**#1 Введение во Flask**

[**Flask**](https://pythonru.com/tag/flask) — это микрофреймворк для Python, созданный в 2010 году разработчиком по имени Армин Ронахер. Но что значит это «микро»?

Это говорит о том, что Flask действительно маленький. У него в комплекте нет ни набора инструментов, ни библиотек, которыми славятся другим популярные фреймворки Python: Django или Pyramid. Но он создан с потенциалом для расширения. Во фреймворке есть набор базовых возможностей, а расширения отвечают за все остальное. «Чистый» Flask не умеет подключаться к базе данных, проверять данные формы, загружать файлы и так далее. Для добавления этих функций нужно использовать расширения. Это помогает использовать только те, которые на самом деле нужны.

Flask также не такой жесткий в отношении того, как разработчик должен структурировать свою программу, в отличие от, например, Django где есть строгие правила. Во Flask можно следовать собственной схеме.

**#2 Установка Flask**

**Примечание**: перед тем как двигаться дальше, нужно удостовериться, что в системе [**установлены Python**](https://pythonru.com/tag/skachat-i-ustanovit-python) и пакет virtualenv.

## Создание виртуальной среды (Virtual Environment)

Виртуальная среда — это изолированная копия Python, куда устанавливаются пакеты, не затрагивающие глобальную версию Python. Начать нужно с создания папки flask\_app. В ней будет храниться [**приложение Flask**](https://pythonru.com/tag/flask).

gvido@vm:~$ mkdir flask\_app

gvido@vm:~$

Важно не забыть сменить рабочий каталог на flask\_app с помощью команды cd.

gvido@vm:~$ cd flask\_app/

gvido@vm:~/flask\_app$

Следующий шаг — создание виртуальной среды внутри папки flask\_app с помощью команды virtualenv.

gvido@vm:~/flask\_app$ virtualenv env

Using base prefix '/usr'

New python executable in /home/gvido/flask\_app/env/bin/python3

Also creating executable in /home/gvido/flask\_app/env/bin/python

Installing setuptools, pip, wheel...done.

gvido@vm:~/flask\_app$

После выполнения вышеуказанной команды в папке flask\_app должна появиться еще одна под названием env. В ней будет храниться отдельная версия Python, включающая все исполняемые скрипты, как и в глобальной версии. Для использования среды ее нужно активировать.

В Linux и Mac OS это делается с помощью следующей команды.

gvido@-vm:~/flask\_app$ source env/bin/activate

(env) gvido@vm:~/flask\_app$

Пользователям Windows нужно использовать следующую команду.

C:\Users\gvido\flask\_app>env\Scripts\activate

(env) C:\Users\gvido\flask\_app>

Стоит обратить внимание, что название виртуальной среды теперь написано в скобках перед активной строкой ввода, например, (env). Это значит, что среда есть и активна. Теперь все установленные пакеты будут доступны только внутри этой среды.

Включение виртуальной среды временно меняет переменную окружения PATH. Так, если сейчас ввести в терминале python, будет вызван интерпретатор внутри среды, то есть, env, вместо глобального.

После окончания работы со средой, ее нужно выключить с помощью команды deactivate.

(env) gvido@vm:~/flask\_app$ deactivate

gvido@vm:~/flask\_app$

Эта же команда снова делает доступным глобальный интерпретатор Python.

## Установка Flask

Для установки Flask внутри виртуальной среды нужно ввести следующую команду.

(env) gvido@vm:~/flask\_app$ pip install flask

Проверить, прошла ли установка успешно, можно, вызвав интерпретатор Python и импортировав Flask.

(env) gvido@vm:~/flask\_app$ python

Python 3.5.2 (default, Nov 17 2016, 17:05:23)

[GCC 5.4.0 20160609] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>> import flask

>>> flask.\_\_version\_\_

'0.12.2'

>>>

Если ошибок нет, значит Flask успешно установился.

**#3 Основы Flask**

## Hello World во фреймворке Flask

Начать [**знакомство с Flask**](https://pythonru.com/uroki/1-vvedenie-vo-flask) можно с создания простого приложения, которое выводит “Hello World”. Создаем новый файл main.py и вводим следующий код.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def index():

return 'Hello World'

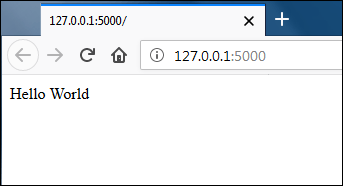
if \_\_name\_\_ == "\_\_main\_\_":

app.run()

Это приложение “Hello World”, созданное с помощью фреймворка Flask. Если код в main.py не понятен, это нормально. В следующих разделах все будет разбираться подробно. Чтобы запустить main.py, нужно ввести следующую команду в виртуальную среду Python.

(env) gvido@vm:~/flask\_app$ python main.py

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

Запуск файла main.py запускает локальный сервер для разработки на порте 5000. Осталось открыть любимый браузер и зайти на https://127.0.0.1:5000/, чтобы увидеть приложение Hello World в действии.  


Остановить сервер можно с помощью комбинации CTRL+C.

## Создание приложения Flask

У каждого Flask-приложения должен быть экземпляр класса. Экземпляр — это WSGI-приложение (WSGI – это интерфейс для взаимодействия сервера с фреймворком), которое показывает, что сервер передает все полученные запросы экземпляру для дальнейшей обработки. Объект класса Flask создается следующим образом:

from flask import Flask

app = Flask(\_\_name\_\_)

В первой строке класс Flask импортируется из пакета flask.

Во второй строке создается объект Flask. Для этого конструктору Flask назначается аргумент \_\_name\_\_. Конструктор Flask должен иметь один обязательный аргумент. Им служит название пакета. В большинстве случаев значение \_\_name\_\_ подходит. Название пакета приложения используется фреймворком Flask, чтобы находить статические файлы, шаблоны и т. д.

## Создание route (путей)

Маршрут (или путь) используется во фреймворке Flask для привязки URL к функции представления. Эта функция отвечает на запрос. Во Flask декоратор route используется, чтобы связать URL адрес с функций. Вот как маршрут создается.

@app.route('/')

def index():

return 'Hello World'

Этот код назначает функцию index() обработчиком корневого URL в приложении. Другими словами, каждый раз, когда приложение будет получать запрос, где путь — /, вызывается функция index(), и на этом запрос завершается.

Как вариант можно использовать метод add\_url\_rule() вместо декоратора route для маршрутизации. add\_url\_rule() — это простой метод, а не декоратор. Помимо URL он принимает конечную точку и название функции представления. Конечная точка относится к уникальному имени маршрута. Обычно, название функции представления — это и есть конечная точка. Flask может генерировать URL из конечной точки, но об этом позже. Предыдущий код аналогичен следующему:

def index():

return 'Hello World'

app.add\_url\_rule('/', 'index', index)

Декоратор route используется в большинстве случаев, но у add\_url\_rule() есть свои преимущества.

Функция представления должна вернуть строку. Если пытаться вернуть что-то другое, сервер ответит ошибкой 500 Internal Sever Error.

Можно создать столько столько, сколько нужно приложению. Например, в следующем списке 3 пути.

@app.route('/')

def index():

return 'Home Page'

@app.route('/career/')

def career():

return 'Career Page'

@app.route('/feedback/')

def feedback():

return 'Feedback Page'

Когда URL в маршруте заканчивается завершающим слешем (/), Flask перенаправляет запрос без слеша на URL со слешем. Так, запрос к /career будет перенаправлен на /career/.

Для одной функции представления может быть использовано несколько URL. Например:

@app.route('/contact/')

@app.route('/feedback/')

def feedback():

return 'Feedback Page'

В этом случае в ответ на запросы /contact/ или /feedback/, будет вызвана функция feedback().

Если перейти по адресу, для которого нет соответствующей функции представления, появится ошибка 404 Not Found.

Эти маршруты статичны. Большая часть современных приложений имеют динамичные URL. Динамичный URL – это адрес, который состоит из одной или нескольких изменяемых частей, влияющих на вывод страницы. Например, при создании веб-приложения со страницами профилей, у каждого пользователя будет уникальный id. Профиль первого пользователя будет на странице /user/1, второго — на /user/2 и так далее. Очень неудобный способ добиться такого результата — создавать маршруты для каждого пользователя отдельно.

