номер версии. Когда потребуется ввести новую версию API, его поддержку можно будет реализовать в виде другого пакета, с другим номером версии, и оба API будут поддерживаться одновременно.

Данный макет API реализует каждый ресурс в виде отдельного модуля. В него также включены модули, решающие задачи аутентификации, обработки ошибок и реализующие дополнительные декораторы. В примере 14.2 приводится конструктор макета.

Пример 14.2 ❖ `app/api_1_0/__init__.py`: конструктор макета API

```
from flask import Blueprint
```

```
api = Blueprint('api', __name__)
```

```
from . import authentication, posts, users, comments, errors
```

Регистрация макета API показана в примере 14.3.

Пример 14.3 ❖ `app/_init_.py`: регистрация макета API

```
def create_app(config_name):
```

```
    # ...
```

```
    from .api_1_0 import api as api_1_0_blueprint
```

```
    app.register_blueprint(api_1_0_blueprint, url_prefix='/api/v1.0')
```

```
    # ...
```

Обработка ошибок

Веб-служба RESTful информирует клиента о результатах обработки запроса, отправляя ответ с соответствующим HTTP-кодом состояния и любыми дополнительными данными в теле ответа. Коды состояния, которые клиенты обычно ожидают получить от веб-службы, перечислены в табл. 14.2.

Таблица 14.2. HTTP-коды состояния, обычно возвращаемые веб-службами

HTTP-код состояния	Имя	Описание
200	OK	Запрос выполнен успешно
201	Created	Запрос выполнен успешно, и новый ресурс благополучно создан
400	Bad request	Ошибочный или противоречивый запрос
401	Unauthorized	Запрос не содержит информацию об аутентификации
403	Forbidden	Информации об аутентификации, переданной с запросом, недостаточно для выполнения запроса
404	Not found	Ресурс, указанный в URL, не найден
405	Method not allowed	Метод запроса не поддерживается для указанного ресурса
500	Internal server error	В процессе обработки запроса возникла неожиданная ошибка

Обработка кодов 404 и 500 представляет небольшую проблему, в том смысле что обработчики этих ошибок в фреймворке Flask воз-

вращают с этими кодами HTML-ответы, которые могут неправильно интерпретироваться клиентом.

Один из способов обеспечить создание ответов, подходящих для любых клиентов, – сделать обработчики ошибок адаптируемыми под формат, запрошенный клиентом. Этот прием называется *сообщением предпочтений*. В примере 14.4 показан усовершенствованный обработчик ошибки 404, возвращающий ответ в формате JSON клиентам веб-службы и разметку HTML всем остальным. Обработчик ошибки 500 реализован аналогично.

Пример 14.4 ❖ `app/main/errors.py`: обработчики ошибок, учитывающие предпочтения клиентов

```
@main.app_errorhandler(404)
def page_not_found(e):
    if request.accept_mimetypes.accept_json and \
        not request.accept_mimetypes.accept_html:
        response = jsonify({'error': 'not found'})
        response.status_code = 404
        return response
    return render_template('404.html'), 404
```

Новая версия обработчика проверяет заголовок `Accept` запроса, который фреймворк Werkzeug декодирует в объект `request.accept_mimetypes`, и определяет, в каком формате клиент хотел бы получить ответ. Обычно браузеры не накладывают ограничений на формат ответа, поэтому данные в формате JSON возвращаются только клиентам, готовым принимать данные в формате JSON и не готовым принимать данных в формате HTML.

Остальные коды состояний генерируются веб-службой явно, поэтому их можно реализовать в виде вспомогательных функций внутри макета, в модуле *errors.py*. В примере 14.5 показана реализация ошибки 403 `error`; остальные реализованы аналогичным образом.

Пример 14.5 ❖ `app/api/errors.py`: обработчик кода состояния 403 для API веб-службы

```
def forbidden(message):
    response = jsonify({'error': 'forbidden', 'message': message})
    response.status_code = 403
    return response
```


Теперь функции представления в веб-службе могут вызывать эти вспомогательные функции для возврата сообщений об ошибках.

Аутентификация пользователей с помощью Flask-HTTPAuth

Веб-службы, подобно обычным веб-приложениям, должны заботиться о защите информации и гарантировать невозможность неавторизованного доступа. Поэтому полнофункциональные интернет-приложения должны запрашивать у пользователя его имя и пароль и передавать их на сервер для проверки.

Как отмечалось выше, одной из особенностей веб-служб RESTful является отсутствие информации о состоянии, а это означает, что сервер не хранит никакой информации о клиенте между запросами. Клиенты обязаны поставлять всю информацию, необходимую для выполнения запроса, в самом запросе, поэтому все запросы должны включать информацию для аутентификации пользователя.

Текущая поддержка аутентификации и авторизации на основе расширения Flask-Login хранит данные в пользовательском сеансе, реализованном в виде cookie – небольшого блока данных, – по умолчанию хранящемся на стороне клиента. То есть на стороне сервера не хранится никакой информации о пользователе – вся она находится на стороне клиента. Казалось бы, такая организация полностью соответствует требованиям архитектурного стиля REST, но это не так, потому что использование cookies в веб-службах RESTful может осложнить реализацию клиентов, отличных от веб-браузеров. По этой причине решение на основе cookie обычно рассматривается как не самое удачное.

 Требование REST не хранить информацию о состоянии на стороне сервера может показаться излишне строгим, но оно не является чьей-то прихотью. Серверы, не хранящие информацию о состоянии, легко поддаются масштабированию. Когда распределенное приложение, выполняющееся на нескольких серверах, хранит информацию о клиентах, оно вынуждено использовать общий кэш, доступный всем серверам, или гарантировать, что на протяжении всего сеанса запросы от одного и того же клиента будут обрабатываться одним и тем же сервером. Оба этих решения весьма сложны в реализации.

Так как архитектура RESTful основана на применении протокола HTTP, *HTTP-аутентификация* оказывается наиболее предпочтительным способом передачи учетных данных. При использовании механизма HTTP-аутентификации учетные данные передаются в заголовке Authorization.

Механизм HTTP-аутентификации достаточно прост, чтобы использовать его непосредственно, однако расширение Flask-HTTPAuth предоставляет еще более удобное решение, скрывающее тонкости взаимодействия с протоколом за декоратором, похожим на декоратор `login_required` из расширения Flask-Login.

Установить Flask-HTTPAuth можно с помощью утилиты pip:

```
(venv) $ pip install flask-httpauth
```

Чтобы инициализировать расширение для использования HTTP-аутентификации в режиме Basic, нужно создать объект класса HTTPBasicAuth. Как и Flask-Login, расширение Flask-HTTPAuth не делает никаких предположений о процедуре проверки учетных данных, поэтому эти данные передаются функции обратного вызова. В примере 14.6 показано, как выполняется инициализация расширения, и приводится функция проверки учетных данных.

Пример 14.6 ❖ app/api_1_0/authentication.py: инициализация Flask-HTTPAuth

```
from flask.ext.httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@auth.verify_password
def verify_password(email, password):
    if email == '':
        g.current_user = AnonymousUser()
        return True
    user = User.query.filter_by(email = email).first()
    if not user:
        return False
    g.current_user = user
    return user.verify_password(password)
```

Так как этот тип аутентификации будет использоваться только в рамках макета API, расширение Flask-HTTPAuth инициализируется в пакете этого макета, а не в пакете приложения, как другие расширения.

Проверка адреса электронной почты и пароля осуществляется с помощью имеющейся поддержки в модели User. Функция проверки возвращает True, проверка увенчалась успехом, и False – в противном случае. Поддерживается также идентификация анонимного пользователя, для чего клиент должен отправить пустой адрес электронной почты.

Функция проверки сохраняет информацию об аутентифицированном пользователе в глобальном объекте g, чтобы позднее функция представления могла получить к ней доступ. Обратите внимание, что для анонимных пользователей функция проверки возвращает True и сохраняет в g.current_user экземпляр класса AnonymousUser.



Поскольку учетные данные пользователя поступают в каждом запросе, очень важно обеспечить доступность всех маршрутов API только при использовании безопасной версии протокола HTTP, чтобы все запросы и ответы передавались по сети в зашифрованном виде.

В случае ошибки аутентификации сервер возвращает клиенту ошибку 401. Расширение Flask-HTTPAuth способно автоматически генерировать ответ с этим кодом ошибки, но, чтобы обеспечить единообразие с другими сообщениями об ошибках, возвращаемыми веб-службой, создание ответа можно реализовать, как показано в примере 14.7.

Пример 14.7 ❖ `_app/api_1_0/authentication.py`: обработчик ошибок для Flask-HTTPAuth

```
@auth.error_handler
def auth_error():
    return unauthorized('Invalid credentials')
```

Для защиты маршрутов можно использовать декоратор `auth.login_required`:

```
@api.route('/posts/')
@auth.login_required
def get_posts():
    pass
```

Но, поскольку в такой защите нуждаются все маршруты в макете, декоратор `login_required` можно применить только к обработчику `before_request`, как показано в примере 14.8.

Пример 14.8 ❖ `app/api_1_0/authentication.py`: обработчик `before_request` с аутентификацией

```
from .errors import forbidden_error

@api.before_request
@auth.login_required
def before_request():
    if not g.current_user.is_anonymous and \
        not g.current_user.confirmed:
        return forbidden('Unconfirmed account')
```

Теперь аутентификация будет выполняться автоматически для всех маршрутов в макете. В качестве дополнительной меры безопасности обработчик `before_request` отвергает запросы от аутентифицированных пользователей, не подтвердивших регистрацию.

Аутентификация на основе маркеров

Клиенты должны отправлять учетные данные для аутентификации с каждым запросом. Чтобы избежать необходимости постоянно передавать уязвимую информацию, можно воспользоваться решением аутентификации на основе маркеров.

В этом случае клиент передает учетные данные на специальный адрес URL, генерирующий маркеры аутентификации. Получив такой маркер, клиент может использовать его для аутентификации запросов взамен учетных данных. По соображениям безопасности, маркеры имеют ограниченный срок действия. Когда срок действия маркера заканчивается, клиент должен повторно выполнить процедуру аутентификации и получить новый маркер. Риск попадания маркера в руки злоумышленника можно ослабить, уменьшая период его действия. В примере 14.9 показаны два новых метода в модели User, реализующих создание и проверку маркеров с помощью itsdangerous.

Пример 14.9 ❖ app/models.py: поддержка аутентификации на основе маркеров

```
class User(db.Model):
    # ...
    def generate_auth_token(self, expiration):
        s = Serializer(current_app.config['SECRET_KEY'],
                       expires_in=expiration)
        return s.dumps({'id': self.id})

    @staticmethod
    def verify_auth_token(token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return None
        return User.query.get(data['id'])
```

Метод `generate_auth_token()` возвращает подписанный маркер, в который зашифровано значение поля `id` пользователя. Срок хранения `expiration` указывается в секундах. Метод `verify_auth_token()` принимает маркер и, если его проверка увенчалась успехом, возвращает соответствующий объект User. Это статический метод, так как пользователь будет известен только после расшифровывания маркера.

Чтобы обеспечить аутентификацию запросов с маркерами, необходимо изменить функцию `verify_password` так, чтобы она принимала не только обычные учетные данные, но и маркер. Измененная версия функции приводится в примере 14.10.

Пример 14.10 ❖ app/api_1_0/authentication.py: усовершенствованная проверка аутентификации с поддержкой маркеров

```
@auth.verify_password
def verify_password(email_or_token, password):
    if email_or_token == '':
```

```

        g.current_user = AnonymousUser()
        return True
    if password == '':
        g.current_user = User.verify_auth_token(email_or_token)
        g.token_used = True
        return g.current_user is not None
    user = User.query.filter_by(email=email_or_token).first()
    if not user:
        return False
    g.current_user = user
    g.token_used = False
    return user.verify_password(password)

```

В новой версии первый аргумент может быть адресом электронной почты или маркером аутентификации. Если этот аргумент содержит пустую строку, предполагается, что запрос послал анонимный пользователь, как и прежде. Если аргумент `password` содержит пустую строку, предполагается, что аргумент `email_or_token` содержит маркер и проверяется в таком качестве. Если оба аргумента содержат непустые строки, предполагается, что производится попытка выполнить обычную процедуру аутентификации. В этой реализации аутентификация на основе маркера не является обязательной – клиент может выбирать, как ему поступить. Чтобы дать функциям представления возможность отличать два способа аутентификации, добавлена переменная `g.token_used`.

Маршрут, возвращающий маркер клиенту, также добавлен в макет API. Его реализация представлена в примере 14.11.

Пример 14.11 ❖ `app/api_1_0/authentication.py`: генератор маркеров аутентификации

```

@api.route('/token')
def get_token():
    if g.current_user.is_anonymous() or g.token_used:
        return unauthorized('Invalid credentials')
    return jsonify({
        'token': g.current_user.generate_auth_token(expiration=3600),
        'expiration': 3600})

```

Так как этот маршрут определен в макете, механизмы аутентификации, добавленные в обработчик `before_request`, также будут применяться к нему. Чтобы воспрепятствовать использованию старого маркера для получения нового, проверяется переменная `g.token_used`, и попытка пройти аутентификацию с использованием маркера отвергается. Функция возвращает маркер в формате JSON и со сроком действия 1 час. Срок действия также включается в ответ.

Преобразование ресурсов в формат JSON и обратно

При создании веб-служб часто бывает необходимо преобразовывать ресурсы из внутреннего представления в формат JSON и обратно для их передачи по протоколу HTTP в запросах и ответах. В примере 14.12 представлен новый метод `to_json()`, добавленный в класс `Post`.

