# Flask Documentation

*Выпуск 0.9 russian*

Armin Ronacher

мар. 09, 2018

# Оглавление

URL этой страницы

**Краткое содержание:** Попытка «человеческого» (не машинного, не кальки с английского) перевода документации к пакету Flask (язык программирования Python 2 и 3), выполненого с учётом синтаксиса русского языка. Приглашаем принять участие в работе по исправлению неизбежных ошибок и неточностей. Перевод выполняется как хобби в свободное время, с целью посильного вклада в развитие и продвижение СПО. При копировании ссылка на первоисточник обязательна.

**Состояние перевода: на 25.03.2014 выполнено 52.0%, начат переход на версию 0.10**.

Замечания и предложения можно оставить здесь: https://bitbucket.org/ferm32/flask-russian-docs/issues

либо отправить на почту ferm32 животное гмыло дот ком

Существенная часть перевода выполнена Владимиром Ступиным


Добро пожаловать в документацию по Flask. Она разбита на части. Мы рекомендуем начать с раздела *Инсталяция*, затем заглянуть в *Быстрый старт*. Помимо раздела «Быстрый старт», есть более детальное руководство - *Учебник*, по которому можно создать вместе с Flask хоть и небольшое, но полноеценное приложение. Если вы хотите погрузиться во «внутренности» Flask, преверьте документацию по API api. Общие шаблоны описаны в разделе *Заготовки для Flask*.

Flask требует наличия двух внешних библиотек: шаблонизатора Jinja2 и инструментария WSGI Werkzeug. Данные библиотеки здесь не рассматриваются. Если вы желаете углубиться в их документацию, воспользуйтесь следующими ссылками:

- Документация по Jinja2
- Документация по Werkzeug

Глава 1

# Руководство пользователя

Эта, по большей части скучная, часть документации начинается с некой справочной информации о Flask, затем шаг за шагом фокусируется на инструкциях по веб-разработке с Flask.

## 1.1 Предисловие

Перед началом работы с Flask прочтите это. Мы надеемся, вы получите ответы на некоторые вопросы о задачах и назначении проекта, а также о том, когда вам следует, а когда не следует его использовать.

### 1.1.1 Что означает «микро»?

“Микро” не означает, что ваше веб-приложение целиком помещается в один файл с кодом на Python, хотя конечно же это может быть и так. Также, это не означает. что Flask испытывает недостаток функциональности. «Микро» в слове «микрофреймворк» означает, что Flask стремится придерживаться простого, но расширяемого ядра. Flask не может за вас решить многие вещи, например, какую базу данных использовать. А те решения, которые он может принять, например, который из движков для работы с шаблонами использовать, легко изменить. Всё остальное зависит от вас, таким образом, может оказаться, что Flask - всё что вам нужно, и ничего лишнего.

По умолчанию, Flask не включает уровень абстракции баз данных, валидации форм или каких-то иных, для чего уже существуют различные занимающиеся этим библиотеки. Вместо этого, Flask поддерживает расширения для добавления подобной функциональности в ваше приложение, таким образом, как если бы это было реализовано в самом Flask. Многочисленные расширения обеспечивают интеграцию с базами данных, валидацию форм, обработку загрузок на сервер, различные открытые технологии аутентификации и так далее. Flask может быть «микро», но при этом он готов для использования в реальных задачах для самых разнообразных нужд.

### 1.1.2 Конфигурация и Соглашения

Flask имеет много параметров конфигурации с разумными значениями по умолчанию, и мало соглашений перед тем, как начать. По соглашению, шаблоны и статические файлы хранятся в поддиректориях внутри дерева исходных текстов на Python, с названиями *templates* и *static* соответственно. Хотя это можно и поменять, обычно этого делать не стоит, особенно в самом начале работы.

### 1.1.3 Расти вместе с Flask

Однажды, скачав и запустив Flask, вы найдёте в сообществе множество доступных его расширений, для их интеграции в ваш реально работающий проект. Команда разработки ядра Flask рассматривает расширения и гарантирует, что одобренные расширения не перестанут работать с будущими релизами.

Пока ваша кодовая база растёт, вы вольны принимать уместные для вашего проекта решения. Flak продолжит обеспечивать очень простой слой для склейки тем наилучшим образом, каким это может предложить Python. Вы можете реализовать расширенные шаблоны в SQLAlchemy или в ином средстве для работы с БД, в необходимых случаях применить надёжные нереляционные базы данных и использовать преимущества изначально не предназначенных для работы с фреймворками средств, построенных для работы с WSGI - веб интерфейсом для языка Python.

Flask включает множество крючков (хуков) для настройки его поведения. Если вам нужны дополнительные настройки, класс Flask построен, чтобы быть готовым к созданию подклассов. Если это вас заинтересовало, проверьте раздел becomingbig. Если вам любопытны принципы, в соответствии с которым спроектирован Flask, посмотрите раздел *Проектные решения во Flask*.

Продолжение: *Инсталяция*, *Быстрый старт*, или *Предисловие для опытных программистов*.

## 1.2 Предисловие для опытных программистов

### 1.2.1 Внутрипоточные объекты (Thread-Locals) во Flask

Одним из проектных решений для Flask было то, что простые задачи должны быть простыми; они не должны требовать большого объёма кода, а ещё они не должны ограничивать вас. Поэтому, при созданиее Flask были выбраны несколько решений, которые для некоторых людей могут показаться неожиданными или оргигинальными. Например, внутри себя Flask использует внутрипоточные объекты, поэтому вам в запросе, чтобы сохранять потокобезопасность (thread-safe), необязательно передавать объекты от функции к функции. Такой подход удобен, но требует действительныйо контекст запроса для внедрения зависимостей или при попытке повторного использования кода, который использует занчение, привязанное к запросу. Flask относится честно к внутрипоточным переменным, не скрывает их и всегда явно указывает в коде и в документации, где бы они не использовались.

### 1.2.2 Разработка для Веб с осторожностью

При создании веб-приложений всегда думайте о безопасности.

Если вы пишете веб-приложение, вы, вероятно, позволяете пользователям регистрироваться и оставлять их данные на сервере. Пользователи доверяют вам свои данные. И даже если вы являетесь единственным пользоваетелем, который может внести данные в приложение, всё равно вы хотите надёжно их сохранить.

К сожалению, существует множество способов, чтобы скомпрометировть защиту с точки зрения безопасности веб-приложения. Flask защищает вас от одной и наиболее распространённых проблем безопасности современных веб-приложений: от межсайтового скриптинга (cross-site scripting - XSS). Если

только вы не пометите небезопасный HTML-код как безопасный, вас прикроет Flask и нижележащий шаблонизатор Jinja2. Но существует и множество других способов вызвать проблемы с безопасностью.

Документация предупердит вас о тех аспектах веб-разработки, которые требуют внимания к безопасности. Некоторые из этих соображений являются гораздо более сложными, чем можно было бы подумать, и иногда все мы недооцениваем вероятность того, что уязвимость будет использована - а в это время толковый злоумышленник вычисляет способы для взлома наших приложений. Н надейтесь, что ваше приложение не настолько важно, чтобы привлечь злоумышленника. При некоторых видах атаки есть вероятность, что автоматизированные боты зондируют различные способы заполнения вашей базы данных спамом, ссылками на вредоносные программы и т. п.

Flask ничем не отличается от других фреймворков в плане того, что вы, как разработчик, при разработке в соответствии с вашей постановкой задачи, должны действовать с осторожностью, при этом следя за попытками взлома.

### 1.2.3 Статус Python 3

В настоящее время сообщество Python находится в процессе совершенствования библиотек для поддержки новой итерации языка программирования Python. Хотя ситуация и резко улучшается, есть ещё некоторые вопросы, из-за которых нам трудно переключиться на Python 3 прямо сейчас. Эти проблемы частично вызваны изменениями в языке, которые не были опробованы в течение достаточно продолжительно времени, частично те, что мы ещё не работали на нижем уровне с теми изменениями уровня API, которые внёс Python 3 в работу с Unicode.

Werkzeug и Flask будут портированы на Python 3, как только решение для подобных изменений будет найдено, и мы обеспечим полезными советами, как обновить существующие приложения для работы с Python 3. Пока этого не произошло, мы настоятельно рекомендуем использовать Python 2.6 и 2.7 с активированными на время разработки предупреждениями о несовместимости с Python 3. Если вы планируете в ближайшем будущем переход на Python 3, мы настоятельно рекомендуем прочитать вам статью Как писать на Python совместимый снизу вверх код.

Продолжение: *Инсталяция* или the *Быстрый старт*.

## 1.3 Инсталяция

Flask зависит от двух внешних библиотек, Werkzeug и Jinja2. Werkzeug - это инструментарий для WSGI, стандартного интерфейса Python между веб-приложениями и различными серверами, предназначенный как для разработки, так и развёртывания. Jinja2 занимается отображением шаблонов.

Итак, как же быстро получить всё необходимое на ваш компьютер? Есть много способов, которыми вы это можете проделать, но самый обалденный - это virtualenv, так что давайте глянем в первую очередь на него.

Для начала вам понадобится Python 2.5 или выше, так что убедитесь, что у вас инсталлирован свежий Python 2.x. На момент написания, спецификация WSGI для Python 3 ещё не была закончена, поэтому Flask не может поддерживать Python версий 3.x.

### 1.3.1 virtualenv

Virtualenv - это возможно, то самое, что вы захотите использовать во время разработки, и если у вас есть доступ к командной оболочке на рабочем сервере, возможно, вы захотите использовать его также и здесь.

Какую из пробем решает virtualenv? Если вам нравится Python так, как он нравится мне, скорее всего, вы захотите использовать его и в других проектах - вне приложений, созданных на базе Flask.

Но чем больше у вас проектов, тем больше вероятность, что вы будете работать с разными версиями самого Python, или по крайней мере с различными версиями библиотек Python. Посмотрим правде в глаза: довольно часто библиотеки нарушают обратную совместимость, и маловерояно, что серьёзное приложение будет работать вообще без каких-либо зависимостей. Так что же делать, если два или более из ваших проектов имеют конфликтующие зависимости?

Наше спасение - Virtualenv! Virtualenv предоставляет несколько соседствующих друг с другом установленных версий Python, по одному для каждого проекта. На самом деле он реально не устанавливает различные версии Python, но обеспечивает хитрый способ создать для проектов несколько изолированных друг от друга окружений. Давайте посмотрим, как работает virtualenv.

Если вы работаете с MacOS X или с Linux, есть вероятность, что заработает одна из следующих двух команд:

```
$ sudo easy_install virtualenv
```

или даже лучше:

```
$ sudo pip install virtualenv
```

Возможно, одна из этих команд установит virtualenv на вашей системе. А может оказаться. что это позволит сделать даже ваш пакетный менеджер. Если вы используете Ubuntu, попробуйте:

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

Если у вас Windows и команда *easy_ install* не работает, вам необходимо сначала установить её. Чтобы получить дополнительную информацию о том, как это можно сделать, проверьте раздел *pip и distribute в MS Windows*. Установив данную команду, запустите её, как указано чуть выше, но без префикса *sudo*.

После того, как virtualenv была установлена, просто запустите командный интерпретатор и создайте ваше собственное окружение. Обычно в таких случаях я создаю папку проекта, а в ней - папку *venv*:

```
$ mkdir myproject
$ cd myproject
$ virtualenv venv
New python executable in venv/bin/python
Installing distribute.............done.
```

Теперь, когда вы захотите работать над проектом, вам необходимо лишь активировать соответствующее окружение. Под MacOS X и Linux, выполните следующее:

```
$ . venv/bin/activate
```

Если вы используете Windows, для вас подойдёт следующая команда:

```
$ venv\scripts\activate
```

В любом случае, теперь вы должны использовать вашу virtualenv (обратите внимание, как изменилось приглашение вашей командной оболочки - для того, чтобы показать активное окружение).

Теперь, чтобы загрузить и активировать в вашем virtualenv Flask, вы можете просто ввести команду:

```
$ pip install Flask
```

Через несколько секунд вы сможете двинуться в дальнейший путь.

### 1.3.2 Установка непосредственно в систему

Возможен и такой вариант установки, но я бы вам его не рекомендовал. Просто запустите *pip* с привелегиями суперпользователя:

```
$ sudo pip install Flask
```

(Под Windows, запустите ту же команду, но только без *sudo*, внутри окна с командной строкой, запущенного с привелегиями администратора системы)

### 1.3.3 Жизнь на переднем краю

Если вы хотите работать с самой последней версией Flask, существует два пути: можно указать *pip*, чтобы он загрузил версию для разработки, или можно работать со срезом с текущего состояния репозитория git. В обоих случаях, рекомендуется пользоваться virtualenv.

Получите срез с последнего состояния git в новом окружении virtualenv и запустите в режиме разработки:

```
$ git clone http://github.com/mitsuhiko/flask.git
Initialized empty Git repository in ~/dev/flask/.git/
$ cd flask
$ virtualenv venv --distribute
New python executable in venv/bin/python
Installing distribute.............done.
$ . venv/bin/activate
$ python setup.py develop
...
Finished processing dependencies for Flask
```

Будет скачанан и активирована в virtualenv текущая версия, соответствующая последнему (головному) срезу из git. В дальнейшем, чтобы обновиться до последней версии, всё, что вам будет необходимо сделать - это выполнить `git pull origin`.

Чтоб получить версию для разработки без git, просто сделайте следующее:

```
$ mkdir flask
$ cd flask
$ virtualenv venv --distribute
$ . venv/bin/activate
New python executable in venv/bin/python
Installing distribute.............done.
$ pip install Flask==dev
...
Finished processing dependencies for Flask==dev
```

### 1.3.4 *pip* и *distribute* в MS Windows

В Window установить *easy_ install* немного сложнее, но всё равно довольно легко. Самый простой путь сделать это - скачав файл distribute_setup.py, запустить его. Простейший способ запустить файл - это открыть папку с загрузками и дважды щёлкнуть мышкой на файле.

Далее, добавьте команду папку с *easy_ install*, а также другими скриптами Python в список путей для поиска команд (переменная окружения *PATH*. Чтобы сделать это, нажмите правой кнопкой мыши на значёк «Мой компьютер» («Компьютер»), находящийся на рабочем столи или в меню «Пуск», и

выберите «Свойства». Далее нажмите «Дополнительные свойства системы» (в Windows XP закладка будет называться «Расширенные»). Далее, нажмите кнопку «Пепеменные среды».

Наконец, дважды кликните на переменной «Path» в секции «Системные переменные», и добавьте путь к папке со скриптами интерпретатора Python. Убидитесь, что вы отделили его от существующих значений точкой с запятой. Предпожим, что вы используете Python 2.7, установленный в папку по умолчанию, тогда добавьте следующее значение:

```
;C:\Python27\Scripts
```

Вот и всё! Чтоб проверить, что всё работает, откройте Командную строку и выполните `easy_install`. Если у вас Windows Vista или Windows 7, и при этом включен Контроль Учётных Записей (UAC), будут запрошены администраторские привилегии.

Теперь, когда у вас есть `easy_install`, вы можете использовать его для установки `pip`:

```
> easy_install pip
```

## 1.4 Быстрый старт

Рвётесь в бой? Эта страница даёт хорошее введие в Flask. Предполагается, что вы уже имеете установленный Flask. Исли это не так, обратитесь к секции *Инсталяция*.

### 1.4.1 Минимальное приложение

Минимальное приложение Flask выглядит примерно так:

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

@app.route('/')
def hello_world():
    return 'Hello World!'

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

Просто сохраните его как *hello.py* (или как-то похоже) и запустите его с помощью вашего интерпретатора Python. Только, пожалуйста, не называйте приложение как *flask.py*, потому что это вызовет конфликт с самим Flask.

```
$ python hello.py
 * Running on http://127.0.0.1:5000/
```

Проследовав по ссылке http://127.0.0.1:5000/ вы увидите ваше приветствие миру.

Итак, что же делает этот код?

1. Сначала мы импортировали класс `Flask`. Экземпляр этого класса и будет вашим WSGI-приложением. Первый аргумент - это имя модуля приложения. Если вы используете единственный модуль (как в этом примере), вам следует использовать *__name__*, потому что в зависимости от того, запущен ли код, как приложение, или был импортирован как модуль, имя будет различным (`'__main__'` или актуальное имя импортированного модуля соответственно. Для дополнительной информации, посмотрите документацию `Flask`.

2. Далее мы создаём экземпляр класса. Мы передаём ему имя модуля или или пакета. Это необходимо, чтобы Flask знал, где искать шаблоны, статические файлы и так далее.
3. Далее, мы используем декоратор `route()`, чтобы сказать Flask, какой из URL должен запускать нашу функцию.
4. Функция, которой дано имя, используемое также для генерации URL-адресов для этой конкретной функции, возвращает сообщение, которое мы хотим отобразить в браузере пользователя.
5. Наконец, для запуска локального сервера с нашим приложением, мы используем функцию `run()`. Проверка `if __name__ == '__main__':` используется, чтобы удостовериться, что сервер запускается только при запуске скрипта непосредственно из интерпретатора Python, а не через импорт модуля.

Для остановки сервера, нажмите Ctrl+C.

---

**Публично доступный сервер**

Если вы запустите сервер, вы заметите, что он доступен только с вашего собственного компьютера, а не с какого-либо другого в сети. Так сделано по умолчанию, потому что в режиме отладки пользователь приложения может выполнить код на Python на вашем компьютере.

Если у вас отключена опция *debug* или вы доверяете пользователям в сети, вы можете сделать сервер публично доступным, просто изменив вызов метода `run()` таким вот образом:

```
app.run(host='0.0.0.0')
```

Это укажет вашей операционной системе, чтобы она слушала сеть со всех публичных IP-адресов.

---

### 1.4.2 Режим отладки

Метод `run()` чудесно подходит для запуска локального сервера для разработки, но вы будете должны перезапускать его всякий раз при изменении вашего кода. Это не очень здорово, и Flask здесь может облегчить жизнь. Если вы включаете поддержку отладки, сервер перезагрузит сам себя при изменении кода, кроме того, если что-то пойдёт не так, это обеспечит вас полезным отладчиком.

Существует два способа включить отладку. Или установите флаг в объекте приложения:

```
app.debug = True
app.run()
```

Или передайте его как параметр при запуске:

```
app.run(debug=True)
```

Оба метода вызовут одинаковый эффект.

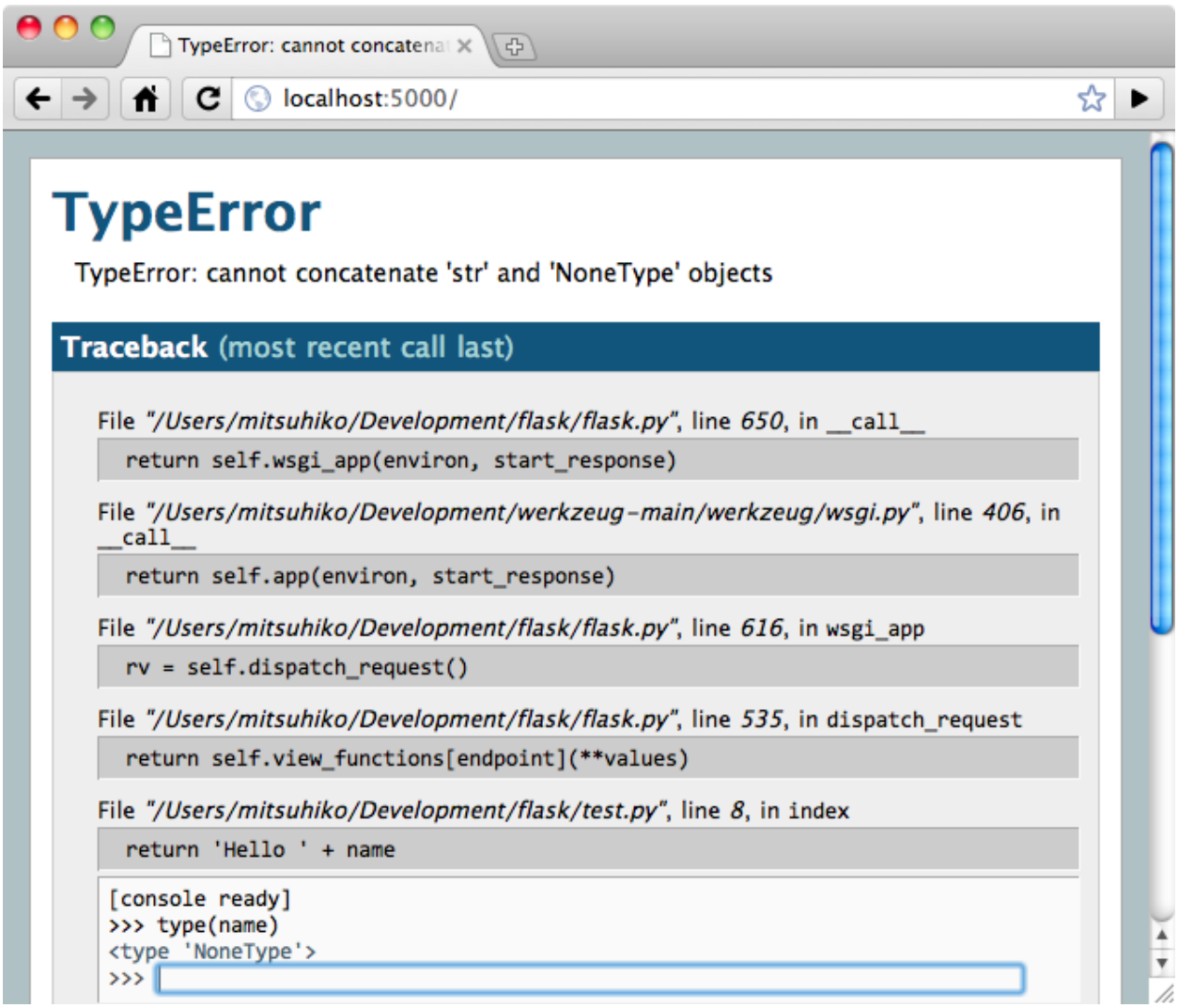
---

**Внимание.**

Несмотря на то, что интерактивный отладчик не работает в многопоточных окружениях (что делает его практически неспособным к использованию на реальных рабочих серверах), тем не менее, он позволяет выполнение произвольного кода. Это делает его главной угрозой безопасности, и поэтому **он никогда не должен использоваться на реальных «боевых» серверах**.

---

Снимок экрана с отладчиком в действии:




Предполагаете использовать другой отладчик? Тогда смотрите *Работа с отладчиками*.

### 1.4.3 Маршрутизация

Современные веб-приложения используют «красивые» URL. Это помогает людям запомнить эти URL, это особенно удобно для приложений, используемых с мобильных устройств с более медленным сетевым соединением. Если пользователь может перейти сразу на желаемую страницу, без предварительного посещения начальной страницы, он с большей вероятностью вернётся на эту страницу и в следующий раз.