Вместе этого можно отметить динамические части URL как <variable\_name> (переменные). Эти части потом будут передавать ключевые слова функции отображения. Следующий код демонстрирует путь с динамическим элементом.

@app.route('/user/<id>/')

def user\_profile(id):

return "Profile page of user #{}".format(id)

В этом примере на месте <id> будет указываться часть URI, которая идет после /user/. Например, если зайти на /user/100/, ответ будет следующим.

Profile page of user #100

Этот элемент не ограничен числовыми id. Адрес может быть /user/cowboy/, /user/foobar10/, /user/@@##/ и так далее. Но он не будет работать со следующими URI: /user/, /user/12/post/. Можно ограничить маршрут, чтобы он работал только с числовыми id после /user/. Это делается с помощью конвертера.

По умолчанию динамические части URL передаются в функцию в виде строк. Это можно изменить с помощью конвертера, который указывается перед динамическими элементами URL с помощью <converter:variable\_name>. Например, /user/<int:id>/ будет работать с адресами /user/1/, /user/200/ и другими. Но /user/cowboy/, /user/foobar10/ и /user/@@##/ не подойдут.

В этом списке все конвертеры, доступные во Flask:

| **Конвертер** | **Описание** |
| --- | --- |
| string | принимает любые строки (значение по умолчанию). |
| int | принимает целые числа. |
| float | принимает числа с плавающей точкой. |
| path | принимает полный путь включая слеши и завершающий слеш. |
| uuid | принимает строки uuid (символьные id). |

## Запуск сервера

Для запуска сервера разработки нужно использовать метод run() объекта Flask.

if \_\_name\_\_ == "\_\_main\_\_":

app.run()

Условие \_\_name\_\_ == "\_\_main\_\_" гарантирует, что метод run() будет вызван только в том случае, если main.py будет запущен, как основная программа. Если попытаться использовать метод run() при импорте main.py в другой модуль Python, он не вызовется.

**Важно**: сервер разработки Flask используется исключительно для тестирования, поэтому его производительность невысокая.

Теперь должно быть понятнее, как работает main.py.

## Режим отладки (Debug)

Баги неизбежны в программировании. Поэтому так важно знать, как находить ошибки в программе и исправлять их. Во Flask есть мощный интерактивный отладчик, который по умолчанию отключен. Когда он выключен, а в программе обнаруживается ошибка, функция показывает 500 Internal Sever Error. Чтобы увидеть это поведение наглядно, можно специально добавить баг в файл main.py.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

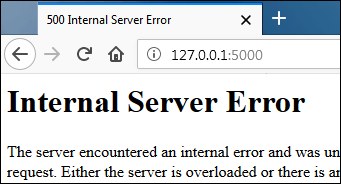
def index():

print(i)

return 'Hello World'

if \_\_name\_\_ == "\_\_main\_\_":

app.run()

В этом случае программа пытается вывести значение неопределенной переменной i, что приводит к ошибке. Если открыть https://127.0.0.1:5000/, то появится ошибка 500 Internal Sever Error:  


Тем не менее сам браузер не сообщает о типе ошибки. Если посмотреть в консоль, то можно увидеть отчет об ошибке. В данном случае он выглядит вот так:

File "/home/gvido/flask\_app/env/lib/python3.5/site-packages/flask/app.py", line 1612, in full\_dispatch\_request

rv = self.dispatch\_request()

File "/home/gvido/flask\_app/env/lib/python3.5/site-packages/flask/app.py", line 1598, in dispatch\_request

return self.view\_functions[rule.endpoint](\*\*req.view\_args)

File "main.py", line 13, in index

print(i)

NameError: name 'i' is not defined

Когда режим отладки выключен, после изменения кода нужно каждый раз вручную запускать сервер, чтобы увидеть изменения. Но режим отладки будет перезапускать его после любых изменений в коде.

Чтобы включить режим, нужно передать аргумент debug=True методу run():

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

Еще один способ — указать значение True для атрибута debug.

from flask import Flask

app = Flask(\_\_name\_\_)

app.debug = True

После обновления файл main.py следующим образом его можно запускать.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def index():

print(i)

return 'Hello World'

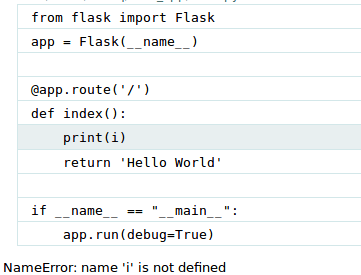
if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True) # add debug mode

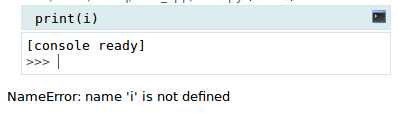
Теперь при открытии https://127.0.0.1:5000/ на странице будет отладчик.

Теперь, когда отладчик включен, вместо ошибки 500 Internal Server на странице будет отображаться отчет об ошибке. Он в полной мере описывает, какая ошибка случилась. Внизу страницы видно, что оператор print пытался вывести значение неопределенной переменной i. Там же указан тип ошибки, NameError, что подтверждает то, что ошибка заключается в том, что имя i не определено.

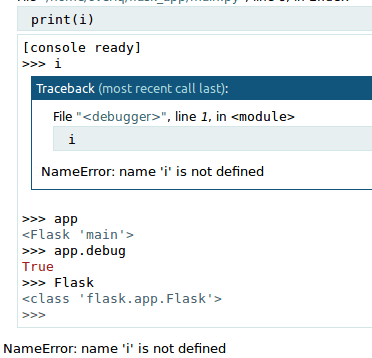
Кликнув на строчку кода на странице вывода ошибки, можно получить исходный код, где эта ошибка обнаружена, а также предыдущие и следующие строчки. Это помогает сразу понять контекст ошибки.



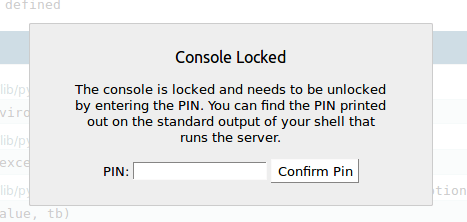
При наведении на строчку кода отображается иконка терминала. Нажав на нее, откроется консоль, где можно ввести любой код Python.



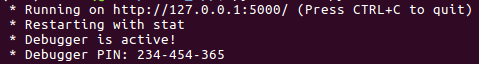
В ней можно проверить локальные переменные.



Если консоль открывается первый раз, то нужно ввести PIN-код.



Это мера безопасности, призванная ограничить доступ к консоли неавторизованным пользователям. Посмотреть код можно в консоли при запуске сервера. Он будет указан в начале вывода.



Завершить урок стоит созданием еще одного [**приложения Flask**](https://pythonru.com/tag/flask) с применением всех имеющихся знаний.

Создаем еще один файл main2.py со следующим кодом:

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def index():

return 'Hello Flask'

@app.route('/user/<int:user\_id>/')

def user\_profile(user\_id):

return "Profile page of user #{}".format(user\_id)

@app.route('/books/<genre>/')

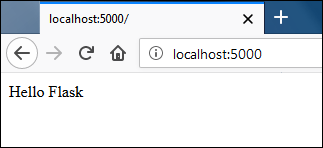
def books(genre):

return "All Books in {} category".format(genre)

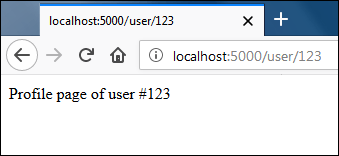
if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

Если запустить файл и зайти на https://127.0.0.1:5000/, браузер поприветствует выражением «Hello Flask»:

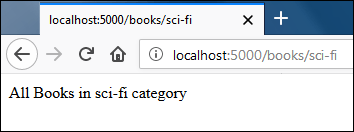


В этой новой версии приложения есть два динамических пути. Если в браузере ввести https://127.0.0.1:5000/user/123/, откроется страница со следующим содержанием:



Стоит заметить, что путь /user/<int:user\_id>/ будет работать только с теми URL, где динамическая часть (user\_id) представлена числом.

Чтобы проверить второй динамический путь, нужно открыть https://127.0.0.1:5000/books/sci-fi/. В этот раз получится следующее:



Если сейчас попробовать открыть URL, который не определен в путях, выйдет ошибка 404 Not Found. Например, такой ответ получите при попытке перейти на https://127.0.0.1:5000/products.

