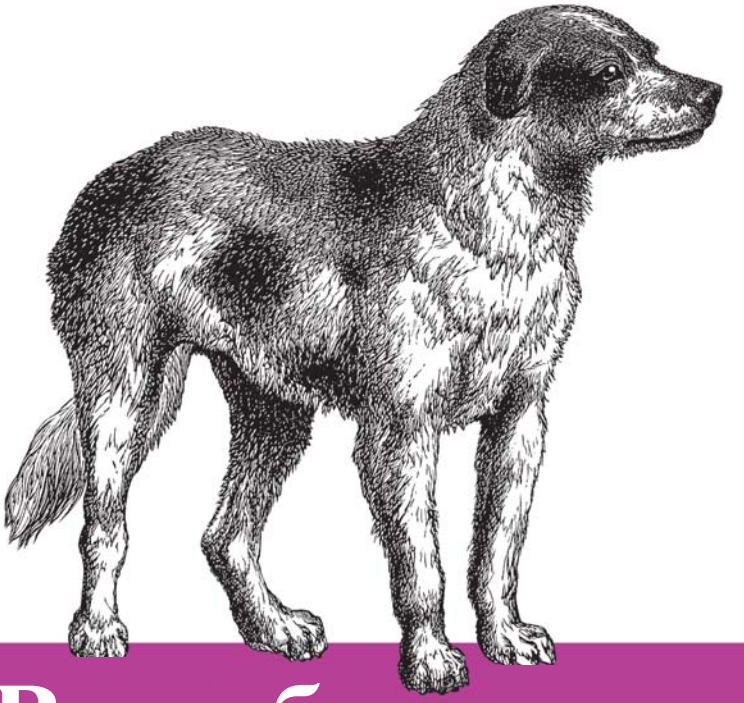

Разработка веб-приложений с использованием Flask на языке Python

Мигель Гринберг

ДМК ПРЕСС издательство

O'REILLY®

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 анг. А. Н. Киселева. – М.: ДМК Пресс, 2016. – 272 с.: ил.

ISBN 978-5-97060-206-5

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

ISBN 978-1-449-37262-0 (анг.)

ISBN 978-5-97060-206-5 (рус.)

Содержание

Глава 1

Установка

Flask[1] – это очень маленький фреймворк, такой маленький, что его часто называют «микрофреймворк». Он настолько мал, что после непродолжительного 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 пактов, 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 `pyvenv`. Команда `pyvenv` может использоваться вместо утилиты `virtualenv`, но имейте в виду, что виртуальные окружения, созданные с помощью `pyvenv` из установки Python 3.3, не включают утилиту `pip`, которую необходимо установить вручную. Это ограничение устранено в Python 3.4, где `pyvenv` 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 утилиту `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.

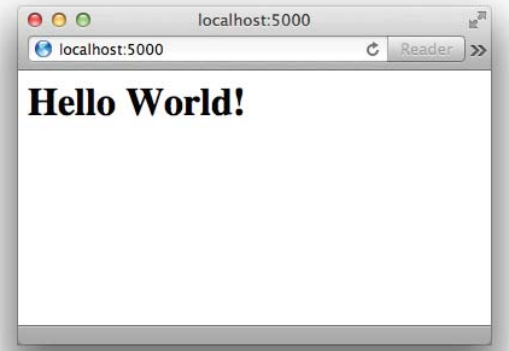

Рис. 2.1 ❖ Flask-приложение hello.py

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.

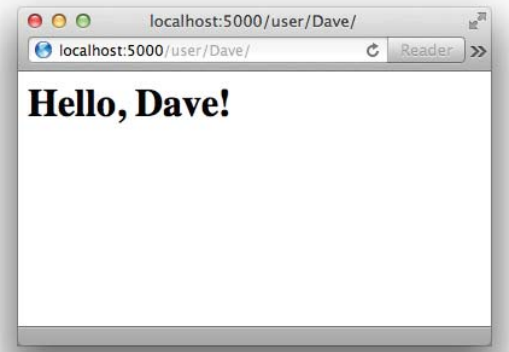

Рис. 2.2 ❖ Динамический маршрут

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
`current_app`	Контекст приложения	Экземпляр активного приложения
`g`	Контекст приложения	Объект, который может использоваться приложением как временное хранилище в процессе обработки запроса. Эта переменная сбрасывается в исходное состояние для каждого нового запроса
`request`	Контекст запроса	Объект запроса, включающий содержимое HTTP-запроса, отправленного клиентом
`session`	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 *переадресацией* (*redirect*). Этот ответ не включает документ страницы; он просто определяет новый адрес URL для браузера, откуда следует загрузить новую страницу. Прием переадресации часто используется в веб-формах, с которыми вы познакомитесь в главе 4.

Обычно ответ-переадресация 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 переадресации 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:
```