Сообщить об ошибке.

Веб фреймворк Flask в Python

Чт…ро-фреймворк Flask.

Мод…ой *микро-фреймворк*, но слово "*микро*" не означает, что веб-приложение, построенное на Flask, дол… с кодом на Python, хотя, это не запрещено. Или Flask испытывает недостаток в фун…вка "*микро*" означает, что Flask придерживаться простого, но расширяемого ядра. Это означает, что…ммиста что и как ему делать. При проектировании веб-приложений на Flask программист все реш…ер, какие паттерны использовать, какую базу данных подключить, использовать ли ORM или раб… не содержит больших слоев абстракции и по этому быстр. В общем Flask - это всё, что нужно, и н…

Фре…много расширений для добавления различной функциональности, которые обеспечивают интеграцию с б…ом, различные технологии аутентификации и так далее. Flask имеет множество параметров кон…ениями по умолчанию, и мало предварительных соглашений. По соглашениям Flask, шаблоны и ста…поддиректориях внутри дерева исходных кодов Python, с названиями templates и static соответственно, а в качестве движка шаблонов рекомендовано использовать модуль Jinja2. Эти соглашения изменяемы, но изменять их не стоит, особенно тем, кто только начинает знакомится с Flask.

> *Основные моменты документации обновлены до версии Flask 2.3.2.*

Содержание.

Установка веб-фреймворка Flask в виртуальное окружение.

Веб-фреймворк Flask зависит от двух внешних библиотек - это Werkzeug и Jinja2. Библиотека Werkzeug - это инструментарий для WSGI - стандартного интерфейса Python между веб-приложениями и различными серверами и предназначен как для ра…, так и развёртывания. Модуль Jinja2 занимается обработкой и визуализацией шаблонов с HTML.

Что бы установить Flask в виртуальное окружение, необходимо выполнить следующие инструкции в терминале вашей системе.

```
# создаем виртуальное окружение, если его нет
$ python3 -m venv .venv --prompt VirtualEnv
# активируем виртуальное окружение
$ source .venv/bin/activate
# ставим веб-фреймворк Flask
(VirtualEnv):~$ python3 -m pip install -U Flask
```

Минимальное веб-приложение на Flask.

Как говорилось ранее, фреймворк Flask очень гибкий и не запрещает размещение всего веб-приложения в одном файле, и может выглядеть примерно следующим образом:

```
# mini-flask.py
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return """<html><body>
        <h1>Главная страница.</h1>
        <h3><a href="/hello/">Страница приветствия...</a></h3>
        </body></html>"""

@app.route('/hello/')
def hello():
    return """<html><body>
        <h1>Привет Flask!</h1>
        <h3><a href="/"><= назад</a></h3>
        </body></html>"""

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

Код выше сохраним в файл с именем mini-flask.py и запустим с помощью Python в виртуальном окружении, куда ранее устанавливался пакет Flask. **Внимание!** никогда не давайте имя файлу flask.py или директории с веб-приложением (если оно большое), это вызовет конфликт с установленным пакетом Flask.

Далее запускаем файл с кодом минимального приложения.

```
# активируем виртуальное окружение
# если оно не активировано
$ source .venv/bin/activate
Теперь запускаем файл `mini-flask.py`
(VirtualEnv):~$ python mini-flask.py
* Serving Flask app "mini-flask" (lazy loading)
 * Environment: production
   WARNING: This is a development server.
   Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Если используется Flask 2.0 и выше, то приложение можно запустить следующим образом:

```
# запуск в Bash Linux
$ export FLASK_APP=mini-flask
$ flask run
 * Running on http://127.0.0.1:5000/
...
# запуск в CMD Windows
> set FLASK_APP=hello
> flask run
 * Running on http://127.0.0.1:5000/
```

Теперь откройте в браузере ссылку http://127.0.0.1:5000/, что бы увидеть работу минимального приложения.