## Как Flask обрабатывает запрос?

Откуда Flask знает, какую функцию выводить, когда он получает запрос от клиента?

Flask сопоставляет URL и функции отображения, которые будут выводиться. Определение соответствий (маршрутизация) создается с помощью декоратора route или метода add\_url\_rule() в экземпляре Flask. Получить доступ к этим соответствиям можно с помощью атрибута url\_map у экземпляра Flask.

>>>

>>> from main2 import app

>>> app.url\_map

Map([<Rule '/' (OPTIONS, GET, HEAD) -> index>,

<Rule '/static/<filename>' (OPTIONS, GET, HEAD) -> static>,

<Rule '/books/<genre>' (OPTIONS, GET, HEAD) -> books>,

<Rule '/user/<user\_id>' (OPTIONS, GET, HEAD) -> user\_profile>])

>>>

Как видно, есть 4 правила. Flask определяет соответствия URL в следующем формате:

url pattern, (comma separated list of HTTP methods handled by the route) -> view function to execute

Путь /static/<filename> автоматически добавляется для статических файлов Flask. О работе со статическими файлами речь пойдет в отдельном уроке «Обслуживание статических файлов во Flask».

**#4 Контексты во Flask**

Flask использует контексты, чтобы временно делать определенные глобальные доступными в глобальной области видимости.

Знакомые с Django могут обратить внимание на то, что функция представления во Flask не принимает request первым аргументом. Во Flask доступ к данным осуществляется с помощью входящего запроса, используя объект request:

from flask import Flask, request

@app.route('/')

def requestdata():

return "Hello! Your IP is {} and you are using {}: ".format(request.remote\_addr,

request.user\_agent)

Код выше может создать впечатление, что request — это глобальный объект, но на самом деле это не так. Если бы request был глобальным объектом, тогда в многопоточной программе приложение не смогло бы различать два одновременных процесса, поскольку программа такого типа распределяет все переменные по потокам. Во Flask используется то, что называется “Контекстами”. Они заставляют отдельные переменные вести себя как глобальные. Обращаясь к этим переменным, пользователь получает доступ к объекту в конкретном потоке. Технически такие переменные называются локальными или внутрипоточными.

Согласно документации Flask существует два вида контекстов:

1. Контекст приложения
2. Контекст запроса

Контекст приложения используется для хранения общих переменных приложения, таких как подключение к базе данных, настройки и т. д. А контекст запроса используется для хранения переменных конкретного запроса.

Контекст приложения предлагает такие объекты как current\_app или g. current\_app ссылается на экземпляр, который обрабатывает запрос, а g используется, чтобы временно хранить данные во время обработки запроса. Когда значение установлено, к нему можно получить доступ из любой функции представления. Данные в g сбрасываются после каждого запроса.

Как и контекст приложения, контекст запроса также предоставляет объекты: request и session. request содержит информацию о текущем запросе, а session — это [**словарь (dict)**](https://pythonru.com/uroki/slovar-dict-uroki-po-python-dlja-nachinajushhih). В нем хранятся значения, которые сохраняются между запросами.

Flask активирует контексты приложения и запроса, когда запрос получен и удаляет их, когда он обработан. Когда используется контекст приложения, все его переменные становятся доступным для потока. То же самое происходит и с контекстом запроса. Когда он активируется, его переменные могут быть использованы в потоке. Внутри функций представления можно получить доступ ко всем объектам контекстов приложения и запроса, так что не стоит волноваться о том, активны ли контексты или нет. Но если попробовать получить к ним доступ вне функции представления или в консоли Python, выйдет ошибка. Следующий пример демонстрирует ее:

>>> from flask import Flask, request, current\_app

>>>

>>> request.method # получаем метод запроса

Traceback (most recent call last):

#...

RuntimeError: Working outside of request context.

This typically means that you attempted to use functionality that needed an active HTTP request. Consult the documentation on testing for information about how to avoid this problem.

>>>

request\_method возвращает HTTP-метод, используемый в запросе, но поскольку самого HTTP-запроса нет, то и контекст запроса не активируется.

Похожая ошибка возникнет, если попытаться получить доступ к объекту, который предоставляется контекстом приложения.

>>> current\_app.name # получим название приложения

Traceback (most recent call last):

#...

RuntimeError: Working outside of application context.

This typically means that you attempted to use functionality that needed to interface with the current application object in a way. To solve

this set up an application context with app.app\_context(). See the documentation for more information.

>>>

Чтобы получить доступ к объектам, предоставляемым контекстами приложения и запроса вне функции представления, нужно сперва создать соответствующий контекст.

Создать контекст приложения можно с помощью метода app\_context() для экземпляра Flask.

>>> from main2 import app

>>> from flask import Flask, request, current\_app

>>>

>>> app\_context = app.app\_context()

>>> app\_context.push()

>>>

>>> current\_app.name

'main2'

Предыдущий код можно упростить используя выражение with следующим образом:

>>> from main2 import app

>>> from flask import request, current\_app

>>>

>>>

>>> with app.app\_context():

... current\_app.name

...

'main2'

>>>

При создании контекстов лучше всего использовать выражение with.

Похожим образом можно создавать контекст запроса с помощью метода test\_request\_context() в экземпляре Flask. Важно запомнить, что когда активируется контекст запроса, контекст приложения создается, если его не было до этого. Следующий код демонстрирует процесс создания контекста запроса:

>>> from main2 import app

>>> from flask import request, current\_app

>>>

>>>

>>> with app.test\_request\_context('/products'):

... request.path # получим полный путь к запрашиваемой странице(без домена).

... request.method

... current\_app.name

...

'/products'

'GET'

'main2'

>>>

Адрес /products выбран произвольно.  
Это все, что нужно знать о контекстах во Flask.

# #5 Ответ сервера и перехват запросов во Flask

## Ответ сервера

Flask предлагает три варианта для создания ответа:

1. В виде строки или с помощью шаблонизатора
2. Объекта ответа
3. Кортежа в формате (response, status, headers) или (response, headers)

Далее о каждом поподробнее.

### Создание ответа в виде строки

@app.route('/books/<genre>')

def books(genre):

return "All Books in {} category".format(genre)

До сих пор этот способ использовался, чтобы отправлять ответ клиенту. Когда Flask видит, что из функции представления возвращается строка, он автоматически конвертирует ее в объект ответа (с помощью метода make\_response()) со строкой, содержащей тело ответа, статус-код HTTP 200 и значение text/html в заголовке content-type. В большинстве случаев это — все, что нужно. Но иногда необходимы дополнительные заголовки перед отправлением ответа клиенту. Для этого создавать ответ нужно с помощью функции make\_response().

### Создание ответа с помощью make\_response()

Синтаксис make\_response() следующий:

res\_obj = make\_response(res\_body, status\_code=200)

res\_body — обязательный аргумент, представляющий собой тело ответа, а status\_code — опциональный, по умолчанию его значение равно 200.

Следующий код показывает, как добавить дополнительные заголовки с помощью функции make\_response().

from flask import Flask, make\_response,

@app.route('/books/<genre>')

def books(genre):

res = make\_response("All Books in {} category".format(genre))

res.headers['Content-Type'] = 'text/plain'

res.headers['Server'] = 'Foobar'

return res

Следующий — демонстрирует, как вернуть ошибку 404 с помощью make\_response().

@app.route('/')

def http\_404\_handler():

return make\_response("<h2>404 Error</h2>", 400)

Настройка куки — еще одна базовая задача для любого веб-приложения. Функция make\_response() максимально ее упрощает. Следующий код устанавливает два куки в клиентском браузере.

@app.route('/set-cookie')

def set\_cookie():

res = make\_response("Cookie setter")

res.set\_cookie("favorite-color", "skyblue")

res.set\_cookie("favorite-font", "sans-serif")

return res

**Примечание**: куки обсуждаются подробно в уроке «Куки во Flask».

Куки, заданные в вышеуказанном коде, будут активны до конца сессии в браузере. Можно указать собственную дату истечения их срока, передав в качестве третьего аргумента в методе set\_cookie() количество секунд. Например:

@app.route('/set-cookie')

def set\_cookie():

res = make\_response("Cookie setter")

res.set\_cookie("favorite-color", "skyblue", 60\*60\*24\*15)

res.set\_cookie("favorite-font", "sans-serif", 60\*60\*24\*15)

return res

Здесь, у куки будут храниться 15 дней.