Пример 14.12 ❖ `app/models.py`: преобразование сообщения в словарь для последующей сериализации в формат JSON

```
class Post(db.Model):
    # ...
    def to_json(self):
        json_post = {
            'url': url_for('api.get_post', id=self.id, _external=True),
            'body': self.body,
            'body_html': self.body_html,
            'timestamp': self.timestamp,
            'author': url_for('api.get_user', id=self.author_id,
                              _external=True),
            'comments': url_for('api.get_post_comments', id=self.id,
                               _external=True)
            'comment_count': self.comments.count()
        }
        return json_post
```

В полях `url`, `author` и `comments` необходимо вернуть адреса URL соответствующих ресурсов, поэтому эти значения генерируются с помощью вызовов функции `url_for()`, которой передаются маршруты, определенные в макете API. Обратите внимание, что во все вызовы `url_for()` добавлен аргумент `_external=True`, чтобы вместо относительных URL, которые обычно используются в контексте традиционного веб-приложения, получить абсолютные URL.

Этот пример также демонстрирует, как можно вернуть «искусственные» атрибуты ресурса. В поле `comment_count` возвращается число комментариев к сообщению в блоге. Хотя этот атрибут не является настоящим атрибутом модели, он включается в представление ресурса для удобства.

Метод `to_json()` для модели `User` можно сконструировать по аналогии с методом `to_json()` в модели `Post`. Этот метод показан в примере 14.13.

Пример 14.13 ❖ `app/models.py`: преобразование информации о пользователе в словарь для последующей сериализации в формат JSON

```
class User(UserMixin, db.Model):
    # ...
```

```

def to_json(self):
    json_user = {
        'url': url_for('api.get_post', id=self.id, _external=True),
        'username': self.username,
        'member_since': self.member_since,
        'last_seen': self.last_seen,
        'posts': url_for('api.get_user_posts', id=self.id, _external=True),
        'followed_posts': url_for('api.get_user_followed_posts',
                                   id=self.id, _external=True),
        'post_count': self.posts.count()
    }
    return json_user

```

Обратите внимание, что по соображениям безопасности в этом методе из ответа исключаются некоторые атрибуты, такие как email и role. Этот пример еще раз демонстрирует, что представление ресурса, предлагаемое клиенту, необязательно должно быть идентично его внутреннему представлению в базе данных.

Обратное преобразование структуры JSON в экземпляр модели представляет некоторые сложности, так как некоторые данные, полученные от клиента, могут быть недопустимыми, ошибочными или ненужными. В примере 14.14 представлен метод, создающий экземпляр модели Post из формата JSON.

Пример 14.14 ❖ app/models.py: создание сообщения в блоге на основе данных в формате JSON

```

from app.exceptions import ValidationError

class Post(db.Model):
    # ...
    @staticmethod
    def from_json(json_post):
        body = json_post.get('body')
        if body is None or body == '':
            raise ValidationError('post does not have a body')
        return Post(body=body)

```

Как видите, эта реализация использует из словаря JSON только атрибут body. Атрибут body_html игнорируется, потому что серверный механизм отображения разметки Markdown автоматически будет вызван событием SQLAlchemy при попытке изменить атрибут body. Атрибут timestamp также не требуется устанавливать, если только клиенту не позволено произвольно устанавливать дату и время для сообщений, что не поддерживается данным приложением. Поле author не используется, потому что клиент не имеет полномочий определять авторство сообщения; единственное возможное значение

для поля `author` – это текущий аутентифицированный пользователь. Атрибуты `comments` и `comment_count` автоматически генерируются из отношения в базе данных, поэтому они не содержат никакой полезной информации, которая могла бы пригодиться при создании экземпляра модели. Наконец, поле `url` игнорируется, потому что в данной реализации адрес URL ресурса определяется сервером, а не клиентом.

Обратите внимание, как выполняется проверка на наличие ошибок. Если поле `body` отсутствует или пустое, возбуждается исключение `ValidationError`. Исключение в данном случае является наиболее подходящей реакцией на ошибку, потому что этот метод не обладает полной информацией, опираясь на которую, можно было бы обработать эту ситуацию. Исключение передаст ошибку вызывающему коду на более высоком уровне, который может более успешно обработать ее. Класс `ValidationError` является простым наследником класса `ValueError`. Его реализация приводится в примере 14.15.

Пример 14.15 ❖ `app/exceptions.py`: исключение `ValidationError`

```
class ValidationError(ValueError):
    pass
```

Теперь приложению нужно обработать это исключение и вернуть соответствующий ответ клиенту. Чтобы избежать необходимости добавлять в функции представления код, который перехватывал бы исключение, можно установить глобальный обработчик исключений. Обработчик для исключения `ValidationError` показан в примере 14.16.

Пример 14.16 ❖ `app/api_1_0/errors.py`: обработчик исключения `ValidationError` в веб-службе

```
@api.errorhandler(ValidationError)
def validation_error(e):
    return bad_request(e.args[0])
```

Здесь используется тот же декоратор `errorhandler`, что применялся для регистрации обработчиков HTTP-кодов состояния, но в данном случае обработчик принимает экземпляр класса `Exception`. Декорированная функция будет вызываться всякий раз, когда будет обнаруживаться исключение указанного класса. Обратите внимание, что декоратор находится в макете API, поэтому данный обработчик будет вызываться только для обработки исключений, возникших при обработке маршрутов, объявленных в макете.

Благодаря этому приему код функций представления получается чище и короче, без захламления его проверками на наличие ошибок. Например:

```
@api.route('/posts/', methods=['POST'])
def new_post():
    post = Post.from_json(request.json)
    post.author = g.current_user
    db.session.add(post)
    db.session.commit()
    return jsonify(post.to_json())
```

Реализация конечных точек ресурсов


Все, что нам осталось, – реализовать маршруты для обработки разных ресурсов. Запросы GET обычно проще в обработке, потому что в ответ достаточно всего лишь вернуть информацию, не внося никаких изменений. В примере 14.17 приводятся два обработчика запросов GET на получение сообщений.

Пример 14.17 ❖ app/api_1_0/posts.py: обработчики GET-запросов на получение сообщений

```
@api.route('/posts/')
@auth.login_required
def get_posts():
    posts = Post.query.all()
    return jsonify({ 'posts': [post.to_json() for post in posts] })

@api.route('/posts/<int:id>')
@auth.login_required
def get_post(id):
    post = Post.query.get_or_404(id)
    return jsonify(post.to_json())
```

Первый маршрут обрабатывает запрос на получение коллекции сообщений. Эта функция генерирует JSON-версию коллекции сообщений с помощью генератора списков (list comprehension). Вторым маршрутом возвращается единственное сообщение и отвечает кодом 404, если сообщение с указанным значением id отсутствует в базе данных.

 Обработчик ошибки с кодом 404 находится на уровне приложения, но он позволяет передавать ответы в формате JSON, если клиент поддерживает его. Если в веб-службе потребуется изменить ответ, можно переопределить обработчик ошибки 404 в макете.

Обработчик запросов POST к ресурсу сообщения добавляет новое сообщение в базу данных. Реализация этого маршрута приводится в примере 14.18.

Пример 14.18 ❖ `app/api_1_0/posts.py`: обработчики POST-запросов, добавляющих новые сообщения

```
@api.route('/posts/', methods=['POST'])
@permission_required(Permission.WRITE_ARTICLES)
def new_post():
    post = Post.from_json(request.json)
    post.author = g.current_user
    db.session.add(post)
    db.session.commit()
    return jsonify(post.to_json()), 201, \
        {'Location': url_for('api.get_post', id=post.id, _external=True)}
```

Эта функция представления обернута декоратором `permission_required` (см. пример 14.19 ниже), гарантирующим доступность данной функции только для аутентифицированных пользователей, имеющих право писать сообщения в блог. Создание самого сообщения выглядит достаточно просто благодаря поддержке обработки ошибок, реализованной выше. Сообщение создается из данных в формате JSON, а автором явно назначается текущий пользователь. После записи модели в базу данных клиенту возвращается ответ с кодом состояния 201 и адресом URL только что созданного ресурса в заголовке `Location`.

Обратите внимание, что для удобства клиента в тело ответа включается и новый ресурс. Это позволит клиенту сэкономить на запросе GET для получения ресурса сразу после его создания.

Декоратор `permission_required`, используемый для предотвращения неавторизованного доступа к функции создания новых сообщений, похож на аналогичный декоратор, используемый в приложении, и адаптирован для нужд макета API. Его реализация приводится в примере 14.19.

Пример 14.19 ❖ `app/api_1_0/decorators.py`: декоратор `permission_required`

```
def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not g.current_user.can(permission):
                return forbidden('Insufficient permissions')
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

Обработчик запросов PUT к существующим ресурсам сообщений используется для их редактирования. Его реализация приводится в примере 14.20.

Пример 14.20 ❖ `app/api_1_0/posts.py`: обработчик PUT-запросов для редактирования сообщений

```
@api.route('/posts/<int:id>', methods=['PUT'])
@permission_required(Permission.WRITE_ARTICLES)
def edit_post(id):
    post = Post.query.get_or_404(id)
    if g.current_user != post.author and \
        not g.current_user.can(Permission.ADMINISTER):
        return forbidden('Insufficient permissions')
    post.body = request.json.get('body', post.body)
    db.session.add(post)
    return jsonify(post.to_json())
```

Проверка привилегий в данном случае выглядит сложнее. Стандартная проверка наличия привилегии писать сообщения выполняется декоратором, но чтобы позволить пользователю изменить сообщение, функция должна также убедиться, что пользователь является автором сообщения или администратором. Эта проверка добавлена в функцию представления. Если такая проверка понадобится в нескольких функциях, тогда лучше будет определить декоратор, чтобы избежать дублирования кода.

Поскольку приложение не позволяет удалять сообщения, обработчик запросов DELETE не требуется.

Обработчики запросов для выполнения операций с пользователями и комментариями реализованы аналогично. В табл. 14.3 перечислены ресурсы, реализованные в этой веб-службе. Исходные тексты реализаций можно получить из репозитория на сайте GitHub.

Таблица 14.3. Ресурсы, поддерживаемые Flasky API

URL ресурса	Методы	Описание
<code>/users/<int:id></code>	GET	Пользователь
<code>/users/<int:id>/posts/</code>	GET	Сообщения, написанные пользователем
<code>/users/<int:id>/timeline/</code>	GET	Сообщения, читаемые пользователем
<code>/posts/</code>	GET, POST	Все сообщения
<code>/posts/<int:id></code>	GET, PUT	Сообщение
<code>/posts/<int:id>/comments/</code>	GET, POST	Комментарии к сообщению
<code>/comments/</code>	GET	Все комментарии
<code>/comments/<int:id></code>	GET	Комментарий

Обратите внимание, что реализованные ресурсы открывают пользователю доступ лишь к подмножеству функциональных возможностей, поддерживаемых веб-приложением. Список поддерживаемых ресурсов можно было бы расширить: дать возможность получать списки читающих и читаемых, модерировать комментарии и реализовать любые другие функции, необходимые клиенту.

Разбивка больших коллекций ресурсов на страницы

Запросы GET, возвращающие очень большие коллекции ресурсов, могут оказаться весьма дорогостоящими и сложными в управлении. Подобно веб-приложениям, веб-службы также могут возвращать коллекции с разбивкой на страницы.

В примере 14.21 представлена возможная реализация разбивки списка сообщений на страницы.

Пример 14.21 ❖ app/api_1_0/posts.py: разбивки списка сообщений на страницы

```
@api.route('/posts/')
def get_posts():
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    prev = None
    if pagination.has_prev:
        prev = url_for('api.get_posts', page=page-1, _external=True)
    next = None
    if pagination.has_next:
        next = url_for('api.get_posts', page=page+1, _external=True)
    return jsonify({
        'posts': [post.to_json() for post in posts],
        'prev': prev,
        'next': next,
        'count': pagination.total
    })
```

Поле `posts` в ответе JSON содержит элементы данных, как и прежде, но теперь это лишь часть всего набора. Поля `prev` и `next` содержат URL ресурсов, соответствующих предыдущей и следующей страницам, если доступны. Поле `count` содержит общее число элементов в коллекции.

Этот прием можно применить к любым маршрутам, возвращающим коллекции.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 14a` и получить эту версию приложения. Чтобы гарантировать установку всех необходимых расширений, выполните также команду `pip install -r requirements/dev.txt`.

Тестирование веб-служб с помощью HTTPie

Чтобы протестировать веб-службу, необходим HTTP-клиент. Наиболее часто для тестирования веб-служб из командной строки используются клиенты *curl* и *HTTPie*. Последний из них имеет более краткие и удобочитаемые команды. Установить HTTPie можно с помощью утилиты `pip`:

```
(venv) $ pip install httpie
```

Ниже показано, как можно выполнить GET-запрос:

```
(venv) $ http --json --auth <email>:<password> GET \
> http://127.0.0.1:5000/api/v1.0/posts
HTTP/1.0 200 OK
Content-Length: 7018
Content-Type: application/json
Date: Sun, 22 Dec 2013 08:11:24 GMT
Server: Werkzeug/0.9.4 Python/2.7.3

{
  "posts": [
    ...
  ],
  "prev": null
  "next": "http://127.0.0.1:5000/api/v1.0/posts/?page=2",
  "count": 150
}
```

Обратите внимание на включенные в ответ ссылки на соседние страницы. Поскольку это первая страница, ссылка на предыдущую страницу не определена, но URL следующей страницы присутствует и указано общее число сообщений.

Тот же запрос можно выполнить от лица анонимного пользователя, отправив пустые строки вместо адреса электронной почты и пароля:

```
(venv) $ http --json --auth : GET http://127.0.0.1:5000/api/v1.0/posts/
```

Следующая команда выполняет запрос POST, чтобы добавить новое сообщение:

```
(venv) $ http --auth <email>:<password> --json POST \
> http://127.0.0.1:5000/api/v1.0/posts/ \
> "body=I'm adding a post from the *command line*."
```



```

HTTP/1.0 201 CREATED
Content-Length: 360
Content-Type: application/json
Date: Sun, 22 Dec 2013 08:30:27 GMT
Location: http://127.0.0.1:5000/api/v1.0/posts/111
Server: Werkzeug/0.9.4 Python/2.7.3