Ра Вверх что здесь происходит: (это необходимо понять)

1. Сначала импортируется из пакета flask класс Flask, который отвечает за всё будущее приложение, состоит ли оно из одного файла или множества файлов, расположенных в папке с веб-приложением (в этом случае импорт и создание экземпляра WSGI-приложения будет происходить в файле __init__.py).
2. Далее создается экземпляр будущего веб-приложением app = Flask(__name__). Первый аргумент конструктора класса Flask - это имя модуля (один файл) или пакета (директория с несколькими модулями). Этот аргумент, сообщает Flask, где искать шаблоны, статические файлы и т.д. Следовательно, если приложение поместилось в один файл, то в качестве аргумента следует использовать __name__ , т.к. в зависимости от того, запущен ли файл с кодом как самостоятельное приложение, или был импортирован как модуль, это имя будет разным ('__main__' или имя импортированного модуля).
3. Затем, используется декоратор @app.route('/') и @app.route('/hello/'), чтобы указать Flask, что для URL-адреса сайта / должна запускаться функция index(), а для /hello/ - hello(). Декорированные таким образом функции, в свою очередь, должны возвращать HTML код отображаемой страницы. Имя декорированной функции, в дальнейшем, можно использовать для автоматического построения URL-адресов для этой конкретной функции. Другими словами, не составлять URL-адрес вручную для страницы, а например написать flask.url_for('hello'), что сгенерирует строку с URL-адресом /hello/ (очень помогает, когда URL многоуровневые).
4. Наконец, для запуска локального сервера с веб-приложением, используется метод экземпляра приложения app.run(). Благодаря конструкции if __name__ == '__main__', сервер запустится только при непосредственном вызове скрипта из интерпретатора Python, а не при его импортировании в качестве модуля.

Как видно из информации, которую выводит сервер отладки при запуске веб-приложения (см. выше), для его остановки, необходимо нажать комбинацию клавиш Ctrl + C .

Публично доступный сервер веб-приложения Flask:

При запуске сервера веб-приложения можно заметить, что он доступен только с локального адреса 127.0.0.1. Такое поведение установлено по умолчанию, т.к. в режиме отладки debug, сторонний пользователь может выполнить нежелательный код Python на компьютере с запущенным приложением. Если вы доверяете пользователям (например локальной сети), то можно сделать сервер с веб-приложением публично доступным, для этого необходимо изменить вызов метода app.run() следующим образом: app.run(host='0.0.0.0'). Это укажет операционной системе, чтобы она слушала все публичные IP-адреса, подключенной к компьютеру сети.

Если используется Flask 2.0 и выше, то сделать приложение общедоступным можно просто добавив --host=0.0.0.0 в командную строку:

```
$ flask run --host=0.0.0.0
```

Режим отладки веб-приложения Flask.

Метод app.run() как нельзя лучше подходит для запуска локального сервера разработки, но при каждом изменении кода, его необходимо перезапускать, что очень напрягает. Но если включить режим отладки debug, то при каждом изменении кода сервер разработки будет перезагружать сам себя, кроме того, если что-то пойдёт не так, режим debug обеспечит полезным отладчиком.

Существует три способа включить отладку:

```
# Нужно добавить `app.run(debug=True)`
$ export FLASK_APP=yourapplication
$ flask run

# !!! этот вариант устарел с версии Flask 2.2.0, удален с версии 2.3.0)
$ export FLASK_ENV=development
$ flask run

# во Flask 2.2.0 появился параметр CLI `--debug`
# его нужно добавить в строку запуска сервера
$ export FLASK_APP=yourapplication
$ flask run --debug
```

Внимание! Включенный интерактивный отладчик позволяет выполнение произвольного кода Python. Это делает его главной угрозой безопасности, и следовательно режим debug никогда не должен использоваться на "*боевых*" серверах.

Экранирование HTML.

При возврате HTML (ответа сервера по умолчанию во Flask) любые предоставленные пользователем значения, отображаемые в выходных данных, должны быть экранированы для защиты от атак с использованием инъекций. HTML-шаблоны, созданные с помощью Ji... Вверх ...ают это автоматически.

Функцию escape(), показанную здесь, можно использовать вручную. В большинстве примеров она опущена для краткости, но всегда необходимо знать и понимать, как используются ненадежные данные.

```
from markupsafe import escape
                            ⋮
@app.route("/<name>")
def hello(name):
    return f"Hello, {escape(name)}!"
```

Если пользователю удалось отправить в качестве name HTML код <script>alert('bad')</script>, то экранирование приводит к тому, что он отображается как текст, а не запускается как сценарий в браузере пользователя.