### Создание ответов с помощью кортежей

Последний способ создать ответ — использовать [**кортежи**](https://pythonru.com/uroki/kortezhi-tuple-uroki-po-python-dlja-nachinajushhih) в одном из следующих форматов:

(response, status, headers)

(response, headers)

(response, status)

response — строка, представляющая собой тело ответа, status — код состояния HTTP, который может быть указан в виде целого числа или строки, а headers — словарь со значениями заголовков.

@app.route('/')

def http\_500\_handler():

return ("<h2>500 Error</h2>", 500)

Функция представления вернет ошибку HTTP 500 Internal Server Error. Поскольку при создании кортежей можно не писать скобки, вышеуказанный код можно переписать следующим образом:

@app.route('/')

def http\_500\_handler():

return "<h2>500 Error</h2>", 500

Следующий код демонстрирует, как указать заголовки с помощью кортежей:

@app.route('/')

def render\_markdown():

return "## Heading", 200, {'Content-Type': 'text/markdown'}

Сможете догадаться, что делает следующая функция?

@app.route('/transfer')

def transfer():

return "", 302, {'location': 'https://localhost:5000/login'}

Функция представления перенаправляет пользователя на https://localhost:5000/login с помощью ответа 302 (временное перенаправление). Перенаправление пользователей — настолько распространенная практика, что во Flask для этого есть даже отдельная функция redirect().

from flask import Flask, redirect

@app.route('/transfer')

def transfer():

return redirect("https://localhost:5000/login")

По умолчанию, redirect() осуществляет 302 редиректы. Чтобы использовать 301, нужно указать это в функции redirect():

from flask import Flask, redirect

@app.route('/transfer')

def transfer():

return redirect("https://localhost:5000/login", code=301)

## Перехват запросов

В веб-приложениях часто нужно исполнить определенный код до или после запроса. Например, нужно вывести весь список IP-адресов пользователей, которые используют приложение или авторизовать пользователя до того как показывать ему скрытые страницы. Вместе того чтобы копировать один и тот же код внутри каждой функции представления, Flask предлагает следующие декораторы:

* before\_first\_request: этот декоратор выполняет функцию еще до обработки первого запроса
* before\_request: выполняет функцию до обработки запроса
* after\_request: выполняет функцию после обработки запроса. Такая функция не будет вызвана при возникновении исключений в обработчике запросов. Она должна принять объект ответа и вернуть тот же или новый ответ.
* teardown\_request: этот декоратор похож на after\_request. Но вызванная функция всегда будет выполняться вне зависимости от того, возвращает ли обработчик исключение или нет.

Стоит отметить, что если функция в before\_request возвращает ответ, тогда обработчик запросов не вызывается.

Следующий код демонстрирует, как использовать эти точки перехвата во Flask. Нужно создать новый файл hooks.py с таким кодом:

from flask import Flask, request, g

app = Flask(\_\_name\_\_)

@app.before\_first\_request

def before\_first\_request():

print("before\_first\_request() called")

@app.before\_request

def before\_request():

print("before\_request() called")

@app.after\_request

def after\_request(response):

print("after\_request() called")

return response

@app.route("/")

def index():

print("index() called")

return '<p>Testings Request Hooks</p>'

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

После этого необходимо запустить сервер и сделать первый запрос, перейдя на страницу https://localhost:5000/. В консоли, где был запущен сервер, должен появиться следующий вывод:

before\_first\_request() called

before\_request() called

index() called

after\_request() called

Примечание: записи о запросах к серверу опущены для краткости.

После перезагрузки страницы отобразится следующий вывод.

before\_request() called

index() called

after\_request() called

Поскольку это второй запрос, функция before\_first\_request() не будет вызываться.

### Отмена запроса с помощью abort()

Flask предлагает функцию abort() для отмены запроса с конкретным типом ошибки: 404, 500 и так далее. Например:

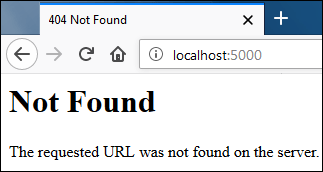
from flask import Flask, abort

@app.route('/')

def index():

abort(404)

# код после выполнения abort() не выполняется

Эта функция представления вернет стандартную страницу ошибки 404, которая выглядит вот так:  


abort() покажет похожие страницы для других типов ошибок. Если нужно изменить внешний вид страниц с ошибками, необходимо использовать декоратор errorhandler.

## Изменение страниц ошибок

Декоратор errorhandler используется для создания пользовательских страниц с ошибками. Он принимает один аргумент — ошибку HTTP, — для которой создается страница. Откроем файл hooks.py для создания кастомных страниц ошибок 404 и 500 с помощью декоратора:

from flask import Flask, request, g, abort

#...

#...

@app.after\_request

def after\_request(response):

print("after\_request() called")

return response

@app.errorhandler(404)

def http\_404\_handler(error):

return "<p>HTTP 404 Error Encountered</p>", 404

@app.errorhandler(500)

def http\_500\_handler(error):

return "<p>HTTP 500 Error Encountered</p>", 500

@app.route("/")

def index():

# print("index() called")

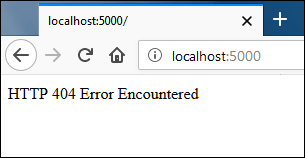
# return '<p>Testings Request Hooks</p>'

abort(404)

if \_\_name\_\_ == "\_\_main\_\_":

#...

Стоит отметить, что оба обработчика ошибок принимают один аргумент error, который содержит дополнительную информацию о типе ошибки.

Если сейчас посетить корневой URL, отобразится следующий ответ:  


# #6 Шаблоны во Flask

До этого момента HTML-строки записывались прямо в функцию представления. Это нормально в демонстрационных целях, но неприемлемо при создании реальных приложений. Большинство современных веб-страниц достаточно длинные и состоят из множества динамических элементов. Вместо того чтобы использовать огромные блоки HTML-кода прямо в функциях (с чем еще и неудобно будет работать), применяются шаблоны.

## Шаблоны

Шаблон — это всего лишь текстовый файл с HTML-кодом и дополнительными элементами разметки, которые обозначают динамический контент. Последний станет известен в момент запроса. Процесс, во время которого динамическая разметка заменяется, и генерируется статическая HTML-страница, называется отрисовкой (или рендерингом) шаблона. Во Flask есть встроенный движок шаблонов Jinja, который и занимается тем, что конвертирует шаблон в статический HTML-файл.

Jinja — один из самых мощных и популярных движков для обработки шаблонов для языка Python. Он должен быть известен пользователям Django. Но стоит понимать, что Flask и Jinja – два разных пакета, и они могут использоваться отдельно.

## Отрисовка шаблонов с помощью render\_template()

По умолчанию, Flask ищет шаблоны в подкаталоге templates внутри папки приложения. Это поведение можно изменить, передав аргумент template\_folder конструктору Flask во время создания экземпляра приложения.

Этот код меняет расположение шаблонов по умолчанию на папку jinja\_templates внутри папки приложения.

app = Flask(\_\_name\_\_, template\_folder="jinja\_templates")

Сейчас в этом нет смысла, поэтому пока стоит продолжать использовать папку templates для хранения шаблонов.

Создаем новую папку templates внутри папки приложения flask\_app. В templates — файл index.html со следующим кодом:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<p>Name: {{ name }}</p>

</body>

</html>

Стоит обратить внимание, что в «базовом» HTML-шаблоне есть динамический компонент {{ name }}. Переменная name внутри фигурных скобок представляет собой переменную, значение которой будет определено во время отрисовки шаблона. В качестве примера можно написать, что значением name будет Jerry. Тогда после рендеринга шаблона выйдет следующий код.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<p>Name: Jerry</p>

</body>

</html>

Flask предоставляет функцию rended\_template для отрисовки шаблонов. Она интегрирует Jinja во Flask. Чтобы отрисовать шаблон, нужно вызвать rended\_template() с именем шаблона и данными, которые должны быть в шаблоне в виде аргументов-ключевых слов. Аргументы-ключевые слова, которые передаются шаблонам, известны как контекст шаблона. Следующий код показывает, как отрисовать шаблон index.html с помощью render\_template().

from flask import Flask, request, render\_template

app = Flask(\_\_name\_\_)

@app.route('/')

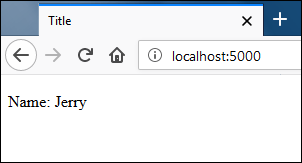
def index():

return render\_template('index.html', name='Jerry')

#...