{
  "author": "http://127.0.0.1:5000/api/v1.0/users/1",
  "body": "I'm adding a post from the *command line*",
  "body_html": "<p>I'm adding a post from the <em>command line</em>.</p>",
  "comments": "http://127.0.0.1:5000/api/v1.0/posts/111/comments",
  "comment_count": 0,
  "timestamp": "Sun, 22 Dec 2013 08:30:27 GMT",
  "url": "http://127.0.0.1:5000/api/v1.0/posts/111"
}

```

Для проверки аутентификации с помощью маркера отправьте следующий запрос:

```

(venv) $ http --auth <email>:<password> --json GET \
> http://127.0.0.1:5000/api/v1.0/token
HTTP/1.0 200 OK
Content-Length: 162
Content-Type: application/json
Date: Sat, 04 Jan 2014 08:38:47 GMT
Server: Werkzeug/0.9.4 Python/3.3.3

{
  "expiration": 3600,
  "token": "eyJpYXQiOjEzODg4MjQ3Mjc5ImV4cCI6MTM4ODgyODMyNywiYWxnIjoiaSFMy..."
}

```

Полученный маркер можно использовать для обращений к API в течение ближайшего часа, передавая его вместе с пустым паролем:

```

(venv) $ http --json --auth eyJpYXQ...: GET http://127.0.0.1:5000/api/v1.0/posts/

```

Когда срок действия маркера истечет, в ответ на такие запросы будет возвращаться код ошибки 401, указывающий на необходимость получения нового маркера.

Примите поздравления! Данная глава завершает вторую часть книги, и на этом разработку приложения Flasky можно считать законченной. Следующий шаг – развертывание приложения, и он влечет за собой новые проблемы, обсуждением которых мы займемся в третьей части.

Часть III



Последняя миля

Глава 15

Тестирование

Существуют две веские причины писать модульные тесты. Когда реализуется новая функциональность, модульные тесты помогут подтвердить, что новый код действует именно так, как задумывалось. Тот же результат можно получить и при тестировании вручную, но совершенно очевидно, что автоматизированные тесты помогут сэкономить время и силы.

Вторая и более веская причина: каждый раз, когда в приложение вносятся изменения, комплект модульных тестов, созданных к этому моменту, поможет убедиться в отсутствии регрессов в существующем коде, то есть что новый код не оказывает влияния на работу ранее написанного кода.

Мы начали писать модульные тесты для приложения Flasky с самого начала. Первые наши тесты предназначались для проверки некоторых его особенностей, реализованных в классах моделей базы данных. Эти классы легко поддаются тестированию вне контекста действующего приложения, поэтому вам не потребуется прилагать много усилий на реализацию модульных тестов для проверки всех особенностей моделей базы данных, а в обмен вы получите уверенность, что хотя бы эта часть приложения работает надежно.

В этой главе мы обсудим способы расширения и улучшения комплекта модульных тестов.

Получение отчета о степени охвата кода тестированием

Иметь комплект тестов важно, но не менее важно знать, насколько он хорош или плох. Инструменты измерения охвата кода тестами позволяют узнать, какая доля приложения покрыта модульными тестами, и получить подробный отчет, позволяющий выяснить, какие части приложения еще не охвачены тестированием. Эта информация бесценна, потому что помогает направить усилия в нужное русло – в создание тестов для участков кода, еще не охваченных тестами.

В Python имеется отличный инструмент измерения степени охвата кода тестированием – пакет, который так и называется: *coverage*. Установить его можно с помощью утилиты *pip*:

```
(venv) $ pip install coverage
```

В состав пакета входит сценарий командной строки, который может запускать любые приложения на Python с включенной поддержкой измерения охвата, но этот пакет предоставляет также более удобный программный доступ к своим внутренним механизмам. Чтобы интегрировать измерение степени охвата в сценарий запуска *manage.py*, можно дополнить команду *test*, реализованную в главе 7, необязательным параметром *--coverage*. Реализация этого параметра приводится в примере 15.1.

Пример 15.1 ❖ manage.py: измерение степени охвата

```
#!/usr/bin/env python
import os
COV = None
if os.environ.get('FLASK_COVERAGE'):
    import coverage
    COV = coverage.coverage(branch=True, include='app/*')
    COV.start()

# ...

@manager.command
def test(coverage=False):
    """Выполняет модульное тестирование."""
    if coverage and not os.environ.get('FLASK_COVERAGE'):
        import sys
        os.environ['FLASK_COVERAGE'] = '1'
        os.execvp(sys.executable, [sys.executable] + sys.argv)
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
    if COV:
        COV.stop()
        COV.save()
        print('Coverage Summary:')
        COV.report()
        basedir = os.path.abspath(os.path.dirname(__file__))
        covdir = os.path.join(basedir, 'tmp/coverage')
        COV.html_report(directory=covdir)
        print('HTML version: file://%s/index.html' % covdir)
        COV.erase()

# ...
```

Расширение Flask-Script значительно упрощает определение новых команд. Чтобы добавить логический параметр в команду `test`, достаточно добавить логический аргумент в функцию `test()`. Расширение Flask-Script автоматически выведет имя параметра командной строки из имени аргумента и передаст функции `True` или `False` в зависимости от наличия или отсутствия этого параметра.

Но интеграция измерения охвата с помощью *manage.py* представляет небольшую проблему. К моменту, когда параметр `--coverage` попадет в функцию `test()`, уже слишком поздно запускать механизм измерения — к этому времени весь код в глобальной области видимости уже будет выполнен. Чтобы получить точные результаты, сценарий перезапускает самого себя после установки переменной окружения `FLASK_COVERAGE`. На втором прогоне код в начале сценария обнаруживает, что переменная окружения установлена, и запускает измерение охвата с самого начала.

Запуск механизма осуществляется вызовом функции `coverage.coverage()`. Параметр `branch=True` включает анализ охвата ветвей условных инструкций, в ходе которого, помимо слежения за выполняемыми строками, проверяется также, все ли ветви условной инструкции были опробованы. Параметр `include` указывает, что анализировать следует только файлы, находящиеся внутри пакета приложения, так как лишь этот код представляет интерес. Без параметра `include` в отчет будут включены все расширения, установленные в виртуальное окружение, и все тесты, что усложнит поиск полезной информации.

После того как все тесты будут выполнены, функция `text()` выведет отчет в консоль, а также сохранит в отформатированном виде в HTML-файле. Версия в формате HTML прекрасно подходит для визуального изучения, потому что в ней строки кода выделяются цветом в зависимости от частоты их использования.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 15a` и получить эту версию приложения. Чтобы гарантировать установку всех необходимых расширений, выполните также команду `pip install -r requirements/dev.txt`.

Ниже приводится пример отчета в простом текстовом формате:

```
(venv) $ python manage.py test --coverage
```

```
...
```

```
-----
Ran 19 tests in 50.609s
```

```
OK
```

Coverage Summary:

Name	Stmts	Miss	Branch	BrMiss	Cover	Missing
...						

app/__init__	33	0	0	0	100%	
app/api_1_0/__init__	3	0	0	0	100%	
app/api_1_0/authentication	30	19	11	11	27%	
app/api_1_0/comments	40	30	12	12	19%	
app/api_1_0/decorators	11	3	2	2	62%	
app/api_1_0/errors	17	10	0	0	41%	
app/api_1_0/posts	35	23	9	9	27%	
app/api_1_0/users	30	24	12	12	14%	
app/auth/__init__	3	0	0	0	100%	
app/auth/forms	45	8	8	8	70%	
app/auth/views	109	84	41	41	17%	
app/decorators	14	3	2	2	69%	
app/email	15	9	0	0	40%	
app/exceptions	2	0	0	0	100%	
app/main/__init__	6	1	0	0	83%	
app/main/errors	20	15	9	9	17%	
app/main/forms	39	7	8	8	68%	
app/main/views	169	131	36	36	19%	
app/models	243	62	44	17	72%	

TOTAL	864	429	194	167	44%	
HTML version: file:///home/flasky/tmp/coverage/index.html						

Отчет показывает, что общий охват составляет 44%, что не так уж и плохо, но и не особенно хорошо. Классы моделей, которым уделялось основное внимание при разработке тестов, содержат 243 инструкции, из которых 72% охвачено тестами. Очевидно, что файлы *views.py* в макетах main и auth, а также маршруты в макете api_1_0 имеют очень низкий показатель охвата тестированием, так как для них не имеется ни одного модульного теста.

Получив эту информацию, легко определить, какие тесты следует добавить, чтобы повысить степень охвата, но, к сожалению, не все части приложения поддаются тестированию так же легко, как модели базы данных. В следующих двух разделах мы обсудим более совершенные стратегии тестирования, которые можно применить к функциям представления, формам и шаблонам.

Обратите внимание, что содержимое столбца *Missing* (Пропущено) в примере отчета было опущено для улучшения форматирования. Этот столбец показывает, какие строки исходного кода были пропущены при тестировании, и содержит длинные списки диапазонов строк.

Тестовый клиент Flask

Некоторые участки приложения сильно зависят от окружения, создаваемого выполняющимся приложением. Например, нельзя просто вызвать функцию представления, чтобы протестировать ее, поскольку этой функции может потребоваться обратиться к глобальным переменным контекста Flask, таким как `request` или `session`, чтобы получить данные из POST-запроса, а некоторые функции представления требуют, чтобы они вызывались только аутентифицированным пользователем. Проще говоря, функции представления могут работать лишь в контексте запроса и приложения.

В состав фреймворка Flask входит специальный тестовый клиент, помогающий хотя бы частично решить описанные выше проблемы. Тестовый клиент воссоздает окружение, имитирующее работающее приложение внутри веб-сервера, позволяя тестам выступать в качестве клиентов, отправляющих запросы.

Функции представления не видят существенных отличий, выполняясь под управлением тестового клиента; запросы принимаются и направляются соответствующим функциям представления, которые генерируют и возвращают ответы. После выполнения функции представления ее ответ передается тесту, где он может быть проверен на наличие ошибок.

Тестирование веб-приложений

В примере 15.2 показана заготовка модульного теста, использующего тестового клиента.

Пример 15.2 ❖ `tests/test_client.py`: заготовка модульного теста, использующего тестового клиента

```
import unittest
from app import create_app, db
from app.models import User, Role

class FlaskClientTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()
        Role.insert_roles()
        self.client = self.app.test_client(use_cookies=True)

    def tearDown(self):
```

```
db.session.remove()
db.drop_all()
self.app_context.pop()

def test_home_page(self):
    response = self.client.get(url_for('main.index'))
    self.assertTrue('Stranger' in response.get_data(as_text=True))
```

Экземпляр `self.client`, создаваемый на этапе настройки теста, – это объект тестового клиента. Данный объект имеет методы для выполнения запросов к приложению. Если тестовый клиент создан с включенным параметром `use_cookies`, он будет принимать и отправлять cookies подобно браузеру, поэтому имеется возможность тестировать функциональность, опирающуюся на использование cookies для сохранения контекста между запросами. В частности, такой подход дает возможность пользоваться сессиями и избежать необходимости постоянно аутентифицировать пользователя.

Тест `test_home_page()` представляет собой простую демонстрацию возможностей тестового клиента. Он отправляет запрос на получение главной страницы приложения. Возвращаемым значением метода `get()` клиента является объект `Response`, содержащий ответ, созданный функцией представления. Чтобы убедиться, что тест выполнен успешно, в теле ответа, возвращаемом вызовом `response.get_data()`, производится поиск слова "Stranger", который является частью приветствия «Hello, Stranger!», отображаемого для анонимных пользователей. Обратите внимание, что по умолчанию `get_data()` возвращает тело ответа в виде массива байтов; если передать этому методу параметр `as_text=True`, он вернет строку Юникода, которая проще в обращении.

С помощью метода `post()` тестового клиента можно также отправлять POST-запросы, включающие данные формы, однако отправка форм имеет свои сложности. Формы, генерируемые с помощью расширения `Flask-WTF`, имеют скрытое поле с маркером `CSRF`, которое должно отправляться вместе с формой. Чтобы воспроизвести эту функциональность, тест должен запросить страницу, включающую форму, проанализировать полученную разметку HTML, извлечь из нее маркер и отправить его вместе с данными формы. Чтобы избежать сложностей с извлечением маркера `CSRF`, лучше вообще запретить защиту форм этим маркером в тестовом окружении. Как это сделать, показано в примере 15.3.

Пример 15.3 ❖ config.py: отключение защиты маркером CSRF в тестовом окружении

```
class TestingConfig(Config):
    #...
    WTF_CSRF_ENABLED = False
```

В примере 15.4 представлен более сложный модульный тест, имитирующий регистрацию новой учетной записи, аутентификацию, подтверждение регистрации с помощью маркера и выход из приложения.

Пример 15.4 ❖ tests/test_client.py: имитация создания и использования новой учетной записи

```
class FlaskClientTestCase(unittest.TestCase):
    # ...
    def test_register_and_login(self):
        # регистрация новой учетной записи
        response = self.client.post(url_for('auth.register'), data={
            'email': 'john@example.com',
            'username': 'john',
            'password': 'cat',
            'password2': 'cat'
        })
        self.assertTrue(response.status_code == 302)

        # аутентификация с новой учетной записью
        response = self.client.post(url_for('auth.login'), data={
            'email': 'john@example.com',
            'password': 'cat'
        }, follow_redirects=True)
        data = response.get_data(as_text=True)
        self.assertTrue(re.search('Hello,\s+john!', data))
        self.assertTrue('You have not confirmed your account yet' in data)

        # отправка маркера подтверждения
        user = User.query.filter_by(email='john@example.com').first()
        token = user.generate_confirmation_token()
        response = self.client.get(url_for('auth.confirm', token=token),
                                   follow_redirects=True)
        data = response.get_data(as_text=True)
        self.assertTrue('You have confirmed your account' in data)