URL маршрутизация в веб-приложении Flask.

Ранее говорилось, что для привязки функции к URL-адресу используется декоратор @app.route(). Но это еще не все, можно сделать определенные части URL динамически меняющимися, при этом задействовать несколько правил.

Правила для переменной части URL-адреса.

Для составления динамически меняющихся URL-адресов, необходимо изменяемые части адреса выделять как <variable_name>. Затем эти части передаются в функцию, в виде аргумента с именем variable_name. Также в изменяемых частях может быть использован конвертер, который задает правила следующего вида <converter:variable_name>.

Несколько интересных примеров:

```
from markupsafe import escape

@app.route('/user/<username>')
def show_user_profile(username):
    # выведет имя профиля пользователя `username`
    return f'Пользователь: {username}'

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # выведет страницу с данным `post_id` (целое число
    return f'Post {post_id}'

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # выведет subpath после /path/
    return f'Subpath {escape(subpath)}'
```

В Flask существуют следующие конвертеры:

- int: принимаются целочисленные значения;
- float: принимаются значения с плавающей точкой;
- path: принимает строки (как по умолчанию), но также принимает слеши;
- uuid: принимает строки UUID.

Уникальные URL-адреса во Flask.

Следующие два правила различаются использованием косой черты в конце URL-адреса.

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

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

В первом случае, канонический URL-адрес /projects/ имеет завершающую косую черту (/ в конце адреса), что похоже на папку в файловой системе. Если обратится к URL-адресу без косой черты в конце, то Flask перенаправляет с кодом ответа сервера 308 (постоянное перенаправление) на канонический URL-адрес с завершающей косой чертой.

Во втором случае, канонический URL-адрес /about не имеет косой черты в конце. Доступ к URL-адресу с косой чертой в конце приведет к ошибке 404 "*Not Found*".

Вверх

Такое поведение Flask помогает сохранить уникальность URL-адресов для этих ресурсов, что помогает поисковым системам избегать двойной индексации одной и той же страницы.

Автоматическое построение (генерация) URL-адресов.

Автоматическое создать URL-адрес для декорированных `@app.route()` функций, используйте функцию `flask.url_for()`. В качестве первого аргумента `flask.url_for()` принимает имя декорированной функции и любое количество ключевых аргументов, каждый из которых соответствует изменяемой части URL-адреса. Если дополнительно указываются переменные, которые отсутствует в изменяемых частях URL-адреса, то все они добавляются к URL в качестве параметров запроса.

```
>>> 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'))
...     # переменной `next` нет в маршруте `/login`
...     print(url_for('login', next='then'))
...     print(url_for('profile', username='admin'))
...
# /
# /login
# /login?next=then
# /user/admin
```

В примере использован метод `app.test_request_context()`, который заставляет Flask вести себя так, как будто он обрабатывает запрос, даже при взаимодействии с ним через интерпретатор Python.

Зачем строить URL-адреса с помощью `flask.url_for()`, ведь их можно составить при помощи переменных в шаблонах?

- Использование `flask.url_for()` часто более понятно (первый аргумент - функция, которая отвечает за страницу), чем жесткое кодирование URL-адресов.
- Можно изменить URL-адреса за один раз, вместо ручного поиска и изменения жестко закодированных URL-адресов.
- Автоматическое построение URL-адресов прозрачно обрабатывает экранирование специальных символов и данных Unicode.
- Сгенерированные пути URL-адреса всегда абсолютны, что позволяет избежать неожиданного поведения относительных путей в браузерах.
- Если приложение размещено за пределами корневого URL-адреса `/`, например в `/myapplication`, то функция `flask.url_for()` обработает все правильно.

HTTP методы (GET, POST и т.д.) URL-адреса во Flask.