Важно обратить внимание, что name в name='Jerry' ссылается на переменную, упомянутую в шаблоне index.html.

Если сейчас зайти на https://localhost:5000/, выйдет следующий ответ:



Если render\_template() нужно передать много аргументов, можно не разделять их запятыми (,), а создать словарь и использовать оператор \*\*, чтобы передать аргументы-ключевые слова функции. Например:

@app.route('/')

def index():

name, age, profession = "Jerry", 24, 'Programmer'

template\_context = dict(name=name, age=age, profession=profession)

return render\_template('index.html', \*\*template\_context)

Шаблон index.html теперь имеет доступ к трем переменным шаблона: name, age и profession.

Что случится, если не определить контекст шаблона?

Ничего не случится, не будет ни предупреждений, ни исключений. Jinja отрисует шаблон как обычно, а на местах пропусков использует пустые строки. Чтобы увидеть это поведение, необходимо изменить функцию представления index() следующим образом:

#...

@app.route('/')

def index():

return render\_template('index.html')

#...

Теперь при открытии https://localhost:5000/ выйдет следующий ответ:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<p>Name: </p>

</body>

</html>

Сейчас должна сложиться картина того, как используются шаблоны во Flask, а в следующем разделе речь пойдет о том, как рендерить их в консоли.

## Отрисовка шаблонов в консоли

Для тестирования рендерить шаблоны можно и в консоли. Это просто и не требует создания нескольких файлов. Для начала нужно запустить Python и импортировать класс Template из пакета jinja2 следующим образом.

>>> from jinja2 import Template

Для создания объекта Templates нужно передать содержимое шаблона в виде строки.

>>> t = Template("Name: {{ name }}")

Чтобы отрендерить шаблон, нужно вызвать метод render() объекта Template вместе с данными аргументами-ключевыми словами

>>> t.render(name='Jerry')

'Name: Jerry'

# #7 Основы шаблонизатора Jinja

Язык шаблонов (или шаблонизатор) Jinja — это маленький набор инструкций, который помогает автоматизировать создание HTML шаблонов.

## Переменные, выражения и вызовы функций

В Jinja двойные фигурные скобки {{ }} позволяют получить результат выражение, переменную или вызвать функцию и вывести значение в шаблоне. Например:

### Определение выражения

>>> from jinja2 import Template

>>>

>>> Template("{{ 10 + 3 }}").render()

'13'

>>> Template("{{ 10 - 3 }}").render()

'7'

>>>

>>> Template("{{ 10 // 3 }}").render()

'3'

>>> Template("{{ 10 / 3 }}").render()

'3.3333333333333335'

>>>

>>> Template("{{ 10 % 3 }}").render()

'1'

>>> Template("{{ 10 \*\* 3 }}").render()

'1000'

>>>

Другие операторы сравнения и присваивания и [**логические операторы**](https://pythonru.com/osnovy/operatory-python) Python также могут использоваться внутри выражений.

### Вывод переменных

>>> Template("{{ var }}").render(var=12)

'12'

>>> Template("{{ var }}").render(var="hello")

'hello'

>>>

Это могут быть не только числа и [**строки Python**](https://pythonru.com/osnovy/stroki-python). Шаблоны Jinja работают со сложными данными, такими как списки, словари, кортежи и даже пользовательские классы.

>>> Template("{{ var[1] }}").render(var=[1,2,3])

'2'

>>> Template("{{ var['profession'] }}").render(var={'name':'tom', 'age': 25, 'profession': 'Manager' })

'Manager'

>>> Template("{{ var[2] }}").render(var=("c", "c++", "python"))

'python'

>>> class Foo:

... def \_\_str\_\_(self):

... return "This is an instance of Foo class"

...

>>> Template("{{ var }}").render(var=Foo())

'This is an instance of Foo class'

>>>

Если обратится к индексу, который не существует, Jinja просто выведет пустую строку.

>>> Template("{{ var[100] }}").render(var=("c", "c++", "python"))

''

>>>

### Вызов функции

В Jinja для определения функции ее нужно просто вызвать.

>>> def foo():

... return "foo() called"

...

>>>

>>> Template("{{ foo() }}").render(foo=foo)

'foo() called'

>>>

## Атрибуты и методы

Для доступа к атрибутам и методам объекта нужно использовать оператор доступ точка (.).

>>> class Foo:

... def \_\_init\_\_(self, i):

... self.i = i

... def do\_something(self):

... return "do\_something() called"

...

>>>

>>> Template("{{ obj.i }}").render(obj=Foo(5))

'5'

>>>

>>> Template("{{ obj.do\_something() }}").render(obj=Foo(5))

'do\_something() called'

>>>

## Комментарии

В Jinja используется следующий синтаксис для добавления комментариев в одну или несколько строк:

{# комментарий #}

{#

это

многострочный

комментарий

#}

## Объявление переменных

Внутри шаблона можно задать переменную с помощью инструкции set.

{% set fruit = 'apple' %}

{% set name, age = 'Tom', 20 %}

Переменные определяются для хранения результатов сложных операций, так чтобы их можно было использовать дальше в шаблоне. Переменные, определенные вне управляющих конструкций (о них дальше), ведут себя как глобальные переменные и доступны внутри любой структуры. Тем не менее переменные, созданные внутри конструкций, ведут себя как локальные переменные и видимы только внутри этих конкретных конструкций. Единственное исключение — инструкция if.

## Цикл и условные выражения

Управляющие конструкции позволяют добавлять в шаблоны элементы управления потоком и циклы. По умолчанию, управляющие конструкции используют разделитель {% … %} вместо двойных фигурных скобок {{ ... }}.

### Инструкция if

Инструкция if в Jinja имитирует выражение if в Python, а значение условия определяет набор инструкции. Например:

{% if bookmarks %}

<p>User has some bookmarks</p>

{% endif %}

Если значение переменной bookmarks – True, тогда будет выведена строка <p>User has some bookmarks</p>. Стоит запомнить, что в Jinja, если у переменной нет значения, она возвращает False.

Также можно использовать условия elif и else, как в обычном коде Python. Например:

{% if user.newbie %}

<p>Display newbie stages</p>

{% elif user.pro %}

<p>Display pro stages</p>

{% elif user.ninja %}

<p>Display ninja stages</p>

{% else %}

<p>You have completed all stages</p>

{% endif %}

Управляющие инструкции также могут быть вложенными. Например:

{% if user %}

{% if user.newbie %}

<p>Display newbie stages</p>

{% elif user.pro %}

<p>Display pro stages</p>

{% elif user.ninja %}

<p>Display ninja stages</p>

{% else %}

<p>You have completed all states</p>

{% endif %}

{% else %}

<p>User is not defined</p>

{% endif %}

В определенных случаях достаточно удобно записывать инструкцию if в одну строку. Jinja поддерживает такой тип записи, но называет это выражением if, потому что оно записывается с помощью двойных фигурных скобок {{ … }}, а не {% … %}. Например:

{{ "User is logged in" if loggedin else "User is not logged in" }}

Здесь если переменная loggedin вернет True, тогда будет выведена строка “User is logged in”. В противном случае — “User is not logged in”.

Условие else использовать необязательно. Если его нет, тогда блок else вернет объект undefined.

{{ "User is logged in" if loggedin }}

Здесь, если переменная loggedin вернет True, будет выведена строка “User is logged in”. В противном случае — ничего.