        # выход
        response = self.client.get(url_for('auth.logout'),
                                   follow_redirects=True)
        data = response.get_data(as_text=True)
        self.assertTrue('You have been logged out' in data)
```

Тест начинается с отправки формы регистрации. Аргумент `data` метода `post()` — это словарь с полями формы, имена элементов в котором должны в точности совпадать с именами полей в форме. Так как теперь в тестовом окружении защита маркером CSRF отключена, поле CSRF можно исключить из формы.

Маршрут `/auth/register` может вернуть два ответа. Если форма содержит допустимые данные, выполняется переадресация на страницу аутентификации. Если попытка регистрации потерпела неудачу, в ответе возвращается все та же форма регистрации, включающая соответствующие сообщения об ошибках. Чтобы убедиться, что регистрация прошла успешно, тест сравнивает код состояния ответа с числом 302, которое соответствует коду операции переадресации.

Во втором разделе теста выполняется попытка пройти аутентификацию с только что зарегистрированным адресом электронной почты и паролем. Для этого по адресу `/auth/login` отправляется POST-запрос. На этот раз в вызов `post()` передается аргумент `follow_redirects=True`, чтобы клиент мог действовать подобно браузеру и автоматически выполнить запрос GET в ответ на переадресацию. В этом случае метод вернет не код состояния 302, а страницу, на которую произошла переадресация.

В случае успешной аутентификации клиенту будет возвращена страница с приветствием, содержащим имя пользователя, и напоминанием о необходимости подтвердить создание учетной записи. Для проверки этой страницы используются две инструкции `assert`. Здесь следует отметить, что простой поиск строки `'Hello, john!'` не даст желаемого результата, потому что эта строка состоит из статической и динамической частей, с дополнительными пробелами между ними. Чтобы избежать проблем, которые могут вызвать лишние пробелы при сравнении, используется более гибкое решение, основанное на регулярных выражениях.

Следующий шаг — подтверждение регистрации, что также сопряжено с некоторыми сложностями. Адрес URL для подтверждения отправляется пользователю на электронную почту, соответственно, внутри теста его нельзя получить автоматически. Решение, представленное в примере, обходит эту сложность, генерируя маркер прямым обращением к экземпляру `User`. Другое возможное решение состоит в том, чтобы извлечь маркер из электронного письма, которое Flask-Mail сохраняет на диске при выполнении в тестовом окружении.

После получения маркера в третьей части теста имитируется щелчок пользователя на ссылке в письме. Делается это путем отправки

GET-запроса по адресу страницы подтверждения с присоединенным к нему маркером. В ответ на этот запрос выполняется переадресация на главную страницу, и снова в вызове метода указан аргумент `follow_redirects=True`, благодаря чему клиент автоматически обработает операцию переадресации. Полученный в результате ответ проверяется на наличие приветствия и сообщения, информирующего об успешном подтверждении.

Последний шаг, выполняемый этим тестом, – отправка GET-запроса маршруту выхода. Чтобы убедиться, что выход состоялся, тест ищет соответствующее сообщение в ответе.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 15b` и получить эту версию приложения.

Тестирование веб-служб

Тестовый клиент из фреймворка Flask может также использоваться и для тестирования веб-служб RESTful. В примере 15.5 приводится реализация двух тестов.

Пример 15.5 ❖ tests/test_api.py: тестирование RESTful API с помощью тестового клиента

```
class APITestCase(unittest.TestCase):
    # ...
    def get_api_headers(self, username, password):
        return {
            'Authorization':
                'Basic ' + b64encode(
                    (username+':'+password).encode('utf-8')).decode('utf-8'),
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        }

    def test_no_auth(self):
        response = self.client.get(url_for('api.get_posts'),
                                    content_type='application/json')
        self.assertTrue(response.status_code == 401)

    def test_posts(self):
        # добавить пользователя
        r = Role.query.filter_by(name='User').first()
        self.assertIsNotNone(r)
        u = User(email='john@example.com', password='cat', confirmed=True,
                  role=r)
        db.session.add(u)
```

```

db.session.commit()

# записать сообщение
response = self.client.post(
    url_for('api.new_post'),
    headers=self.get_auth_header('john@example.com', 'cat'),
    data=json.dumps({'body': 'body of the blog post'}))
self.assertTrue(response.status_code == 201)
url = response.headers.get('Location')
self.assertIsNotNone(url)

# получить новое сообщение обратно
response = self.client.get(
    url,
    headers=self.get_auth_header('john@example.com', 'cat'))
self.assertTrue(response.status_code == 200)
json_response = json.loads(response.data.decode('utf-8'))
self.assertTrue(json_response['url'] == url)
self.assertTrue(json_response['body'] == 'body of the *blog* post')
self.assertTrue(json_response['body_html'] ==
    '<p>body of the <em>blog</em> post</p>')

```

В модульном тесте, предназначенном для тестирования веб-службы, используются те же методы `setUp()` и `tearDown()`, что и в модульном тесте для обычного приложения, была убрана только поддержка cookies, потому что веб-служба не использует ее. Метод `get_api_headers()` — это вспомогательный метод, возвращающий общие заголовки, которые необходимы во всех запросах. В их число входят заголовок, необходимый для аутентификации, и заголовки, определяющие MIME-типы. Эти заголовки потребуются в большинстве запросов.

Тест `test_no_auth()` просто проверяет, отвергаются ли запросы, не содержащие учетных данных, с кодом ошибки 401. Тест `test_posts()` добавляет пользователя в базу данных, затем добавляет новое сообщение обращением к веб-службе RESTful и читает его обратно. Все данные, отправляемые в теле запроса, должны кодироваться с помощью `json.dumps()`, потому что тестовый клиент не осуществляет автоматического преобразования в формат JSON. Аналогично данные, возвращаемые в ответах в формате JSON, должны декодироваться вызовом `json.loads()` перед их исследованием.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 15c` и получить эту версию приложения.

Сквозное тестирование с помощью Selenium

Тестовый клиент в фреймворке Flask не способен полностью имитировать окружение, создаваемое работающим приложением. Например, любое приложение, опирающееся на сценарии JavaScript, которые должны выполняться на стороне клиента, нельзя проверить полностью, потому что код JavaScript, возвращаемый в ответах, нельзя выполнить с помощью тестового клиента и убедиться в его работоспособности.

Когда требуется протестировать окружение полностью, нет иного выбора, как использовать настоящий веб-браузер для взаимодействия с приложением, выполняющимся под управлением настоящего веб-сервера. К счастью, большинство браузеров поддерживает возможность автоматизации операций. Selenium¹ – это инструмент автоматизации управления браузером, поддерживающий большинство популярных веб-браузеров в трех основных операционных системах.

Интерфейс Python для Selenium можно установить с помощью утилиты pip:

```
(venv) $ pip install selenium
```

Для проведения тестирования с помощью Selenium необходимо запустить приложение на веб-сервере, готовом принимать HTTP-запросы. Способ тестирования, представленный в этом разделе, заключается в том, чтобы запустить приложение с сервером для разработки в фоновом потоке, а тесты запускать в основном потоке выполнения. Под управлением Selenium будет запускать браузер и подключать его к приложению для выполнения требуемых операций.

Проблема такого подхода состоит в том, что после завершения всех тестов сервер Flask необходимо останавливать, в идеале предусмотренными для этого стандартными средствами, чтобы фоновые задания, такие как оценка охвата кода тестированием, могли корректно завершить свою работу. Веб-сервер Werkzeug предусматривает возможность остановки, но, поскольку он выполняется изолированно, в отдельном потоке, единственный способ остановить его – послать обычный HTTP-запрос. В примере 15.6 представлена реализация маршрута, осуществляющего остановку сервера.

¹ <http://www.seleniumhq.org/>.

Пример 15.6 ❖ `_app/main/views.py`: маршрут для остановки сервера

```
@main.route('/shutdown')
def server_shutdown():
    if not current_app.testing:
        abort(404)
    shutdown = request.environ.get('werkzeug.server.shutdown')
    if not shutdown:
        abort(500)
    shutdown()
    return 'Shutting down...'
```

Этот маршрут выполняет свою работу, только когда приложение выполняется в тестовом окружении; в любых других случаях он возвращает код ошибки 404. Фактическая процедура остановки сервера заключается в вызове функции `shutdown`, экспортируемой сервером Werkzeug. После вызова этой функции и возврата ответа на запрос веб-сервер выполнит корректную остановку.

В примере 15.7 показана заготовка модульного теста, настроенного на выполнение под управлением Selenium.

Пример 15.7 ❖ `tests/test_selenium.py`: заготовка модульного теста, выполняющегося под управлением Selenium

```
from selenium import webdriver

class SeleniumTestCase(unittest.TestCase):
    client = None

    @classmethod
    def setUpClass(cls):
        # запуск Firefox
        try:
            cls.client = webdriver.Firefox()
        except:
            pass

        # пропустить следующие тесты, если браузер не запустился
        if cls.client:
            # создать приложение
            cls.app = create_app('testing')
            cls.app_context = cls.app.app_context()
            cls.app_context.push()

            # подавить вывод отладочных сообщений, чтобы очистить
            # вывод от лишнего мусора
            import logging
            logger = logging.getLogger('werkzeug')
```

```

logger.setLevel("ERROR")

# создать базу данных и наполнить ее фиктивными данными
db.create_all()
Role.insert_roles()
User.generate_fake(10)
Post.generate_fake(10)

# добавить учетную запись администратора
admin_role = Role.query.filter_by(permissions=0xff).first()
admin = User(email='john@example.com',
              username='john', password='cat',
              role=admin_role, confirmed=True)
db.session.add(admin)
db.session.commit()

# запустить сервер Flask в отдельном потоке
threading.Thread(target=cls.app.run).start()

@classmethod
def tearDownClass(cls):
    if cls.client:
        # остановить сервер и закрыть браузер
        cls.client.get('http://localhost:5000/shutdown')
        cls.client.close()

        # уничтожить базу данных
        db.drop_all()
        db.session.remove()

        # удалить контекст приложения
        cls.app_context.pop()


def setUp(self):
    if not self.client:
        self.skipTest('Web browser not available')

def tearDown(self):
    pass

```

Методы класса `setUpClass()` и `tearDownClass()` вызываются до и после выполнения тестов, определенных в этом классе. Процедура настройки начинается с запуска браузера Firefox посредством прикладного интерфейса webdriver Selenium, создания экземпляра приложения и наполнения базы данных информацией, используемой в процессе тестирования. Приложение запускается в отдельном потоке вызовом стандартного метода `app.run()`. По окончании тестирования приложению посылается запрос с адресом `/shutdown`, что вызы-

вает его остановку. Затем закрывается окно браузера и уничтожается тестовая база данных.

 Selenium поддерживает множество других браузеров, кроме Firefox. Если вы собираетесь использовать другой браузер, обращайтесь за справками к документации с описанием Selenium: <http://docs.seleniumhq.org/docs/>.

Метод `setUp()`, который вызывается перед каждым тестом, пропускает тестирование, если Selenium не удалось запустить браузер в методе `setUpClass()`. В примере 15.8 приводится тест, реализованный на основе Selenium.

Пример 15.8 ❖ `tests/test_selenium.py`: пример модульного теста на основе Selenium

```
class SeleniumTestCase(unittest.TestCase):
    # ...

    def test_admin_home_page(self):
        # перейти на главную страницу
        self.client.get('http://localhost:5000/')
        self.assertTrue(re.search('Hello,\s+Stranger!',
                                   self.client.page_source))

        # перейти на страницу аутентификации
        self.client.find_element_by_link_text('Log In').click()
        self.assertTrue('<h1>Login</h1>' in self.client.page_source)

        # выполнить аутентификацию
        self.client.find_element_by_name('email').\
            send_keys('john@example.com')
        self.client.find_element_by_name('password').send_keys('cat')
        self.client.find_element_by_name('submit').click()
        self.assertTrue(re.search('Hello,\s+john!', self.client.page_source))