Веб-приложения могут использовать разные HTTP методы при доступе к URL-адресу. Необходимо иметь базовое понимание HTTP методов при работе с Flask. По умолчанию маршрут URL-адреса отвечает только на GET-запросы. Для обработки URL-адреса другим методом или несколькими HTTP-методами необходимо использовать аргумент methods декоратора `@app.route()`.

```
from flask import request

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

Если присутствует HTTP-метод `GET`, то Flask автоматически добавляет поддержку HTTP-метода `HEAD` и обрабатывает запросы `HEAD` в соответствии с HTTP RFC. Аналогичным образом, автоматически реализуется HTTP-метод `OPTIONS`.

Вверх

Бы введение в HTTP-методы.

HTTP-метод сообщает серверу, что хочет сделать клиент с запрашиваемой страницей.

- GET: клиент сообщает серверу, что он получил информацию, хранимую на этой странице и отобразил/обработал её. Это самый распространённый HTTP-метод.
- HEAD: клиент запрашивает у сервера информацию только о заголовках страницы для установки или обработки cookies и т.д., без ее содержимого. В Flask, не требуется иметь дело с этим методом, так как нижележащая библиотека Werkzeug сделает всё сама.
- POST: клиент сообщает серверу, что он хочет передать этому URL какую-то новую информацию, при сервер должен обработать и сохранить эти данные. Обычно, методом POST передаются данные пользовательских форм на сервер.
- PUT: HTTP-метод похож на POST, только сервер может вызвать процедуру сохранения несколько раз, перезаписывая старые значения более одного раза. Другими словами, веб-приложение, ориентируясь на метод PUT удаляет старую запись и записывает новую, при этом она доступна по тому же URL.
- DELETE: удаляет информацию, расположенную в указанном месте.
- OPTIONS обеспечивает быстрый способ выяснения клиентом поддерживаемых для данного URL методов. Это HTTP-метод запускается автоматически.

Подключение статических файлов CSS и JavaScript.

Динамическим веб-приложениям также нужны статические файлы - это CSS и JavaScript файлы. Веб-сервер Flask настроен на их обслуживание в директории static. Во время разработки нужно создать папку с именем static в пакете с веб-приложением или рядом с приложением-модулем (приложение состоит из одного файла), и она будет доступна в приложении по URL /static.

Чтобы сгенерировать URL-адреса для статических файлов, необходимо использовать специальное имя конечной точки 'static':

```
# в коде Python
url_for('static', filename='style.css')

# в шаблонах Jinja2
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
```

Файл должен храниться в файловой системе как static/style.css.

Использование шаблонизатора Jinja2 в приложении Flask.

Генерировать HTML из Python - это неправильно и довольно громоздко, плюс ко всему этому встает необходимость самостоятельно выполнять экранирование HTML, чтобы обеспечить безопасность приложения. Что бы избежать всего этого фреймворк Flask автоматически настраивает шаблонизатор Jinja2.

Для визуализации шаблона используется функция flask.render_template(). Все, что нужно сделать, это указать имя подключаемого шаблона и передать переменные (как словарь или как ключевые аргументы) в подключенный шаблон. Вот простой пример того, как визуализировать шаблон:

```
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 с шаблонами templates. Итак, если приложение является модулем (веб-приложение поместилось в один файл), то папка templates находится рядом с этим модулем (файлом), если это пакет, то templates находится внутри пакета (директории с веб-приложением):

Случай 1: приложение-модуль.

```
/application.py
/static
    /style.css
/templates
    /hello.html
```

Случай 2: приложение-пакет.

```
/application
    /__init__.py
    /o.py
    /ic
```

Вверх

```
        /style.css
    /templates
        /hello.html
```

В шаблонах используйте всю мощь движка Jinja2. Для получения дополнительной информации перейдите к документации по Jinja2.

Вот пример шаблона:

```
<!doctype html>
<html lang="ru">
<head>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<title>Привет из Flask</title>
</head>
<body>
{% if name %}
  <h1>Привет {{ name }}!</h1>
{% else %}
  <h1>Привет, Flask!</h1>
{% endif %}
</body>
</html>
```

Внутри шаблонов, также есть доступ к объектам flask.request, flask.session и flask.g, а также к функции flask.get_flashed_messages().

Движок Jinja2 особенно полезен при использовании наследования шаблонов. В основном, наследование шаблонов позволяет указать определенные элементы в базовом шаблоне, а потом переопределить их в дочерних шаблонах на каждой странице (например, заголовок, блок навигации, нижний колонтитул и т.д.).