Как вы увидели ранее, декоратор `route()` используется для привязки функции к URL. Вот простейшие примеры:

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello World'
```

Но это еще не все! Вы можете сделать определенные части URL динамически меняющимися и задействовать в функции несколько правил.

### Правила для переменной части

Чтобы добавлять к адресу URL переменные части, можно эти особые части выделить как `<variable_name>`. Подобные части затем передаются в вашу функцию в качестве аргумента - в виде ключевого слова. Также может быть указан конвентер, с помощью задния правила в виде `<converter:variable_name>`. Вот несколько интересных примеров

```
@app.route('/user/<username>')
def show_user_profile(username):
    # показать профиль данного пользователя
    return 'User %s' % username

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # вывести сообщение с данным id, id - целое число
    return 'Post %d' % post_id
```

Существуют следующие конвертеры:

| | |
|---|---|
| *int* | принимаются целочисленные значения |
| *float* | как и *int*, только значения с плавающей точкой |
| *path* | подобно поведению по умолчанию, но допускаются слэши |

**Уникальные URL / Перенаправления**

Правила для URL, работающие в Flask, основаны на модуле маршрутизации Werkzeug. Этот модуль реализован в соответствие с идеей обеспечения красивых и уникальных URL-адресов на основе исторически попавшего в обиход из поведения Apache и более ранних HTTP серверов.

Возьмём два правила:

```
@app.route('/projects/')
def projects():
    return 'The project page'

@app.route('/about')
def about():
    return 'The about page'
```

Хоть они и выглядят довольно похожими, есть разница в использовании слэша в *определении* URL. В первом случае, канонический URL имеет завершающую часть *projects* со слэшем в конце. В этом смысле он похож на папку в файловой системе. В данном случае, при доступе к URL без слэша, Flask перенаправит к каноническому URL с завершающим слэшем.

Однако, во втором случае, URL определен без косой черты - как путь к файлу на UNIX-подобных системах. Доступ к URL с завершающей косой чертой будет приводить к появлению ошибки 404 «Not Found».

Такое поведение позволяет продолжить работать с родственными URL - в случаях, когда пользователь пытается получить доступ к странице, забыв указать в конце строки URL слэш, в соответствии с тем, как работают Apache и другие сервера. Кроме того, URL-адреса будут оставаться уникальными, что поможет поисковым системам избежать повторного индексирования страницы.

### Построение (генерация) URL

Раз Flask может искать соответствия в URL, может ли он их генерировать? Конечно, да. Для построения URL для специфической функции, вы можете использовать функцию `url_for()`. В качестве первого аргумента она принимает имя функции, кроме того она принимает ряд именованных аргументов, каждый из которых соответствует переменной части правила для URL. Неизвестные переменные части добавляются к URL в качестве параметров запроса. Вот некоторые примеры:

```
>>> from flask import Flask, url_for
>>> app = Flask(__name__)
>>> @app.route('/')
... def index(): pass
...
>>> @app.route('/login')
... def login(): pass
...
>>> @app.route('/user/<username>')
... def profile(username): pass
...
>>> with app.test_request_context():
...  print url_for('index')
...  print url_for('login')
...  print url_for('login', next='/')
...  print url_for('profile', username='John Doe')
...
/
/login
/login?next=/
/user/John%20Doe
```

(Здесь также использован метод `test_request_context()`, который будет объяснён ниже. Он просит Flask вести себя так, как будто он обрабатывает запрос, даже если мы взаимодействуем с ним через оболочку Python. Взгляните на нижеследующее объяснение. *Локальные объекты контекста (context locals)*).

Зачем Вам может потребоваться самим строить URL вместо того, чтобы их жёстко задать внутри ваших шаблонов? Для этого есть три веские причины:

1. По сравнению с жёстким заданием URL внутри кода обратный порядок часто является более наглядным. Более того, он позваляет менять URL за один шаг, и забыть про необходимость изменять URL повсюду. 2. Построение URL будет прозрачно для вас осуществлять экранирование специальных символов и данных Unicode, так что вам не придётся отдельно иметь с ними дела. 3. Если ваше приложение размещено не в корневой папке URL root (а, скажем, в `/myapplication` вместо `/`), данную ситуацию нужным для вас образом обработает функция `url_for()`.

### Методы HTTP

HTTP (протокол, на котором общаются веб-приложения) может использовать различные методы для доступка к URL-адресам. По умолчанию, route отвечает лишь на запросы типа *GET*, но это можно изменить, снабдив декоратор `route()` аргументом *methods*. Вот некоторые примеры:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

Если присутствует метод *GET*, то автоматически будет добавлен и *HEAD*. Вам не придётся иметь с ним дело. Также, при этом можно быть уверенным, что запросы *HEAD* будут обработаны в соответствии с требованиями HTTP RFC (документ с описанием протокола HTTP), так что вам не требуется ничего знать об этотй части спецификации HTTP. Кроме того, начиная с Flask версии 0.6, для вас будет автоматически реализован метод *OPTIONS* автоматически.

Не имеете понятия, что такое метод HTTP? Не беспокойтесь, здесь приводится быстрое введение в методы HTTP, и почему они важны:

HTTP-метод (также часто называемый командой) сообщает серверу, что хочеть сделать клиент с запрашиваемой страницей. Очень распространены Следующие методы:

***GET*** Браузер говорит серверу, чтобы он просто получил информацию, хранимую на этой странице, и отослал её. Возможно, это самый распространённый метод.

***HEAD*** Браузер просит сервер получить информацию, но его интересует только *заголовки*, а не содержимое страницы. Приложение предполагает обработать их так же, как если бы был получен запрос *GET*, но без доставки фактического содержимого. В Flask, вам вовсе не требуется иметь дело с этим методом, так как нижележащая библиотека Werkzeug сделает всё за вас.

***POST*** Бразузер говорит серверу, что он хочет сообщить этому URL некоторую новую информацию, и что сервер должен убедиться, что данные сохранены и сохранены в единожды. Обычно, аналогичным образом происходит передача из HTML форм на сервер данных.

***PUT*** Похоже на *POSt*, только сервер может вызвать процедуру сохранения несколько раз, перезаписывая старые значения более одного раза. Здесь вы можете спросить, зачем это нужно, и есть несколько веских причин, чтобы делать это подобным образом. Предположим, во время передачи произошла потеря соединения: в этой ситуации система между браузером и сервером, ничего не нарушая, может совершенно спокойно получить запрос во второй раз. С *POST* такое было бы невозможно, потому что он может быть вызван только один раз.

***DELETE*** Удалить информацию, расположенную в указанном месте.

***OPTIONS*** Обеспечивает быстрый способ выяснения клиентом поддерживаемых для данного URL методов. Начиная с Flask 0.6, это работает для вас автоматически.

Теперь самое интересное: в HTML 4 и XHTML1, единственными методами, которыми форма может отправить серверу данные, являются *GET* и *POST*. Но для JavaScript и будущих стандартов HTML вы также можете использовать и другие методы. Кроме того, в последнее время HTTP стал довольно популярным, и теперь браузеры уже не единственные клиенты, использующие HTTP. Например, его используют многие системы контроля версий.

### 1.4.4 Статические файлы

Динамеческие веб-приложение также нуждаются и в статических файлах. Обычно, именно из них берутся файлы CSS и JavaScript. В идеале ваш веб-сервер уже сконфигурирован так, чтобы обслуживать их для вас, однако в ходе разработке это также может делать и сам Flask. Просто создайте внутри вашего пакета или модуля папку с названием *static*, и она будет доступна из приложения как */static*.

Чтобы сформировать для статических файлов URL, используйте специальное окончание `'static'`:

```
url_for('static', filename='style.css')
```

Этот файл должен храниться в файловой системе как `static/style.css`.

### 1.4.5 Визуализация шаблонов

Генерация HTML из Pyhton - невесёлое и на самом деле довольно сложное занятие, так как вам необходимо самостоятельно заботиться о безопасности приложения, производя для HTML обработку специальных последовательностей (escaping). Поэтому внутри Flask уже автоматически преднастроен шаблонизатор Jinja2.

Для визуализации шаблона вы можете использовать метод `render_template()`. Всё, что вам необходимо - это указать имя шаблона, а также переменные в виде именованных аргументов, которые вы хотите передать движку обработки шаблонов:

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Flask будет искать шаблоны в папке *templates*. Поэтому, если ваше приложение выполнено в виде модуля, эта папка будет рядом с модулем, а если в виде пакета, она будет внутри вашего пакета:

**Первый случай** - модуль:

```
/application.py
/templates
    /hello.html
```

**Второй случай** - пакет:

```
/application
    /__init__.py
    /templates
        /hello.html
```

При работе с шаблонами вы можете использовать всю мощь Jinja2. За дополнительной информацией обратитесь к официальной Документации по шаблонам Jinja2

Вот пример шаблона:

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
  <h1>Hello {{ name }}!</h1>
{% else %}
  <h1>Hello World!</h1>
{% endif %}
```

Также, внутри шаблонов вы имеете доступ к обхектам `request`, `session` и `g` [#]_, а также к функции `get_flashed_messages()`.

Шаблоны особенно полезны при использовании наследования. Если вам интересно, как это работает, обратитесь к документации по заготовкам template-inheritance. Проще говоря, наследование шаблонов позволяет размезтить определённые элементы (такие, как заголовки, элементы навигации и «подвал» страницы) на каждой странице.

Автоматическая обработка специальных (escape-) последовательностей (escaping) включена по умолчанию, поэтому если *name* содержит HTML, он будет экранирован автоматически. Если вы можете доверять переменной и знаете, что в ней будет безопасный HTML (например, потому что он пришёл из модуля конвертирования разметки wiki в HTML), вы можете пометить её в шаблоне, как безопасную

- с использованием класса `Markup` или фильтра `|safe`. За дополнительными примерами обратитесь к документации по Jinja2.

Вот основные возможности по работе с классом `Markup`:

```
>>> from flask import Markup
>>> Markup('<strong>Hello %s!</strong>') % '<blink>hacker</blink>'
Markup(u'<strong>Hello &lt;blink&gt;hacker&lt;/blink&gt;!</strong>')
>>> Markup.escape('<blink>hacker</blink>')
Markup(u'&lt;blink&gt;hacker&lt;/blink&gt;')
>>> Markup('<em>Marked up</em> &raquo; HTML').striptags()
u'Marked up \xbb HTML'
```

Изменено в версии 0.5: Автоматическая обработка escape-последовательностей больше не активирована для всех шаблонов. Вот расширения шаблонов, которые активизируют автообработку: `.html`, `.htm`, `.xml`, `.xhtml`. Шаблоны, загруженные из строк, не будут обрабатывать специальные последовательности.

[#] Затрудняетесь понять, что это за объект - `g`? Это то, в чём вы можете хранить информацию для ваших собственных нужд, для дополнительной информации смотрите документацию на этот объект

(g) и sqlite3.

### 1.4.6 Доступ к данным запроса

Для веб-приложений важно, чтобы они реагировали на данные, которые клиент отправляет серверу. В Flask эта информация предоставляется глобальным объектом `request`. Если у вас есть некоторый опыт по работе с Python, вас может удивить, как этот объект может быть глобальным, и как Flask при этом умудрился остаться ориентированным на многопоточное выполнение.

#### Локальные объекты контекста (context locals)

**Информация от инсайдера**

Прочтите этот раздел, если вы хотите понять, как это работает, и как вы можете реализовать тесты с локльными переменными контекста. Если вам это неважно, просто пропустите его.

Некоторые объекты в Flask являются глобальными, но необычного типа. Эти объекты фактически являются прокси (посредниками) к объектам, локальным для конкретного контекста. Труднопроизносимо. Но на самом деле довольно легко понять.

Представьте себе контекст, обрабатывающий поток. Приходит запрос, и веб-сервер решает породить новый поток (или нечно иное - базовый объект может иметь дело с системой параллельного выполнения не на базе потоков). Когда Flask начинает осуществлять свою внутреннюю обработку запроса, он выясняте, что текущий поток является активным контекстом и связывает текущее приложение и окружение WSGI с этим контекстом (потоком). Он делает это с умом - так, что одно приложение может, не ломаясь, вызывать другое приложение.

Итак, что это означает для вас? В принципе, вы можете полностью игнорировать, что это так, если вы не делаете чего-либо вроди тестирования модулей. Вы заметите, что код, зависящий от объекта запроса, неожиданно будет работать неправильно, так как отсутствует объект запроса. Решением является самостоятельное создание объекта запроса и его привязка к контексту. Простейшим решением для тестирования модулей является импользование менеджера конекстов `test_request_context()`. В сочетании с оператором *with* этот менеджер свяжет тестовый запрос так, что вы сможете с ним взаимодействовать. Вот пример:

```
from flask import request

with app.test_request_context('/hello', method='POST'):
    # now you can do something with the request until the
    # end of the with block, such as basic assertions:
    assert request.path == '/hello'
    assert request.method == 'POST'
```

Другая возможность - это передача целого окружения WSGI методу `request_context()` method:

```
from flask import request

with app.request_context(environ):
    assert request.method == 'POST'
```

### Объект запроса

Объект запроса документирован в секции API, мы не будем рассматривать его здесь подробно (смотри `request`). Вот широкий взгляд на некоторые наиболее распространённые операции. Прежде всего, вам необходимо импортировать его из модуля *flask*:

```
from flask import request
```

В настоящее время метод запроса доступен через использование атрибута `method`. Для доступа к данным формы (данным, которые передаются в запросах типа *POST* или *PUT*), вы можете использовать атрибут `form`. Вот полноценный прмер работы с двумя упомянутыми выше атрибутами:

```
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'])
        else:
            error = 'Invalid username/password'
    # следущий код выполняется при методе запроса GET
    # или при признании полномочий недействительными
    return render_template('login.html', error=error)
```

Что произойдёт, если ключ, указанный в атрибуте *form*, не существует? В этом случае будет возбуждена специальная ошибка `KeyError`. Вы можете перехватить её подобно стандартной `KeyError`, но если вы этого не сделаете, вместо этого будет показана страница с ошибкой *HTTP 400 Bad Request*. Так что, во многих ситуаций вам не придётся иметь дело с этой проблемой.

Для доступа к параметрам, представленным в URL (`?ключ=значение`), вы можете использовать атрибут `args`:

```
searchword = request.args.get('key', '')
```

Мы рекомендуем доступ к параметрам внутри URL через *get* или через перехват *KeyError*, так как пользователь может изменить URL, а предъявление ему страницы с ошибкой 400 *bad request* не является дружественным.

За полным списком методов и атрибутов объекта запроса, обратитесь к следующей документации: `request`.

### Загрузка файлов на сервер

В Flask обработка загружаемых на сервер файлов является несложным занятием. Просто убедитесь, что вы в вашей HTML-форме не забыли установить атрибут `enctype="multipart/form-data"`, в противном случае браузер вообще не передаст файл.

Загруженные на сервер файлы сохраняются в памяти или во временной папке внутри файловой системы. Вы можете получить к ним доступ, через атрибут объекта запроса `files`. Каждый загруженный файл сохраняется в этом словаре. Он ведёт себя также, как стандартный объект Python `file`, однако он также имеет метод `save()`, который вам позволяет сохранить данный файл внутрь файловой системы сервера. Вот простой пример, показывающий, как это работает:

```
from flask import request

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/uploaded_file.txt')
    ...
```

Если вы хотите до загрузки файла в приложение узнать, как он назван на стороне клиента, вы можете просмотреть атрибут `filename`. Однако, имейте в виду, что данному значению никогда не стоит доверять, потому что оно может быть подделано. Если вы хотите использовать им файла на клиентской стороне для сохранения файла на сервере, пропустите его через функцию `secure_filename()`, которой вас снабдил Werkzeug:

```
from flask import request
from werkzeug import secure_filename

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/' + secure_filename(f.filename))
    ...
```

Некоторые более удачные примеры можно найти в разделе заготовок: *Загрузка файлов*.

### Cookies

Для доступа к cookies можно использовать атрибут `cookies`. Для установки cookies можно использовать метод объектов ответа `set_cookie`. Атрибут объектов запроса `cookies` - это словать со всеми cookies, которые передаёт клиент. Если вы хотите использовать сессии, то не используйте cookies напрямую, вместо этого используйте во Flask :ref:‘sessions, который при работе с cookies даст вам некоторую дополнительную безопасность.

Чтение cookies:

```
from flask import request

@app.route('/')
def index():
    username = request.cookies.get('username')
    # Чтобы не получить в случае отсутствия cookie ошибку KeyError
    # используйте cookies.get(key) вместо cookies[key]
```

Сохранение cookies:

```
from flask import make_response

@app.route('/')
def index():
    resp = make_response(render_template(...))
    resp.set_cookie('username', 'the username')
    return resp
```

Заметьте, что cookies устанавливаются в объектах ответа. Так как вы обычно просто возвращаете строки из функций представления, Flask конвертирует их для вас в объекты ответа. Если вы это хотите сделать явно, то можете использовать функцию, `make_response()`, затем изменив её.

Иногда вы можете захотеть установить cookie в точке, где объект ответа ещё не существует. Это можно сделать, использовав заготовку deferred-callbacks.

Также Об этом можно почитать здесь *Об ответах*.

### 1.4.7 Ошибки и перенаправления

Чтобы перенаправить пользователя в иную конечную точку, используйте функциюе `redirect()`; для того, чтобы преждевременно прервать запрос с кодом ошибки, используйте функцию `abort()` function:

```
from flask import abort, redirect, url_for

@app.route('/')
def index():
    return redirect(url_for('login'))

@app.route('/login')
def login():
    abort(401)
    this_is_never_executed()
```

Это довольно бессмысленный пример, потому что пользователь будет перенаправлен с индексной страницы на страницу, на которую у него нет доступа (401 означает отказ в доступе), однако он показывает, как это работает.

По умолчанию, для каждого кода ошибки отображается чёрно-белая страница с ошибкой. Если вы хотите видоизменить страницу с ошибкой, то можете использовать декоратор `errorhandler()`:

```
from flask import render_template

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

Обратите внимание на 404 после вызова `render_template()`. Это сообщит Flask, что код статуса для этой страницы должен быть 404, что означает «не найдено». По умолчанию предполагается код «200», который означает «всё прошло хорошо».

### 1.4.8 Об ответах

Возвращаемое из функции представления значение автоматически для вас конвертируется вас в объект ответа. Если возвращаемое значение является строкой, оно конвертируется в объект ответа в строку в

виде тела ответа, код ошибки `200 OK` и в mimetype со значением `text/html`. Логика, которую применяет Flask для конвертации возвращаемых значений в объекты ответа следующая:

1. Если возвращается объект ответа коррекного типа, он прямо возвращается из представления.
2. Если это строка, создаётся объект ответа с этими же данными и параметрами по умолчанию.
3. Если возвращается кортеж, его элементы могут предоставлять дополнительную информацию. Такие кортежи должны соответствовать форме `(ответ, статус, заголовки)`, кортеж должен содержать как минимум один из перечисленных элементов. Значение *статус* заменит код статуса, а элемент *заголовки* может быть или списком или словарём с дополнительными значениями заголовка.
4. Если ничего из перечисленного не совпало, Flask предполагает, что возвращаемое значение - это допустимая WSGI-заявка, и конвертирует его в объект ответа.

Если вы хотите в резуельтате ответа заполучить объект внутри представления, то можете использовать функцию `make_response()`.

Представим, что вы имеете подобное представление:

```
@app.errorhandler(404)
def not_found(error):
    return render_template('error.html'), 404
```

Вам надо всего лишь обернуть возвращаемое выражение функцией `make_response()` и получить объект результата для его модификации, а затем вернуть его:

```
@app.errorhandler(404)
def not_found(error):
    resp = make_response(render_template('error.html'), 404)
    resp.headers['X-Something'] = 'A value'
    return resp
```

### 1.4.9 Сессии

В дополнение к объекту ответа есть ещё один объект, называемый `session`, который позволяет вам сохранять от одного запроса к другому информацию, специфичную для пользователя. Это реализовано для вас поверх cookies, при этом используется крипотографическая подпись этих cookie. Это означает, что пользователь может посмотреть на содержимое cooke, но не может ничего в ней изменить, если он конечно не знает значение секретного ключа, использованного для создания подписи.

В случае использования сессий вам необходимо установить значение этого секретного ключа. Вот как работают сессии:

```
from flask import Flask, session, redirect, url_for, escape, request

app = Flask(__name__)

@app.route('/')
def index():
    if 'username' in session:
        return 'Logged in as %s' % escape(session['username'])
    return 'You are not logged in'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
```

```
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form action="" method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>
    '''

@app.route('/logout')
def logout():
    # удалить из сессии имя пользователя, если оно там есть
    session.pop('username', None)
    return redirect(url_for('index'))

# set the secret key.  keep this really secret:
app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
```

Упомянутая `escape()` осуществляет для вас обработку специальных последовательностей (escaping), что необходимо, если вы не используете движок шаблонов (как в этом примере).

---

**Как генерировать хорошие секретные ключи**

Проблемой случайных занчений является то, что трудно сказать, что является на является действительно случайным. А секретный ключ должен быть настолько настолько случайным, насколько это возможно. У вашей операционной системы есть способы для генерации весьма случайных значений на базе криптографического случайного генератора, который может быть использован для получения таких ключей:

```
>>> import os
>>> os.urandom(24)
'\xfd{H\xe5<\x95\xf9\xe3\x96.5\xd1\x01O<!\xd5\xa2\xa0\x9fR"\xa1\xa8'
```

Просто возьмите скопируйте/вставьте это в ваш код, вот и готово.

---

Замечание об сессиях на базе cookie: Flask возьмёт значения, которые вы помещаете в объект сессии, и сериализует их в cookie. Если вы обнаружили какие-либо значения, которые не сохраняются между запросами, а cookies реально включены, а никаких ясных сообщений об ошибках не было, проверьте размер cookie в ответах вашей страницы и сравните с размером, поддерживаемым веб-браузером.

### 1.4.10 Message Flashing

Хорошие приложения и интерфейсы пользователя дают обратную связь. Если пользователь не получает достаточной обратной связи, вскоре он может начать ненавидеть приложение. Flask предоставляет по-настоящему простой способ дать обратную связь пользователю при помощи системы всплывающих сообщений. Система всплывающих сообщений обычно делает возможным записать сообщение в конце запроса и получить к нему доступ во время обработки следующего и только следующего запроса. Обычно эти сообщения используются в шаблонах макетов страниц, которые его и отображают.

Чтобы вызвать всплывающие сообщения, используйте метод `flash()`, чтобы заполучить сообщения, можно использовать метод, также доступный для шаблонов - `get_flashed_messages()`. Полный пример приведён в разделе *Всплывающие сообщения*.

### 1.4.11 Ведение журналов

Добавлено в версии 0.3.

Иногда может возникнуть ситуация, в которой вы имеете дело с данными, которые должны быть корректными, но в действительности это не так. К примеру, у вас может быть некий код клиентской стороны, который посылает HTTP-запрос к серверу, однако он очевидным образом неверен. Это может произойти из-за манипуляции пользователя с данными, или из-за неудачной работы клиентского кода. В большинстве случаев ответом, адекватным ситуации будет `400 Bad Request`, но иногда, когда надо, чтобы код продолжал работать, это не годится.

Вы по-прежнему хотите иметь журнал того, что пошло не так. Вот где могут пригодиться объекты создания журнала *logger*. Начиная с Flask 0.3, инструмент для журналирования уже настроен для использования.

Вот некоторые примеры вызовов функции журналирования:

```
app.logger.debug('Значение для отладки')
app.logger.warning('Предупреждение: (%d яблок)', 42)
app.logger.error('Ошбибка')
```

Прилагаемый `logger` это стандартный класс журналирования `Logger`, так что за подробностями вы можете обратиться к официальной документации по журналированию.

### 1.4.12 Как зацепиться (hooking) к промежуточному слою WSGI

Если вы хотите добавить в ваше приложение слой промежуточного, или связующего для WSGI программного обеспечения (middleware), вы можете обернуть внутреннее WSGI-приложение. К примеру, если вы хотите использовать одно из middleware из пакета Werkzeug для обхода известных багов в lighttpd, вы можете сделать это подобным образом:

```
from werkzeug.contrib.fixers import LighttpdCGIRootFix
app.wsgi_app = LighttpdCGIRootFix(app.wsgi_app)
```

### 1.4.13 Развёртывание приложения на веб-сервере

Готовы к развёртыванию на сервере вашего нового приложения Flask? В завершение краткого руководства, вы можете немедленно развернуть приложение на одной из платформ хостинга, предоставляющих бесплатное размещение для малых проектов:

- Развёртывание приложения Flask на Heroku
- Развёртывание WSGI в dotCloud с специфическими для Flask замечаниями

Другие места, где можно разместить ваше приложение:

- Развёртывание приложения Flask на Webfaction
- Развёртывание приложения Flask в Google App Engine
- Общий доступ к локальному хосту с помощью Localtunnel

Если вы управляете собственными хостами и желаете разместиться у себя, смотрите раздел *Варианты развёртывания*.

## 1.5 Учебник

Хотите разрабатоть приложение на Python под Flask? Здесь вам даётся такой шанс - изучить как это делается на примере. В этом учебнике мы создадим простое приложение микроблога. Оно будет поддерживать одного пользователя, который сможет создавать чисто текстовые записи, без возможности подписки или комментирования, однако в ней будет всё, что необходимо для того, чтобы начать работу. Мы будем использовать Flask и SQLite в качестве базы данных, которая идёт в комплекте с Python, поэтому вам больше ничего не понадобится.

Если вам нужны все исходные тексты, чтобы не вводить их вручную либо для сравнения, посмотрите их здесь исходные текств примеров.

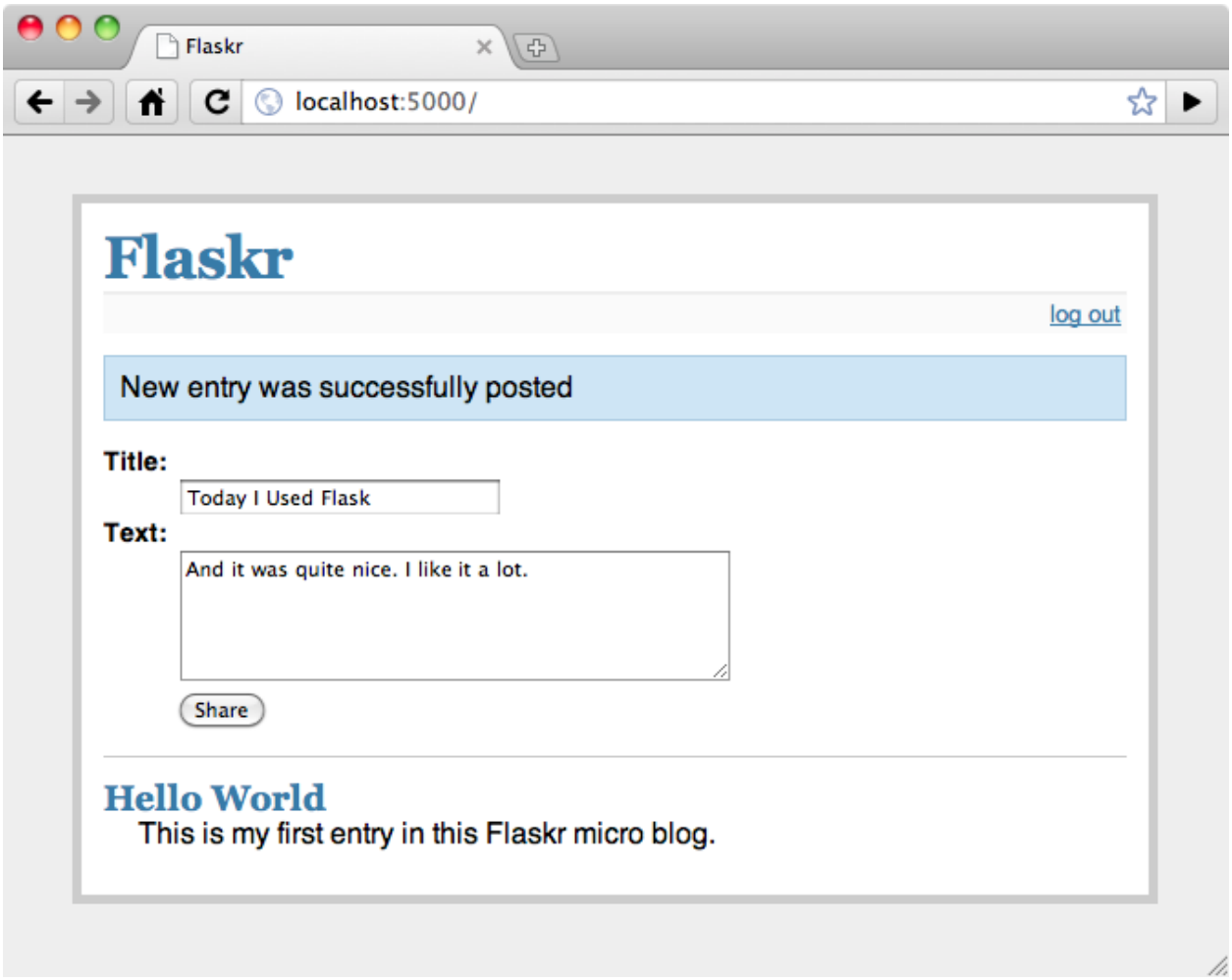
### 1.5.1 Введение в Flaskr

Здесь мы будем называть наше приложение для блога flaskr, а вы можете свободно выбрать менее оригинальное имя (прим. переводчика: очевидно, имеется ввиду шутливая аллюзия на название известного web-2.0 ресурса Flickr.com). ;) Вот те основные возможности, которые хотелось бы получить от данного приложения:

1. вход и выход пользователя с учётными данными, заданными в конфигурации. Требуется поддержка одного пользователя.
2. если пользователь вошёл, он может добавлять ана страницу новые записи, состоящие из чисто текстовых заголовков и некоторого HTML кода для самого текста. Мы полностью доверяем пользователю, поэтому этот HTML проверять не требуется.
3. на странице отображаются все записи до самой последней, в обратном порядке (свежие наверху), и пользователь после входа может добавлять новые прямо с этой же страницы.

Мы будем использовать для этого приложения прямые обращения к SQLite3, так как это хорошо подходит для приложения подобного размера. Однако, для более крупных приложений, имеет болшой смысл использовать SQLAlchemy, которая более разумным способом обрабатывает соединения с базой данных, что позволяет вам использовать одновременно различные целевые реляционные базы данных, а также многое другое. Также вы можете подумать над использованием одной из популярных не-SQL баз данных, если она лучше подходит для ваших данных.

Вот снимок экрана конечного варианта приложения:




Продолжение: *Шаг 0: Создание папок*.

### 1.5.2 Шаг 0: Создание папок

Перед тем, как мы начнём, давайте создадим папки, необходимые для нашего приложения:

```
/flaskr
    /static
    /templates
```

Папка *flaskr* не является пакетом Python, это просто некоме нместо, куда мы будем класть наши файлы. Далее, на следующих шагах, прямо в эту папку мы поместим - в добавок к главному модулю - схему нашей базы данных. Файлы внутри папки *static* доступны пользователям приложения по протоколу HTTP. Это место, куда попадут файлы css и javascript. В папке *templates* Flask будет искать шаблоны Jinja2. В эту папку будут помещены шаблоны, которые вы будете создавать в ходе работы с этим руководством.

Продолжение: *Шаг 1: Схема базы данных*.

### 1.5.3 Шаг 1: Схема базы данных

Сперва мы хотим создать схему базы данных. Это очень просто, потому что для этого приложения требуется единственная таблица, и мы хотим поддерживать только SQLite. Просто поместите следующий текст в файл с именем *schema.sql* в только что созданной папке *flaskr*:

```
drop table if exists entries;
create table entries (
  id integer primary key autoincrement,
  title string not null,
  text string not null
);
```

Эта схема состоит из единственной таблице с названием *entries*, и кажда запись в ней содержит поля *id*, *title* и *text*. *id* это автоинкрементируемое целое, которое одновременно является первичным ключом, а два других поля - это строки, которые не могут быть пустыми.

Продолжение: *Шаг 2: Код для настройки приложения*.

### 1.5.4 Шаг 2: Код для настройки приложения

Сейчас, когда наша схема уже на месте, мы можем создать модуль приложения. Назовём его *flaskr.py* и разместим в папке *flaskr*. Для начала мы добавим те импорты, которые нам потребуются, а также раздел конфигурации. Для небольших приложений есть возможность разместить конфигурацию прямо внтри модуля, что мы сейчас исделаем. Однако, более прозрачным решением было бы создание отдельного файла *.ini* или *.py* и загрузка или импорт значений оттуда.

В *flaskr.py*:

```
# все импорты
import sqlite3
from flask import Flask, request, session, g, redirect, url_for, \
     abort, render_template, flash

# конфигурация
DATABASE = '/tmp/flaskr.db'
DEBUG = True
SECRET_KEY = 'development key'
USERNAME = 'admin'
PASSWORD = 'default'
```

Далее, мы должны создать наше текущее приложение и инициализировать его в помощью конфигурации из того же файла, т. е. flaskr.py:

```
# создаём наше маленькое приложение :)
app = Flask(__name__)
app.config.from_object(__name__)
```

`from_object()` будет смотреть на данный объект (если это строка, он импортирует её) и затем будет искать все определённые здесь переменные, названные заглаавными буквами. В нашем случае, конфигурация, которую мы только что записали, это несколько вышеприведённых строк кода. Вы также можете перенести их в отдельный файл.

Обычно хорошей идеей является загрузка конфигурации из отдельного файла конфигурации. Это именно то, чем занимается метод `from_envvar()`, которым следует заменить вышеприведённю строчку `from_object()`:

```
app.config.from_envvar('FLASKR_SETTINGS', silent=True)
```

Таким образом, кто-то может установить переменную окружения с именем `FLASKR_SETTINGS`, тем самым указав, из какого файла будет загружена конфигурация, при этом она переопределит значения по умолчанию. Параметр-переключатель *silent* просто говорит Flask не ругаться, если вышеуказанная переменная окружения не задана.

*secret_key* необходим для обеспечения безопасности сессий на клиентской стороне. Выбирайте этот ключ с умом, и настолько трудный для разгадывания и сложный, насколько это возможно. Флаг DEBUG включает или отключает интерактивный отладчик. *Никогда не оставляйте режим отладки ативированным при реальном использовании системы*, потому что это позолит пользоватлям исполнять код на сервере!

Мы также добавили метод для простого соединения с указанной базой данных. Он может быть использован для открытия соединения по запросу, а также из интерактивной командной оболочки Python или из скрипта. Это пригодится в дальнейшем.

```
def connect_db():
    return sqlite3.connect(app.config['DATABASE'])
```

Наконец, мы просто добовляем строчку в конце файла, которая запускает сервер, если мы хотим запустить этот файл как отдельное приложение:

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

Отвлекаясь на минутку, приведём способ, которым достигается запуск приложения без каких-либо проблем - это делается с помощью следующих команд:

```
python flaskr.py
```

Вы увидете сообщение, которое сообщает вам о том, что сервер запущен, и адрес, по которому вы можете к нему получить доступ.

Когда вы зайдёте с помощью своего браузера на сервер, вы получите ошибку 404 (страница не найдена), так как у нас пока нет ни одного представления (view). Однако, мы сфокусируемся на этом чуть позже. Для начала мы должны добиться, чтобы заработала база данных.

---

**Сервер, доступный из внешнего мира**

Хотите, чтобы ваш сервер был публично доступным? Для получения дополнительной информации обратитесь к разделу *externally visible server*.

---

Продолжение: *Шаг 3: Создание базы данных*.

### 1.5.5 Шаг 3: Создание базы данных

Как было указано ранее, Flaskr это приложение с базой данных «под капотом», а если более точно, то приложение с системой на базе реляционной базы данных «под капотом». Такие системы нуждаются в схеме, котораясообщает им, как хранить информацию. Поэтому важно создать схему перед тем, как запустить в первый раз сервер.

Такая схема может быть создана с помощью перенаправления через канал (pipe) содержимого файла *schema.sql* команде *sqlite3* следующим образом:

```
sqlite3 /tmp/flaskr.db < schema.sql
```

Недостатком такого способа является необходимость наличия в системе команды sqlite3, которая в некоторых системах может отсутствовать. Также, здесь указан путь к базе данных, что является ещё одной возможностью наделать ошибок. Хорошей идеей было бы добавить функцию, которая иициализировала бы для вашего приложения базу данных.

Если вы хотите это сделать, сначала необходимо импортировать функцию `contextlib.closing()` из пакета contextlib. Если вы хотите использовать Python 2.5, то необходимо также сначала добавить оператор *with* (импорты из *__future__* должны быть самыми первыми импортами). Соответственно, добавьте в файле *flaskr.py* следующие строки к уже существующим импортам:

```
from __future__ import with_statement
from contextlib import closing
```

Далее, мы можем создать функцию с именем *init_db*, которая инициализирует базу данных. Для этого мы можем использовать функцию *connect_db*, которую мы определили ранее. Просто добавьте в *flaskr.py* после функции *connect_db* следующую функцию

```
def init_db():
    with closing(connect_db()) as db:
        with app.open_resource('schema.sql') as f:
            db.cursor().executescript(f.read())
        db.commit()
```

Функция-помощник (helper) `closing()` позволяет нам держать соединение открытым на протяжении блока *with*. Метод объекта приложения `open_resource()` поддерживает такую функциональность «из коробки», так что он может быть использован непосредственно внутри блока *with*. Эта функция открывает файл из папки, содержащей ресурсы (в нашем случае папка *flaskr*), и позволяет вам из него читать. Мы используем её здесь для того, чтобы выполнить скрипт при установлении соединения с базой данных.

Когда мы соединяемся с базой данных, мы получаем объект соединения (упоминается далее под именем *db*), который предоставляет нам курсор. У этого курсора есть метод для исполнения полноценного скрипта. И напоследок, нам всего лишь надо зафиксировать изменения. SQLite 3 и другие транзакционные базы данных не будут фиксировать изменения, если им это не будет указано в явной форме.

Теперь мы можем из командной оболочки Python импортировать и вызвать эту функцию, тем самым создав базу данных:

```
>>> from flaskr import init_db
>>> init_db()
```

**Поиск и устранение возможных проблем**

Если появилось исключение, что таблица не может быть найдена, проверьте, что вы назвали функцию *init_db*, и что имена ваших таблиц заданы корректным образом (проверьте, например, на отсутствие в именах ошибок, связанных с использованием единственного и множественного числа).

Продолжение: *Шаг 4: Запрос соединения с базой данных*

### 1.5.6 Шаг 4: Запрос соединения с базой данных

Теперь мы знаем, как открыть соединение с базой данных, и использовать его для скриптов, но как это элегантно сделать для запросов? Нам будет необходимо соединение с базой данных во всех на-

ших функциях, так что имеет смысл инициализировать его перед каждым запросом, а впоследствии закрывать его.

Flask позволяет нам это сделать с помощью декораторов `before_request()`, `after_request()` и `teardown_request()`:

```
@app.before_request
def before_request():
    g.db = connect_db()

@app.teardown_request
def teardown_request(exception):
    g.db.close()
```

Функции с меткой `before_request()` вызываются до запроса, им не передаётся аргументов. Функции с меткой `after_request()` вызываются после запроса и им передаётся ответ, который будет послан клиенту. Они должны вернуть этот или другой объект ответа. Однако, нет гарантии их выполнения в случае возникновения ситуации исключения, в таких ситуациях приходит время для вызова функции `teardown_request()`. Они вызываются после того, как ответ будет построен. Они не позволяют изменять запрос, а возвращаемые ими значения игнорируются. Если исключение появилось во время обработки запроса, оно передаётся каждой функции; в обратном случае ничего не передаётся ничего (*None*).

Мы храним наши текущие соединения с базой данных в специальном объекте, которым нас снабдил Flask - `g`. Этот объект сохраняет информацию только для одного запроса, и доступен изнутри каждой функции. Никогда не храните подобные вещи в других объектах, потому что это не будет работать в многопоточных окружениях. Для того, чтобы всё пошло как надо, этот особый объект - `g` - производит за сценой определённые «магические» действия.

Продолжение: *Шаг 5: Функции представления*.

---

**Подсказка:** Куда мне поместить этот код?

Если вы следовали указаниям этого учебника, вас может немного озадачить, куда поместить код из этого и следующего шага. Было бы логичным сгруппировать эти функции уровня модуля вместе, и разместить ваши новые функции `before_request` и `teardown_request` под функцией `init_db` (аккуратно следуя учебнику).

Если вам нужно время, чтобы сориентироваться, взгляните как организованы исходные тексты примера. В Flask, вы можете поместить весь код вашего приложения в единственный модуль Python. Однако это вовсе необязательно, и если ваше приложение *растёт larger*, будет разумным этого не делать.

---

### 1.5.7 Шаг 5: Функции представления

Теперь, когда соединения с базой данных уже работают, мы можем заняться написанием функций представления. Нам нужны четыре из них:

#### Показать записи

Это представление показывает все записи, хранящиеся в базе данных. Оно соответствует главной странице вашего приложения, и выбирает все заголовки и тексты из базы данных. Запись с наибольшим id (последняя по времени) будет наверху. Строки, возвращаемые курсором - это кортежи с такими же колонками, какие указаны в операторе select. Это приемлемо для небольших приложений подобных

нашему, но вы можете захотеть конвертировать их в словарь. Если вам интересно, как это делается, смотрите пример: easy-querying.

Функция представления передаёт записи в виде словаря шаблону *show_ entries.html* и возвращает сформированное отображение:

```
@app.route('/')
def show_entries():
    cur = g.db.execute('select title, text from entries order by id desc')
    entries = [dict(title=row[0], text=row[1]) for row in cur.fetchall()]
    return render_template('show_entries.html', entries=entries)
```

### Добавть новую запись

Это представление позволяет пользователю, если он осуществил вход, добавлять новые записи. Оно реагирует только на запросы типа *POST*, а фактическая форма отображается на странице *show_ entries*. Если всё работает хорошо, наше сообщение будет передано (`flash()`) следующему запросу и произойдёт возврат через перенаправление на страницу *show_ entries*:

```
@app.route('/add', methods=['POST'])
def add_entry():
    if not session.get('logged_in'):
        abort(401)
    g.db.execute('insert into entries (title, text) values (?, ?)',
                 [request.form['title'], request.form['text']])
    g.db.commit()
    flash('New entry was successfully posted')
    return redirect(url_for('show_entries'))
```

Заметьте, что здесь есть проверка на то, что пользователь вошёл (ключ внутри сессии *logged_ in* присутствует и установлен в *True*).

---

**Замечание, касающееся безопасности**

Убедитесь пожалуйста, что при формировании оператора SQL, как и в примере выше, были использованы вопросительные знаки. В обратном случае, при форматировании строк для построения оператора SQL, ваше приложение станет уязвимым для SQL-инъекций. За подробностями обратитесь к разделу sqlite3.

---

### Вход и выход

Эти функции используются для того, чтобы пользователь мог войти под собой и выйти. При входе производится проверка имя пользователя и пароля с значениями, хранимыми в конфигурации, и в сессии устанавливается ключ *logged_ in*. Если пользователь зашёл успешно, этот ключ устанавливается в *True*, и пользователь возвращается обратно к странице *show_ entries*. К тому же, появляется всплывающее сообщение, что он или она зашли успешно. При возникновении ошибки, шаблон об этом получает уведомление, и происходит повторный запрос у пользователя имени пользователя и пароля:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME']:
            error = 'Invalid username'
```

```
        elif request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid password'
        else:
            session['logged_in'] = True
            flash('You were logged in')
            return redirect(url_for('show_entries'))
    return render_template('login.html', error=error)
```

Функция выхода, с другой стороны, удаляет обратно этот ключ из сессии. Здесь мы используем ловкий трюк: если вы используете метод словаря `pop()` и передаёте ему второй параметр (по умолчанию), метод удаляет ключ из словаря при его наличии или ничего не делает если такого ключа нет. Это полезно, потому что теперь нам не надо делать проверку, вошёл ли пользователь или нет.

```
@app.route('/logout')
def logout():
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('show_entries'))
```

Продолжение: *Шаг 6: Шаблоны*.

### 1.5.8 Шаг 6: Шаблоны

Теперь нам следует поработать над шаблонами. Если мы будем сейчас запрашивать URL, мы получим лишь исключения, означающие, что Flask не может найти шаблоны. Шаблоны используют синтаксис Jinja2 и по умолчанию имеют автоматическую обработку специальных последовательностей (autoescaping). Это означает, что если вы в шаблоне вы не пометили значение с помощью `Markup` или с помощью фильтра `|safe`, Jinja2 гарантирует, что особые символы, например `<` или `>`, будут заменены спецпоследовательностями их XML-эквивалентов.

Также мы используем наследование шаблонов, что делает возможным повторно использовать макет веб-сайта на всех страницах.

Поместите в папку *templates* следующие шаблоны:

#### layout.html

Этот шаблон содержит скелет HTML, заголовок и ссылку для входа на сайт (или выхода, если пользователь уже вошёл). Он таже отображает всплывающие сообщения, если они есть. Блок `{% block body %}` может быть заменён в дочернем шаблоне на блок с тем же именем (`body`).

Также из шаблона доступен словарь `session`, и вы можете его использовать для проверки - вошёл ли пользователь или нет. Заметим, что в Jinja вы можете иметь достуул к отсутствующим атрибутам и элементам объектов / словарей, что делает следующий код работоспособным, даже при отсутствии в сессии ключа `'logged_in'`:

```
<!doctype html>
<title>Flaskr</title>
<link rel=stylesheet type=text/css href="{{ url_for('static', filename='style.css') }}">
<div class=page>
  <h1>Flaskr</h1>
  <div class=metanav>
  {% if not session.logged_in %}
    <a href="{{ url_for('login') }}">log in</a>
  {% else %}
```

```
    <a href="{{ url_for('logout') }}">log out</a>
  {% endif %}
  </div>
  {% for message in get_flashed_messages() %}
    <div class=flash>{{ message }}</div>
  {% endfor %}
  {% block body %}{% endblock %}
</div>
```

### show_entries.html

Этот шаблон расширет вышеприведённый шаблон *layout.html* для отображения сообщений. Заметим, что цикл *for* итерирует сообщения которые мы передаём внутрь шаблона с помощью функции `render_template()`. Также мы сообщаем форме, чтобы она предала данные вашей функции *add_ entry*, используя при этом *POST* в качестве метода *HTTP*:

```
{% extends "layout.html" %}
{% block body %}
  {% if session.logged_in %}
    <form action="{{ url_for('add_entry') }}" method=post class=add-entry>
      <dl>
        <dt>Title:
        <dd><input type=text size=30 name=title>
        <dt>Text:
        <dd><textarea name=text rows=5 cols=40></textarea>
        <dd><input type=submit value=Share>
      </dl>
    </form>
  {% endif %}
  <ul class=entries>
  {% for entry in entries %}
    <li><h2>{{ entry.title }}</h2>{{ entry.text|safe }}
  {% else %}
    <li><em>Unbelievable.  No entries here so far</em>
  {% endfor %}
  </ul>
{% endblock %}
```

### login.html

Наконец, шаблон для осуществления входа, который просто-напросто отображает форму, позволяющую пользователю залогиниться:

```
{% extends "layout.html" %}
{% block body %}
  <h2>Login</h2>
  {% if error %}<p class=error><strong>Error:</strong> {{ error }}{% endif %}
  <form action="{{ url_for('login') }}" method=post>
    <dl>
      <dt>Username:
      <dd><input type=text name=username>
      <dt>Password:
      <dd><input type=password name=password>
      <dd><input type=submit value=Login>
    </dl>
```

```
  </form>
{% endblock %}
```

Продолжение: *Шаг 7: Добавление стиля*.