        # перейти на страницу профиля пользователя
        self.client.find_element_by_link_text('Profile').click()
        self.assertTrue('<h1>john</h1>' in self.client.page_source)
```

Этот тест выполняет процедуру аутентификации с использованием учетной записи администратора, созданной методом `setUpClass()`, и затем открывает страницу профиля. Обратите внимание, насколько отличается методология тестирования с помощью Selenium от методологии тестирования с помощью тестового клиента Flask. Когда тестирование выполняется с помощью Selenium, тесты посылают команды веб-браузеру и никогда не взаимодействуют с приложением непосредственно. Имена команд очень похожи на описание действий, выполняемых пользователем с помощью мыши и клавиатуры.

Тест начинается с вызова метода `get()` для получения главной страницы приложения. Это приводит к вводу указанного URL в адресную строку браузера. Чтобы проверить результат, полученный на этом шаге, выполняется поиск приветствия «Hello, Stranger!» в исходном коде страницы.

Чтобы перейти на страницу аутентификации, тест отыскивает ссылку с текстом «Log In» (Войти) вызовом `find_element_by_link_text()` и затем вызывает метод `click()`, имитирующий щелчок мышью на ссылке. Selenium предоставляет целое семейство вспомогательных методов `find_element_by...`(), позволяющих искать элементы по разным параметрам.

Для аутентификации в приложении тест находит поля для ввода адреса электронной почты и пароля по их именам, используя для этого `find_element_by_name()`, и затем вызовом `send_keys()` вводит в них текст. Форма отправляется на сервер вызовом метода `click()` для кнопки отправки. Чтобы убедиться в успешности аутентификации, проверяется персонализированное приветствие.

На заключительном этапе тест отыскивает ссылку «Profile» в строке навигации и имитирует щелчок на ней. Чтобы убедиться, что страница профиля благополучно загрузилась, в исходном коде страницы выполняется поиск заголовка с именем пользователя.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 15d` и получить данную версию приложения. Это обновление содержит файл миграции базы данных, поэтому не забудьте выполнить команду `python manage.py db upgrade` после извлечения кода из репозитория. Чтобы гарантировать установку всех необходимых расширений, выполните также команду `pip install -r requirements/dev.txt`.

Насколько это необходимо?

У многих из вас может появиться вопрос: стоит ли вообще мучиться с тестовым клиентом Flask или Selenium? Хороший вопрос, но на него нет простого ответа.

Хотите вы того или нет, но приложение в любом случае будет тестироваться. Если вы не протестируете его сами, оно невольно будет тестироваться пользователями, которые будут находить ошибки, и вам придется исправлять их. Простые и ограниченные тесты, подобные тем, что были написаны нами для тестирования моделей базы данных и других элементов приложения, которые можно выполнять

вне контекста приложения, создавать нужно всегда, так как они просты в реализации и помогают убедиться в правильной работе основных элементов приложения.

Сквозное тестирование с применением тестового клиента Flask или Selenium тоже желательно выполнить, но из-за повышенной сложности создания таких тестов можно ограничиться тестированием функциональности, которую нельзя проверить отдельно. Код приложения должен быть организован так, чтобы как можно больше прикладной логики содержалось в моделях базы данных или в других вспомогательных классах, не зависящих от контекста приложения и потому легко поддающихся тестированию. Код в функциях представления должен быть максимально простым и действовать как тонкая обертка, принимающая запросы и вызывающая соответствующие операции в других классах или функциях, инкапсулирующих прикладную логику.

И да, тестирование стоит усилий, потраченных на реализацию тестов. Но так же важно тщательно обдумывать стратегию тестирования и писать код, который сможет извлечь выгоду из этого.

Глава 16

Производительность

Никому не нравится работать с медленными приложениями. Длительное ожидание, пока загрузится страница, приводит пользователей в уныние, поэтому так важно своевременно выявлять и устранять проблемы, отрицательно сказывающиеся на производительности. В этой главе мы рассмотрим два важнейших аспекта, влияющих на производительность веб-приложений.

Регистрация медленных запросов к базе данных

Когда производительность приложения падает с течением времени, это зачастую вызвано падением производительности запросов к базе данных из-за увеличивающегося объема хранимых данных. Иногда для оптимизации производительности базы данных достаточно добавить дополнительные индексы в таблицы, но иногда требуется организовать кэширование информации между базой данных и приложением. Инструкция `explain`, имеющаяся в языке запросов большинства баз данных, показывает, какие операции выполняет база данных для выполнения указанного запроса, и часто помогает выявить недостатки в структуре базы данных или в индексах, ухудшающие производительность.

Но, прежде чем приступить к оптимизации запросов, необходимо определить, какие именно запросы следует оптимизировать. Типичное обращение к базе данных может вызывать передачу множества запросов, поэтому часто бывает сложно выявить, какой из этих запросов оказывается «слабым звеном». Расширение `Flask-SQLAlchemy` позволяет сохранять статистику запросов к базе данных, выполняемых в процессе обращения к ней. В примере 16.1 показано, как можно использовать эту возможность для получения информации о запросах, выполняющихся медленнее определенного порога.

Пример 16.1 ❖ app/main/views.py: отчет о скорости выполнения запросов к базе данных

```

from flask.ext.sqlalchemy import get_debug_queries

@main.after_app_request
def after_request(response):
    for query in get_debug_queries():
        if query.duration >= current_app.config['FLASKY_SLOW_DB_QUERY_TIME']:
            current_app.logger.warning(
                'Slow query: %s\nParameters: %s\nDuration: %fs\nContext: %s\n' %
                (query.statement, query.parameters, query.duration,
                 query.context))
    return response

```

Эта функциональность заключена в обработчик `after_app_request`, который напоминает обработчик `before_app_request`, но вызывается после того, как функция представления обработает запрос. Flask передает обработчику `after_app_request` объект ответа на случай, если потребуется внести в него какие-либо изменения.

В данном случае обработчик `after_app_request` не изменяет ответ; он просто извлекает время выполнения запроса, записанное расширением Flask-SQLAlchemy, и выводит информацию о запросе, если он выполнялся слишком медленно.

Функция `get_debug_queries()` возвращает выполнявшиеся запросы в виде списка. В табл. 16.1 перечислена информация, которая возвращается для каждого запроса.

Таблица 16.1. Сведения о запросах, сохраняемые расширением Flask-SQLAlchemy

Имя	Описание
<code>statement</code>	Инструкция SQL
<code>parameters</code>	Параметры, использованные в инструкции SQL
<code>start_time</code>	Время начала выполнения запроса
<code>end_time</code>	Время окончания выполнения запроса
<code>duration</code>	Продолжительность выполнения запроса в секундах
<code>context</code>	Строка, определяющая строку в исходном коде, где был запущен запрос

Обработчик `after_app_request` обходит список и журналирует все запросы, выполняющиеся дольше установленного порога. Журналирование выполняется с уровнем важности `WARNING`. Если повысить уровень важности до `ERROR`, это заставит систему журналирования сообщать о медленных запросах еще и по электронной почте.

По умолчанию функция `get_debug_queries()` работает только в режиме отладки. К сожалению, проблемы производительности базы данных редко проявляются на этапе разработки из-за небольшого объема базы данных. По этой причине иногда полезно использовать эту возможность в эксплуатационном окружении. В примере 16.2 показано, какие изменения в настройках необходимо произвести, чтобы включить измерение производительности в эксплуатационном режиме.

Пример 16.2 ❖ `config.py`: настройка регистрации медленных запросов

```
class Config:
    # ...
    SQLALCHEMY_RECORD_QUERIES = True
    FLASKY_DB_QUERY_TIMEOUT = 0.5
    # ...
```

Параметр `SQLALCHEMY_RECORD_QUERIES` сообщает расширению Flask-SQLAlchemy о необходимости включить запись информации о запросах. Порог, выше которого запросы считаются медленными, установлен равным полсекунды. Обе переменные включены в базовый класс `Config`, поэтому они доступны во всех конфигурациях.


Всякий раз, когда очередной запрос классифицируется как медленный, информация об этом запросе передается механизму журналирования в приложении. Чтобы эта информация сохранялась, необходимо настроить механизм журналирования. Порядок настройки журналирования в значительной степени зависит от платформы, на которой выполняется приложение. Некоторые примеры можно найти в главе 17.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 16a` и получить эту версию приложения.

Профилирование исходного кода


Другим возможным источником проблем с производительностью является высокое потребление CPU, вызванное выполнением интенсивных и продолжительных вычислений. В поиске медленных участков внутри приложения вам могут помочь профилировщики. Профилировщик следит за выполнением приложения и записывает, какие функции вызывались и как долго они выполнялись. Затем он создает подробный отчет, показывающий наиболее медленные функции.

 Обычно профилирование выполняется в окружении разработки. Профилировщики исходного кода замедляют скорость работы приложения, потому что им приходится следить за всем, что происходит. Производить профилирование во время эксплуатации не рекомендуется, если только не используется какой-нибудь легковесный профилировщик, созданный специально для использования в промышленном окружении.

Веб-сервер из фреймворка Flask, предназначенный для использования на этапе разработки, может выполнять профилирование обработки каждого запроса. В примере 16.3 показано, как добавить в приложение новую команду, запускающую профилирование.

Пример 16.3 ❖ manage.py: запуск приложения с профилированием запросов

```
@manager.command
def profile(length=25, profile_dir=None):
    """Запускает приложение в режиме профилирования кода."""
    from werkzeug.contrib.profiler import ProfilerMiddleware
    app.wsgi_app = ProfilerMiddleware(app.wsgi_app, restrictions=[length],
                                     profile_dir=profile_dir)
    app.run()
```

 Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 16b` и получить эту версию приложения.

Если запустить приложение командой `python manage.py profile`, на консоль будет выводиться информация, собранная профилировщиком для каждого запроса, включая 25 самых медленных функций. Изменить число функций, отображаемых в отчете, можно с помощью параметра `--length`. Если задан параметр `--profile-dir`, данные профилирования для каждого запроса будут сохраняться в файл в указанном каталоге. Файлы с этими данными можно использовать для создания более подробных отчетов, включающих *дерево вызовов*. За дополнительной информацией о профилировщике для Python обращайтесь к официальной документации по адресу: <http://bit.ly/py-profile>.

Подготовка приложения к развертыванию закончена. В следующей главе вы узнаете, чего ждать при развертывании приложения.

Глава 17

Развертывание

Веб-сервер, предназначенный для разработки и входящий в состав Flask, не обладает ни надежностью, ни безопасностью, ни достаточной эффективностью, чтобы его можно было использовать в промышленном окружении. В этой главе мы исследуем возможные варианты развертывания приложений на основе фреймворка Flask.

Порядок развертывания

Независимо от выбранной платформы есть ряд операций, которые необходимо выполнить при установке приложения на промышленный сервер. Примером таких операций может служить создание или обновление таблиц базы данных.

Выполнение этих операций вручную при каждой установке или обновлении приложения может требовать немалого времени и чревата ошибками, поэтому лучше добавить в сценарий *manage.py* команду, которая будет выполнять все необходимые операции.

В примере 17.1 показана реализация команды `deploy` для приложения Flasky.

Пример 17.1 ❖ `manage.py`: команда `deploy`


```
@manager.command
def deploy():
    """Выполняет операции, связанные с развертыванием."""
    from flask.ext.migrate import upgrade
    from app.models import Role, User

    # обновить базу данных до последней версии
    upgrade()

    # создать роли для пользователей
    Role.insert_roles()

    # объявить всех пользователей как читающих самих себя
    User.add_self_follows()
```

Все функции, вызываемые этой командой, были созданы ранее; просто здесь впервые они вызываются все вместе.

 Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 17a` и получить эту версию приложения.

Все эти функции спроектированы так, что не вызывают проблем при многократном вызове. Такой подход делает возможным выполнять команду `deploy` всякий раз, когда производится установка или обновление.

Журналирование ошибок во время эксплуатации

Когда приложение выполняется в режиме отладки, при возникновении ошибки в работу вмешивается интерактивный отладчик из пакета Werkzeug. Он отображает *трассировку стека* в веб-странице, дает возможность просматривать исходный код и даже вычислять выражения в контексте каждого кадра стека.

Отладчик — отличный инструмент для отладки проблем во время разработки, но по понятным причинам он не может использоваться в промышленном окружении. Ошибки, возникающие на этапе эксплуатации, подавляются, и вместо развернутых сообщений пользователь получает ошибку с кодом 500. Но трассировка при этом не теряется, потому что Flask записывает ее в *файл журнала*.

В момент запуска приложения фреймворк Flask создает экземпляр класса `logging.Logger` и добавляет его в экземпляр приложения как `app.logger`. В режиме отладки этот механизм журналирования выводит сообщения в консоль, но в режиме промышленной эксплуатации отсутствуют обработчики, настраиваемые по умолчанию, — в отсутствие обработчиков журналируемые сообщения не сохраняются. В примере 17.2 показана настройка обработчика, отправляющего сообщения об ошибках по адресам электронной почты администраторов, перечисленным в параметре настройки `FLASKY_ADMIN`.

Пример 17.2 ❖ `config.py`: отправка электронных писем с сообщениями об ошибках

```
class ProductionConfig(Config):
    # ...
    @classmethod
    def init_app(cls, app):
```



```

Config.init_app(app)