В шаблонах включено автоматическое экранирование, следовательно, если переменная содержит HTML, то содержимое будет экранировано автоматически. Если вы доверяете содержимому переменной и знаете, что это будет безопасный HTML (например, он сгенерирован кодом веб-приложения), то можно пометить его как безопасный, используя класс flask.Markup или фильтр шаблона {{ var | safe }}.

Базовое введение в то, как работает класс Markup:

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

Локальные переменные контекста запроса.

Данные, которые клиент отправляет на сервер передаются в глобальном объекте request. Этот объект фактически является прокси для объектов, которые в свою очередь являются локальными для определенного контекста, немного запутанно, но это легко понять.

Представьте, что контекст является потоком обработки. Приходит запрос, и веб-сервер решает создать новый поток или что-то еще, т.к. базовый объект может работать с системами, отличными от потоков. Когда Flask начинает обработку внутреннего запроса, он определяет, является ли текущий поток активным контекстом, и связывает текущее приложение и среду WSGI с этим контекстом (потоком). Он делает это так, что одно приложение может вызывать другое приложение без прерывания работы.

Что это значит? По сути, это можно полностью игнорировать, если только не используется модульное тестирование. Код, зависящий от объекта запроса, сломается из-за отсутствия объекта запроса. Решение состоит в том, чтобы самостоятельно создать объект запроса и привязать его к контексту. Самым простым решением для модульного тестирования является использование диспетчера контекста app.test_request_context(). В сочетании с оператором with он связывает тестовый запрос для взаимодействия с ним. Вот пример:

```
from flask import request

with app.test_request_context('/hello', method='POST'):
    # теперь можно что-то делать с запросом до конца
    # блока `with`, например:
```

Вверх

```
    assert request.path == '/hello'
    assert request.method == 'POST'
```

Другая возможность передать среду WSGI методу app.request_context():

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

Доступ к данным запроса во Flask.

Здесь не будет рассматриваться подробно объект запроса flask.Request, а рассмотрим некоторые из наиболее распространенных операций. Прежде всего, необходимо импортировать его из модуля flask.

Текущий HTTP-метод запроса к серверу будет доступен с помощью атрибута request.method. Для доступа к данным формы (передаваемым в запросах POST или PUT) используйте атрибут request.form, представляющий собой словарь, ключи которого имеют значения name HTML-тэгов <input> в форме (например <input name="username">).

```
from flask import request

@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 = 'Неверное имя пользователя или пароль'
    # код ниже выполнится, если метод запроса был GET
    # или учетные данные были недействительными
    return render_template('login.html', error=error)
```

Что произойдет, если атрибута формы не существует? В этом случае возникает особая ошибка KeyError. Ее можно поймать как стандартную ошибку KeyError, но если этого не сделать, то будет показана страница с ошибкой HTTP 400 Bad Request. Так что во многих ситуациях не придется сталкиваться с этой проблемой.

Для доступа к параметрам, представленным в URL-адресе как ?key=value, можно использовать атрибут request.args:

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

Так как request.args представляет собой словароподобный объект Python, то рекомендуется обращаться к параметрам URL-адреса с помощью метода словаря dict.get() (как показано выше) или путем перехвата KeyError и выдачи собственной страницы с ошибкой HTTP 404 Not Found. Ведь пользователи могут изменить URL-адрес, а автоматически созданная Flask станица ошибки HTTP 400 Bad Request неудобна для пользователя.

Полный список методов и атрибутов объекта запроса можно найти в документации по flask.Request.

Загрузка файлов на сервер.

Можно легко обрабатывать загруженные файлы с помощью Flask, при этом необходима установка атрибута enctype='multipart/form-data' в HTML-форме, иначе браузер вообще не будет передавать файлы.