Как и в Python можно использовать операторы сравнения, присваивания и логические операторы для управляющих конструкций, чтобы создавать более сложные условия. Вот несколько примеров:

{# Если user.count ревен 1000, код '<p>User count is 1000</p>' отобразится #}

{% if users.count == 1000 %}

<p>User count is 1000</p>

{% endif %}

{# Если выражение 10 >= 2 верно, код '<p>10 >= 2</p>' отобразится #}

{% if 10 >= 2 %}

<p>10 >= 2</p>

{% endif %}

{# Если выражение "car" <= "train" верно, код '<p>car <= train</p>' отобразится #}

{% if "car" <= "train" %}

<p>car <= train</p>

{% endif %}

{#

Если user залогинен и superuser, код

'<p>User is logged in and is a superuser</p>' отобразится

#}

{% if user.loggedin and user.is\_superuser %}

<p>User is logged in and is a superuser</p>

{% endif %}

{#

Если user является superuser, moderator или author, код

'<a href="#">Edit</a>' отобразится

#}

{% if user.is\_superuser or user.is\_moderator or user.is\_author %}

<a href="#">Edit</a>

{% endif %}

{#

Если user и current\_user один и тот же объект, код

<p>user and current\_user are same</p> отобразится

#}

{% if user is current\_user %}

<p>user and current\_user are same</p>

{% endif %}

{#

Если "Flask" есть в списке, код

'<p>Flask is in the dictionary</p>' отобразится

#}

{% if ["Flask"] in ["Django", "web2py", "Flask"] %}

<p>Flask is in the dictionary</p>

{% endif %}

Если условия становятся слишком сложными, или просто есть желание поменять приоритет оператора, можно обернуть выражения скобками ():

{% if (user.marks > 80) and (user.marks < 90) %}

<p>You grade is B</p>

{% endif %}

### Цикл for

[**Цикл for**](https://pythonru.com/osnovy/sintaksis-vozmozhnosti-i-podvodnye-kamni-cikla-for-v-python-3) позволяет перебирать последовательность. Например:

{% set user\_list = ['tom', 'jerry', 'spike'] %}

<ul>

{% for user in user\_list %}

<li>{{ user }}</li>

{% endfor %}

</ul>

Вывод:

<ul>

<li>tom</li>

<li>jerry</li>

<li>spike</li>

</ul>

Вот как можно перебирать значения словаря:

{% set employee = { 'name': 'tom', 'age': 25, 'designation': 'Manager' } %}

<ul>

{% for key in employee.items() %}

<li>{{ key }} : {{ employee[key] }}</li>

{% endfor %}

</ul>

Вывод:

<ul>

<li>designation : Manager</li>

<li>name : tom</li>

<li>age : 25</li>

</ul>

**Примечание**: в Python элементы словаря не хранятся в конкретном порядке, поэтому вывод может отличаться.

Если нужно получить ключ и значение словаря вместе, используйте метод items().

{% set employee = { 'name': 'tom', 'age': 25, 'designation': 'Manager' } %}

<ul>

{% for key, value in employee.items() %}

<li>{{ key }} : {{ value }}</li>

{% endfor %}

</ul>

Вывод:

<ul>

<li>designation : Manager</li>

<li>name : tom</li>

<li>age : 25</li>

</ul>

Цикл for также может использовать дополнительное условие else, как в Python, но зачастую способ его применения отличается. Стоит вспомнить, что в Python, если else идет следом за циклом for, условие else выполняется только в том случае, если цикл завершается после перебора всей последовательности, или если она пуста. Оно не выполняется, если цикл остановить оператором break.

Когда условие else используется в цикле for в Jinja, оно исполняется только в том случае, если последовательность пустая или не определена. Например:

{% set user\_list = [] %}

<ul>

{% for user in user\_list %}

<li>{{ user }}</li>

{% else %}

<li>user\_list is empty</li>

{% endfor %}

</ul>

Вывод:

<ul>

<li>user\_list is empty</li>

</ul>

По аналогии с вложенными инструкциями if, можно использовать [**вложенные циклы**](https://pythonru.com/osnovy/6-python-dlja-data-science-obedinenie-cikla-for-i-instrukcii-if) for. На самом деле, любые управляющие конструкции можно вкладывать одна в другую.

{% for user in user\_list %}

<p>{{ user.full\_name }}</p>

<p>

<ul class="follower-list">

{% for follower in user.followers %}

<li>{{ follower }}</li>

{% endfor %}

</ul>

</p>

{% endfor %}

Цикл for предоставляет специальную переменную loop для отслеживания прогресса цикла. Например:

<ul>

{% for user in user\_list %}

<li>{{ loop.index }} - {{ user }}</li>

{% endfor %}

</ul>

loop.index внутри цикла for начинает отсчет с 1. В таблице упомянуты остальные широко используемые атрибуты переменной loop.

| **Метод** | **Значение** |
| --- | --- |
| loop.index0 | то же самое что и loop.index, но с индексом 0, то есть, начинает считать с 0, а не с 1. |
| loop.revindex | возвращает номер итерации с конца цикла (считает с 1). |
| loop.revindex0 | возвращает номер итерации с конца цикла (считает с 0). |
| loop.first | возвращает True, если итерация первая. В противном случае — False. |
| loop.last | возвращает True, если итерация последняя. В противном случае — False. |
| loop.length | возвращает длину цикла(количество итераций). |

**Примечание**: полный список есть в документации Flask.

## Фильтры

Фильтры изменяют переменные до процесса рендеринга. Синтаксис использования фильтров следующий:

variable\_or\_value|filter\_name

Вот пример:

{{ comment|title }}

Фильтр title делает заглавной первую букву в каждом слове. Если значение переменной comment — "dust in the wind", то вывод будет "Dust In The Wind".

Можно использовать несколько фильтров, чтобы точнее настраивать вывод. Например:

{{ full\_name|striptags|title }}

Фильтр striptags удалит из переменной все HTML-теги. В приведенном выше коде сначала будет применен фильтр striptags, а затем — title.

У некоторых фильтров есть аргументы. Чтобы передать их фильтру, нужно вызвать фильтр как функцию. Например:

{{ number|round(2) }}

Фильтр round округляет число до конкретного количества символов.

В следующей таблице указаны широко используемые фильтры.

| **Название** | **Описание** |
| --- | --- |
| upper | делает все символы заглавными |
| lower | приводит все символы к нижнему регистру |
| capitalize | делает заглавной первую букву и приводит остальные к нижнему регистру |
| escape | экранирует значение |
| safe | предотвращает экранирование |
| length | возвращает количество элементов в последовательности |
| trim | удаляет пустые символы в начале и в конце |
| random | возвращает случайный элемент последовательности |

Примечание: полный список фильтров доступен [**здесь**](https://jinja.pocoo.org/docs/2.10/templates/#list-of-builtin-filters).

## Макросы

Макросы в Jinja напоминают [**функции в Python**](https://pythonru.com/uroki/funkcii-v-python-uroki-po-python-dlja-nachinajushhih). Суть в том, чтобы сделать код, который можно использовать повторно, просто присвоив ему название. Например:

{% macro render\_posts(post\_list, sep=False) %}

<div>

{% for post in post\_list %}

<h2>{{ post.title }}</h2>

<article>

{{ post.html|safe }}

</article>

{% endfor %}

{% if sep %}<hr>{% endif %}

</div>

{% endmacro %}

В этом примере создан макрос render\_posts, который принимает обязательный аргумент post\_list и необязательный аргумент sep. Использовать его нужно следующим образом:

{{ render\_posts(posts) }}

Определение макроса должно идти до первого вызова, иначе выйдет ошибка.

Вместо того чтобы использовать макросы прямо в шаблоне, лучше хранить их в отдельном файле и импортировать по надобности.

Предположим, все макросы хранятся в файле macros.html в папке templates. Чтобы импортировать их из файла, нужно использовать инструкцию import:

{% import "macros.html" as macros %}

Теперь можно ссылаться на макросы в файле macros.html с помощью переменной macros. Например:

{{ macros.render\_posts(posts) }}

Инструкция {% import “macros.html” as macros %} импортирует все макросы и переменные (определенные на высшем уровне) из файла macros.html в шаблон. Также можно импортировать определенные макросы с помощью from:

{% from "macros.html" import render\_posts %}

При использовании макросов будут ситуации, когда потребуется передать им произвольное число аргументов.

По аналогии с \*args и \*\*kwargs в Python внутри макросов можно получить доступ к varargs и kwargs.