# отправить администраторам письма с сообщениями об ошибках
import logging
from logging.handlers import SMTPHandler
credentials = None
secure = None
if getattr(cls, 'MAIL_USERNAME', None) is not None:
    credentials = (cls.MAIL_USERNAME, cls.MAIL_PASSWORD)
    if getattr(cls, 'MAIL_USE_TLS', None):
        secure = ()
mail_handler = SMTPHandler(
    mailhost=(cls.MAIL_SERVER, cls.MAIL_PORT),
    fromaddr=cls.FLASKY_MAIL_SENDER,
    toaddrs=[cls.FLASKY_ADMIN],
    subject=cls.FLASKY_MAIL_SUBJECT_PREFIX + ' Application Error',
    credentials=credentials,
    secure=secure)
mail_handler.setLevel(logging.ERROR)
app.logger.addHandler(mail_handler)

```

Напомню, что все экземпляры конфигураций имеют статический метод `init_app()`, который вызывается методом `create_app()`. В реализации этого метода для класса `ProductionConfig` механизм журналирования настраивается на отправку сообщений об ошибках по электронной почте.

Уровень журналирования для обработчика, рассылающего электронные письма, установлен в значение `logging.ERROR`, поэтому только сообщения о серьезных проблемах будут рассылаться по электронной почте. Сообщения с более низкими уровнями значимости можно сохранять в файле, в системном журнале, или использовать любой другой поддерживаемый способ хранения, добавляя соответствующие обработчики. Способ журналирования в значительной степени зависит от используемой платформы.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 17b` и получить эту версию приложения.

Развертывание в облаке

В последнее время для развертывания приложений часто используются облачные платформы. Эта технология, которая официально называется «платформа как услуга» (Platform as a Service, PaaS), освобождает разработчиков от рутины, связанной с установкой и сопровождением аппаратных и программных платформ, где выполняет

ся приложение. В модели PaaS сопровождение платформы полностью осуществляется поставщиком услуг. Разработчикам предоставляются инструменты и библиотеки для интеграции приложения с платформой. Затем приложение выгружается на сервер, сопровождаемый поставщиком услуг, и разворачивается в течение нескольких секунд. Большинство поставщиков услуг PaaS предлагает возможность динамического «масштабирования» приложения путем добавления или удаления серверов в зависимости от колебаний нагрузки.

Развертывание в облаке дает огромную гибкость и относительную простоту в настройке, однако все эти преимущества имеют свою цену. В следующем разделе мы поближе познакомимся с компанией Heroku, одним из популярных поставщиков услуг PaaS, предлагающих великолепную поддержку Python.

Платформа Heroku

Компания Heroku стала одним из первых поставщиков услуг PaaS, начав свой бизнес в 2007 году. Платформа Heroku очень гибкая и поддерживает длинный список языков программирования. Чтобы развернуть приложение на платформе Heroku, разработчик должен поместить его в репозиторий Git. Команда `git push` автоматически запустит установку, настройку и развертывание приложения на сервере Heroku.


Для определения стоимости услуг в компании Heroku используется единица измерения вычислительной мощности, которая называется *dyno*. Наиболее типичной разновидностью *dyno* является *web dyno*, единица измерения, представляющая экземпляр веб-сервера. Приложение может увеличить вычислительные мощности, выделяемые для обработки запросов, используя больше единиц *web dyno*. Другой разновидностью *dyno* является единица измерения *worker dyno*, используемая для определения вычислительной мощности, выделяемой для выполнения фоновых заданий или других сопутствующих задач.

Платформа предоставляет огромное число расширений и дополнений поддержки баз данных, электронную почту и многие другие службы. В следующих разделах мы подробнее обсудим порядок развертывания приложения Flasky на платформе Heroku.

Подготовка приложения

Для развертывания на платформе Heroku приложение должно находиться в репозитории Git. Если вы используете удаленный Git-

сервер, такой как GitHub или BitBucket, создайте локальную копию репозитория для передачи его на платформу Heroku. Если приложение еще не помещено в Git-репозиторий, это необходимо сделать на компьютере, где ведется разработка.

 Если вы планируете развернуть свое будущее приложение на платформе Heroku, с самого начала создайте для него репозиторий Git. Инструкции по установке для трех основных систем можно найти на сайте GitHub, в справочном руководстве по адресу: <http://help.github.com/>.

Создание учетной записи Heroku

Прежде чем вы сможете воспользоваться услугами компании Heroku, необходимо создать учетную запись¹. Вы можете зарегистрировать свое приложение на самом низком уровне обслуживания и пользоваться услугами бесплатно, что делает эту платформу отличным испытательным полигоном.

Установка комплекта инструментов Heroku Toolbelt


Удобнее всего управлять своими приложениями на платформе Heroku с помощью специализированного комплекта инструментов командной строки Heroku Toolbelt. В состав Toolbelt входят два приложения:

- `heroku`: клиент службы Heroku, используется для создания приложений и управления ими;
- `foreman`: инструмент, имитирующий окружение Heroku на локальном компьютере для тестирования.

Обратите внимание, что если у вас еще не установлен клиент Git, мастер установки Toolbelt автоматически установит Git.

Клиенту Heroku необходимы ваши учетные данные для подключения к службе. Ввести их можно с помощью команды `heroku login`:

```
$ heroku login
Enter your Heroku credentials.
Email: <your-email-address>
Password (typing will be hidden): <your-password>
Uploading ssh public key .../id_rsa.pub
```

 Очень важно выгрузить на сервер Heroku свой публичный ключ SSH, чтобы получить возможность выполнить команду `git push`. Обычно команда `login` создает и выгружает этот ключ автоматически, однако то же самое можно сделать вручную командой `heroku keys:add`, что может пригодиться для загрузки дополнительных ключей.

¹ <https://www.heroku.com/>.

Создание приложения

Следующий шаг – создание приложения с помощью клиента Heroku. Предварительно необходимо поместить приложение в репозиторий Git, если этого еще не сделано, и затем выполнить следующую команду из каталога верхнего уровня:

```
$ heroku create <appname>
Creating <appname>... done, stack is cedar
http://<appname>.herokuapp.com/ | git@heroku.com:<appname>.git
Git remote heroku added
```

Приложения, размещаемые на платформе Heroku, должны иметь уникальные имена, поэтому вам придется подобрать имя, не занятое другими приложениями. Как следует из вывода команды create, после развертывания приложение будет доступно по адресу: `http://<appname>.herokuapp.com`. К имени приложения можно также присоединить собственное доменное имя.

В процессе создания приложения Heroku создаст сервер Git: `git@heroku.com:<appname>.git`. К имени приложения можно также присоединить собственное доменное имя. Команда create добавит этот сервер в ваш локальный репозиторий, так что его можно будет увидеть под именем heroku, выполнив команду `git remote`.

Подготовка базы данных

Heroku поддерживает базы данных Postgres в виде дополнений. Небольшие базы данных, до 10 000 записей, можно использовать в приложении бесплатно:

```
$ heroku addons:add heroku-postgresql:dev
Adding heroku-postgresql:dev on <appname>... done, v3 (free)
Attached as HEROKU_POSTGRESQL_BROWN_URL
Database has been created and is available
! This database is empty. If upgrading, you can transfer
! data from another database with pgbackups:restore.
Use `heroku addons:docs heroku-postgresql:dev` to view documentation.
```

Здесь `HEROKU_POSTGRESQL_BROWN_URL` – это имя переменной окружения, хранящей URL базы данных. Обратите внимание, что, пытаясь выполнить эту команду у себя, вы можете получить другой цвет в имени переменной¹. Heroku позволяет использовать в одном приложении несколько баз данных, и для каждой из них будет создана

¹ Здесь часть «BROWN» в имени переменной – это название коричневого цвета. – *Прим. перев.*

переменная окружения с URL своего цвета. После этого URL базы данных можно экспортировать с переменной окружения `DATABASE_URL`. Данную операцию с базой данных, созданной выше, выполняет следующая команда:

```
$ heroku pg:promote HEROKU_POSTGRES_SQL_BROWN_URL
Promoting HEROKU_POSTGRES_SQL_BROWN_URL to DATABASE_URL... done
```

Формат переменной окружения `DATABASE_URL` в точности соответствует требованиям SQLAlchemy. Напомню, что в сценарии *config.py* используется значение переменной `DATABASE_URL`, если оно определено, поэтому теперь автоматически будет выполняться подключение к базе данных Postgres.

Настройка журналирования

Журналирование фатальных ошибок и рассылка сообщений о них по электронной почте были добавлены ранее, но в дополнение к этому очень важно настроить журналирование сообщений менее важных категорий. Хорошим примером таких сообщений могут служить предупреждения о медленной работе запросов к базе данных, добавленные в главе 16.

Приложения на платформе Heroku должны осуществлять журналирование в поток вывода `stdout` или `stderr`. Эти сообщения будут автоматически перехватываться и сохраняться платформой, после чего их можно прочитать с помощью клиента Heroku, командой `heroku logs`.

Настройку журналирования можно добавить в класс `ProductionConfig`, в его статический метод `init_app()`, но, поскольку данный способ журналирования является характерной особенностью Heroku, необходимо создать новую конфигурацию конкретно для этой платформы, оставив за классом `ProductionConfig` роль базовой конфигурации для разных платформ. Определение класса `HerokuConfig` показано в примере 17.3.

Пример 17.3 ❖ config.py: конфигурация Heroku

```
class HerokuConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # журналирование в поток stderr
        import logging
        from logging import StreamHandler
```

```
file_handler = StreamHandler()
file_handler.setLevel(logging.WARNING)
app.logger.addHandler(file_handler)
```

Когда приложение запускается под управлением платформы Heroku, ему необходимо сообщить, что оно должно использовать эту конфигурацию. Чтобы узнать, какую конфигурацию использовать, приложение проверяет переменную окружения `FLASK_CONFIG`, соответственно, эту переменную нужно установить в окружении Heroku. Установка переменных окружения выполняется командой `config:set` клиента Heroku:

```
$ heroku config:set FLASK_CONFIG=heroku
Setting config vars and restarting <appname>... done, v4
FLASK_CONFIG: heroku
```

Настройка электронной почты

Heroku не предоставляет свой сервер SMTP, поэтому для рассылки электронных писем необходимо настроить доступ к внешнему серверу. Существует несколько сторонних дополнений, интегрирующих надежные механизмы рассылки электронной почты в Heroku, но для тестирования и оценки вполне достаточно конфигурации Gmail, которая уже определена в базовом классе `Config`.

Так как внедрение учетных данных непосредственно в сценарий представляет определенную опасность, имя пользователя и пароль для доступа к SMTP-серверу Gmail определяются с помощью переменных окружения:

```
$ heroku config:set MAIL_USERNAME=<your-gmail-username>
$ heroku config:set MAIL_PASSWORD=<your-gmail-password>
```

Запуск промышленного веб-сервера

Платформа Heroku не предоставляет веб-сервера для своих приложений. Вместо этого предполагается, что приложения будут запускать собственные серверы, которые будут прослушивать порт с номером, указанным в переменной окружения `PORT`.

Веб-сервер, входящий в состав фреймворка Flask и предназначенный для разработки, имеет относительно невысокую производительность, потому что не предполагалось использовать его в промышленном окружении. Однако есть два надежных веб-сервера, которые прекрасно работают с приложениями на основе Flask: Gunicorn¹ и uWSGI².

¹ <http://gunicorn.org/>.

² <http://bit.ly/uwsgi-proj>.

Чтобы протестировать конфигурацию Нероки локально, можно установить веб-сервер в виртуальное окружение. Например, веб-сервер Gunicorn можно установить командой:

```
(venv) $ pip install gunicorn
```

Запуск приложения под управлением Gunicorn осуществляется следующей командой:

```
(venv) $ gunicorn manage:app
2013-12-03 09:52:10 [14363] [INFO] Starting gunicorn 18.0
2013-12-03 09:52:10 [14363] [INFO] Listening at: http://127.0.0.1:8000 (14363)
2013-12-03 09:52:10 [14363] [INFO] Using worker: sync
2013-12-03 09:52:10 [14368] [INFO] Booting worker with pid: 14368
```

Аргумент `manage:app` содержит слева от двоеточия пакет или модуль, определяющий приложение, и имя экземпляра приложения внутри пакета справа. Обратите внимание, что по умолчанию веб-сервер Gunicorn использует порт 8000, а не 5000, как Flask.

Добавление файла со списком зависимостей

Нероки загружает пакеты зависимостей из файла *requirements.txt*, хранящегося в папке верхнего уровня. Все зависимости из этого файла будут импортироваться в виртуальное окружение, созданное платформой Нероки в ходе развертывания.

Файл *requirements.txt* для Нероки должен включать все общие зависимости для промышленной версии приложения, а также пакет *psycopg2* поддержки базы данных Postgres и веб-сервер Gunicorn. В примере 17.4 приводится пример файла *requirements.txt*.

Пример 17.4 ❖ *requirements.txt*: файл со списком зависимостей для Нероки

```
-r requirements/prod.txt
gunicorn==18.0
psycopg2==2.5.1
```

Добавление файла Procfile

Платформе Нероки необходимо сообщить, какую команду использовать для запуска приложения. Эта команда определяется в специальном файле с именем *Procfile*. Данный файл должен находиться в папке верхнего уровня приложения.

В примере 17.5 показано содержимое этого файла.

Пример 17.5 ❖ Procfile: файл Procfile для Нероки

```
web: gunicorn manage:app
```

Формат файла *Procfile* очень прост: в каждой строке определяется имя задания, за которым через двоеточие указывается команда запуска этого задания. Имя `web` является специальным; оно интерпретируется платформой Heroku как задание, запускающее веб-сервер. Heroku создает для этого задания переменную окружения `PORT` и присваивает ей номер порта, на котором приложение будет принимать запросы. По умолчанию веб-сервер Gunicorn учитывает значение переменной `PORT`, если оно определено, поэтому нет необходимости включать номер порта в команду запуска.



Приложения могут объявлять дополнительные задания в файле *Procfile* с именами, отличными от `web`. Это могут быть другие службы, необходимые приложению. При развертывании приложения Heroku запустит все задания, перечисленные в *Procfile*.

Тестирование с помощью Foreman

В состав комплекта инструментов Heroku Toolbelt входит еще одна утилита с именем *Foreman*, которую можно использовать для запуска приложения локально с целью тестирования. Переменные окружения, такие как `FLASK_CONFIG`, устанавливаемые с помощью клиента Heroku, доступны лишь на серверах Heroku, поэтому необходимо также определить эти переменные и в окружении, создаваемом утилитой *Foreman*. Эта утилита ищет определения переменных окружения в файле с именем *.env*, находящемся в каталоге приложения верхнего уровня. Например, файл *.env* может содержать следующие определения переменных:

```
FLASK_CONFIG=heroku
MAIL_USERNAME=<имя_пользователя>
MAIL_PASSWORD=<пароль>
```



Так как файл *.env* содержит пароли и другую конфиденциальную информацию, он никогда не должен добавляться в репозиторий Git.

Утилита *Foreman* может вызываться разными способами, но основными являются `foreman run` и `foreman start`. Команду `run` можно использовать для запуска произвольных команд в окружении приложения, в частности для выполнения команды `deploy`, используемой приложением для создания базы данных:

```
(venv) $ foreman run python manage.py deploy
```

Команда `start` читает содержимое файла *Procfile* и выполняет все задания, перечисленные в нем:


```
(venv) $ foreman start
22:55:08 web.1 | started with pid 4246
22:55:08 web.1 | 2013-12-03 22:55:08 [4249] [INFO] Starting gunicorn 18.0
22:55:08 web.1 | 2013-12-03 22:55:08 [4249] [INFO] Listening at: http://...
22:55:08 web.1 | 2013-12-03 22:55:08 [4249] [INFO] Using worker: sync
22:55:08 web.1 | 2013-12-03 22:55:08 [4254] [INFO] Booting worker with pid: 4254
```

Foreman объединяет вывод журналируемых записей от всех запущенных заданий и выводит их в консоль, добавляя в начало каждой записи текущее время и имя задания.

С помощью флага `-c` можно имитировать потребление вычислительных мощностей в *dyno*. Например, следующая команда запустит три веб-сервера, принимающих запросы на разных портах:

```
(venv) $ foreman start -c web=3
```

Включение безопасного протокола HTTP с помощью Flask-SSLify

Когда пользователь выполняет вход в приложение, отправляя имя пользователя и пароль с веб-формой, во время передачи по сети эти учетные данные могут быть перехвачены третьей стороной, как уже отмечалось несколько раз выше. Чтобы предотвратить перехват учетных данных таким способом, необходимо использовать безопасную версию протокола HTTP, который шифрует все данные, передаваемые между клиентом и сервером с использованием публичного ключа.

Платформа Heroku обеспечивает доступ ко всем приложениям в домене *herokuapp.com* по обоим протоколам, *http://* и *https://*, без необходимости настраивать использование сертификата SSL. Единственное, что нужно сделать в приложении, — перехватывать все запросы к интерфейсу *http://* и переадресовывать их на интерфейс *https://*, что и делает расширение Flask-SSLify.

Это расширение необходимо добавить в файл *requirements.txt*. В примере 17.6 показано, как активировать данное расширение.

Пример 17.6 ❖ `app/__init__.py`: переадресация всех запросов на защищенный интерфейс *https://*

```
def create_app(config_name):
    # ...
    if not app.debug and not app.testing and not app.config['SSL_DISABLE']:
        from flask.ext.sslify import SSLify
        sslify = SSLify(app)
    # ...
```

Поддержка SSL необходима только в режиме промышленной эксплуатации и только если платформа обеспечивает такую возможность. Чтобы упростить включение и выключение поддержки SSL, была добавлена новая переменная `SSL_DISABLE`. Базовый класс `Config` устанавливает ее в значение `True`, то есть по умолчанию SSL не используется, а класс `HerokuConfig` переопределяет ее. Реализация установки этой переменной показана в примере 17.7.

Пример 17.7 ❖ `config.py`: настройка использования SSL

```
class Config:
    # ...
    SSL_DISABLE = True

class HerokuConfig(ProductionConfig):
    # ...
    SSL_DISABLE = bool(os.environ.get('SSL_DISABLE'))
```

Значение для поля `SSL_DISABLE` в `HerokuConfig` извлекается из одноименной переменной окружения. Если переменная окружения имеет любое произвольное значение, кроме пустой строки, преобразование в логическое значение вернет `True`, что приведет к отключению SSL. Если переменная окружения отсутствует или содержит пустую строку, преобразование в логическое значение вернет `False`. Чтобы выключить SSL при использовании утилиты `Foreman`, в файл `.env` необходимо добавить `SSL_DISABLE=1`.

С этими настройками пользователи будут вынуждены использовать SSL, но есть еще кое-какие мелочи, необходимые, чтобы обеспечить полноту поддержки. При использовании `Heroku` клиенты подключаются не к приложениям непосредственно, а к обратному прокси-серверу, переадресующему запросы приложениям. В такой конфигурации только прокси-сервер действует в режиме поддержки SSL; приложения принимают все запросы от прокси-сервера по простому протоколу HTTP, потому что во внутренней сети `Heroku` нет необходимости в дополнительной защите. Это обстоятельство может превратиться в проблему, когда приложению потребуется сгенерировать абсолютный URL, соответствующий требованиям безопасности, потому что при использовании обратного прокси-сервера поле `request.is_secure` всегда будет иметь значение `False`.

Примером ситуаций, когда это может стать проблемой, является создание адресов URL аватаров. Если вы помните, в главе 10 говорилось, что метод `gravatar()` модели `User`, генерирующий адреса URL, ведущие на сайт `Gravatar`, проверяет поле `request.is_secure`

и на основе его значения создает защищенные или незащищенные URL. Использование незащищенных адресов аватаров на странице, которая сама была запрошена по защищенному интерфейсу, может заставить некоторые браузеры вывести предупреждение, поэтому интерфейсы во всех адресах на странице должны быть одинаково защищены.

Прокси-серверы передают информацию, описывающую оригинальный запрос, полученный от клиента, в нестандартных HTTP-заголовках, благодаря этому есть возможность определить, используется ли защищенный протокол для взаимодействия с приложением. В пакет Werkzeug входит модуль WSGI промежуточной обработки, проверяющий наличие заголовков, сформированных прокси-сервером, и вносит необходимые изменения в объект запроса, например присваивает полю `request.is_secure` значение, соответствующее защищенности запроса, отправленного клиентом обратному прокси-серверу. В примере 17.8 показано, как добавить модуль ProxyFix в приложение.

Пример 17.8 ❖ config.py: поддержка прокси-серверов

```
class HerokuConfig(ProductionConfig):
    # ...
    @classmethod
    def init_app(cls, app):
        # ...

        # обработка заголовков прокси-сервера
        from werkzeug.contrib.fixers import ProxyFix
        app.wsgi_app = ProxyFix(app.wsgi_app)
```

Включение в работу модуля промежуточной обработки производится в методе инициализации конфигурации Нероку. Добавление промежуточных уровней WSGI, таких как ProxyFix, производится путем обертывания WSGI-приложения. Когда поступает новый запрос, промежуточные модули получают возможность исследовать окружение и внести изменения в запрос до того, как он будет передан дальше для обработки. Модуль ProxyFix может потребоваться не только в случае использования Нероку, но и в любом другом окружении, где используется обратный прокси-сервер.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 17c` и получить эту версию приложения. Чтобы гарантировать установку всех необходимых расширений, выполните также команду `pip install -r requirements/dev.txt`.

Развертывание командой `git push`

Последний шаг в процессе развертывания – выгрузка приложения на серверы Нероку. Убедитесь, что все изменения сохранены в локальном репозитории Git, и затем выполните команду `git push heroku master`, чтобы выгрузить приложение:

```
$ git push heroku master
Counting objects: 645, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (315/315), done.
Writing objects: 100% (645/645), 95.52 KiB, done.
Total 645 (delta 369), reused 457 (delta 288)

.---> Python app detected
.----> No runtime.txt provided; assuming python-2.7.4.
.----> Preparing Python runtime (python-2.7.4)
...
-----> Compiled slug size: 32.8MB
-----> Launching... done, v8
http://<appname>.herokuapp.com deployed to Heroku

To git@heroku.com:<appname>.git
* [new branch]      master -> master
```

Теперь приложение развернуто и запущено, но пока оно работает неправильно, потому что еще не была выполнена команда `deploy`. Выполнить эту команду можно с помощью клиента Нероку:

```
$ heroku run python manage.py deploy
Running `python manage.py predeploy` attached to terminal... up, run.8449
INFO [alembic.migration] Context impl PostgresqlImpl.
INFO [alembic.migration] Will assume transactional DDL.
...
```

После создания и настройки таблиц в базе данных приложение можно перезапустить, чтобы запуск прошел чисто:

```
$ heroku restart
Restarting dynos... done
```

Теперь приложение должно быть доступно по адресу: **`https://<имя_приложения>.herokuapp.com`**.

Просмотр журналов

Вывод журналируемых записей, производимый приложением, перехватывается платформой Нероку. Получить содержимое журнала можно командой `logs`:

```
$ heroku logs
```

В ходе тестирования также может пригодиться возможность наблюдать за появлением новых записей в журнале:

```
$ heroku logs -t
```

Развертывание и обновление

Когда потребуется обновить приложение на платформе Нероку, придется вновь повторить некоторые шаги. После сохранения изменений в репозитории Git вызовите следующие команды, чтобы выполнить обновление:

```
$ heroku maintenance:on  
$ git push heroku master  
$ heroku run python manage.py deploy  
$ heroku restart  
$ heroku maintenance:off
```

Параметр `maintenance`, поддерживаемый клиентом Нероку, отключит приложение от сети на время обновления и обеспечит отображение статической страницы, информирующей пользователей, что сайт временно недоступен.

Традиционный хостинг

Традиционный хостинг подразумевает покупку или аренду сервера, физического или виртуального, и самостоятельную настройку всех необходимых компонентов. Обычно это обходится дешевле, чем услуги хостинга в облаке, но требует больше усилий. В следующих разделах вы получите представление о том, какие работы подразумевает традиционный хостинг.

Настройка сервера

Существует несколько задач администрирования, которые необходимо выполнить на сервере, прежде чем на нем можно будет размещать приложения:

- установить сервер баз данных, такой как MySQL или Postgres; можно также использовать базу данных SQLite, но для промышленной эксплуатации этот вариант не рекомендуется из-за множества ограничений;
- установить транспортный агент электронной почты (Mail Transport Agent, MTA), такой как Sendmail, чтобы отправлять почту пользователям;

- установить надежный, промышленный веб-сервер, такой как *Gunicorn* или *uWSGI*;
- приобрести, установить и настроить сертификат SSL для поддержки защищенного протокола HTTPS;
- (необязательно, но рекомендуется) установить обратный прокси-сервер, такой как *nginx* или *Apache*; этот процесс будет обслуживать статические файлы непосредственно и передавать все остальные запросы веб-серверу приложения, прослушивающему скрытый порт по адресу *localhost*;
- защитить сервер; под этим подразумевается несколько задач, целью которых является уменьшение уязвимости сервера, таких как установка брандмауэра, удаление неиспользуемого программного обеспечения и служб и т. д.

Импортирование переменных окружения

Как и на платформе Нероки, приложение, выполняющееся на отдельном сервере, также опирается на некоторые настройки, такие как URL базы данных, учетные данные для доступа к почтовому серверу и имя конфигурации. Все эти настройки хранятся в переменных окружения и должны импортироваться перед запуском приложения.

Поскольку в этом случае нет ни клиента Нероки, ни утилиты Foreman, с помощью которых можно было бы импортировать переменные, эту задачу необходимо выполнить в самом приложении во время запуска. В примере 17.9 приводится короткий фрагмент кода, загружающий и интерпретирующий файл *.env* подобно тому, как это делает утилита Foreman. Этот код можно добавить в сценарий запуска *manage.py* перед созданием экземпляра приложения.

Пример 17.9 ❖ manage.py: импортирование окружения из файла *.env*

```
if os.path.exists('.env'):
    print('Importing environment from .env...')
    for line in open('.env'):
        var = line.strip().split('=')
        if len(var) == 2:
            os.environ[var[0]] = var[1]
```

Файл *.env* должен содержать хотя бы переменную `FLASK_CONFIG`, определяющую имя используемой конфигурации.

Настройка журналирования

При использовании серверов на основе Unix журналирование можно осуществлять с помощью системного домена *syslog*. Новую конфигурацию для Unix можно создать в виде класса, наследующего класс *ProductionConfig*, как показано в примере 17.10.

Пример 17.10 ❖ config.py: пример конфигурации для Unix