Загруженные файлы хранятся в памяти или во временном каталоге файловой системы. Получить доступ к файлам, можно при помощи атрибута request.files объекта запроса. Каждый загруженный файл хранится в этом словаре. Он ведет себя так же, как стандартный файловый объект Python, но также имеет метод fp.save(), который позволяет сохранять файл в указанном месте файловой системе. Простой пример, как это работает:

```
from flask import request

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

Если необходимо знать, как изначально назывался файл, до того, как он был загружен, то можно получить доступ к атрибуту имени файла fp.filename. Имейте в виду, что это значение может быть подделано, следовательно доверять этому значению не[illegible]сли необходимо использовать изначальное имя файла для его хранения на сервере, то лучше получить его через функцию werkzeug.utils.secure_filename(), которую предоставляет пакет Werkzeug:

```python
from werkzeug.utils import secure_filename

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

Дополнительно смотрите раздел "Загрузка файлов во Flask".

Установка и получение файлов cookie во Flask.

Для доступа к файлам cookie можно использовать атрибут request.cookie. Чтобы установить файлы cookie, можно использовать метод response.set_cookie() объекта ответа. Атрибут cookie объектов запроса Request представляет собой словарь со всеми файлами cookie, передаваемыми клиентом. Если нужно использовать сессии/сеансы, то не используйте cookie напрямую, а вместо этого используйте встроенные сессии Sessions во Flask, которые повышают безопасность файлов cookie.

Чтение файлов cookie:

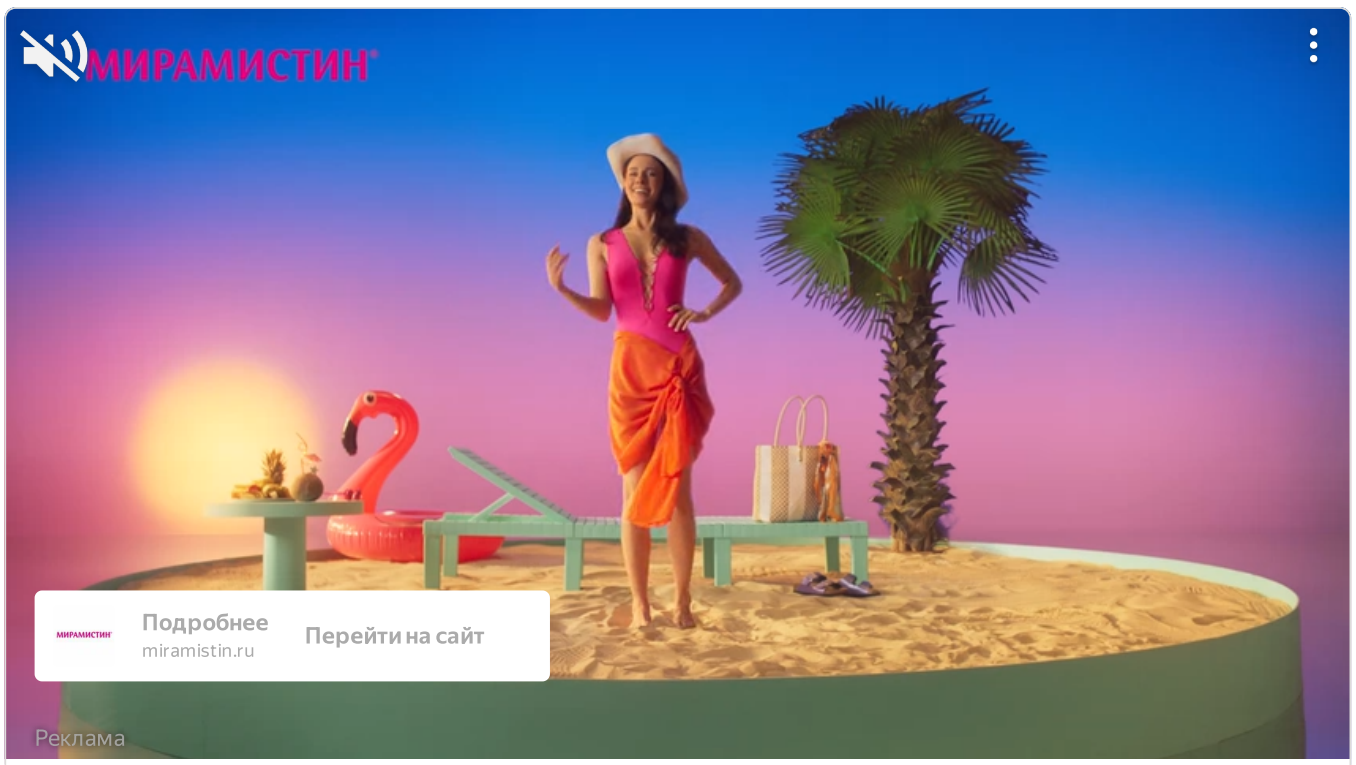
```python
from flask import request

@app.route('/')
def index():
    # Используйте `request.cookies.get(key)` вместо `cookies[key]`,
    # чтобы не получать `KeyError`, если cookie отсутствует.
    username = request.cookies.get('username')
```