### 1.5.9 Шаг 7: Добавление стиля

Теперь, когда всё остальное работает, пришло время, чтобы придать приложению немного стиля. Просто создайте в папке *static*, которую мы создавали до этого, таблицу стилей, в файле с именем *style.css*:

```
body            { font-family: sans-serif; background: #eee; }
a, h1, h2       { color: #377BA8; }
h1, h2          { font-family: 'Georgia', serif; margin: 0; }
h1              { border-bottom: 2px solid #eee; }
h2              { font-size: 1.2em; }

.page           { margin: 2em auto; width: 35em; border: 5px solid #ccc;
                  padding: 0.8em; background: white; }
.entries        { list-style: none; margin: 0; padding: 0; }
.entries li     { margin: 0.8em 1.2em; }
.entries li h2  { margin-left: -1em; }
.add-entry      { font-size: 0.9em; border-bottom: 1px solid #ccc; }
.add-entry dl   { font-weight: bold; }
.metanav        { text-align: right; font-size: 0.8em; padding: 0.3em;
                  margin-bottom: 1em; background: #fafafa; }
.flash          { background: #CEE5F5; padding: 0.5em;
                  border: 1px solid #AACBE2; }
.error          { background: #F0D6D6; padding: 0.5em; }
```

Продолжение: *Бонус: Тестирование приложения*.

### 1.5.10 Бонус: Тестирование приложения

Теперь, когда вы завершили работу над приложением и всё работает как и ожидалось, нам - чтобы упростить его дальнейшую модификацию - вероятно, было бы неплохо добавить автоматизированные тесты. Вышеприведённое приложение используется в разделе документации *Тестирование приложений Flask* в качестве простейшего примера того, как следует выполнять тестирование модулей. Перейдите туда, чтобы увидеть, насколько просто тестировать приложения Flask.

## 1.6 Шаблоны

Flask использует в качестве системы шаблонизации Jinja2. Можно использовать другие системы шаблонизации, но для запуска Flask всё равно необходимо установить Jinja2. Это необходимо для использования дополнительных возможностей. Расширения могут зависеть от наличия Jinja2.

Этот раздел предоставляет лишь краткое описание интеграции Jinja2 во Flask. Если вам нужна информация о синтаксисе самой системы шаблонизации, за более подробной информацией обратитесь к официальной документации по шаблонам Jinja2.

### 1.6.1 Установка Jinja

По умолчанию Flask настраивает Jinja2 следующим образом:

- включено автоматическое экранирование для всех шаблонов, с раширениями `.html`, `.htm`, `.xml`, `.xhtml`
- шаблон может включать или отключать автоматическое экранирование при помощи тега `{% autoescape %}`.
- Flask добавляет пару функций и хелперов в контекст Jinja2, дополнительно к значениям, имеющимся по умолчанию.

### 1.6.2 Стандартный контекст

По умолчанию из шаблонов Jinja2 доступны следующие глобальные переменные:

`config`
: Объект текущей конфигурации (`flask.config`)

  Добавлено в версии 0.6.

  Изменено в версии 0.10: Теперь он доступен всегда, даже в импортированных шаблонах.

`request`
: Объект текущего запроса (`flask.request`). Эта переменная недоступна, если шаблон отрисован без контекста активного запроса.

`session`
: Объект текущего сеанса (`flask.session`). Эта переменная недоступна, если шаблон отрисован без контекста активного запроса.

`g`
: Связанный с запросом объект с глобальными переменными (`flask.g`). Эта переменная недоступна, если шаблон отрисован без контекста активного запроса.

`url_for()`
: Функция `flask.url_for()`.

`get_flashed_messages()`
: Функция `flask.get_flashed_messages()`.

---

**Контекстное поведение Jinja**

Эти переменные добавляются к переменным контекста, но это не глобальные переменные. Отличие заключается в том, что по умолчанию эти переменные отсутствуют в контексте импортируемых шаблонов. Отчасти это сделано для повышения производительности, отчасти - из-за предпочтения явного поведения неявному.

Какое это имеет значение? Если вам нужно получить доступ из макроса к объекту запроса, есть две возможности:

1. явным образом передать объект запроса или его атрибут в макрос в качестве параметра.
2. импортировать макрос с контекстом, указав ключевые слова «with context».

Импорт с контекстом выглядит следующим образом:

```
{% from '_helpers.html' import my_macro with context %}
```

---

### 1.6.3 Стандартные фильтры

В дополнение к собственным фильтрам Jinja2, доступны следующие фильтры:

`tojson()`
: Эта функция конвертирует переданный объект в JSON-представление. Это может быть полезно, когда нужно на лету сгенерировать JavaScript.

  Отметим, что внутри тегов *script* не должно производиться экранирование, поэтому убедитесь в том, что отключили экранирование при помощи фильтра `|safe`, если собираетесь использовать фильтр *tojson* внутри тегов *script*:

```
<script type=text/javascript>
    doSomethingWith({{ user.username|tojson|safe }});
</script>
```

  Фильтр `|tojson` правильно экранирует прямую косую черту.

### 1.6.4 Управление автоэкранированием

Автоэкранирование - это автоматическое экранирование специальных символов. Специальными символами в HTML (а также в XML и в XHTML) являются `&`, `>`, `<`, `"` и `'`. Поскольку эти символы имеют особое значение в документах, для использования в тексте их нужно заменить на так называемые «сущности». Если этого не сделать, это не только может повлиять на невозможность использования этих символов пользователем, но и привести к проблемам с безопасностью (см. xss).

Однако, иногда в шаблонах может потребоваться отключить автоэкранирование. Это может понадобиться, если нужно явным образом вставить в страниц фрагмент HTML, если фрагмент поступил из системы генерации безопасного HTML, например, из преобразователя markdown в HTML.

Для достижения этого есть три способа:

- В коде Python обернуть строку HTML в объект `Markup` перед передачей в шаблон. Это рекомендуемый способ.
- Внутри шаблона, воспользовавшись фильтром `|safe` для явной отметки строки, как безопасного HTML (`{{ myvariable|safe }}`)
- Временно отключить систему автоэкранирования.

Для отключения системы автоэкранирования в шаблонах можно воспользоваться блоком `{% autoescape %}`:

```
{% autoescape false %}
    <p>autoescaping is disabled here
    <p>{{ will_not_be_escaped }}
{% endautoescape %}
```

Соблюдайте осторожность и всегда следите за переменными, которые помещаете в этот блок.

### 1.6.5 Регистрация фильтров

Если нужно зарегистрировать собственные фильтры в Jinja2, у есть два способа. Можно просто поместить их вручную в атрибут `jinja_env` приложения или использовать декоратор `template_filter()`.

Следующие примеры делают одно и то же, переставляя элементы объекта в обратном порядке:

```
@app.template_filter('reverse')
def reverse_filter(s):
    return s[::-1]

def reverse_filter(s):
    return s[::-1]
app.jinja_env.filters['reverse'] = reverse_filter
```

При использовании декоратора указывать аргумент не обязательно, если вы хотите чтобы имя фильтра совпадало с именем функции. Однажды зарегистрировав фильтр, вы можете использовать его в шаблонах точно так же, как и встроенные фильтры Jinja2, например, если имеется список Python, имеющий в контексте имя *mylist*:

```
{% for x in mylist | reverse %}
{% endfor %}
```

### 1.6.6 Процессоры контекста

Для автоматической вставки в контекст шаблона новых переменных существуют процессоры контекста Flask. Процессоры контекста запускаются перед отрисовкой шаблона и позволяют добавить новые переменные в контекст. Процессор контекста - это функция, возвращающая словарь, который будет объединён с контекстом шаблона, для всех шаблонов в приложении. Например, для app:

```
@app.context_processor
def inject_user():
    return dict(user=g.user)
```

Процессор контекста, приведённый выше, сделает переменную *g.user* доступной из шаблона под именем *user*. Этот пример не очень интересен, поскольку *g* и так доступна в шаблонах, но даёт представление о том, как это работает.

Переменные не ограничены только своими значениями; процессор контекста может передавать в шаблон не только переменные, но и функции (поскольку Python позволяет передавать функции):

```
@app.context_processor
def utility_processor():
    def format_price(amount, currency=u'€'):
        return u'{0:.2f}{1}'.format(amount, currency)
    return dict(format_price=format_price)
```

Вышеприведённый процессор контекста сделает функцию *format_price* доступной для всех шаблонов:

```
{{ format_price(0.33) }}
```

Вы также можете встроить *format_price* как фильтр шаблона (см. выше раздел *Регистрация фильтров*), но этот пример демонстрирует, как передавать функции в контекст шаблона.

## 1.7 Тестирование приложений Flask

**То, что не протестировано, не работает.**

Оригинал цитаты неизвестен, и хотя она не совсем правильная, всё же она недалека от истины. Не подвергнутые тестированию приложения затрудняют развитие существующего кода, а разработчики непротестированных приложений, как правило, подвергаются влиянию паранойи. Если у приложения

есть автоматизированные тесты, можно безопасно вностить в него изменения и узнавать сразу, если что-то пошло не так.

Flask предоставляет вам способ тестирования вашего приложения с помощью теста из состава Werkzeug `Client` и через обработку локальных переменных конекста. В дальнейшем вы можете это использовать с вашим любимым средством тестирования. В данной документации мы будем использовать идущий в комплекте с Python пакет `unittest`.

### 1.7.1 Приложение

Для начала, нам нужно иметь приложение для тестирования; мы будем использовать приложение из *Учебник*. Если у вас пока нет этого приложения, возьмите его исходные тексты из примеров.

### 1.7.2 Скелет для тестирования

Для того, чтобы протестировать приложение, мы добавим второй модуль (*flaskr_ tests.py*) и создадим здесь скелет для использования модуля unittest:

```
import os
import flaskr
import unittest
import tempfile

class FlaskrTestCase(unittest.TestCase):

    def setUp(self):
        self.db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
        flaskr.app.config['TESTING'] = True
        self.app = flaskr.app.test_client()
        flaskr.init_db()

    def tearDown(self):
        os.close(self.db_fd)
        os.unlink(flaskr.app.config['DATABASE'])

if __name__ == '__main__':
    unittest.main()
```

Код метода `setUp()` создаёт нового клиента тестирования и инициализирует новую базу данных. Эта функция вызывается перед запуском каждой индивидуальной функции тестирования. Для удаления базы данных после окончани теста, мы закрываем файл и удалем его из файловой системы в методе `tearDown()`. Дополнительно, в процессе настройки активируется флаг конфигурации `TESTING`. Это приводит к отключению отлова ошибок во время обработки запроса и вы получаете более качественные отчёты об ошбиках при выполнении по отношению к приложению тестовых запросов.

Этот тестовый клиент даст нам простой интерфейс к приложению. Мы можем запускать тестовые запросы к приложению, а клиент будет также отслеживать для нас cookies.

Так как SQLite3 использует файловую систему, для создания временной базы данных и её инициализации мы можем по-простому использовать модуль tempfile. Функция `mkstemp()` делает для нас две вещи: она возвращает низкоуровневый обработчик файла и случайное имя файла, последний из которых мы будем использовать как имя базы данных. Нам всего лишь надо сохранить значение *db_ fd*, чтобы в дальнейшем мы могли использовать функцию `os.close()` для закрытия файла.

Если мы запустим сейчас наш набор тестов, мы должны получить следующий результат:

```
$ python flaskr_tests.py

----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

Даже без запуска реальных тестов мы уже будем знать, является ли наше приложение flaskr допустимым синтаксически, в случае, если это не так, операция импорта завершится с возникновением исключения.

### 1.7.3 Первый тест

Сейчас самое время начать тестирование функциональности приложения. Давайте проверим, что при попытке доступа к корню приложения (/) оно отображает «Здесь пока нет ни одной записи». Чтобы это сделать, добавим новый метод тестирования в наш класс, следующим образом:

```
class FlaskrTestCase(unittest.TestCase):

    def setUp(self):
        self.db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
        self.app = flaskr.app.test_client()
        flaskr.init_db()

    def tearDown(self):
        os.close(self.db_fd)
        os.unlink(flaskr.DATABASE)

    def test_empty_db(self):
        rv = self.app.get('/')
        assert 'Здесь пока нет ни одной записи' in rv.data
```

Заметим, что наша тестовая функция начинается со слова *test*; это позволяет модулю `unittest` определить метод как тест для запуска.

С помощью использования *self.app.get* мы можем послать приложению HTTP-запрос *GET* с заданным путём. Возвращаемым значением будет объект `response_class`. Теперь мы можем для проверки возвращаемого значения (как строки) использовать атрибут `data`. В данном случае, мы хотим убедиться в том, что в выводе присутствует `'Здесь пока нет ни одной записи'`.

Запустим всё заново, при похождении теста должно появиться:

```
$ python flaskr_tests.py
.
----------------------------------------------------------------------
Ran 1 test in 0.034s

OK
```

### 1.7.4 Вход и выход

Большинство функций нашего приложения доступны только для администратора, поэтому нам необходим способ для осуществления нашим клиентом тестирования входа и выхода из приложения. Чтобы реализовать это. мы запустим несколько запросов к страницам входа и входа с необходимыми данными

форм (именем пользователя и паролем). Так как страницы входа и выхода приводят к редиректу, мы попросим это позволить нашему клиенту: *follow_ redirects*.

Добавьте два следующих метода к вашему классу *FlaskrTestCase*:

```
def login(self, username, password):
    return self.app.post('/login', data=dict(
        username=username,
        password=password
    ), follow_redirects=True)

def logout(self):
    return self.app.get('/logout', follow_redirects=True)
```

Теперь мы с лёгкостью можем проверить, что вход и выход работают и что они оканчиваются неудачей при неверных учётных данных. Добавьте к нашему классу новый тест:

```
def test_login_logout(self):
    rv = self.login('admin', 'default')
    assert 'You were logged in' in rv.data
    rv = self.logout()
    assert 'You were logged out' in rv.data
    rv = self.login('adminx', 'default')
    assert 'Invalid username' in rv.data
    rv = self.login('admin', 'defaultx')
    assert 'Invalid password' in rv.data
```

### 1.7.5 Тестирование добавления сообщений

Нам необходимо также проверить как работает добавление сообщений. Добавьте новый метод тестирования следующего вида:

```
def test_messages(self):
    self.login('admin', 'default')
    rv = self.app.post('/add', data=dict(
        title='<Hello>',
        text='<strong>HTML</strong> allowed here'
    ), follow_redirects=True)
    assert 'No entries here so far' not in rv.data
    assert '&lt;Hello&gt;' in rv.data
    assert '<strong>HTML</strong> allowed here' in rv.data
```

Здесь мы проверяем, что запись соответсвуте установленным правилам - HTML может присутствовать в тексте, но не заголовке.

Теперь при запуске мы должны увидеть, что пройдено 3 теста:

```
$ python flaskr_tests.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.332s

OK
```

Чтобы посмотреть более сложные тесты заголовков и статусных кодов, обратитесь к исходникам Flask MiniTwit Example, в которых есть более развёрнутый набор тестов.

### 1.7.6 Другие трюки с тестами

Кроме вышеуказанного использования клиента тестирования, есть ещё метод `test_request_context()`, который может быть использован в комбинации с оператором *with* для временной активации контекста запроса. С помощью него вы можете получить доступ к объектам `request`, `g` и `session`, подобно функции представления. Вот полный пример, который демонстрируют подобный подход:

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    assert flask.request.path == '/'
    assert flask.request.args['name'] == 'Peter'
```

Подобным способом могут использоваться и все другие объекты, связанные с контекстом.

Если вы хотите протестировать ваше приложение в другой конфигурации, а хорошего способа, чтобы это сделать, нет, подумайте над переходом на фабрики приложений (см. app-factories).

Заметим, однако, что если вы используете контекст тестового запроса, функции `before_request()` автоматически не выполняются для подобных функций `after_request()`. Однако, функции `teardown_request()` действительно выполняются, когда контекст тестового запроса покидает блок *with*. Если вы хотите, чтобы функции `before_request()` также вызывались, вам необходимо вручную использовать вызов `preprocess_request()`.

app = flask.Flask(__name__)

**with app.test_request_context(„/?name=Peter“):** app.preprocess_request() ...

Это может понадобиться для открытия соединений с базой данных или для чего-то подобного, в зависимости от того, как построено ваше приложение.

Если вы хотите вызвать функции `after_request()`, вам необходимо воспользоваться вызовом `process_response()`, однако он требует, чтобы вы передали ему объект ответа:

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    resp = Response('...')
    resp = app.process_response(resp)
    ...
```

Обычно, это не так полезно, так как в данный момент вы можете прямо стартовать с помощью тестового клиента.

### 1.7.7 Сохранение окружения контекста

Добавлено в версии 0.4.

Иногда бывает полезно вызвать регулярный запрос, сохранив при этом немного дольше окружение контекста, так чтобы могла произойти дополнительная интроспекция. Это стало возможным начиная с Flask версии 0.4 с использованием `test_client()` совместно с блоком *with*:

```
app = flask.Flask(__name__)

with app.test_client() as c:
    rv = c.get('/?tequila=42')
    assert request.args['tequila'] == '42'
```

Если вы использовали просто `test_client()` без блока *with*, *assert* завершится аварийно с ошибкой, так как *request* более недоступен (потому что вы попытались использовать его вне актуального запроса).

### 1.7.8 Доступ к сессиям и их изменение

Добавлено в версии 0.8.

Иногда бывает очень полезно получить доступ к сессиям или изменить их из клиента тестирования. Обычно, чтобы сделать это, существует два способа. Если вы просто хотите убедиться, что в сессии есть определённый набор ключей, которым присвоены определённые значения, вам надо просто сохранить окружение контекста и получить доступ к `flask.session`:

```
with app.test_client() as c:
    rv = c.get('/')
    assert flask.session['foo'] == 42
```

Однако, это не даёт возможности изменять сессию или иметь доступ к сессии до запуска запроса. Начиная с Flask 0.8, мы предоставили вам так называемую "транзакцию сессии", которая имитирует соответствующие вызовы для открытия сессии в контексте клиента тестирования, в том числе для её изменения. В конце транзакции сессия сохраняется. Это работает вне зависимости от того, какой из бэкэндов сессии был использован:

```
with app.test_client() as c:
    with c.session_transaction() as sess:
        sess['a_key'] = 'a value'

    # здесь сессия будет сохранена
```

Заметим, что в данном случае вам необходимо использовать вместо прокси `flask.session` объект `sess`. Однако, объект сам по себе обеспечит вас тем же интерфейсом.

## 1.8 Журналирование ошибок приложения

Добавлено в версии 0.3.

В приложениях и серверах иногда происходят ошибки. Рано или поздно вы увидите исключение на сервере в эксплуатации. Даже если ваш код на 100% правильный, вы всё равно будете время от времени видеть исключения. Почему? Потому что может сломаться что-то другое. Вот некоторые ситуации, в которых совершенный код может приводить к ошибкам на сервере:

- клиент завершил запрос раньше, а приложение по-прежнему ожидает поступления данных.
- сервер базы данных был перегружен и не смог обработать запрос.
- в файловой системе закончилось место.
- сломался жёсткий диск.
- перегружен сервер-бэкэнд.
- ошибка в используемой библиотеке.
- ошибка сетевого соединения сервера с другой системой.

И это только небольшой список причин, который можно продолжить. Как же справляться с проблемами такого рода? По умолчанию, если приложение запущено в рабочем режиме, Flask покажет очень простую страницу и занесёт исключение в журнал `logger`.

Но для обработки ошибок можно сделать и больше, если задать соответствующие настройки.

### 1.8.1 Письма об ошибках

Если приложение запущено на сервере в эксплуатации, по умолчанию нежелательно показывать сообщения об ошибках. Почему? Flask пытается быть фреймворком, не требующим настройки. Куда он должен складывать сообщения об ошибках, если это не указано в настройках? Автоматически выбранное место может не подойти, потому что у пользователя может не быть прав на создание там журналов. К тому же, в большинстве никто не станет читать журналы небольших приложений.

На деле я предполагаю, что вы не станете заглядывать в журнал ошибок, даже если настроите его, до тех пор пока вам не понадобится увидеть исключение для отладки проблемы, о которой сообщил пользователь. Более полезной может оказаться отправка письма в случае возникновения исключения. Тогда вы получите оповещение и сможете что-нибудь с ним сделать.

Flask использует встроенную систему журналирования Python, и действительно может отправлять письма об ошибках, чем вы можете воспользоваться. Вот как можно настроить систему журналирования Flask для отправки писем об исключениях:

```
ADMINS = ['yourname@example.com']
if not app.debug:
    import logging
    from logging.handlers import SMTPHandler
    mail_handler = SMTPHandler('127.0.0.1',
                               'server-error@example.com',
                               ADMINS, 'YourApplication Failed')
    mail_handler.setLevel(logging.ERROR)
    app.logger.addHandler(mail_handler)
```

Что это даст? Мы создали новый обработчик `SMTPHandler`, который отправит письма через почтовый сервер с IP-адресом `127.0.0.1` на все адреса из *ADMINS* с адреса *server-error@example.com* и с темой «YourApplication Failed» («Сбой в ВашемПриложении»). Если почтовый сервер требует авторизации, можно указать необходимые для неё данные. За информацией о том, как это сделать, обратитесь к документации на `SMTPHandler`.

Мы также сообщили обработчику, что он должен отправлять письма только об ошибках или более важных сообщениях, потому что нам явно не нужны письма о предупреждениях или других бесполезных записях в журнале, которые могут появиться в процессе обработки запроса.

Перед тем, как применить эти настройки на сервере в эксплуатации, обратитесь к разделу *Управление форматом журнала* ниже, чтобы в письма помещалась необходимая информация и вы не растерялись, получив письмо.

### 1.8.2 Журналирование в файл

Даже если вы получили письмо, вам скорее всего захочется посмотреть и на предупреждения в журнале. Хорошо бы сохранить как можно больше информации, полезной для отладки проблемы. Отметим, что ядро Flask само по себе не выдаёт предупреждений, поэтому вам самим нужно позаботиться о том, чтобы генерировать предупреждения в вашем коде, в случае если что-то пошло не так.

Эта пара обработчиков поставляется в комплекте с системой журналирования, но не каждый из них подходит для начального журналирования ошибок. Наиболее интересными могут показаться следующие:

- `FileHandler` - ведёт журнал ошибок в файле.

- `RotatingFileHandler` - ведёт журнал ошибок в файле и создаёт новый файл после определённого количества сообщений.
- `NTEventLogHandler` - использует системный журнал событий Windows. Может пригодиться при развёртывании приложения на Windows-компьютере.
- `SysLogHandler` - отправляет сообщения в syslog, системный журнал UNIX.

Как только вы подберёте подходящий обработчик журнала, можете настроить обработчик SMTP из примера выше. Просто убедитесь в том, что снизили порог критичности сообщений (я рекомендую *WARNING*):

```
if not app.debug:
    import logging
    from themodule import TheHandlerYouWant
    file_handler = TheHandlerYouWant(...)
    file_handler.setLevel(logging.WARNING)
    app.logger.addHandler(file_handler)
```

### 1.8.3 Управление форматом журнала

По умолчанию обработчик будет лишь записывать строку с сообщением в файл или отправлять эту строку по почте. Записи журнала содержат больше информации и стоит настроить обработчик так, чтобы он содержал больше полезной информации, по которой можно понять, что случилось и, что более важно, где это произошло.

Средство форматирования можно настроить при помощи строки формата. Отметим, что отчёт о трассировке добавляется к записи в журнале автоматически, для этого не требуется каких-то специальных настроек в строке формата.

Вот примеры настройки:

#### Журналирование на почту

```
from logging import Formatter
mail_handler.setFormatter(Formatter('''
Message type:       %(levelname)s
Location:           %(pathname)s:%(lineno)d
Module:             %(module)s
Function:           %(funcName)s
Time:               %(asctime)s

Message:

%(message)s
'''))
```

#### Журналирование в файл

```
from logging import Formatter
file_handler.setFormatter(Formatter(
    '%(asctime)s %(levelname)s: %(message)s '
    '[in %(pathname)s:%(lineno)d]'
))
```

#### Сложное форматирование журналов

Вот список полезных переменных форматирования для подстановки в строку формата. Отметим, что этот список не полный, полный список можно найти в официальной документации пакета журналирования `logging`.

| Format | Description |
|---|---|
| `%(levelname)s` | Уровень серьёзности сообщения (`'DEBUG'` - отладочное, `'INFO'` - информационное, `'WARNING'` - предупреждение, `'ERROR'` - ошибка, `'CRITICAL'` - критичное). |
| `%(pathname)s` | Полный путь к файлу с исходным текстом, из которого была вызвана функция журналирования (если доступен). |
| `%(filename)s` | Имя файла с исходным текстом. |
| `%(module)s` | Модуль (часть имени файла). |
| `%(funcName)s` | Имя функции, из который была вызвана функция журналирования. |
| `%(lineno)d` | Номер строки в файле исходного текста, в которой произошёл вызов функции журналирования (если доступна). |
| `%(asctime)s` | Время создания записи в журнале в человеко-читаемом виде. По умолчанию используется формат `"2003-07-08 16:49:45,896"` (числа после запятой - это миллисекунды). Можно изменить путём создания класса-наследника от formatter и заменой метода `formatTime()`. |
| `%(message)s` | Журнальное сообщение, полученное из выражения `msg % args` |

Если вы хотите выполнить тонкую настройку форматирования, нужно создать класс-наследник от formatter. formatter имеет три полезных метода:

`format()`: Занимается собственно форматированием. Принимает объект `LogRecord` и возвращает отформатированную строку.

`formatTime()`: Вызывается для форматирования *asctime*. Если нужно задать другой формат времени, можно заменить этот метод.

`formatException()` Вызывается для форматирования исключений. Принимает кортеж `exc_info` и возвращает строку. В большинстве случаев подойдёт метод по умолчанию, поэтому скорее всего вам не потребуется его заменять.

За более подробной информацией обратитесь к официальной документации.

### 1.8.4 Другие библиотеки

Таким образом мы настроили журналирование событий, порождаемых самим приложением. Другие библиотеки могут вести собственный журнал. Например, SQLAlchemy широко использует журналирование в собственном ядре. Хотя этот способ пригоден для настройки сразу всех средств журналирования в пакете `logging`, пользоваться им не рекомендуется. Может возникнуть ситуация, когда нужно различать приложения, работающие в пределах одного интерпретатора Python, но будет невозможно сделать для них отдельные настройки журналирования.

Вместо этого рекомендуется выяснить, какие средства журналирования нужны, получить их с помощью функции `getLogger()` и перебрать все присоединённые к ним обработчики:

```
from logging import getLogger
loggers = [app.logger, getLogger('sqlalchemy'),
           getLogger('otherlibrary')]
for logger in loggers:
```

```
    logger.addHandler(mail_handler)
    logger.addHandler(file_handler)
```

## 1.9 Отладка ошибок приложения

Для приложений в эксплуатации настройте журналирование и уведомления так, как описано в разделе *Журналирование ошибок приложения*. Этот раздел предоставит указания для отладки конфигурации развёртывания и более глубокого исследования с использованием полнофункционального отладчика Python.

### 1.9.1 В случае сомнений запускайте вручную

Возникли проблемы с настройкой приложения в эксплуатации? Если имеется доступ к командной строке на сервере, проверьте, можете ли вы запустить приложение вручную из командной строки в режиме разработки. Убедитесь, что запустили его под той же учётной записью, под которой оно установлено, чтобы отследить ошибки, связанные с неправильной настройкой прав доступа. Можете воспользоваться встроенным во Flask сервером разработки, передав ему аргумент *debug=True* на сервере эксплуатации, что может помочь в отлове проблем с настройками, но **убедитесь, что делаете это временно в управляемом окружении**. Не запускайте приложение в эксплуатацию с аргументом debug=True.

### 1.9.2 Работа с отладчиками

Для более глубокого исследования можно выполнить трассировку кода. Flask содержит отладчик в стандартной поставке (смотрите *Режим отладки*). Если вам нравится пользоваться другим отладчиком Python, учтите, что они могут мешать друг другу. Можно указать несколько опций, чтобы использовать ваш любимый отладчик:

- `debug` - указывает, нужно ли включить режим отладки и захват исключений.
- `use_debugger` - указывает, нужно ли использовать отладчик, встроенный во Flask.
- `use_reloader` **- указывает, нужно ли перезагружать и перезапускать процесс,** если произошло исключение.

Опция `debug` должна иметь значение True (то есть, исключения должны захватываться), для того чтобы учитывались две следующие опции.

Если вы используете Aptana/Eclipse для отладки, вам нужно установить обе опции `use_debugger` и `use_reloader` в False.

Возможно, лучшие всего настроить эти опции в файле config.yaml (измените блок, так как вам нужно):

```
FLASK:
    DEBUG: True
    DEBUG_WITH_APTANA: True
```

В точке входа в ваше приложение (main.py), нужно написать что-то вроде этого:

```
if __name__ == "__main__":
    # To allow aptana to receive errors, set use_debugger=False
    app = create_app(config="config.yaml")

    if app.debug: use_debugger = True
    try:
```

```
        # Disable Flask's debugger if external debugger is requested
        use_debugger = not(app.config.get('DEBUG_WITH_APTANA'))
    except:
        pass
    app.run(use_debugger=use_debugger, debug=app.debug,
            use_reloader=use_debugger, host='0.0.0.0')
```

## 1.10 Обработка конфигурации Flask

Добавлено в версии 0.3.

Приложения требуют настройки. Здесь описаны различные настройки, которые можно менять в зависимости от окружения, в котором работает приложение: переключение режима отладки, настройки секретного ключа и т.п.

Flask спроектирован так, что обычно требует настройки при запуске приложения. Вы можете вшить настройки в код, что не так уж плохо для многих небольших приложений, но имеются способы лучше.

Вне зависимости от того, каким образом загружена конфигурация, существует объект конфигурации, содержащий загруженные параметры: атрибут `config` объекта `Flask`. Это место, где Flask содержит свои настройки, а также то место, куда расширения могут поместить собственные настройки. Но здесь можно размещать и конфигурацию вашего приложения.

### 1.10.1 Основы конфигурации

`config` на самом деле является подклассом словаря и может изменяться точно так же, как и любой словарь:

```
app = Flask(__name__)
app.config['DEBUG'] = True
```

Некоторые параметры конфигурации передаются в объект `Flask`, которые тоже можно читать и писать:

```
app.debug = True
```

Для обновления нескольких ключей за раз можно воспользоваться методом словаря `dict.update()`:

```
app.config.update(
    DEBUG=True,
    SECRET_KEY='...'
)
```

### 1.10.2 Встроенные параметры конфигурации

Сам Flask использует следующие параметры конфигурации:

| | |
|---|---|
| DEBUG | Включить/выключить режим отладки. |
| TESTING | Включить/выключить режим тестирования. |
| PROPAGATE_EXCEPTIONS | Явное включение или отключение исключений. Если не задано или явным образом задано значение *None*, то подразумевается истина, если истиной является *TESTING* или *DEBUG*. |
| PRESERVE_CONTEXT_ON_EXCEPTION | По умолчанию в режиме отладки при возникновении исключения контекст запроса не извлекается из стека, позволяя отладчику анализировать данные. Такое поведение можно отключить с помощью этого параметра. Также можно воспользоваться этим параметром для его принудительного включения, если это может помочь в отладке приложений в эксплуатации (однако, это не рекомендуется). |
| SECRET_KEY | Секретный ключ. |
| SESSION_COOKIE_NAME | Имя переменной (cookie) браузера для хранения сеанса. |
| SESSION_COOKIE_DOMAIN | Домен переменной браузера, используемой для хранения сеанса. Если не задан, переменная браузера будет действительной для всех поддоменов SERVER_NAME. |
| SESSION_COOKIE_PATH | Путь к переменной браузера, используемой для хранения сеанса. Если не задан, переменная браузера будет действительной для всех APPLICATION_ROOT, если не задано значение '/'. |
| SESSION_COOKIE_HTTPONLY | Указывает, должен ли у переменной браузера устанавливаться флаг httponly (что защищает переменную от доступа со стороны скриптов, работающих внутри браузера - прим. перев.). По умолчанию - *True*. |
| SESSION_COOKIE_SECURE | Указывает, должен ли у переменной браузера устанавливаться флаг secure (что позволяет передавать переменную только по защищённому протоколу HTTPS - прим. перев.). По умолчанию - *False*. |
| PERMANENT_SESSION_LIFETIME | Непрерывное время жизни сеанса, как объект datetime.timedelta. Начиная с Flask 0.8 этот параметр может быть задан в виде целого числа с количеством секунд. |
| USE_X_SENDFILE | Включить/отключить x-sendfile. (При использовании этой возможности представление может вернуть специально сформированный ответ со ссылкой на статический файл. Получив такой ответ от представления, веб-сервер отдаёт клиенту вместо ответа представления сам статический файл, найдя его по ссылке в локальной файловой системе. Это позволяет перенести нагрузку по отдаче больших файлов на веб-сервер, если перед отдачей файла представление должно решить, можно ли отдавать этот файл клиенту и какой именно файл нужно отдать по этой ссылке - прим. перев.) |
| LOGGER_NAME | Имя средства журналирования. |
| SERVER_NAME | Имя и номер порта сервера. Необходимо для поддержки поддоменов (например: 'myapp.dev:5000'). Отметим, что localhost не поддерживает поддомены, поэтому установка параметра в значение "localhost" не поможет. Настройка SERVER_NAME также по умолчанию включает генерацию URL'ов без контекста запроса, но с контекстом приложения. |
| APPLICATION_ROOT | Если приложение не занимает целый домен или под- |

**Подробнее о SERVER_NAME**

SERVER_NAME - это параметр, который используется для поддержки поддоменов. Flask не может догадаться о том, какая часть доменного имени является поддоменом, не зная имя сервера. Этот же параметр используется для настройки переменной браузера, в которой хранится сеанс.

Помните, что не только Flask не может узнать поддомен, ваш веб-браузер тоже не может. Большинство соверменных браузеров не разрешают междоменные переменные браузера, если в имени сервера нет точек. Поэтому если имя сервера `'localhost'`, вы не сможете задать переменную браузера для `'localhost'` и каждого из его поддоменов. В этом случае выберите другое имя сервера, например `'myapplication.local'` и добавьте это имя и поддомены, которые вы хотите использовать, в файл hosts или настройте локальный bind.

Добавлено в версии 0.4: LOGGER_NAME

Добавлено в версии 0.5: SERVER_NAME

Добавлено в версии 0.6: MAX_CONTENT_LENGTH

Добавлено в версии 0.7: PROPAGATE_EXCEPTIONS, PRESERVE_CONTEXT_ON_EXCEPTION

Добавлено в версии 0.8: TRAP_BAD_REQUEST_ERRORS, TRAP_HTTP_EXCEPTIONS, APPLICATION_ROOT, SESSION_COOKIE_DOMAIN, SESSION_COOKIE_PATH, SESSION_COOKIE_HTTPONLY, SESSION_COOKIE_SECURE

Добавлено в версии 0.9: PREFERRED_URL_SCHEME

### 1.10.3 Задание конфигурации с помощью файлов

Конфигурация становится более удобной, если разместить её в отдельном файле. Лучше, если он находится за пределами пакета с приложением. Это позволяет создавать пакеты и распространять приложения с помощью различных инструментов обработки пакетов (distribute-deployment) и впоследствии - изменять конфигурацию.

Далее показан обычный пример::

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

Сначала грузится конфигурация из модуля *yourapplication.default_ settings*, а затем её значения заменяет содержимое файла, указанного в переменной окружения YOURAPPLICATION_SETTINGS. Эта переменная окружения может быть задана в Linux или OS X при помощи команды export оболочки перед запуском сервера:

```
$ export YOURAPPLICATION_SETTINGS=/path/to/settings.cfg
$ python run-app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader...
```

В системах Windows воспользуйтесь встроенной командой *set*:

```
>set YOURAPPLICATION_SETTINGS=\path\to\settings.cfg
```

Файлы конфигурации являются обычными файлами Python. В объект конфигурации сохраняются только переменные с именами в верхнем регистре, так что убедитесь в том, что имена ваших параметров заданы в верхнем регистре.

Вот пример файла конфигурации:

```
# Пример конфигурации
DEBUG = False
SECRET_KEY = '?\xbf,\xb4\x8d\xa3"<\x9c\xb0@\x0f5\xab,w\xee\x8d$0\x13\x8b83'
```

Убедитесь в том, что файл загружается как можно раньше, чтобы расширения могли получить доступ к собственным настройкам при запуске. Существуют другие методы загрузки объекта конфигурации из отдельных файлов. За более полной информацией обратитесь к документации объекта `Config`.

### 1.10.4 Лучшие способы задания конфигурации

Недостаток описанного выше подхода заключается в усложнении тестирования. Нет стопроцентного способа решения этой проблемы, но вот несколько рекомендаций опытных пользователей:

1. Создайте ваше приложение внутри функции и зарегистрируйте в ней blueprint'ы. Таким образом вы можете создать несколько экземпляров вашего приложения с разными конфигурациями, что значительно упростит модульное тестирование. Вы можете воспользоваться функцией, чтобы передать в неё необходимую конфигурацию.
2. Не пишите код, которому требуется конфигурация при импорте. Если ограничиться чтением конфигурации по запросу, возможно будет переконфигурировать объект позже.

### 1.10.5 Режим разработки и режим эксплуатации

Большинству приложений нужно более одной конфигурации. По меньшей мере нужна отдельная конфигурация для рабочего сервера и ещё одна для разработки. Простейший способ управления ими - это создать конфигурацию по умолчанию, которая загружается всегда и которую можно поместить в систему управления версиями, и частичные конфигурации, которые заменяют необходимые значения следующим образом:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

Теперь просто создайте отдельный файл *config.py*, выполните команду export `YOURAPPLICATION_SETTINGS=/path/to/config.py` и готово. Однако, существуют альтернативные способы. Например, можно воспользоваться импортом и подклассами.

В мире Django распространён следующий способ: сделать явный импорт конфигурации, добавив строку `from yourapplication.default_settings import *` в начале файла, а затем заменить значения вручную. Можно сделать также, затем взять из переменной окружения вида `YOURAPPLICATION_MODE` необходимый режим - *production*, *development* и т.п., а затем импортировать заранее определённые файлы, основываясь на этом значении.

Другой любопытный способ - воспользоваться классами и наследованием конфигурации:

```
class Config(object):
    DEBUG = False
    TESTING = False
    DATABASE_URI = 'sqlite://:memory:'

class ProductionConfig(Config):
    DATABASE_URI = 'mysql://user@localhost/foo'

class DevelopmentConfig(Config):
```

```
    DEBUG = True

class TestingConfig(Config):
    TESTING = True
```

Для включения такой конфигурации, вам просто нужно вызвать `from_object()`:

```
app.config.from_object('configmodule.ProductionConfig')
```

There are many different ways and it's up to you how you want to manage your configuration files. However here a list of good recommendations:

- keep a default configuration in version control. Either populate the config with this default configuration or import it in your own configuration files before overriding values.
- use an environment variable to switch between the configurations. This can be done from outside the Python interpreter and makes development and deployment much easier because you can quickly and easily switch between different configs without having to touch the code at all. If you are working often on different projects you can even create your own script for sourcing that activates a virtualenv and exports the development configuration for you.
- Use a tool like fabric in production to push code and configurations separately to the production server(s). For some details about how to do that, head over to the fabric-deployment pattern.

Есть много разных способов, которые можно выбрать для управления файлами конфигурации. Вот список хороших советов:

- Храните файл конфигурации по умолчанию в системе управления версиями. Заполните объект конфигурации значениями по умолчанию или импортируйте его в ваших собственных файлах конфигурации перед тем, как заменить значения.
- Воспользуйтесь переменной окружения для переключения между конфигурациями. Это можно сделать вне интерпретатора Python и это позволит упростить разработку и развёртывание, потому что вы можете быстро и легко переключаться между разными конфигурациями, совсем не прикасаясь к коду. Если вы часто работаете над разными проектами, можно создать собственный скрипт, который будет определять текущий каталог проекта, активировать virtualenv и экспортировать конфигурацию режима разработки.
- Используйте инструмент fabric для внесения изменений на сервер эксплуатации и раздельных конфигураций на серверы эксплуатации. За более подробным описанием того, как это сделать, обратитесь к главе fabric-deployment.

### 1.10.6 Каталоги экземпляров

Добавлено в версии 0.8.

Во Flask 0.8 появились каталоги экземпляров. Flask долгое время позволял ссылаться на пути относительно каталога приложения (с помощью `Flask.root_path`). Поэтому многие разработчики хранили конфигурацию рядом с приложением. К несчастью, это возможно только если приложение не находится внутри пакета, так как в таком случае root_path указывает внутрь пакета.

Во Flask 0.8 был введён новый атрибут: `Flask.instance_path`. Он вводит новое понятие, которое называется "каталогом экземпляра". Каталог экземпляра задуман как каталог, не управляемый системой контроля версий и относящийся к развёрнутому приложению. Это подходящее место для того, чтобы поместить в него файлы, изменяемые в процессе работы или файлы конфигурации.

Можно явным образом указать путь к каталогу экземпляра при создании приложения Flask или можно разрешить Flask'у самому выбрать каталог экземпляра. Чтобы задать его явным образом, воспользуйтесь параметром *instance_path*:

```
app = Flask(__name__, instance_path='/path/to/instance/folder')
```

Помните, что если этот путь указан, он *должен* быть абсолютным.

Если параметр *instance_path* не указан, по умолчанию используются следующие места:

- Не установленный модуль:

  ```
  /myapp.py
  /instance
  ```

- Не установленный пакет:

  ```
  /myapp
      /__init__.py
  /instance
  ```

- Установленный модуль или пакет:

  ```
  $PREFIX/lib/python2.X/site-packages/myapp
  $PREFIX/var/myapp-instance
  ```

  `$PREFIX` - это префикс, с которым установлен Python. Это может быть каталог `/usr` или путь в каталоге с виртуальным окружением, созданном virtualenv. Можно вывести на экран значение `sys.prefix`, чтобы увидеть его действительное значение.

Как только стало возможным загружать объект конфигурации из файлов с относительными именами, мы добавили возможность загружать файлы с именами относительно каталога экземпляра. При помощи переключателя *instance_relative_config* в конструкторе приложения можно указать, должны ли интерпретироваться относительные пути файлов "относительно корня приложения" (по умолчанию) или "относительно каталога экземпляра":

```
app = Flask(__name__, instance_relative_config=True)
```

Ниже представлен полный пример настройки Flask для предварительной загрузки конфигурации из модуля и последующей замены параметров значениями из файла в каталоге конфигурации, если он существует:

```
app = Flask(__name__, instance_relative_config=True)
app.config.from_object('yourapplication.default_settings')
app.config.from_pyfile('application.cfg', silent=True)
```

Путь к каталогу экземпляра может быть найден при помощи `Flask.instance_path`. Flask также предоставляет более короткий способ открытия файлов из каталога экземпляра при помощи `Flask.open_instance_resource()`.

Вот пример для обоих способов:

```
filename = os.path.join(app.instance_path, 'application.cfg')
with open(filename) as f:
    config = f.read()

# или при помощи open_instance_resource:
with app.open_instance_resource('application.cfg') as f:
    config = f.read()
```

### 1.10.7 Примечания переводчика

В качестве перевода для термина `cookie` было использовано понятие «переменных бразуера».

Информация о флагах *httponly* и *secure* взята из статьи HTTP cookie.

Информация о *x-sendfile* взята из статьи Передача файлов с помощью XSendfile с помощью NGINX.

## 1.11 Контекст приложения Flask

Добавлено в версии 0.9.

Одно из проектных решений Flask заключается в том, что есть два разных “состояния”, в которых выполняется код. Состояние настройки приложения, которое подразумевается на уровне модуля. Оно наступает в момент, когда создаётся экземпляр объекта `Flask`, и заканчивается когда поступает первый запрос. Пока приложение находится в этом состоянии, верно следующее:

- программист может безопасно менять объект приложения.
- запросы ещё не обрабатывались.
- у вас имеется ссылка на объект приложения, чтобы изменить его, нет необходимости пользоваться каким-либо посредником для того, чтобы получить ссылку на созданный или изменяемый объект приложения.

Напротив, во время обработки запроса верны следующие правила:

- пока активен запрос, объекты локального контекста (`flask.request` и другие) указывают на текущий запрос.
- любой код может в любое время может заполучить эти объекты.

Есть и третье состояние, которое располагается между ними. Иногда можно работать с приложением так же, как и во время обработки запроса, просто в этот момент нет активного запроса. Например, вы можете работать с интерактивной оболочкой Python и взаимодействовать с приложением или запустить приложение из командной строки.

Контекст приложения - это то, чем управляет локальный контекст `current_app`.

### 1.11.1 Назначение контекста приложения

Основная причина существования контекста приложений состоит в том, что в прошлом большая доля функциональности была привязана к контексту запроса за неимением лучшего решения. Тогда одной из целей, учитываемых при проектировании Flask, было обеспечение возможности иметь несколько приложений в рамках одного процесса Python.

Каким образом код находит “правильное” приложение? В прошлом мы рекомендовали явную передачу приложений, но появились проблемы с библиотеками, которые не были спроектированы с учётом этого.

Обходной путь решения этой проблемы заключался в том, чтобы использовать посредника `current_app`, привязанного ссылкой к текущему запросу приложения. Но поскольку создание такого контекста запроса является не оправданно дорогостоящим в случае отсутствия запроса, был введён контекст приложения.

### 1.11.2 Создание контекста приложения

Для создания контекста приложения есть два способа. Первый из них - неявный: когда поступает контекст запроса, при необходимости также создаётся и контекст приложения. В результате вы можете игнорировать существование контекста приложения до тех пор, пока он вам не понадобится.

Второй способ - это явное создание контекста при помощи метода `app_context()`:

```
from flask import Flask, current_app

app = Flask(__name__)
with app.app_context():
    # within this block, current_app points to app.
    print current_app.name
```

Контекст приложения также используется функцией `url_for()` в случае, если было настроено значение параметра конфигурации `SERVER_NAME`. Это позволяет вам генерировать URL'ы даже при отсутствии запроса.

### 1.11.3 Локальность контекста

Контекст приложения создаётся и уничтожается при необходимости. Он никогда не перемещается между потоками и не является общим для разных запросов. Поэтому - это идеальное место для хранения информации о подключении к базе данных и т.п. Внутренний объект стека называется `flask._app_ctx_stack`. Расширения могут хранить дополнительную информацию на самом верхнем уровне, если предполагается, что они выбрали достаточно уникальное имя.

За дополнительной информацией по теме обратитесь к разделу extension-dev.

### 1.11.4 Использование контекста

Обычно контекст используют для кэширования в нём ресурсов, которые необходимо создавать в случае отдельных запросов или для постоянного использования. Например, соединения к базе данных предназначены для хранения именно там. Для сохранения чего-либо в контексте приложения необходимо выбирать уникальные имена, так как это то место, которое является общим для приложений и расширений Flask.

Наиболее распространённым использованием является разделение управления ресурсами на две части:

1. неявное кэширование ресурсов в контексте.
2. освобождение ресурса, основанное на демонтировании контекста.

В обычном случае, должна присутсвовать функция `get_X()`, которая создаёт ресурс `X`, если он ещё не существует, и, если это не так, возвращает тот же самый ресурс, а также функция `teardown_X()`, которая регистрирует обработчик демонтирования.

Вот пример соединения с базой данных:

```
import sqlite3
from flask import _app_ctx_stack

def get_db():
    top = _app_ctx_stack.top
    if not hasattr(top, 'database'):
        top.database = connect_to_database()
    return top.database
```

```
@app.teardown_appcontext
def teardown_db(exception):
    top = _app_ctx_stack.top
    if hasattr(top, 'database'):
        top.database.close()
```

Соединение будет установлено, когда `get_db()` вызывается в первый раз. Для того, чтобы сделать это неявно, можно использовать класс `LocalProxy`:

```
from werkzeug.local import LocalProxy
db = LocalProxy(get_db)
```

С использованием этого способа пользователь может иметь прямой доступ к `db`, которая сама внутри себя вызовет `get_db()`.

## 1.12 Контекст запроса Flask

Этот документ описывает поведение Flask 0.7, которое в основном совпадает со старым, но имеет некоторые небольшие отличия.

Рекомендуем сначала прочитать главу *Контекст приложения Flask*.

### 1.12.1 Подробнее о локальных объектах контекста

Представим, что имеется служебная функция, которая возвращает URL, на который нужно перенаправить пользователя. Представим, что всегда нужно перенаправлять на URL из параметра `next` или на страницу, с которой перешли на текущую страницу, или на страницу-индекс:

```
from flask import request, url_for

def redirect_url():
    return request.args.get('next') or \
           request.referrer or \
           url_for('index')
```

Можно заметить, что функция обращается к объекту запроса. Если попытаться запустить её из оболочки Python, будет выброшено исключение:

```
>>> redirect_url()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'request'
```

В этом есть определённый смысл, потому что в данный момент нет запроса, к которому мы пытаемся получить доступ. Итак, нам нужно создать запрос и связать его с текущим контекстом. Создадим `RequestContext` с помощью метода `test_request_context`:

```
>>> ctx = app.test_request_context('/?next=http://example.com/')
```

Этот контекст можно использовать одним из двух способов - используя выражение *with* или вызвав методы `push()` и `pop()`:

```
>>> ctx.push()
```

После чего можно работать с объектом запроса:

```
>>> redirect_url()
u'http://example.com/'
```

И так до тех пор, пока вы не вызовете *pop*:

```
>>> ctx.pop()
```

Поскольку контекст запроса изнутри представляет собой элемент стека, можно добавлять и вынимать его из стека множество раз. Это очень полезно для реализации таких функций, как внутреннее перенаправление.

За более подробной информацией об использовании контекста запроса из интерактивной оболочки Python, обратитесь к главе shell.

### 1.12.2 Как работает контекст

Если посмотреть изнутри на то, как работает приложение Flask WSGI, можно обнаружить фрагмент кода, который выглядит очень похожим на следующий:

```
def wsgi_app(self, environ):
    with self.request_context(environ):
        try:
            response = self.full_dispatch_request()
        except Exception, e:
            response = self.make_response(self.handle_exception(e))
        return response(environ, start_response)
```

Метод `request_context()` возвращает новый объект `RequestContext` и использует его в выражении *with* для связывания контекста. Всё, что будет вызвано из этого потока, начиная с этой точки и до конца выражения *with*, будет иметь доступ к глобальному объекту запроса (`flask.request` и т.п.).

Контекст запроса изнутри работает как стек: на самом верху стека находится текущий активный запрос. `push()` добавляет контекст на верхушку стека, а `pop()` вынимает его из стека. При изъятии также вызываются функции `teardown_request()` приложения.

Стоит также отметить, что при добавлении контекста запроса в стек также создаётся контекст приложения, если его ещё не было. (Not completed: Another thing of note is that the request context will automatically also create an *application context* when it's pushed and there is no application context for that application so far.)

### 1.12.3 Функции обратного вызова и ошибки

Что случится, если произойдёт ошибка во время обработки запроса во Flask? Частично это поведение изменилось в версии 0.7, потому что желательно знать, что на самом деле произошло. Новое поведение очень простое:

1. Перед каждым запросом выполняются функции `before_request()`. Если одна из этих функций вернула ответ, другие функции не выполняются. Однако, в любом случае, это значение трактуется как значение, возвращённое представлением.
2. Если функции `before_request()` не вернули ответ, обработка запроса прекращается и он передаётся в подходящую функцию представления, которая может вернуть ответ.

3. Значение, возвращённое из функции представления, преобразуется в настоящий объект ответа и обрабатывается функциями `after_request()`, которые могут заменить его целиком или отредактировать.

4. В конце запроса выполняются функции `teardown_request()`. Это происходит независимо от того, было ли выброшено необработанное исключение, были ли вызваны функции `before_request()`, или произошло всё сразу (например, в тестовом окружении обработка функций обратного вызова `before_request()` иногда может быть отключена).

Итак, что же происходит в случае ошибки? В рабочем режиме неотловленные исключения приводят к тому, что обработчик выводит сообщение об ошибке 500 на сервере. В режиме разработки, однако, приложение не обрабатывает исключение и передаёт его наверх, серверу WSGI. Таким образом, средства интерактивной отладки могут предоставить информацию для отладки.

Важное изменение в версии 0.7 заключается в том, что внутреннее сообщение сервера об ошибке теперь больше не подвергается пост-обработке с помощью функций обратного вызова `after_request()` и больше нет гарантии того, что они будут выполнены. Таким образом, внутренняя обработка кода выглядит понятнее и удобнее в настройке.

Предполагается, что вместо них должны использоваться новые функции `teardown_request()`, специально предназначенные для действий, которые нужно выполнять по окончании запроса при любом его исходе.

### 1.12.4 Функции обратного вызова teardown_request

Функции обратного вызова `teardown_request()` - это особые функции обратного вызова, которые выполняются отдельно. Строго говоря, они не зависят от действительной обработки запроса и связаны с жизненным циклом объекта `RequestContext`. Когда контекст запроса вынимается из стека, вызываются функции `teardown_request()`.

Это важно знать, если жизнь контекста запроса будет удлинена при использовании клиента для тестировании с помощью выражения *with* или при использовании контекста запроса из командной строки:

```
with app.test_client() as client:
    resp = client.get('/foo')
    # the teardown functions are still not called at that point
    # even though the response ended and you have the response
    # object in your hand

# only when the code reaches this point the teardown functions
# are called.  Alternatively the same thing happens if another
# request was triggered from the test client
```

В этом можно убедиться, воспользовавшись командной строкой:

```
>>> app = Flask(__name__)
>>> @app.teardown_request
... def teardown_request(exception=None):
...     print 'this runs after request'
...
>>> ctx = app.test_request_context()
>>> ctx.push()
>>> ctx.pop()
this runs after request
>>>
```

Учтите, что функции обратного вызова выполняются всегда, независимо от того, были ли выполнены функции обратного вызова `before_request()` и произошло ли исключение. Некоторые части системы

тестирования могут также создавать временный контекст без вызова обработчиков `before_request()`. Убедитесь, что ваши обработчики `teardown_request()` в таких случаях никогда не приводят к ошибкам.

### 1.12.5 Замечания о посредниках

Некоторые из объектов, предоставляемых Flask, являются посредниками к другим объектам. Причина в том, что эти посредники являются общими для потоков и они скрыто передают объект для обработки соответствующему потоку.

В большинстве случаев об этом не стоит беспокоиться, но существует несколько исключительных случаев, когда хорошо знать, что объект на самом деле является посредником:

- Объекты-посредники не подделывают наследуемые типы, поэтому если понадобится провести проверки над реальным экземпляром объекта, это можно сделать это над экземпляром, который доступен через посредника (см. ниже *_get_current_object*).
- Если ссылка на объект имеет значение (например, для отправки signals).

Если нужно получить доступ к объекту, доступному через посредника, можно воспользоваться методом `_get_current_object()`:

```
app = current_app._get_current_object()
my_signal.send(app)
```

### 1.12.6 Защита контекста при ошибках

Случилась ли ошибка или нет, в конце обработки запроса контекст запроса извлекается из стека и все связанные с ним данные удаляются. Однако, во время разработки это может стать проблемой, потому что может потребоваться извлечь информацию из запроса, если произошло исключение. Во Flask 0.6 и более ранних версиях в режиме отладки, если произошло исключение, контекст запроса не извлекается, так что средство интерактивной отладки всё ещё может предоставить вам необходимую информацию.

Начиная со Flask 0.7 имеется возможность управлять этим поведением при помощи настройки параметра конфигурации `PRESERVE_CONTEXT_ON_EXCEPTION`. По умолчанию он связан с настройкой `DEBUG`. Если приложение находится в отладочном режиме, то контекст защищается, а если в рабочем, то - нет.

Не следует принудительно включать `PRESERVE_CONTEXT_ON_EXCEPTION` в рабочем режиме, потому что это может привести к утечке памяти приложения при каждом исключении. Однако, эта настройка может оказаться полезной в режиме разработки, чтобы воспроизвести ошибку, которая происходит только с настройками рабочего режима.

## 1.13 Модульные приложения Flask с использованием blueprint'ов

Добавлено в версии 0.7.

Flask использует концепцию *blueprint'ов* («blueprint» - «эскиз») для создания компонентов приложений и поддержки общих шаблонов внутри приложения или между приложениями. Blueprint'ы могут как значительно упростить большие приложения, так и предоставить общий механизм регистрации в приложении операций из расширений Flask. Объект `Blueprint` работает аналогично объекту приложения `Flask`, но в действительности он не является приложением. Обычно это лишь *эскиз* для сборки или расширения приложения.

### 1.13.1 Для чего нужны blueprint'ы?

Blueprint'ы во Flask могут пригодиться в случае, если нужно:

- Разделить приложения на набор blueprint'ов. Они идеальны для больших приложений; проект должен создать объект приложения, инициализировав несколько расширений, и зарегистрировав набор blueprint'ов.
- Зарегистрировать blueprint в приложении по определённом префиксу URL и/или в поддомене. Параметры в префиксе URL или поддомене становятся обычными аргументами представлений (со значениями по умолчанию) для всех функций представлений в blueprint'е.
- Зарегистрировать blueprint несколько раз в приложении с разными правилами URL.
- Предоставить фильтры шаблонов, статический файлы, шаблоны и другие вспомогательные средства с помощью blueprint'ов. Blueprint не является реализацией приложения или функций представлений.
- Зарегистрировать blueprint в приложении в любом из этих случаев при инициализации расширения Flask.

Blueprint во Flask не является подключаемым приложением, потому что это на самом деле не приложение – это набор операций, которые могут быть зарегистрированы в приложении, возможно даже не один раз. Почему бы не воспользоваться несколькими объектами приложений? Вы можете это сделать (обратитесь к разделу app-dispatch), но ваши приложения будут иметь раздельные файлы конфигурации и будут управляться слоем WSGI.

Вместо этого, blueprint'ы предоставляют разделение на уровне Flask, позволяя использовать общий файл конфигурации приложения и могут менять объект приложения необходимым образом при регистрации. Побочным эффектом будет невозможность отменить регистрацию blueprint'а, если приложение уже было создано, если только не уничтожить целиком весь объект приложения.

### 1.13.2 Концепция blueprint'ов

Основная концепция blueprint'ов заключается в том, что они записывают операции для выполнения при регистрации в приложении. Flask связывает функции представлений с blueprint'ами при обработке запросов и генерировании URL'ов от одной конечной точки к другой.

### 1.13.3 Мой первый blueprint

Приведём пример того, как выглядит основа простейшего blueprint'а. В данном случае мы хотим реализовать blueprint, который выполняет простую отрисовку статических шаблонов:

```
from flask import Blueprint, render_template, abort
from jinja2 import TemplateNotFound

simple_page = Blueprint('simple_page', __name__,
                        template_folder='templates')

@simple_page.route('/', defaults={'page': 'index'})
@simple_page.route('/<page>')
def show(page):
    try:
        return render_template('pages/%s.html' % page)
    except TemplateNotFound:
        abort(404)
```

При связывании функции при помощи декоратора `@simple_page.route`, blueprint записывает намерение зарегистрировать в приложении функцию *show*, когда blueprint будет зарегистрирован. Кроме того, декоратор предварит название конечной точки префиксом - именем blueprint'а, который был указан конструктору `Blueprint` (в данном случае это тоже `simple_page`).

### 1.13.4 Регистрация blueprint'ов

Как теперь зарегистрировать этот blueprint? Например, так:

```
from flask import Flask
from yourapplication.simple_page import simple_page

app = Flask(__name__)
app.register_blueprint(simple_page)
```

Если теперь посмотреть на правила, зарегистрированные в приложении, то можно обнаружить следующее:

```
[<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/' (HEAD, OPTIONS, GET) -> simple_page.show>]
```

Первым обычно является правило для статических файлов самого приложения. Следующие два правила - правила для функции *show* из blueprint'а `simple_page`. Как можно заметить, они тоже предварены именем blueprint'а и отделены от него точкой (`.`).

Однако, blueprint'ы можно связывать с другими местами:

```
app.register_blueprint(simple_page, url_prefix='/pages')
```

И, чтобы убедиться в этом, посмотрим на правила, сгенерированные на этот раз:

```
[<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/pages/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/pages/' (HEAD, OPTIONS, GET) -> simple_page.show>]
```

Плюс ко всему, можно зарегистрировать blueprint'ы несколько раз, хотя не каждый blueprint будет работать правильно. Это зависит от того, был ли реализован blueprint'е с учётом возможности многократного монтирования.

### 1.13.5 Ресурсы blueprint'а

Blueprint'ы могут, кроме всего прочего, предоставлять ресурсы. Иногда может потребоваться ввести дополнительный blueprint только ради предоставления ресурсов.

#### Каталог ресурсов blueprint'а

Как и обычные приложения, blueprint'ы задуманы для размещения в отдельном каталоге. Хотя несколько blueprint'ов можно разместить в одном и том же каталоге, так делать не рекомендуется.

Имя каталога берётся из второго аргумента `Blueprint`'а, которым обычно является *__name__*. Этот аргумент указывает, какой логический модуль или пакет Python соответствует blueprint'у. Если он указывает на существующий пакет Python (который является каталогом файловой системы), то он и будет каталогом ресурсов. Если это модуль, то каталогом ресурсов будет тот каталог, в котором

содержится модуль. Можно обратиться к свойству `Blueprint.root_path`, чтобы увидеть, что это за каталог:

```
>>> simple_page.root_path
'/Users/username/TestProject/yourapplication'
```

Для быстрого открытия ресурсов из этого каталога можно воспользоваться функцией `open_resource()`:

```
with simple_page.open_resource('static/style.css') as f:
    code = f.read()
```

#### Статические файлы

Blueprint может выставлять наружу каталог со статическими файлами, если в его конструкторе указан каталог файловой системы с помощью аргумента с ключевым словом *static_folder*. Аргумент может быть абсолютным путём или каталогом относительно каталога blueprint'а:

```
admin = Blueprint('admin', __name__, static_folder='static')
```

По умолчанию самая правая часть пути выставляется наружу в веб. Поскольку в данном случае указан каталог с именем `static`, он будет располагаться внутри каталога blueprint'а и будет называться `static`. В данном случае при регистрации blueprint'а в каталоге `/admin`, каталог `static` будет находиться в `/admin/static`.

Конечная точка будет иметь имя *blueprint_name.static*, так что можно генерировать URL'ы точно так же, как это делается для статического каталога приложения:

```
url_for('admin.static', filename='style.css')
```

#### Шаблоны

Если нужно выставить наружу каталог с шаблонами, это можно сделать указав параметр *template_folder* конструктору `Blueprint`:

```
admin = Blueprint('admin', __name__, template_folder='templates')
```

Как и в случае статических файлов, путь может быть абсолютным или располагаться в каталоге ресурсов blueprint'а. Каталог шаблона добавляется к пути поиска шаблонов, но с меньшим приоритетом, чем каталог шаблонов самого приложения. Таким образом, можно легко заменить шаблоны blueprint'а в самом приложении.

Например, если есть blueprint в каталоге `yourapplication/admin` и нужно отрисовать шаблон `'admin/index.html'`, а в параметре *template_folder* указан каталог *templates*, тогда нужно создать файл `yourapplication/admin/templates/admin/index.html`.

### 1.13.6 Генерирование URL'ов

Если нужно вставить ссылку с одной страницы на другую, можно воспользоваться функцией `url_for()`, как обычно: нужно просто добавить к конечной точке URL'а префикс с именем blueprint'а и точкой (`.`):

```
url_for('admin.index')
```

Наконец, если в функции представления blueprint'а или в отрисованном шаблоне нужно добавить ссылку на другую конечную точку того же blueprint'а, можно воспользоваться относительным перенаправлением, добавив префикс, состоящий только из точки:

```
url_for('.index')
```

Получится ссылка на `admin.index` в случае обработки текущего запроса в любой другой конечной точке blueprint'а.

## 1.14 Расширения Flask

Расширения Flask различным образом расширяют функциональность Flask. Например, добавляют поддержку баз данных и т.п.

### 1.14.1 Поиск расширений

Расширения Flask перечислены по ссылке Реестр расширений Flask и могут быть скачаны при помощи `easy_install` или `pip`. Если добавить расширение Flask в качестве зависимости в файл `requirements.rst` или `setup.py`, то обычно их можно установить с помощью простой команды или при установке приложения.

### 1.14.2 Использование расширений

Обычно расширения снабжены документацией, которая объясняет как их использовать. Нет общих правил обращения с расширениями, но их можно импортировать из обычных мест. Если у вас имеется расширение под названием `Flask-Foo` или `Foo-Flask`, в любом случае его можно импортировать из `flask.ext.foo`:

```
from flask.ext import foo
```

### 1.14.3 Flask до версии 0.8

Во Flask версии 0.7 и более ранних пакет `flask.ext` отсутствует, а вместо него можно импортировать расширение из `flaskext.foo` или `flask_foo`, в зависимости от способа поставки расширения. Если вы хотите разрабатывать приложения, поддерживающие Flask 0.7 и более ранние, импортировать всё равно нужно из пакета `flask.ext`. Есть модуль, предоставляющий этот пакет и позволяющий достичь совместимости со старыми версиями Flask. Его можно скачать с github: flaskext_compat.py

Вот пример его использования:

```
import flaskext_compat
flaskext_compat.activate()

from flask.ext import foo
```

После активации модуля `flaskext_compat` появится `flask.ext`, из которого можно импортировать расширения.

## 1.15 Заготовки для Flask

Некоторые вещи являются настолько общими, что существует большой шанс найти их в большинстве веб-приложений. К примеру, довольно много приложений используют реляционные базы данных и аутентификацию пользователей. В этом случае, скорее всего, ими в начале запроса будут открываться подключения к базе данных и получаться информация о пользователе, вошедшем в систему. В конце запроса, соединение с базой данной вновь закрывается.

В Архиве фрагментов для Flask можно найти много поддерживаемых энтузиастами фрагментов и заготовок.

### 1.15.1 Большие приложения во Flask

В больших приложениях лучше использовать пакеты вместо модулей. Это очень просто. Представьте, что небольшое приложение выглядит так:

```
/yourapplication
    /yourapplication.py
    /static
        /style.css
    /templates
        layout.html
        index.html
        login.html
        ...
```

#### Простые пакеты

Чтобы преобразовать его в чуть большее, просто создайте новый каталог приложения *yourapplication* внутри существующего и поместите всё в него. Переименуйте *yourapplication.py* в *__init__.py*. (Сначала убедитесь, что удалили все файлы *.pyc*, иначе скорее всего оно перестанет работать).

У вас должно получиться что-то такое:

```
/yourapplication
    /yourapplication
        /__init__.py
        /static
            /style.css
        /templates
            layout.html
            index.html
            login.html
            ...
```

В итоге должно получиться что-то вроде этого:

Но как теперь запустить приложение? Простой запуск `python yourapplication/__init__.py` не сработает. Скажем так, Python не хочет запускать модули в пакете, как программы. Но это не проблема, просто добавим во внутренний каталог *yourapplication* новый файл *runserver.py* со следующим содержимым:

```
from yourapplication import app
app.run(debug=True)
```

Что это нам даст? Теперь мы можем реструктурировать приложение и поделить его на несколько модулей. Единственное, о чём нужно помнить и соблюдать, это:

1. Создание приложения Flask должно происходить в файле *__init__.py*. Так каждый модуль сможет безопасно импортировать его, а переменная *__name__* примет значение имени соответствующего пакета.
2. Все функции представлений, к которым применён декоратор `route()`, должны быть импортированы в файл *__init__.py*. Не сам объект, но модуль с ними. Импорт модуля представлений производится **после создания объекта**.

Вот пример *__init__.py*:

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

import yourapplication.views
```

При этом *views.py* должен выглядеть так:

```
from yourapplication import app

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

В итоге должно получиться что-то вроде этого:

```
/yourapplication
    /runserver.py
    /yourapplication
        /__init__.py
        /views.py
        /static
            /style.css
        /templates
            layout.html
            index.html
            login.html
            ...
```

**Взаимный импорт**

Каждый Python-программист его ненавидит, но мы его добавили: два модуля зависят друг от друга. В данном случае *views.py* зависит от __init__.py. Мы предостерегаем вас от подобного приёма, но здесь он оправдан. Причина заключается в том, что мы на самом деле не используем представления в *__init__.py*, а просто убеждаемся в том, что модуль импортирован и делаем это в конце файла.

Однако, у этого подхода всё-же есть проблемы, но если вы хотите использовать декораторы, без него не обойтись. Обратитесь к разделу becomingbig за идеями о том, как с этим справиться.

### Работа с Blueprint'ами

Если у вас большое приложение, рекомендуется поделить его на небольшие группы - так, чтобы каждая группа была реализована с помощью blueprint'ов. Для плавного вникания в данную тему обратитесь к разделу *Модульные приложения Flask с использованием blueprint'ов* данной документации.

### 1.15.2 SQLAlchemy во Flask

Многие люди для доступа к базам данных предпочитают использовать SQLAlchemy. В этом случае для написания приложений на Flask больше подходят не модули, а пакеты, так как в этом случае можно поместить модели в отдельный модуль (*Большие приложения во Flask*). Хоть это и не обязательно, но всё же имеет смысл.

Есть четыре обычных способа использования SQLAlchemy. Остановимся на каждом из них подробнее:

#### Расширение Flask-SQLAlchemy

Поскольку SQLAlchemy - это обобщённый слой абстракции над базами данных и объектно-реляционное отображение, требующее предварительной настройки, существует расширение Flask, делающее всё необходимое за вас. Если нужно быстро начать работу, рекомендуем воспользоваться им.

Расширение Flask-SQLAlchemy можно скачать из PyPI.

#### Declarative

Расширение declarative в SQLAlchemy - это один из наиболее частых способов использования SQLAlchemy. Оно позволяет вам определять таблицы и модели одновременно, примерно так, как это делается в Django. В дополнение к следующему тексту рекомендуется обратиться к официальной документации по расширению declarative.

Вот пример модуля *database.py* для приложения:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
Base = declarative_base()
Base.query = db_session.query_property()

def init_db():
    # Здесь нужно импортировать все модули, где могут быть определены модели,
    # которые необходимым образом могут зарегистрироваться в метаданных.
    # В противном случае их нужно будет импортировать до вызова init_db()
    import yourapplication.models
    Base.metadata.create_all(bind=engine)
```

Для определения собственных моделей наследуйте от класса *Base*, который создан вышеприведённым кодом. Если вы удивлены, почему в этом примере не нужно заботиться о потоках (как мы делали в примере для SQLite3 с объектом `g` выше), то это потому что SQLAlchemy делает это самостоятельно при помощи `scoped_session`.