```
class UnixConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # использовать syslog для журналирования
        import logging
        from logging.handlers import SysLogHandler
        syslog_handler = SysLogHandler()
        syslog_handler.setLevel(logging.WARNING)
        app.logger.addHandler(syslog_handler)
```

Согласно этой конфигурации, файлы журналов приложения будут сохраняться в каталоге */var/log/messages*. Службу *syslog* можно настроить на запись в отдельный файл журнала или отправлять журналируемые записи на другой сервер.



Если у вас уже есть копия репозитория с исходными текстами приложений с сайта GitHub, вы можете выполнить команду `git checkout 17d` и получить эту версию приложения.

Глава 18

Дополнительные ресурсы

Поздравляю, вы практически закончили чтение книги! Я надеюсь, что темы, представленные мною, помогли вам сформировать надежный фундамент для построения своих будущих приложений с применением фреймворка Flask. Примеры кода, приводившиеся в книге, выпущены под открытой, либеральной лицензией, поэтому вы свободно можете использовать любой мой код в своих проектах, даже если вы собираетесь распространять их на коммерческой основе. В этой короткой последней главе я хочу дать вам несколько советов и указать несколько дополнительных ресурсов, которые могут вам пригодиться при использовании Flask.

Использование интегрированной среды разработки

Применение интегрированной среды разработки (Integrated Development Environment, IDE) для создания приложений на основе Flask несет дополнительные удобства, поскольку такие функции, как автодополнение кода и интерактивный отладчик, способны существенно ускорить процесс программирования. Ниже перечислены некоторые IDE, имеющие прекрасную поддержку Flask.

- PyCharm¹: коммерческая IDE от компании JetBrains, доступная в редакциях Community (бесплатная) и Professional (платная). Обе редакции пригодны для разработки приложений на основе Flask. Доступны версии для Linux, Mac OS X и Windows.
- PyDev²: открытая IDE, основанная на Eclipse. Доступны версии для Linux, Mac OS X и Windows.

¹ <http://bit.ly/py-charm>.

² <http://pydev.org>.

- Python Tools for Visual Studio¹: бесплатная IDE, встраиваемая как расширение в Microsoft Visual Studio. Доступна только версия для Microsoft Windows.



При настройке приложения на основе Flask для запуска под управлением отладчика добавьте параметры `--passthrough-errors` `--no-reload` в команду `runserver`. Первый параметр запрещает перехват ошибок фреймворком Flask, благодаря чему ошибки, возникшие во время обработки запросов, будут возбуждать исключения, перехватываемые отладчиком. Второй параметр отключает модуль `reloader`, который вводит в заблуждение некоторые отладчики.

Поиск расширений для Flask

Примеры в этой книге опираются на некоторые расширения и пакеты, однако существует еще масса подобных расширений, не обсуждавшихся в этой книге, которые могут пригодиться вам. Ниже приводится короткий список некоторых пакетов, заслуживающих особого внимания:

- Flask-Babel²: поддержка интернационализации и локализации;
- Flask-RESTful³: комплект инструментов для создания веб-служб RESTful;
- Celery⁴: очередь фоновых заданий;
- Frozen-Flask⁵: преобразует приложение на основе Flask в статический веб-сайт;
- Flask-DebugToolbar⁶: инструменты отладки для браузера;
- Flask-Assets⁷: реализует слияние, минификацию и компиляцию ресурсов CSS и JavaScript;
- Flask-OAuth⁸: аутентификация посредством провайдеров OAuth;
- Flask-OpenID⁹: аутентификация посредством провайдеров OpenID;
- Flask-WhooshAlchemy¹⁰: полнотекстовый поиск для применения в моделях Flask-SQLAlchemy, основанных на Whoosh¹¹;

¹ <http://pytools.codeplex.com>.

² <http://bit.ly/fl-babel>.

³ <http://bit.ly/fl-rest>.

⁴ <http://bit.ly/celery-doc>.

⁵ <http://bit.ly/flask-frozen>.

⁶ <http://bit.ly/flask-debug>.

⁷ <http://bit.ly/fl-assets>.

⁸ <http://bit.ly/fl-oauth>.

⁹ <http://bit.ly/fl-opID>.

¹⁰ <http://bit.ly/fl-whoosh>.

¹¹ <http://pythonhosted.org/Whoosh/>.

- Flask-KVsession¹: альтернативная реализация пользовательских сеансов, использующая хранилище на стороне сервера.

Если среди этих расширений и пакетов, упоминавшихся в книге, вы не найдете функциональность, необходимую вашему проекту, тогда в первую очередь загляните в официальный реестр расширений для Flask². Поискать можно также на сайтах Python Package Index³, GitHub⁴ и BitBucket⁵.

Участие в разработке Flask

Фреймворк Flask не получился бы таким замечательным без участия в работе над ним обширного сообщества разработчиков. Так как теперь вы становитесь частью этого сообщества и начинаете пользоваться плодами трудов множества добровольцев, подумайте о возможности внести и свой вклад в развитие фреймворка. Ниже перечислены некоторые направления, с которых вы можете начать:

- обзор документации для Flask или любого родственного проекта и предложение своих корректировок и улучшений;
- перевод документации на другие языки;
- ответы на вопросы на сайтах-форумах, таких как Stack Overflow⁶;
- популяризация фреймворка среди своих коллег на встречах и конференциях;
- исправление ошибок или внесение улучшений в используемые вами пакеты;
- создание новых расширений для Flask и выпуск их с открытым исходным кодом;
- выпуск своих приложений с открытым исходным кодом.

Я надеюсь, что вы примете участие в развитии фреймворка, выбрав любое направление приложения усилий, какое сами сочтете нужным, и заранее благодарен вам!

¹ <http://bit.ly/fl-kvses>.

² <http://bit.ly/fl-exreg>.

³ <http://pypi.python.org>.

⁴ <http://github.org>.

⁵ <http://bitbucket.org>.

⁶ <http://stackoverflow.com/>.

Предметный указатель

Символы

.env, файл, 256, 262

B

Bleach, 168

F

Flask

- abort, функция, 36
- add_url_rule, функция, 33
- after_app_request, обработчик, 244
- after_request, обработчик, 34
- app_errorhandler, декоратор, 103, 211
- before_app_request, обработчик, 131
- before_first_request, обработчик, 34
- before_request, обработчик, 34, 214
- cookies, 190
- current_app, переменная контекста, 33, 107
- debug, аргумент, 28
- errorhandler, декоратор, 49, 102, 219
- flash, функция, 67
- flask.ext, пространство имен, 38, 46
- get_flashed_messages, функция шаблонов, 67
- g, переменная контекста, 33, 35
- jsonify, функция, 209
- make_response, функция, 35, 191
- methods, аргумент, 62
- redirect, функция, 36, 66
- render_template, функция, 41, 66
- request, переменная контекста, 33
- Response, класс, 35
- route, декоратор, 33, 101
- run, метод, 28
- session, переменная контекста, 33, 66
- set_cookie, функция, 36, 191
- SQLALCHEMY_DATABASE_URI, параметр настройки, 99
- static, папка, 53
- teardown_request, обработчик, 35
- templates, папка, 41
- url_for, функция, 52, 66, 103, 130
- url_prefix, аргумент, 115
- динамические маршруты, 27
- контексты, 31, 107
- макеты, 101, 114, 209
- остановка сервера, 237
- процессор контекста, 141

реестр расширений, 266

ссылки, 52

статические файлы, 53

тестовый клиент, 231

фабричная функция приложения, 100

Flask-Bootstrap, 46

quick_form, макрос, 61

блоки, 48

Flask-HTTPAuth, 212

Flask-Login, 115

AnonymousUserMixin, класс, 140

current_user, переменная, 119

LoginManager, класс, 117

login_required, декоратор, 117, 133

login_user, функция, 120

logout_user, функция, 121

user_loader, декоратор, 117

UserMixin, класс, 116

Flask-Mail, 91

асинхронная отправка

электронной почты, 95

интеграция поддержки электронной почты в приложение, 93

настройка подключения к Gmail, 92

Flask-Migrate, 87

Flask-Moment, 54

format, метод, 55

fromNow, метод, 55

lang, метод, 56

Flask-PageDown, 167

PageDownField, класс, 168

Flask-Script, расширение, 37

Flask-SQLAlchemy, 74

add, метод сеанса, 81, 82

create_all, метод, 80

delete_all, метод, 80

delete, метод сеанса, 82

filter_by, фильтр запроса, 85

get_debug_queries, функция, 244

paginate, метод, 164

Pagination, класс, 164

query, объект, 82

SECRET_KEY, параметр настройки, 99

SQLALCHEMY_COMMIT_ON_TEARDOWN, параметр, 75

SQLALCHEMY_DATABASE_URI, параметр, 75

модели, 75

настройка MySQL, 74

настройка Postgres, 74
настройка SQLite, 74
настройки столбцов, 77
методы выполнения запросов, 84
типы столбцов, 76
фильтры запросов, 83
Flask-SSLify, 257
Flask-WTF, 57, 167
BooleanField, класс, 59, 118
CSRF Cross-Site Request Forgery, 57
DateField, класс, 59
DateTimeField, класс, 59
DecimalField, класс, 59
Email, валидатор, 118
EqualTo, валидатор, 124
FieldList, класс, 59
FileField, класс, 59
FloatField, класс, 59
FormField, класс, 59
Form, класс, 58
HiddenField, класс, 59
IntegerField, класс, 59
PasswordField, класс, 59, 118
RadioField, класс, 59
Regexp, валидатор, 124
Required, валидатор, 59
SelectField, класс, 59
SelectMultipleField, класс, 59
StringField, класс, 59
SubmitField, класс, 59
TextAreaField, класс, 59
validate_on_submit, функция, 62, 120
валидаторы, 58, 60
нестандартные валидаторы, 124
отображение форм, 60
подделка межсайтовых запросов, 57
поля форм, 59
Flask, класс, 26
foreman, утилита, 251, 256
ForgeryPy, пакет, 161

G, H, I

Git, система управления версиями, 250
Gunicorn, веб-сервер, 254, 256
heroku, клиент, 251
Heroku, платформа, 250
HTTTPie, 224
HTTP, протокол, коды состояния, 210
itsdangerous, 126, 215

J

JavaScript Object Notation (JSON),
формат записи объектов JavaScript, 207
сериализация, 217

Jinja2, 22, 41
block, директива, 44, 47
capitalize, фильтр, 43
extends, директива, 45, 47
for, директива, 43
if, директива, 43, 61
import, директива, 44, 61
include, директива, 44
lower, фильтр, 43
macro, директива, 44
safe, фильтр, 43
set, директива, 200
striptags, фильтр, 43
super, макрос, 45
title, фильтр, 43
trim, фильтр, 43
upper, фильтр, 43
наследование шаблонов, 44
переменные, 42
фильтры, 42

M

manage.py, 98, 104, 262
coverage, команда, 228
db, команда, 88
deploy, команда, 247, 256
profile, команда, 246
runserver, команда, 38
shell, команда, 38, 86
test, команда, 107
Markdown, 167, 168

P

PaaS (Platform as a Service –
платформа как услуга), 249
PageDown, 167
pip, утилита, 25
Post/Redirect/Get, шаблон, 65
Procfile, файл, 255

R

requirements.txt, 98, 105
requirements.txt, файл, 255
REST (Representational State Transfer),
архитектурный стиль, 204
Rich Internet Application (RIA), полно-
функциональные
интернет-приложения, 204

S, T, U, V, W

Selenium, 237
syslog, 263
Twitter Bootstrap, 45

unittest, 106
 uWSGI, веб-сервер, 254
 virtualenv, утилита, 23
 activate, команда, 24
 deactivate, команда, 25
 Werkzeug, 110

А

Аватары, 152
 Аутентификация, 212, 214

Б

Базы данных
 filter_by, фильтр запросов, 188
 join, фильтр запросов, 188
 NoSQL, 71
 SQL, 70
 ассоциативные таблицы, 177
 миграция, 87
 отношения, 78, 84, 176, 195
 самоссылочные, 179
 производительность, 243
 реляционная модель, 70
 соединения, 186
 Блоггинг, 156

В

Виджет постраничного
 отображения, 164
 Виртуальные окружения, 23
 Всплывающие сообщения, 67

Д, Ж, И

Декораторы, 141
 Журналирование, 244, 248, 253,
 260, 263
 Интегрированная среда
 разработки, 264

К

Коды состояния HTTP, 210
 Конфигурация, 248, 262

М

«Многие к одному», отношения, 177
 «Многие ко многим», отношения, 177

О

Облако, 249
 Обработка ошибок, 210
 Обработка текста на сервере, 169
 «Один к одному», отношения, 177
 «Один ко многим», отношения, 177

Отладка, 248
 Охват кода тестированием, 227

П

Пароли, защищенность, 111
 Платформа как услуга (Platform as
 a Service, PaaS), 249
 Постоянные ссылки,
 на сообщения, 171
 Привилегии, 135
 Прикладные программные интерфейсы
 поддержка версий, 208
 ресурсы, 205, 220
 Производительность, 245
 Прокси-серверы, 259
 Профили пользователей, 143
 Профилирование, 245, 246
 исходного кода, 245

Р

Разбивка коллекций ресурсов
 на страницы, 223
 Редактирование профилей, 147
 на уровне администратора, 149
 на уровне пользователя, 147
 Роли пользователей, 135

С

Сообщения
 отправка и отображение, 156
 постоянные ссылки, 171
 постраничный вывод, 160
 фиктивные, для отладки, 161
 Списки, постраничный вывод, 160

Т

Тестирование, 224, 227
 веб-приложений, 231
 веб-служб, 235
 модульное тестирование, 113
 модульные тесты, 106, 141

Ф

Файл зависимостей, 105
 Формат записи объектов JavaScript
 (JSON), 207
 сериализация, 217
 Фрагменты URL, 198

Х, Э

Хэширование паролей, 111
 Электронная почта, 254

Об авторе

Мигель Гринберг (Miguel Grinberg) имеет более чем 25-летний опыт разработки программного обеспечения. На своей работе он возглавляет группу инженеров, занимающихся созданием программного обеспечения обработки видео для индустрии средств вещания. Мигель ведет свой блог (<http://blog.miguelgrinberg.com>), где пишет статьи на разные темы, в том числе о разработке веб-приложений, робототехнике, фотографии, и иногда оставляет обзоры фильмов. Он живет в городе Портленде, штат Орегон, со своей женой, четырьмя детьми, двумя собаками и кошкой.

Выходные данные

На обложке книги «Веб-фреймворк Flask» изображен пиренейский мастиф (вид *Canis lupus familiaris* – собака). Эти гигантские испанские собаки ведут свою родословную от древней римской сторожевой собаки (Molossus), выведенной греками и римлянами и ныне вымершей. Однако, как известно, римская сторожевая также является предком для многих других пород собак, распространенных в наши дни, таких как ротвейлер, датский дог, ньюфаундленд и итальянская сторожевая (Cane Corso). Пиренейские мастифы были признаны самостоятельной породой в 1977 году. Эта порода догов популяризуется в США американским клубом владельцев пиренейских мастифов (Pyrenean Mastiff Club of America) как домашнее животное.

После гражданской войны в Испании¹ популяция пиренейских мастифов на их родине резко сократилась, и порода сохранилась только благодаря усилиям нескольких разобщенных заводчиков. Современный генофонд пиренейцев берет начало в этой немногочисленной послевоенной популяции, что обуславливает склонность к генетическим заболеваниям, таким как дисплазия тазобедренного сустава. В настоящее время наиболее ответственные владельцы тщательно проверяют своих собак на наличие заболеваний и подвергают их рентгеновскому исследованию на наличие аномалий развития тазобедренного сустава, прежде чем заводить от них потомство.

Взрослые кобели пиренейского мастифа могут достигать в весе 90 килограммов, поэтому владение этой собакой накладывает на владельца обязательства по дрессировке и выгулу. Несмотря на размеры и историю происхождения (эта порода выводилась как охотничья, для охоты на медведей и волков), пиренейцы отличаются очень спокойным характером и являются отличными семейными собаками. Они очень дружелюбны по отношению к детям и являются верными охранниками дома, в то же время они достаточно равнодушны по отношению к другим собакам. При надлежащей социализации и строгом воспитании пиренейские мастифы прекрасно чувствуют себя в домашней обстановке и являются отличными опекунами и друзьями.

Изображение для обложки взято из книги «Animate Creation», написанной Вудом Джоном Джорджем (Wood, J. G.). Текст на обложке набран шрифтами URW Typewriter и Guardian Sans. Текст книги набран шрифтом Adobe Minion Pro; заголовки набраны шрифтом Adobe Myriad Condensed; программный код набран шрифтом Ubuntu Mono Далтона Мара (Dalton Maag).

¹ 17 июля 1936 г. – 1 апреля 1939 г. – *Прим. перев.*

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:

115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.aliants-kniga.ru**.

Оптовые закупки: тел. **(499) 782-38-89**.

Электронный адрес: **books@aliants-kniga.ru**.

Мигель Гринберг

Разработка веб-приложений с использованием Flask на языке Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод *Киселев А. Н.*

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16 .

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 17. Тираж 200 экз.

Веб-сайт издательства: www.dmk.ru