Установка файлов cookie:

```python
from flask import make_response

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



Обратите внимание, что файлы cookie устанавливаются для объектов Responses. Поскольку вы обычно просто возвращаете строки из функций просмотра, Flask преобразует их для вас в объекты ответа. Если вы явно хотите сделать это, вы можете использовать функцию make_response (), а затем изменить ее.

Иногда вам может потребоваться установить cookie в точке, где объект ответа еще не существует. Это возможно с помощью шаблона обратных вызовов отложенного запроса.

Дополнительно смотрите ниже подраздел "Ответ сервера во Flask".

Перенаправление/редиректы во Flask.

Вверх

Чтобы перенаправить пользователя на другую страницу сайта, используйте функцию flask.redirect(). Чтобы преждевременно прервать запрос с установленным кодом ошибки, используйте функцию flask.abort():

```
from flask import abort, redirect, url_for

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

@app.route('/login')
def login():
    abort(401)
    # эта функция никогда не исполнится
    this_is_never_executed()
```

Бессмысленный пример, так как пользователь будет перенаправлен с главной страницы на страницу, к которой он не может получить доступ (код HTTP 401 означает отказ в доступе), но этот пример показывает, как это работает.

Вызов страницы с HTTP-ошибкой 404, 500 и т. д. во Flask.

По умолчанию для каждого кода ошибки (в примере выше строка с abort(401)) отображается стандартная страница ошибки. Если необходимо настроить собственную страницу с ошибкой, то можно использовать декоратор @app.errorhandler():

```
from flask import render_template

@app.errorhandler(404)
# кастомная страница с ошибкой
def page_not_found(error):
    return render_template('page_not_found.html'), 404
```

Обратите внимание на 404 после вызова flask.render_template(). Это сообщает Flask, что код состояния этой страницы должен быть 404 (обязательно нужно указать). По умолчанию функция будет возвращать код 200, что означает: все прошло хорошо.

Дополнительные сведения смотрите. В разделе "Обработка ошибок Flask, пользовательские страницы ошибок".

Ответ сервера во Flask.

Возвращаемое значение из функции представления, автоматически преобразуется в объект ответа сервера. Если возвращаемое значение является строкой, то оно преобразуется в объект ответа со строкой в качестве тела ответа, кодом состояния 200 OK и mimetype text/html. Если возвращаемое значение - словарь dict, то для получения ответа вызывается flask.jsonify(). Логика, которую Flask применяет для преобразования возвращаемых значений в объекты ответа, выглядит следующим образом:

- Если возвращается объект ответа правильного типа, то он возвращается непосредственно из функции представления.
- Если это строка, то создается объект ответа с этими данными и параметрами по умолчанию.
- Если это dict, то объект ответа создается с помощью flask.jsonify().
- Если возвращается кортеж, то элементы в кортеже предоставляют дополнительную информацию. Такие кортежи должны выглядеть следующим образом (response, status), (response, headers) или (response, status, headers). Значение кортежа status будет иметь приоритет над кодом статуса сервера, а заголовки headers могут быть списком или словарем дополнительных значений заголовков.

Если ничего из этого не сработает, то Flask будет считать, что возвращаемое значение является допустимым приложением WSGI, и преобразует его в объект ответа.

Если необходимо получить объект response внутри функции представления, то можно использовать функцию flask.make_response().

Представьте, что есть такая функция-представление:

```
from flask import render_template

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

Для получения объекта response, нужно обернуть возвращаемое выражение с помощью flask.make_response(), далее можно его изменить, а затем вернуть его:

```
from flask import make_response

@a rhandler(404)
de ound(error):
```

Вверх

```
    # получение объекта `response`
    resp = make_response(render_template('error.html'), 404)
    # добавление заголовков к объекту `response`
    resp.headers['X-Something'] = 'A value'
    return resp
```

Создание API с JSON во Flask.