Чтобы использовать SQLAlchemy в приложении декларативным образом, необходимо поместить в модуль вашего приложения следующий код. Flask автоматически удалит за вас сеанс базы данных в конце запроса:

```
from yourapplication.database import db_session

@app.teardown_request
```

```
def shutdown_session(exception=None):
    db_session.remove()
```

Вот пример модели (поместите его, например, в *models.py*):

```
from sqlalchemy import Column, Integer, String
from yourapplication.database import Base

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True)
    email = Column(String(120), unique=True)

    def __init__(self, name=None, email=None):
        self.name = name
        self.email = email

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

Для создания базы данных можно воспользоваться функцией *init_ db*:

```
>>> from yourapplication.database import init_db
>>> init_db()
```

Вот так можно добавить новые записи в базу данных:

```
>>> from yourapplication.database import db_session
>>> from yourapplication.models import User
>>> u = User('admin', 'admin@localhost')
>>> db_session.add(u)
>>> db_session.commit()
```

Пример запроса:

```
>>> User.query.all()
[<User u'admin'>]
>>> User.query.filter(User.name == 'admin').first()
<User u'admin'>
```

### Ручное объектно-реляционное отображение

Ручное объектно-реляционное отображение имеет некоторые преимущества и недостатки по сравнению с декларативным подходом, рассмотренным выше. Главное отличие заключается в том, что таблицы и классы определяются раздельно, а затем создаётся их взаимное отображение. Этот подход более гибок, однако и более трудоёмок. В целом он работает подобно декларативному подходу, поэтому убедитесь в том, что поделили ваше приложение на несколько модулей в пакете.

Вот пример модуля *database.py* для вашего приложения:

```
from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import scoped_session, sessionmaker

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
metadata = MetaData()
```

```
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
def init_db():
    metadata.create_all(bind=engine)
```

Как и при декларативном подходе, вам необходимо закрывать сеанс после каждого запроса. Поместите следующие строки в модуль вашего приложения:

```
from yourapplication.database import db_session

@app.teardown_request
def shutdown_session(exception=None):
    db_session.remove()
```

Вот пример таблицы и модели (поместите их в *models.py*):

```
from sqlalchemy import Table, Column, Integer, String
from sqlalchemy.orm import mapper
from yourapplication.database import metadata, db_session

class User(object):
    query = db_session.query_property()

    def __init__(self, name=None, email=None):
        self.name = name
        self.email = email

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

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50), unique=True),
    Column('email', String(120), unique=True)
)
mapper(User, users)
```

Запрос и вставка записей делается точно так же, как в примере выше.

### Слой абстракции над SQL

Если вы хотите использовать только слой абстракции к базам данных (и SQL), вам потребуется только объект engine:

```
from sqlalchemy import create_engine, MetaData

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
metadata = MetaData(bind=engine)
```

Теперь можно объявить таблицы в коде, как в примере выше или автоматически загрузить их:

```
users = Table('users', metadata, autoload=True)
```

Чтобы вставить данные, вы можете воспользоваться методом *insert*. Прежде чем совершить транзакцию, необходимо сначала получить подключение:

```
>>> con = engine.connect()
>>> con.execute(users.insert(), name='admin', email='admin@localhost')
```

SQLAlchemy автоматически подтвердит транзакцию.

Для выполнения запросов можно воспользоваться напрямую объектом engine, либо использовать подключение:

```
>>> users.select(users.c.id == 1).execute().first()
(1, u'admin', u'admin@localhost')
```

С результатом запроса можно обращаться как со словарём:

```
>>> r = users.select(users.c.id == 1).execute().first()
>>> r['name']
u'admin'
```

В метод `execute()` можно также передавать строки с выражениями SQL:

```
>>> engine.execute('select * from users where id = :1', [1]).first()
(1, u'admin', u'admin@localhost')
```

За более подробной информацией о SQLAlchemy обратитесь к вебсайту website.

Примечание переводчика:

В сети можно найти русскоязычный викиучебник по использованию SQLAlchemy: website.

### 1.15.3 Загрузка файлов

Ну да, старая добрая проблема загрузки файлов. Основная мысль загрузки файлов на самом деле очень проста. В общих чертах это работает так:

1. Тег `<form>` помечается атрибутом `enctype=multipart/form-data`, а в форму помещается тег `<input type=file>`.
2. Приложение получает доступ к файлу через словарь `files` в объекте запроса.
3. Воспользуйтесь методом `save()` для того, чтобы сохранить временный файл в файловой системе для последующего использования.

#### Введение

Начнём с простейшего приложения, которое загружает файл в определённый каталог и отображает его пользователю. Вот начало нашего приложения:

```
import os
from flask import Flask, request, redirect, url_for
from werkzeug import secure_filename

UPLOAD_FOLDER = '/path/to/the/uploads'
ALLOWED_EXTENSIONS = set(['txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'])

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

Сначала нужно выполнить серию импортов. Большая часть понятна, `werkzeug.secure_filename()` рассматривается чуть позже. *UPLOAD_FOLDER* - это путь, куда будут загружаться файлы, а *ALLOWED_EXTENSIONS* - это набор допустимых расширений. Теперь вручную добавим в приложение правило для URL. Обычно мы так не делаем, но почему мы делаем это сейчас? Причина в том, что мы хотим заставить веб-сервер (или наш сервер приложения) обслуживать эти файлы и поэтому нам нужно генерировать правила для связывания URL с этими файлами.

Почему мы ограничили список допустимых расширений? Наверное вам совсем не хочется, чтобы пользователи могли загружать что угодно, если сервер напрямую отправляет данные клиенту. В таком случае вам нужно быть уверенными в том, что пользователи не загрузят файлы HTML, которые могут вызвать проблему XSS (см. Cross-Site Scripting (xss) - межсайтовый скриптинг). Также убедитесь в том, что запрещены файлы с расширением *.php*, если сервер их выполняет. Правда, кому нужен PHP на сервере? :)

Следующая функция проверяет, что расширение файла допустимо, загружает файл и перенаправляет пользователя на URL с загруженным файлом:

```
def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1] in ALLOWED_EXTENSIONS

@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        file = request.files['file']
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            return redirect(url_for('uploaded_file',
                                    filename=filename))
    return '''
    <!doctype html>
    <title>Upload new File</title>
    <h1>Upload new File</h1>
    <form action="" method=post enctype=multipart/form-data>
      <p><input type=file name=file>
         <input type=submit value=Upload>
    </form>
    '''
```

Что делает функция `secure_filename()`? Мы исходим из принципа «никогда не доверяй тому, что ввёл пользователь». Это справедливо и для имени загружаемого файла. Все отправленные из формы данные могут быть поддельными и имя файла может представлять опасность. Сейчас главное запомнить: всегда используйте эту функцию для получения безопасного имени файла, если собираетесь поместить файл прямо в файловую систему.

### Информация для профи

Может быть вам интересно, что делает функция `secure_filename()` и почему нельзя обойтись без её использования? Просто представьте, что кто-то хочет отправить следующую информацию в ваше приложение в качестве имени файла:

```
filename = "../../../../home/username/.bashrc"
```

Если считать, что `../` - это нормально, то при соединении этого имени с *UPLOAD_FOLDER*, пользователь может получить возможность изменять на файловой системе сервера те файлы, который он не должен изменять. Нужно немного разбираться в устройстве вашего приложения, но поверьте мне,

хакеры настойчивы :)

Посмотрим, как отработает функция:

```
>>> secure_filename('../../../../home/username/.bashrc')
'home_username_.bashrc'
```

---

Осталась последняя вещь: обслуживание загруженных файлов. Начиная с Flask 0.5 для этого можно использовать соответствующую функцию:

```
from flask import send_from_directory

@app.route('/uploads/<filename>')
def uploaded_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                               filename)
```

Другая возможность - зарегистрировать *uploaded_file* с помощью правила *build_only* и воспользоваться `SharedDataMiddleware`. Такой вариант будет работать и в более старых версиях Flask:

```
from werkzeug import SharedDataMiddleware
app.add_url_rule('/uploads/<filename>', 'uploaded_file',
                 build_only=True)
app.wsgi_app = SharedDataMiddleware(app.wsgi_app, {
    '/uploads':  app.config['UPLOAD_FOLDER']
})
```

Теперь, если запустить приложение, всё должно работать как положено.

### Улучшение загрузки

Добавлено в версии 0.6.

Как на самом деле Flask обрабатывает загрузку? Если файл достаточно мал, он сохраняется в памяти веб-сервера. В противном случае он помещается во временное место (туда, куда укажет `tempfile.gettempdir()`). Но как указать максимальный размер файла, после которого загрузка файла должна быть прервана? По умолчанию Flask не ограничивает размер файла, но вы можете задать лимит настройкой ключа конфигурации `MAX_CONTENT_LENGTH`:

```
from flask import Flask, Request

app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024
```

Код выше ограничит максимальный размер файла 16 мегабайтами. Если передаваемый файл окажется больше, Flask сгенерирует исключение `RequestEntityTooLarge`.

Эта функциональность была добавлена во Flask 0.6, но может быть реализована и в более ранних версиях при помощи наследовании от класса request. За более подробной информацией обратитесь к документации Werkzeug об обработке файлов.

### Индикаторы процесса загрузки

Многие разработчики придумывают считывать файл мелкими частями, сохранять процент загрузки в базу данных и давать возможность JavaScript считывать эти данные из клиента. Короче говоря,

клиент спрашивает у сервера каждые 5 секунд, сколько уже было передано. Почувствовали иронию ситуации? Клиент спрашивает у сервера о том, что уже и так знает.

Сейчас существуют способы получше, которые работают быстрее и более надёжны. В последнее время в вебе многое изменилось и теперь можно использовать HTML5, Java, Silverlight или Flash, чтобы сделать загрузку удобнее со стороны клиента. Посмотрите на следующие библиотеки, предназначенные именно для этого:

- Plupload - HTML5, Java, Flash
- SWFUpload - Flash
- JumpLoader - Java

#### Простейшее решение

Поскольку общая процедура загрузки файлов остаётся неизменной для всех приложений, занимающихся загрузкой файлов, для Flask есть расширение под названием Flask-Uploads, которое реализует полностью самостоятельный механизм загрузки с белым и чёрным списком расширений и т.п.

### 1.15.4 Кэширование

Когда приложение работает медленно, можно попробовать воспользоваться кэшированием. По крайней мере это самый простой способ ускорения программы. Как работает кэширование? Допустим, имеется функция, которая работает довольно долго, но результаты её работы будут пригодны для использования даже через 5 минут. Смысл в том, что на самом деле мы на некоторое время помещаем результаты расчётов в кэш.

Сам по себе Flask не умеет кэшировать, но Werkzeug, одна из библиотек, на которой он основан, имеет базовую поддержку кэширования. Библиотека поддерживает различные средства кэширования, но скорее всего вам захочется воспользоваться сервером memcached.

#### Настройка кэша

Объект кэша создаётся единожды и продолжает работать примерно так же, как это происходит с объектами `Flask`. Если речь идёт о сервере разработки, то можно создать объект `SimpleCache`, который представляет собой простейший кэш, хранящий элементы в памяти интерпретатора Python:

```
from werkzeug.contrib.cache import SimpleCache
cache = SimpleCache()
```

Для использования memcached нужно установить один из поддерживаемых модулей memcache (его можно взять на PyPI) и запустить где-нибудь сервер memcached. Теперь можно подключиться к серверу memcached:

```
from werkzeug.contrib.cache import MemcachedCache
cache = MemcachedCache(['127.0.0.1:11211'])
```

Если вы пользуетесь App Engine, можно легко подключиться к серверу memcache из App Engine:

```
from werkzeug.contrib.cache import GAEMemcachedCache
cache = GAEMemcachedCache()
```

#### Использование кэша

Как теперь воспользоваться этим кэшем? Имеется две очень важные операции: `get()` и `set()`. Вот как их использовать:

Чтобы получить элемент из кэша, вызовите `get()`, указав ей ключ - имя элемента. Если что-нибудь есть в кэше, оно будет возвращено. В противном случае функция вернёт *None*:

```
rv = cache.get('my-item')
```

Чтобы добавить элемент в кэш, воспользуйтесь методом `set()`. Первый аргумент - это ключ, а второй - его значение. Также можно указать время кэширования, по истечении которого элемент будет автоматически удалён.

Вот полный пример того, как это обычно выглядит:

```
def get_my_item():
    rv = cache.get('my-item')
    if rv is None:
        rv = calculate_value()
        cache.set('my-item', rv, timeout=5 * 60)
    return rv
```

#### Примечания переводчика

Под App Engine подразумевается облачный сервис веб-приложений от Google: https://appengine.google.com/start. Чуть подробнее о нём можно узнать из статьи на Википедии: Google App Engine.

### 1.15.5 Декораторы представлений во Flask

В Python имеется любопытная функциональность, которая называется декораторами функций. Декораторы позволяют сделать веб-приложения изящнее. Поскольку каждое представление во Flask является функцией, можно использовать декораторы для добавления дополнительной функциональности к одной или более функций. Декоратор `route()` - один из тех, который вы возможно уже используете. Но можно реализовать собственные декораторы. Например, представьте, что есть представление, которое должно быть доступно только аутентифицированным пользователям. Если пользователь вошёл на сайт, но не аутентифицировался, его нужно перенаправить на страницу аутентификации. Это хороший случай, в котором декоратор может прийтись как нельзя кстати.

#### Декоратор «необходима аутентификация»

Ну что ж, давайте реализуем этот декоратор. Декоратор - это функция, возвращающая функцию. На самом деле очень просто. Единственное, о чём нужно помнить при реализации чего-то подобного, это об обновлении *__name__*, *__module__* и других атрибутов функции. Часто об этом забывают, нам не потребуется делать это вручную, потому что есть функция, которая может сделать это за нас и используется в качестве декоратора (`functools.wraps()`).

Пример подразумевает, что страница для ввода учётных данных называется `'login'`, а текущий пользователь хранится в *g.user*. Если пользователь не аутентифицирован, то в *g.user* хранится *None*:

```
from functools import wraps
from flask import g, request, redirect, url_for

def login_required(f):
```

```
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function
```

Итак, как же теперь воспользоваться этим декоратором? Применим его в качестве наиболее глубоко вложенного декоратора. При дальнейшем применении декораторов помните, что декоратор `route()` должен быть самым внешним:

```
@app.route('/secret_page')
@login_required
def secret_page():
    pass
```

### Кэширующий декоратор

Представьте, что у вас есть функция представления, которая выполняет сложные расчёты и поэтому вам хочется, чтобы результаты расчётов в течение некоторого времени выдавались из кэша. Для решения этой задачи подойдёт декоратор. Подразумевается, что кэш уже настроен, как описано в разделе *Кэширование*.

Вот пример функции кэширования. Она использует в качестве ключа для кэша некий префикс (на самом деле это строка формата) и текущий путь запроса. Отметим, что мы создадим функцию, которая создаст декоратор, с помощью которого мы задекорируем функцию. Звучит пугающе? На самом деле это впрямь немного сложнее, но код остаётся достаточно прямолинейным для того, чтобы его ещё можно было прочитать.

Задекорированная функция будет работать следующим образом:

1. приготовит уникальный ключ кэша для текущего запроса, основываясь на текущем пути.
2. получит значение для этого ключа из кэша. Если кэш вернул что-нибудь, мы вернём это значение.
3. в противном случае будет вызвана оригинальная функция, возвращённое значение будет помещено в кэш с указанным временем кэширования (по умолчанию - 5 минут).

Вот код:

```
from functools import wraps
from flask import request

def cached(timeout=5 * 60, key='view/%s'):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            cache_key = key % request.path
            rv = cache.get(cache_key)
            if rv is not None:
                return rv
            rv = f(*args, **kwargs)
            cache.set(cache_key, rv, timeout=timeout)
            return rv
        return decorated_function
    return decorator
```

Notice that this assumes an instantiated *cache* object is available, see *Кэширование* for more information. В этом примере подразумевается, что объект *cache* уже инициализирован, как описано в разделе *Кэширование*.

### Шаблонизирующий декоратор

В TurboGears некоторое время назад было распространено использование шаблонизирующих декораторов. Смысл этого декоратора заключается в том, что функция представления возвращает словарь со значениями для шаблона, а шаблон отрисовывается автоматически. В этом случае следующие три примера делают одно и то же:

```
@app.route('/')
def index():
    return render_template('index.html', value=42)

@app.route('/')
@templated('index.html')
def index():
    return dict(value=42)

@app.route('/')
@templated()
def index():
    return dict(value=42)
```

Как можно заметить, если имя шаблона не указано, используется конечная точка карты URL с точками, преобразованными в косые черты и с добавленным справа текстом `'.html'`. В противном случае, используется шаблон с указанным именем. Когда задекорированная функция завершается, возвращённый ею словарь передаётся в функцию отрисовки шаблона. Если ничего не возвращено, подразумевается пустой словарь, а если возвращён не словарь, мы возвращаем это значение неизменным. Таким образом по-прежнему можно пользоваться функцией redirect или возвращать обычные строки.

Вот код этого декоратора:

```
from functools import wraps
from flask import request

def templated(template=None):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            template_name = template
            if template_name is None:
                template_name = request.endpoint \
                    .replace('.', '/') + '.html'
            ctx = f(*args, **kwargs)
            if ctx is None:
                ctx = {}
            elif not isinstance(ctx, dict):
                return ctx
            return render_template(template_name, **ctx)
        return decorated_function
    return decorator
```

#### Декоратор конечной точки

Если хочется воспользоваться системой маршрутизации werkzeug для достижения большей гибкости, нужно отобразить конечную точку в соответствии с правилом `Rule` для для функции представления. Это можно сделать с помощью декоратора. Например:

```
from flask import Flask
from werkzeug.routing import Rule

app = Flask(__name__)
app.url_map.add(Rule('/', endpoint='index'))

@app.endpoint('index')
def my_index():
    return "Hello world"
```

### 1.15.6 Всплывающие сообщения

Хорошие приложения и интерфейсы пользователя дают обратную связь. Если пользователь не получает достаточной обратной связи, вскоре он может начать ненавидеть приложение. Flask предоставляет по-настоящему простой способ дать обратную связь пользователю при помощи системы всплывающих сообщений. Система всплывающих сообщений обычно делает возможным записать сообщение в конце запроса и получить к нему доступ во время обработки следующего и только следующего запроса. Обычно эти сообщения используются в шаблонах макетов страниц, которые его и отображают.

#### Пример всплывающих сообщений

Вот полный пример:

```
from flask import Flask, flash, redirect, render_template, \
     request, url_for

app = Flask(__name__)
app.secret_key = 'some_secret'

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

@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != 'admin' or \
                request.form['password'] != 'secret':
            error = 'Invalid credentials'
        else:
            flash('You were successfully logged in')
            return redirect(url_for('index'))
    return render_template('login.html', error=error)

if __name__ == "__main__":
    app.run()
```

А вот шаблон макета `layout.html`, отображающий сообщение:

```
<!doctype html>
<title>My Application</title>
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
      <li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
{% block body %}{% endblock %}
```

А это - шаблон index.html:

```
{% extends "layout.html" %}
{% block body %}
  <h1>Overview</h1>
  <p>Do you want to <a href="{{ url_for('login') }}">log in?</a>
{% endblock %}
```

И, конечно, шаблон страницы входа:

```
{% extends "layout.html" %}
{% block body %}
  <h1>Login</h1>
  {% if error %}
    <p class=error><strong>Error:</strong> {{ error }}
  {% endif %}
  <form action="" method=post>
    <dl>
      <dt>Username:
      <dd><input type=text name=username value="{{
          request.form.username }}">
      <dt>Password:
      <dd><input type=password name=password>
    </dl>
    <p><input type=submit value=Login>
  </form>
{% endblock %}
```

### Всплывающие сообщения с категориями

Добавлено в версии 0.3.

Всплывающему сообщению можно назначить категорию. По умолчанию, если категория не указана, используется категория `'message'` - сообщение. Для более качественной обратной связи с пользователем можно указывать другие категории. Например, сообщения об ошибках должны отображаться на красном фоне.

Для вывода всплывающего сообщения с другой категорией, просто передайте её вторым аргументом функции `flash()`:

```
flash(u'Invalid password provided', 'error')
```

Внутри шаблона можно сообщить функции `get_flashed_messages()`, что нужно вернуть ещё и категорию. В этом случае цикл выглядит несколько иначе:

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    <ul class=flashes>
    {% for category, message in messages %}
      <li class="{{ category }}">{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

Это просто пример того, как можно отображать всплывающие сообщения. Можно использовать категорию для добавления к сообщению префикса, например, `<strong>Ошибка:</strong>`.

#### Фильтрация всплывающих сообщений

Добавлено в версии 0.9.

При желании можно передать список категорий, который будет использован функцией `get_flashed_messages()` для фильтрации сообщений. Это полезно в тех случаях, если нужно выводить сообщения каждой категории в отдельном блоке.

```
{% with errors = get_flashed_messages(category_filter=["error"]) %}
{% if errors %}
<div class="alert-message block-message error">
  <a class="close" href="#">×</a>
  <ul>
    {%- for msg in errors %}
    <li>{{ msg }}</li>
    {% endfor -%}
  </ul>
</div>
{% endif %}
{% endwith %}
```

#### Примечания переводчика

В оригинале были «мигающие» или «вспыхивающие» сообщения, но я счёл дословный перевод не вполне соответствующим истине - эти сообщения не мигают и не вспыхивают. Хотя они также и не всплывают, но это слово показалось мне более удачным.

### 1.15.7 Собственные страницы ошибок

Flask поставляется с удобной функцией `abort()`, которая досрочно прерывает запрос с кодом ошибки HTTP. В комплекте с ней идёт простая чёрно-белая страница ошибки с небольшим описанием, ничего особого.

В зависимости от кода ошибки, с разной степенью вероятности, пользователь может увидеть эту страницу.

#### Обычные коды ошибок

Чаще всего, даже если приложение работает правильно, пользователь может увидеть следующие коды ошибок:

*404 Не найдено*

Старое доброе сообщение «чувак, ты допустил ошибку при вводе URL». Встречается настолько часто, что даже новички в интернете знают, что означает 404: блин, тут нет того, что я искал. Неплохо поместить на этой странице что-то более полезное, хотя-бы ссылку на стартовую страницу.

***403 Доступ запрещён*** Если на сайте есть какая-нибудь разновидность ограничения доступа, может понадобиться отправить код 403 при попытке доступа к ресурсам, доступ к которым запрещён. Позволяет не потерять пользователя, который попытался получить доступ к запрещённому ресурсу.

***410 Удалено*** Вы знаете, что у «404 Не найдено» есть брат по имени «410 Удалено»? Мало кто делает так, что прежде существовавшие, но ныне удалённые ресурсы отвечают ошибкой 410 вместо 404. Если вы не удаляете документы из базы данных навсегда, а просто помечаете их как удалённые, можно сделать пользователю одолжение и воспользоваться кодом 410, а не отображать сообщение о том, что искомое было удалено навечно.

***500 Внутренняя ошибка сервера*** Обычно происходит в случае ошибок в программе или при повышенной нагрузке на сервер. Ужасно хорошо было бы иметь на этот случай приятную страницу, потому что рано или поздно в приложении может случится сбой (см. также: *Журналирование ошибок приложения*).

### Обработчики ошибок

Обработчик ошибок - это функция, похожая на функцию представления, только она принимает возникающие ошибки. Большинство ошибок - это ошибки типа `HTTPException`, но это могут быть и другие ошибки, потому что в обработчик внутренних ошибок сервера попадают все остальные не перехваченные экземпляры исключений.

Обработчик ошибок регистрируется при помощи декоратора `errorhandler()` с кодом ошибки. Помните, что Flask не устанавливает код ошибки за вас, поэтому удостоверьтесь, что возвращаете в ответе код статуса HTTP.