**varags**: сохраняет дополнительные позиционные аргументы, переданные макросу, в виде кортежа.

**lwargs**: сохраняет дополнительные позиционные аргументы, переданные макросу, в виде словаря.

Хотя к ним можно получить доступ внутри макроса, объявлять их отдельно в заголовке макроса не нужно. Вот пример:

{% macro custom\_renderer(para) %}

<p>{{ para }}</p>

<p>varargs: {{ varargs }}</p>

<p>kwargs: {{ kwargs }}</p>

{% endmacro %}

{{ custom\_renderer("some content", "apple", name='spike', age=15) }}

В этом случае дополнительный позиционный аргумент, "apple", присваивается varargs, а дополнительные аргументы-ключевые слова (name=’spike’, age=15) — kwargs.

## Экранирование

Jinja по умолчанию автоматически экранирует вывод переменной в целях безопасности. Поэтому если переменная содержит, например, такой HTML-код: "<p>Escaping in Jinja</p>", он отрендерится в виде "&lt;p&gt;Escaping in Jinja&lt;/p&gt;". Благодаря этому HTML-коды будут отображаться в браузере, а не интерпретироваться. Если есть уверенность, что данные безопасны и их точно можно рендерить, стоит воспользоваться фильтром safe. Например:

{% set html = "<p>Escaping in Jinja</p>" %}

{{ html|safe }}

Вывод:

<p>Escaping in Jinja</p>

Использовать фильтр safe в большом блоке кода будет неудобно, поэтому в Jinja есть оператор autoescape, который используется, чтобы отключить экранирование для большого объема данных. Он может принимать аргументы true или false для включения и отключения экранирования, соответственно. Например:

{% autoescape true %}

Escaping enabled

{% endautoescape %}

{% autoescape false %}

Escaping disabled

{% endautoescape %}

Все между {% autoescape false %} и {% endautoescape %} отрендерится без экранирования символов. Если нужно экранировать отдельные символы при выключенном экранировании, стоит использовать фильтр escape. Например:

{% autoescape false %}

<div class="post">

{% for post in post\_list %}

<h2>{{ post.title }}</h2>

<article>

{{ post.html }}

</article>

{% endfor %}

</div>

<div>

{% for comment in comment\_list %}

<p>{{ comment|escape }}</p> # escaping is on for comments

{% endfor %}

</div>

{% endautoescape %}

## Вложенные шаблоны

Инструкция include рендерит шаблон внутри другого шаблона. Она широко используется, чтобы рендерить статический раздел, который повторяется в разных местах сайта. Вот синтаксис include:

Предположим, что навигационное меню хранится в файле nav.html, сохраненном в папке templates:

<nav>

<a href="/home">Home</a>

<a href="/blog">Blog</a>

<a href="/contact">Contact</a>

</nav>

Чтобы добавить это меню в home.html, нужно использовать следующий код:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

{# добавляем панель навигации из nav.html #}

{% include 'nav.html' %}

</body>

</html>

Вывод:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<nav>

<a href="/home">Home</a>

<a href="/blog">Blog</a>

<a href="/contact">Contact</a>

</nav>

</body>

</html>

### Наследование шаблонов

Наследование шаблонов — один из самых мощных элементов шаблонизатора Jinja. Его принцип похож на ООП (объектно-ориентированное программирование). Все начинается с создания базового шаблона, который содержит в себе скелет HTML и отдельные маркеры, которые дочерние шаблоны смогут переопределять. Маркеры создаются с помощью инструкции block. Дочерние шаблоны используют инструкцию extends для наследования или расширения основного шаблона. Вот пример:

{# Это шаблон templates/base.html #}

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>{% block title %}Default Title{% endblock %}</title>

</head>

<body>

{% block nav %}

<ul>

<li><a href="/home">Home</a></li>

<li><a href="/api">API</a></li>

</ul>

{% endblock %}

{% block content %}

{% endblock %}

</body>

</html>

Это базовый шаблон base.html. Он создает три блока с помощью block, которые впоследствии будут заполнены дочерними шаблонами. Инструкция block принимает один аргумент — название блока. Внутри шаблона это название должно быть уникальным, иначе возникнет ошибка.

Дочерний шаблон — это шаблон, который растягивает базовый шаблон. Он может добавлять, перезаписывать или оставлять элементы родительского блока. Вот как можно создать дочерний шаблон.

{# Это шаблон templates/child.html #}

{% extends 'base.html' %}

{% block content %}

{% for bookmark in bookmarks %}

<p>{{ bookmark.title }}</p>

{% endfor %}

{% endblock %}

Инструкция extends сообщает Jinja, что child.html — это дочерний элемент, наследник base.html. Когда Jinja обнаруживает инструкцию extends, он загружает базовый шаблон, то есть base.html, а затем заменяет блоки контента внутри родительского шаблона блоками с теми же именами из дочерних шаблонов. Если блок с соответствующим названием не найден, используется блок родительского шаблона.

Стоит отметить, что в дочернем шаблоне перезаписывается только блок content, так что содержимое по умолчанию из title и nav будет использоваться при рендеринге дочернего шаблона. Вывод должен выглядеть следующим образом:

<!DOCTYPE html>

<head>

<meta charset="UTF-8">

<title>Default Title</title>

</head>

<body>

<ul>

<li><a href="/home">Home</a></li>

<li><a href="/api">API</a></li>

</ul>

<p>Bookmark title 1</p>

<p>Bookmark title 2</p>

<p>Bookmark title 3</p>

<p>Bookmark title 4</p>

</body>

</html>

Если нужно, можно поменять заголовок по умолчанию, переписав блок title в child.html:

{# Это шаблон templates/child.html #}

{% extends 'base.html' %}

{% block title %}

Child Title

{% endblock %}

{% block content %}

{% for bookmark in bookmarks %}

<p>{{ bookmark.title }}</p>

{% endfor %}

{% endblock %}

После перезаписи блока на контент из родительского шаблона все еще можно ссылаться с помощью функции super(). Обычно она используется, когда в дополнение к контенту дочернего шаблона нужно добавить содержимое из родительского. Например:

{# Это шаблон templates/child.html #}

{% extends 'base.html' %}

{% block title %}

Child Title

{% endblock %}

{% block nav %}

{{ super() }} {# referring to the content in the parent templates #}

<li><a href="/contact">Contact</a></li>

<li><a href="/career">Career</a></li>

{% endblock %}

{% block content %}

{% for bookmark in bookmarks %}

<p>{{ bookmark.title }}</p>

{% endfor %}

{% endblock %}

Вывод:

<!DOCTYPE html>

<head>

<meta charset="UTF-8">

<title>Child Title</title>

</head>

<body>

<ul>

<li><a href="/home">Home</a></li>

<li><a href="/api">API</a></li>

<li><a href="/contact">Contact</a></li>

<li><a href="/career">Career</a></li>

</ul>

<p>Bookmark title 1</p>

<p>Bookmark title 2</p>

<p>Bookmark title 3</p>

<p>Bookmark title 4</p>

</body>

</html>

Это все, что нужно знать о шаблонах Jinja. В следующих уроках эти знания будут использованы для созданы крутых шаблонов.

# #8 Создание URL во Flask

[**Flask**](https://pythonru.com/tag/flask) может генерировать URL с помощью функции url\_for() из пакета flask. URL можно задавать вручную в шаблонах и функциях представления, но это не очень хорошая практика. Предположим, возникла необходимость поменять структуру ссылок для блога с /<id>/<post-title>/ на /<id>/post/<post-title>/ . Если URL были заданы вручную в шаблонах и функциях, тогда придется вручную редактировать их во всех местах. Функция url\_for() позволяет произвести то же изменение одним щелчком.

Функция url\_for() принимает конечную точку и возвращает URL в виде строки. Стоит напомнить, что конечная точка ссылается на уникальное имя URL и в большинстве случае — это имя функции представления. Например, сейчас main2.py имеет определенный корневой путь(/):

#...

@app.route('/')

def index():

return render\_template('index.html', name='Jerry')

#...

Чтобы сгенерировать корневой URL, нужно вызвать url\_for() следующим образом: url\_for(‘index’). Выводом будет '/'. Следующий код демонстрирует, как использовать url\_for() в консоли.

>>> from main2 import app

>>> from flask import url\_for

>>>

>>> with app.test\_request\_context('/api'): # путь /api выбран произвольно

... url\_for('index')

...

'/'

>>>

Стоит обратить внимание, что сперва создается контекст запроса (и таким образом — контекст приложения). Если попробовать использовать url\_for() внутри консоли без вызова контекста, выйдет ошибка. Больше о контекстах запросов и приложения можно прочитать [**здесь**](https://pythonru.com/uroki/4-konteksty-vo-flask).