Распространенным форматом ответа при написании API является JSON. Начать писать такой API с помощью Flask несложно. Если возвращать словарь dict из функции-представления, то он будет преобразован в ответ JSON.

```
@app.route("/me")
def me_api():
    user = get_current_user()
    return {
        "username": user.username,
        "theme": user.theme,
        "image": url_for("user_image", filename=user.image),
    }
```

В зависимости от дизайна API можно создавать ответы JSON для типов, отличных от словаря dict. В этом случае используйте функцию flask.jsonify(), которая сериализует любой поддерживаемый тип данных JSON. Или используйте расширения сообщества Flask, которые поддерживают более сложные приложения.

```
from flask import jsonify

@app.route("/users")
def users_api():
    users = get_all_users()
    return jsonify([user.to_json() for user in users])
```

Сессии/сеансы во Flask.

В дополнение к объекту flask.request существует также второй объект flask.session, называемый сеансом/сессией, который позволяет хранить информацию, которая будет передаваться от одного запроса к другому. Сессии реализованы поверх файлов cookie, при этом cookie криптографически подписываются. Это означает, что пользователь может просматривать содержимое файла cookie, но не зная секретного ключа изменить его не сможет.

Чтобы использовать сессии/сеансы, необходимо установить секретный ключ, при помощи которого будут подписываться сессионные cookie.

```
from flask import session

# установим секретный ключ для подписи. Держите это в секрете!
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
    if 'username' in session:
        return f'Вошел как {session["username"]}'
    return 'Вы не авторизованы'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>
    '''

@app.route('/logout')
def logout():
    # удаляем имя пользователя из сеанса, если оно есть
    on.pop('username', None)
    n redirect(url_for('index'))
```

Вверх

Как сгенерировать хорошие секретные ключи?

Секретный ключ должен быть как можно более случайным. В операционной системе есть способы сгенерировать случайные данные на основе криптографического генератора случайных чисел. Используйте функцию os.urandom(), чтобы быстро сгенерировать значение для Flask.secret_key или app.config['SECRET_KEY']). Например:

```
$ python -c 'import os; print(os.urandom(16))'
b'\x00\x1b\x1f:\xad\x08s>\x84\xf5\xd2Q?}a\xef'
```

*Примечание о сеансах на основе файлов cookie:

Flask принимает значения, которые вводятся в объект сессии/сеанса, и преобразует их в файл cookie. Если вдруг обнаружите, что некоторые значения не сохраняются в запросах (четкой ошибки не будет), то необходимо проверить в настройках Вашего браузера, что файлы cookie действительно включены. Также проверьте размер cookie в ответах страницы, он должен быть меньше размера, поддерживаемым веб-браузерами.

Помимо клиентских сеансов/сессий встроенных в Flask, есть несколько расширений, которые добавляют возможность обрабатывать сеансы на стороне сервера.

Ведение журнала во Flask.

Flask использует стандартный модуль logging для ведение журнала. Сообщения в приложении Flask регистрируются с помощью app.logger, имя которого совпадает с именем app.name. Этот регистратор также может использоваться для регистрации собственных сообщений.

```
@app.route('/login', methods=['POST'])
def login():
    user = get_user(request.form['username'])

    if user.check_password(request.form['password']):
        login_user(user)
        app.logger.info('%s успешно вошел в систему', user.username)
        return redirect(url_for('index'))
    else:
        app.logger.info('%s не удалось войти в систему', user.username)
        abort(401)
```

Если специально не настраивать ведение журнала, то уровень журнала Python по умолчанию обычно warning.

Еще несколько примеров вызовов журнала:

```
app.logger.debug('A value for debugging')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
```

Встроенный app.logger является стандартным регистратором ведения журнала, поэтому для получения дополнительной информации смотрите документацию модуля logging.

Дополнительно смотрите раздел "пользовательская настройка ведения журнала во Flask"

Содержание раздела:

Вверх