Здесь приведён пример обработки исключения «404 Страница не найдена»:

```
from flask import render_template

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

Шаблон для этого примера может быть таким:

```
{% extends "layout.html" %}
{% block title %}Page Not Found{% endblock %}
{% block body %}
  <h1>Page Not Found</h1>
  <p>What you were looking for is just not there.
  <p><a href="{{ url_for('index') }}">go somewhere nice</a>
{% endblock %}
```

### Примечания переводчика

После прочтения перевода у вас может возникнуть недоумение - что за «чувак» и «клёво»? Я не стал бы добавлять в текст эти слова по собственной инициативе - я просто попытался в меру своих скудных

способностей воспроизвести стиль оригинального текста. В оригинале этим словам соответствуют chap и nice.

Более подробный список кодов статуса HTTP можно увидеть здесь: Список кодов состояния HTTP.

## 1.16 Варианты развёртывания

В зависимости от того, что вы имеете в наличии, есть несколько вариантов запуска приложений Flask. Во время разработки вы можете использовать встроенный сервер, но для развёртывания приложений для промышленного использования необходимо использовать вариант полноценного развёртывания (не используйте в качестве «боевого» встроенный сервер для разработки). Несколько вариантов развёртывания докементированы в данном разделе.

Если у вас другой WSGI сервер, чтобы понять, как с ним использовать WSGI-приложения, читайте его документацию. Просто не забывайте, что фактически объект приложения `Flask` и есть WSGI-приложение.

Для того, чтобы быстро взять и применить приведённые варианты, читайте *Развёртывание приложения на веб-сервере* в разделе Быстрый старт.

### 1.16.1 Настройка mod_wsgi (Apache) для Flask

Если у вас веб-сервер Apache, рекомендуется использовать mod_wsgi.

---

**Предварительная проверка**

Удостоверьтесь, что все вызовы `app.run()` в файле приложения находятся внутри блока `if __name__ == '__main__':` или вынесены в отдельный файл. Просто убедитесь в отсутствии подобных вызовов, потому что если вы решили воспользоваться mod_wsgi для запуска приложения, то запускать локальный сервер WSGI не нужно.

---

#### Установка *mod_wsgi*

Если *mod_wsgi* ещё не установлен, его можно установить с помощью менеджера пакетов или собрать самостоятельно. В инструкции по установке mod_wsgi описывается установка из исходных текстов в UNIX-системах.

Если вы используете Ubuntu/Debian, можно воспользоваться apt-get:

```
# apt-get install libapache2-mod-wsgi
```

На FreeBSD *mod_wsgi* можно установить сборкой из порта *www/mod_wsgi* или при помощи pkg_add:

```
# pkg_add -r mod_wsgi
```

Если используется pkgsrc, можно установить *mod_wsgi*, собрав из пакета *www/ap2-wsgi*.

Если случится ошибка сегментации дочернего процесса после первой перезагрузки apache, можно спокойно проигнорировать её. Просто перезапустите сервер.

### Создание файла *.wsgi*

Для запуска приложения нужен файл *yourapplication.wsgi*. Этот файл содержит код, выполняемый *mod_wsgi* для получения объекта приложения Объект с именем *application* в этом файле будет использоваться в качестве приложения.

Для большинства приложений будет достаточно такого файла:

```
from yourapplication import app as application
```

Если у вас нет фабричной функции для создания приложений, но есть экземпляр-одиночка, можно просто импортировать его как *application*.

Сохраните этот файл где-нибудь, где сможете его найти (например, в */var/www/yourapplication*) и удостоверьтесь, что *yourapplication* и все используемые им библиотеки находятся в списке путей загрузки python. Если вы не хотите делать его общедоступным для всей системы, воспользуйтесь виртуальным экземпляром python. Помните, что при этом вам нужно будет установить ваше приложение внутрь virtualenv. Или можно отредактировать переменную path внутри *.wsgi* перед импортом:

```
import sys
sys.path.insert(0, '/path/to/the/application')
```

### Настройка Apache

Наконец, нужно создать файл с настройками Apache для запуска приложения. В следующем примере мы говорим *mod_wsgi* выполнить приложение от имени отдельного пользователя в целях безопасности:

```
<VirtualHost *>
    ServerName example.com

    WSGIDaemonProcess yourapplication user=user1 group=group1 threads=5
    WSGIScriptAlias / /var/www/yourapplication/yourapplication.wsgi

    <Directory /var/www/yourapplication>
        WSGIProcessGroup yourapplication
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

Замечание: WSGIDaemonProcess не реализован в Windows и Apache не запустится с указанной выше конфигурацией. В системе Windows эти строки нужно удалить:

```
<VirtualHost *>
        ServerName example.com
        WSGIScriptAlias / C:\yourdir\yourapp.wsgi
        <Directory C:\yourdir>
                Order deny,allow
                Allow from all
        </Directory>
</VirtualHost>
```

За более подробной информацией обратитесь к странице mod_wsgi wiki.

### Решение проблем

Если приложение не запускается, попробуйте следующие решения:

**Проблема:** приложение не запускается, в журнале ошибок появляется сообщение SystemExit ignored

Произошёл вызов `app.run()` в файле вашего приложения, в котором не было предохранительного условия `if __name__ == '__main__':`. Либо удалите из файла этот вызов `run()`, либо поместите его в отдельный файл *run.py*, либо поместите его в подобный блок if.

**Проблема: приложение сообщает об ошибках доступа** Возможно это вызвано тем, что ваше приложение работает от неправильного пользователя. Проверьте, что каталоги, к которым необходим доступ из приложения, имеют правильные разрешения, а приложение запущено от правильного пользователя (параметры `user` и `group` в директиве *WSGIDaemonProcess*).

**Проблема: приложение завершается с выводом сообщения об ошибке** Помните, что mod_wsgi запрещает делать что-либо с `sys.stdout` и `sys.stderr`. Можно выключить эту защиту, прописав в конфигурации следующую настройку:

```
WSGIRestrictStdout Off
```

Или можно заменить стандартный вывод в файле .wsgi на другой поток:

import sys sys.stdout = sys.stderr

**Проблема: доступ к ресурсам приводит к ошибкам ввода-вывода** Возможно ваше приложение является символической ссылкой на один из файлов .py, находящихся в каталоге пакетов сайта. Такой приём не работает, поэтому удостоверьтесь, что поместили каталог в пути поиска python или преобразуйте ваше приложение в пакет.

Причина заключается в том, что имя файла модуля используется для поиска ресурсов, а в случае символической ссылки будет использоваться неправильное имя файла.

### Поддержка автоматической перезагрузки

Чтобы облегчить работу инструментов установки, можно включить поддержку автоматической перезагрузки. Когда файл *.wsgi* изменится, *mod_wsgi* перезагрузит для нас все процессы демона.

Для этого просто добавьте следующие директивы в раздел *Directory*:

```
WSGIScriptReloading On
```

### Работа с виртуальными окружениями

Польза от виртуальных окружений заключается в том, что они позволяют не устанавливать необходимые зависимости на уровне всей системы, что позволяет достичь большего контроля над используемыми пакетами. Если вы решили воспользоваться виртуальным окружением совместно с mod_wsgi, нужно слегка изменить файл *.wsgi*.

Добавьте следующие строки в начало файла *.wsgi*:

```
activate_this = '/path/to/env/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

Эти строки настроят пути загрузки в соответствии с настройками виртуального окружения. Помните, что это должен быть абсолютный путь.

### 1.16.2 FastCGI

FastCGI - это один из вариантов развёртывания приложения на таких серверах, как nginx, lighttpd и cherokee; за описанием других опций обратитесь к разделам deploying-uwsgi и deploying-wsgi-standalone. Для использования приложения WSGI с любым из этих серверов необходимо сначала настроить сервер FastCGI. Наиболее популярен flup, который будет использоваться в этом руководстве. Убедитесь в том, что установили его, Прежде чем продолжить чтение убедитесь, что он установлен.

---

**Предварительная проверка**

Удостоверьтесь, что вызовы `app.run()` в файле приложения находятся внутри блока `if __name__ == '__main__':` или вынесены в отдельный файл. Просто убедитесь в отсутствии подобных вызовов, потому что если вы решили воспользоваться FastCGI для запуска приложения, то запускать локальный сервер WSGI не нужно.

---

#### Создание файла *.fcgi*

Для начала нужно создать файл сервера FastCGI. Давайте назовём его *yourapplication.fcgi*:

```
#!/usr/bin/python
from flup.server.fcgi import WSGIServer
from yourapplication import app

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

Этого достаточно для работы Apache, однако nginx и старые версии lighttpd требуют явного указания сокетов для связи с сервером FastCGI. Для этого нужно передать путь к сокет-файлу в `WSGIServer`:

```
WSGIServer(application, bindAddress='/path/to/fcgi.sock').run()
```

Этот путь должен быть точно таким же, какой был указан в настройках сервера.

Сохраните файл *yourapplication.fcgi* где-нибудь, где вы сможете потом найти его. Неплохо положить его в */var/www/yourapplication* или в какое-то другое подходящее место.

Убедитесь, что у этого файла установлен флаг выполнения, чтобы сервер мог его выполнить:

```
# chmod +x /var/www/yourapplication/yourapplication.fcgi
```

#### Настройка Apache

Приведённый выше пример достаточно хорош для того, чтобы использовать его при развёртывании с Apache, однако файл *.fcgi* будет встречаться в URL приложения, например: example.com/yourapplication.fcgi/news/. Есть несколько способов настройки приложения для того, чтобы убрать yourapplication.fcgi из URL. Предпочтительный способ - это использование директивы конфигурации ScriptAlias:

```
<VirtualHost *>
    ServerName example.com
    ScriptAlias / /path/to/yourapplication.fcgi/
</VirtualHost>
```

Если задать ScriptAlias нельзя, например на веб-узле, настроенном для нескольких пользователей, то можно воспользоваться промежуточным приложением WSGI для удаления yourapplication.fcgi из URL. Настройте .htaccess:

```
<IfModule mod_fcgid.c>
   AddHandler fcgid-script .fcgi
   <Files ~ (\.fcgi)>
       SetHandler fcgid-script
       Options +FollowSymLinks +ExecCGI
   </Files>
</IfModule>

<IfModule mod_rewrite.c>
   Options +FollowSymlinks
   RewriteEngine On
   RewriteBase /
   RewriteCond %{REQUEST_FILENAME} !-f
   RewriteRule ^(.*)$ yourapplication.fcgi/$1 [QSA,L]
</IfModule>
```

Теперь настроим yourapplication.fcgi:

```
#!/usr/bin/python
#: optional path to your local python site-packages folder
import sys
sys.path.insert(0, '<your_local_path>/lib/python2.6/site-packages')

from flup.server.fcgi import WSGIServer
from yourapplication import app

class ScriptNameStripper(object):
   def __init__(self, app):
       self.app = app

   def __call__(self, environ, start_response):
       environ['SCRIPT_NAME'] = ''
       return self.app(environ, start_response)

app = ScriptNameStripper(app)

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

### Настройка lighttpd

Базовая настройка FastCGI для lighttpd выглядит следующим образом:

```
fastcgi.server = ("/yourapplication.fcgi" =>
    ((
        "socket" => "/tmp/yourapplication-fcgi.sock",
        "bin-path" => "/var/www/yourapplication/yourapplication.fcgi",
        "check-local" => "disable",
        "max-procs" => 1
    ))
)

alias.url = (
```

```
    "/static/" => "/path/to/your/static"
)

url.rewrite-once = (
    "^(/static($|/.*))$" => "$1",
    "^(/.*)$" => "/yourapplication.fcgi$1"
```

Не забудьте включить модули FastCGI, alias и rewrite. Эта настройка закрепит приложение за */yourapplication*. Если нужно, чтобы приложение работало в корне URL, понадобится обойти недоработку lighttpd при помощи промежуточного приложения `LighttpdCGIRootFix`.

Убедитесь, что применяете его лишь в том случае, если подключили приложение к корню URL. А также, обратитесь к документации Lighttpd за более подробной информацией сюда FastCGI and Python (отметим, что явная передача сокет-файла в run() больше не требуется).

### Настройка nginx

Установка приложений FastCGI в nginx немного отличается, потому что по умолчанию программе не передаются параметры FastCGI.

Базовая конфигурация FastCGI nginx для flask выглядит следующим образом:

```
location = /yourapplication { rewrite ^ /yourapplication/ last; }
location /yourapplication { try_files $uri @yourapplication; }
location @yourapplication {
    include fastcgi_params;
    fastcgi_split_path_info ^(/yourapplication)(.*)$;
    fastcgi_param PATH_INFO $fastcgi_path_info;
    fastcgi_param SCRIPT_NAME $fastcgi_script_name;
    fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}
```

Эта конфигурация привязывает приложение к */yourapplication*. Привязать приложение к корню URL несколько проще, потому что не нужно думать о том, какие значения использовать в *PATH_INFO* и *SCRIPT_NAME*:

```
location / { try_files $uri @yourapplication; }
location @yourapplication {
    include fastcgi_params;
    fastcgi_param PATH_INFO $fastcgi_script_name;
    fastcgi_param SCRIPT_NAME "";
    fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}
```

### Запуск процессов FastCGI

Поскольку Nginx и другие серверы не загружают приложения FastCGI, это нужно сделать самостоятельно. Процессами FastCGI может управлять программа Supervisor. Можно поискать другие диспетчеры процессов FastCGI или написать сценарий для запуска файла *.fcgi* во время загрузки, например, с помощью сценария SysV `init.d`. В качестве временного решения может подойти запуск сценария *.fcgi* из программы GNU screen. Обратитесь к странице руководства screen за более подробной информацией, однако стоит заметить, что после перезагрузки системы запуск придётся повторять вручную:

```
$ screen
$ /var/www/yourapplication/yourapplication.fcgi
```

### Отладка

На большинстве веб-серверов становится всё труднее отлаживать приложения FastCGI. Довольно часто единственное, о чём сообщают журналы сервера - это о неожиданном окончании заголовков. Для отладки приложения единственное подходящее средство диагностики - это переключиться на нужного пользователя и запустить приложение вручную.

В следующем примере предполагается, что приложение называется *application.fcgi*, а веб-сервер работает от имени пользователя *www-data*:

```
$ su www-data
$ cd /var/www/yourapplication
$ python application.fcgi
Traceback (most recent call last):
  File "yourapplication.fcgi", line 4, in <module>
ImportError: No module named yourapplication
```

In this case the error seems to be «yourapplication» not being on the python path. Common problems are:

- Relative paths being used. Don't rely on the current working directory
- The code depending on environment variables that are not set by the web server.
- Different python interpreters being used.

В данном случае ошибка вызвана тем, что «yourapplication» не найден в путях поиска python. Обычно это происходит по одной из следующих причин:

- Указаны относительные пути, которые не работают относительно текущего каталога.
- Выполнение программы зависит от переменных окружения, которые не заданы для веб-сервера.
- Используется интерпретатор python другой версии.

### Примечания переводчика

В случае настройки Lighttpd не нужно писать никаких сценариев SysV *init.d*, потому что:

1. Lighttpd может сам управлять FastCGI-процессами на локальном компьютере, самостоятельно порождая необходимое их количество (с учётом настроенного лимита),
2. в рамках проекта Lighttpd разрабатывается собственный диспетчер процессов FastCGI - *spawn-fcgi*, который не настолько продвинут, чтобы регулировать количество необходимых процессов, но по крайней мере указанное количество процессов запустить и поддерживать сможет.

Обычно *spawn-fcgi* применяется в тех случаях, когда приложение FastCGI работает на отдельном от веб-сервера компьютере или нужно запустить приложение от имени другого пользователя, например, для изоляции друг от друга приложений разных пользователей, работающих на одном сервере. Например, так: Настройка FastCGI и PHP с индивидуальными правами пользователей.

И, наконец, никто не мешает использовать spawn-fcgi совместно с nginx.

Глава 2

# Справка об API

Если Вы ищите информацию по конкретным функции, классу или мотоду, эта часть документации для Вас.

Глава 3

# Дополнительные заметки

Здесь заметки для заинтересованных - о дизайне, правовая информация и лог изменений.

## 3.1 Проектные решения во Flask

Если вам кажется странным, почему Flask делает некоторые вещи именно так, а не как-нибудь иначе, этот раздел для вас. После его прочтения у вас должно появиться понимание, почему некоторые проектные решения, на первый взгляд кажутся случайными и неожиданными, особенно по сравнению с другими фреймворками.

### 3.1.1 Явный объект приложения

Веб-приложения на Python, основанные на WSGI, должны обладать центральным вызываемым объектом, реализующим приложение. Во Flask это экземпляр класса `Flask`. Каждое приложение Flask должно само создать экземпляр этого класса и передать ему имя модуля, но почему Flask не может сделать это сам?

Если бы не было явного объекта приложения, то следующий код:

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

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

Выглядел бы так:

```
from hypothetical_flask import route

@route('/')
```

```
def index():
    return 'Hello World!'
```

Для этого есть три основных причины. Наиболее важная заключается в том, что неявный объект приложения может существовать одновременно только в одном экземпляре. Есть способы сымитировать множество приложений в одном объекте приложения, например, поддерживая пачку приложений, но это вызовет некоторые проблемы, в детали которых мы не станем вдаваться. Следующий вопрос: в каких случаях микрофреймворку может понадобиться создавать больше одного приложения одновременно? Хорошим примером является модульное тестирование. Когда вам нужно протестировать что-нибудь, может оказаться полезным создать минимальное приложение для тестирования определённого поведения. Когда объект приложения удалятся, занятые им ресурсы снова становятся свободными.

Другой возможной причиной, почему могут пригодиться явные объекты приложений, может быть необходимость обернуть объект в дополнительный код, что можно сделать создав подкласс базового класса (`Flask`) для изменения части его поведения. Это невозможно сделать без костылей, если объект был создан до этого, на основе класса, который вам не доступен.

Но существует и ещё одна очень важная причина, почему Flask зависит от явных экземпляров классов: имя пакета. Когда бы вы ни создавали экземпляр Flask, обычно вы передаёте ему *__name__* в качестве имени пакета. Flask использует эту информацию для правильной загрузки ресурсов относящихся к модулю. При помощи выдающихся возможностей Python к рефлексии, вы можете получить доступ к пакету, чтобы узнать, где хранятся шаблоны и статические файлы (см. `open_resource()`). Очевидно, что фреймворкам не требуется какая-либо настройка и они по-прежнему смогут загружать шаблоны относительно модуля вашего приложения. Но они используют для этого текущий рабочий каталог, поэтому нет возможности надёжно узнать, где именно находится приложение. Текущий рабочий каталог одинаков в рамках процесса, поэтому если запустить несколько приложений в рамках одного процесса (а это может происходить в веб-сервере, даже если вы об этом не догадываетесь), путь может измениться. Даже хуже: многие веб-серверы устанавливают рабочим каталогом не каталог приложения, а корневой каталог документов, который обычно является другим каталогом.

Третья причина заключается в том, что «явное лучше неявного». Этот объект - ваше приложение WSGI, вам больше не нужно помнить о чём-то ещё. Если вам нужно воспользоваться промежуточным модулем WSGI, просто оберните его этим модулем и всё (хотя есть способы лучше, чтобы сделать это, если вы не хотите потерять ссылки на объект приложения `wsgi_app()`).

Кроме того, такой дизайн делает возможным использование фабричных методов для создания приложения, которое может оказаться полезным для модульного тестирования и других подобных вещей (app-factories).

### 3.1.2 Система маршрутизации

Flask использует систему маршрутизации Werkzeug, который был спроектирован для того, чтобы автоматически упорядочивать маршруты в зависимости от их сложности. Это означает, что вы можете объявлять маршруты в произвольном порядке и они всё равно будут работать ожидаемым образом. Это необходимо, если вы хотите, чтобы маршруты, основанные на декораторах, работали правильно, потому что декораторы могут активироваться в неопределённом порядке, если приложение разделено на несколько модулей.

Другое проектное решение системы маршрутизации Werkzeug заключается в том, что Werkzeug старается, чтобы URL'ы были уникальными. Werkzeug настолько последователен в этом, что автоматически выполнит переадресацию на канонический URL, если маршрут окажется неоднозначным.

### 3.1.3 Одна система шаблонизации

Flask остановился на одной системе шаблонизации: Jinja2. Почему Flask не имеет интерфейса для подключения систем шаблонизации? Очевидно, что вы можете использовать любые системы шаблонизации, но Flask всё равно настроит Jinja2. Хотя *всегда* настроенная Jinja2, возможно и не потребуется, скорее всего нет причин, чтобы не принять решение воспользоваться единой системой шаблонизации.

Системы шаблонизации похожи на языки программирования и каждая из них обладает собственным мировоззрением. Внешне они все работают одинаково: вы просите систему шаблонизации заполнить шаблон набором переменных и получаете его в виде строки.

Но на этом сходство заканчивается. Jinja2, например, обдает системой расширяемых фильтров, возможностью наследования шаблонов, поддержкой повторного использования блоков (макросов), которые можно использовать внутри шаблонов и из кода на Python для всех операций, использует Unicode, поддерживает поэтапную отрисовку шаблонов, настраиваемый синтаксис и многое другое. С другой стороны, системы шаблонизации вроде Genshi основываются на потоковом заполнении XML, наследовании шаблонов с помощью XPath и т.п. Mako интерпретирует шаблоны аналогично модулям Python.

Когда нужно соединить систему шаблонизации с приложением или фреймворком, требуется больше чем просто отрисовка шаблонов. Например, Flask широко использует поддержку экранирования из Jinja2. А ещё он позволяет получить доступ к макросам из шаблонов Jinja2.

Слой абстракции над системами шаблонизации, учитывающий уникальные особенности каждой из них - это почти наука, слишком сложная для поддержки в микрофреймворке вроде Flask.

Кроме того, расширения из-за этого могут оказаться слишком зависимыми от имеющейся системы шаблонизации. Вы можете воспользоваться собственным языком шаблонов, но расширения по-прежнему будут зависеть от Jinja.

### 3.1.4 Микро, но с зависимостями

Почему Flask называет себя микрофреймворком, если он зависит от двух библиотек (а именно - от Werkzeug и Jinja2)? Если посмотреть на Ruby со стороны веб-разработки, то и там имеется протокол очень похожий на WSGI. Только там он называется Rack, но всё же он больше похож на реализацию WSGI для Ruby. Но практически ни одно приложение для Ruby не работает с Rack напрямую, используя для этого одноимённую библиотеку. У библиотеки Rack в Python имеется два аналога: WebOb (прежде известная под именем Paste) и Werkzeug. Paste всё ещё существует, но насколько я понимаю, он уступил своё место в пользу WebOb. Разработчики WebOb и Werkzeug развивались параллельно, имея схожие цели: сделать хорошую реализацию WSGI для использования в других приложениях.

Фреймворк Flask использует преимущества уже реализованные в Werkzeug для правильного взаимодействия с WSGI (что иногда может оказаться сложной задачей). Благодаря недавним разработкам в инфраструктуре пакетов Python, пакеты с зависимостями больше не являются проблемой и осталось мало причин,чтобы воздерживаться от использования библиотек, зависящих от других библиотек.

### 3.1.5 Локальные объекты потоков

Flask использует объекты, локальные внутри потока (фактически - объекты локального контекста, по сути они поддерживают контексты гринлетов) для запроса, сеанса и для дополнительного объекта, в который можно положить собственные объекты (g). Почему сделано так и почему это не так уж плохо?

Да, обычно это не лучшая мысль, использовать локальные объекты потоков. Они могут вызвать проблемы с серверами, которые не основаны на понятии потоков и приводят к сложностям в сопровождении больших приложений. Однако, Flask просто не предназначен для больших приложений или асинхронных серверов. Flask стремится упростить и ускорить разработку традиционных веб-приложений.

Также обратитесь к разделу документации becomingbig за мыслями о том, как стоит писать большие приложения на основе Flask.

### 3.1.6 Что есть Flask и чем он не является

Во Flask никогда не будет слоя для работы с базами данных. В нём нет библиотеки форм или чего-то подобного. Flask сам по себе является мостом к Werkzeug для реализации хорошего приложения WSGI и к Jinja2 для обработки шаблонов. Он также связан с несколькими стандартными библиотеками, например, с библиотекой журналирования logging. Всё остальное отдаётся на откуп расширениям.

Почему так? Потому что люди обладают разными предпочтениями и требованиями и Flask не сможет удовлетворить их, если всё это будет частью ядра. Большинству веб-приложений нужна какая-нибудь система шаблонизации. Однако не каждому приложению необходима база данных SQL.

Идея Flask заключается в создании хорошей основы для других приложений. Всё остальное должны сделать вы сами или расширения.

# Алфавитный указатель

## Symbols



## F



## Y