Если url\_for() не может создать URL, она вызовет исключение BuildError.

>>>

>>> with app.test\_request\_context('/api'):

... url\_for('/api')

...

Traceback (most recent call last):

...

werkzeug.routing.BuildError: Could not build url for endpoint '/api

Did you mean 'static' instead?

>>>

Чтобы сгенерировать абсолютной URL, нужно передать функции url\_for() аргумент external=True:

>>>

>>> with app.test\_request\_context('/api'):

... url\_for('index', \_external=True)

...

'https://localhost:5000/'

>>>

Вместо того чтобы прописывать URL в функции redirect(), стоит всегда использовать url\_for() для этого. Например:

@app.route('/admin/')

def admin():

if not loggedin:

return redirect(url\_for('login')) # если не залогинен, выполнять редирект на страницу входа

return render\_template('admin.html')

Чтобы сгенерировать URL для динамических адресов, нужно передать динамические части в виде аргументов-ключевых слов. Например:

>>>

>>> with app.test\_request\_context('/api'):

... url\_for('user\_profile', user\_id = 100)

...

'/user/100/'

>>>

>>>

>>> with app.test\_request\_context('/api'):

... url\_for('books', genre='biography')

...

'/books/biography/'

>>>

Дополнительные аргументы-ключевые слова, переданные функции url\_for(), будут добавлены к URL в виде строки запроса.

>>>

>>> with app.test\_request\_context('/api'):

... url\_for('books', genre='biography', page=2, sort\_by='date-published')

...

'/books/biography/?page=2&sort\_by=date-published'

>>>

url\_for() — одна из тех функций, которую можно использовать внутри шаблона. Чтобы сгенерировать URL внутри шаблонов, нужно просто вызвать url\_for() внутри фигурных скобок {{ … }}:

<a href="{{ url\_for('books', genre='biography') }}">Books</a>

Вывод:

<a href="/books/biography/">Books</a>

# #9 Работа со статическими файлами во Flask

Статические файлы — это файлы, которые не изменяются часто. Это, например, файлы CSS, JavaScript, шрифты и так далее. По умолчанию Flask ищет статические файлы в папке static, которая хранится в папке приложения. Это поведение можно поменять, передав аргументу-ключевому слову static\_folder название новой папки при создании экземпляра приложения:

app = Flask(\_\_name\_\_, static\_folder="static\_dir")

Это изменит расположение статических файлов по умолчанию на папку static\_dir внутри папки приложения.

Пока что можно остановиться на папке по умолчанию, statiс. Сперва нужно создать папку static в папке flask\_app. В static создаем CSS-файл style.css со следующим содержимым.

body {

color: red

}

Стоит вспомнить, что в уроке «[**Основы Flask**](https://pythonru.com/uroki/3-osnovy-flask)» речь шла о том, что Flask автоматически добавляет путь в формате /static/<filename> для обработки статических файлов. Поэтому все, что остается — создать URL с помощью функции url\_for():

<script src="{{ url\_for('static', filename='jquery.js') }}"></script>

Вывод:

<script src="/static/jquery.js"></script>

Дальше необходимо открыть [**шаблон**](https://pythonru.com/uroki/6-shablony-vo-flask) index.html и добавить тег <link>:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

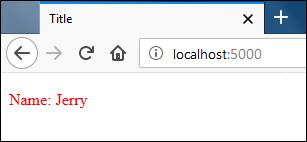
<title>Title</title>

<link rel="stylesheet" href="{{ url\_for('static', filename='style.css') }}">

</head>

...

Если сервер не запущен, его нужно запустить и открыть https://localhost:5000/. Там будет страница с красным текстом:



*Этот метод работы со статическими файлов подходит только для разработки. При создании реальных приложений используются реальные веб-сервера, такие как Nginx или Apache.*

# #10 Расширение возможностей Flask с помощью Flask-Script

## Расширения Flask

Расширения Flask — это пакеты, которые можно установить, чтобы расширить возможности Flask. Их суть в том, чтобы обеспечить удобный и понятный способ интеграции пакетов во Flask. Посмотреть все доступные расширения можно на странице [**https://flask.pocoo.org/extenstions/**](https://flask.pocoo.org/extenstions/). На странице есть пакеты, возможности которых варьируются от отправки email до создания полноценных интерфейсов администратора. Важно помнить, что расширять возможности Flask можно не только с помощью его расширений. На самом деле, подойдет любой пакет из стандартной библиотеки Python или PyPi. Оставшаяся часть урока посвящена тому, как установить и интегрировать удобное расширение для Flask под названием Flask-Script.

## Расширение Flask-Script

Flask-Script — это удобное миниатюрное расширение, которое позволяет создавать интерфейсы командной строки, запускать сервер и консоль Python в контексте приложений, делать определенные переменные видимыми в консоли автоматически и так далее.

Стоит напомнить то, что обсуждалось в уроке «[**Основы Flask**](https://pythonru.com/uroki/3-osnovy-flask)». Для запуска сервера разработки на конкретном хосте и порте, их нужно передать в качестве аргументов-ключевых слов методу run():

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True, host="127.0.0.10", port=9000)

Проблема в том, что такой подход не гибкий. Намного удобнее передать хост и порт в виде параметров командной строки при запуске сервера. Flask-Script позволяет сделать это. Установить Flask-Script можно с помощью [**pip**](https://pythonru.com/baza-znanij/ustanovka-pip-dlja-python-i-bazovye-komandy):

(env) gvido@vm:~/flask\_app$ pip install flask-script

Чтобы использовать Flask-Script сперва нужно импортировать класс Manager из пакета flask\_script и создать [**экземпляр объекта**](https://pythonru.com/primery/primery-raboty-s-klassami-v-python) Manager, передав ему экземпляр приложения. Таким образом расширения Flask интегрируются. Сначала импортируется нужный класс из пакета, а затем создается экземпляр с помощью передачи ему экземпляра приложения. Нужно открыть файл main2.py и изменить его следующим образом:

from flask import Flask, render\_template

from flask\_script import Manager

app = Flask(\_\_name\_\_)

manager = Manager(app)

#...

Созданный объект Manager также имеет метод run(), который помимо запуска сервера разработки может считывать аргументы командной строки. Следует заменить строку *app.run(debug=True)* на manager.run(). К этому моменту main2.py должен выглядеть вот так:

from flask import Flask, render\_template

from flask\_script import Manager

app = Flask(\_\_name\_\_)

manager = Manager(app)

@app.route('/')

def index():

return render\_template('index.html', name='Jerry')

@app.route('/user/<int:user\_id>/')

def user\_profile(user\_id):

return "Profile page of user #{}".format(user\_id)

@app.route('/books/<genre>/')

def books(genre):

return "All Books in {} category".format(genre)

if \_\_name\_\_ == "\_\_main\_\_":

manager.run()

Теперь у приложения есть доступ к базовым командам. Чтобы посмотреть, какие из них доступны, необходимо запустить файл main2.py:

(env) gvido@vm:~/flask\_app$ python main2.py

usage: main2.py [-?] {shell,runserver} ...

positional arguments:

{shell,runserver}

shell Runs a Python shell inside Flask application context.

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

optional arguments:

-?, --help show this help message and exit

Как показывает вывод, сейчас есть всего две команды: shell и runserver. Начнем с команды runserver.

runserver запускает веб-сервер. По умолчанию, он запускается на 127.0.0.1 на порте 5000. Чтобы увидеть варианты для любой команды нужно ввести --help и саму команду. Например:

(env) gvido@vm:~/flask\_app$ python main2.py runserver --help

usage: main2.py runserver [-?] [-h HOST] [-p PORT] [--threaded]

[--processes PROCESSES] [--passthrough-errors] [-d]

[-D] [-r] [-R] [--ssl-crt SSL\_CRT]

[--ssl-key SSL\_KEY]

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

optional arguments:

-?, --help show this help message and exit

-h HOST, --host HOST

-p PORT, --port PORT

--threaded

--processes PROCESSES

--passthrough-errors

-d, --debug enable the Werkzeug debugger (DO NOT use in production

code)

-D, --no-debug disable the Werkzeug debugger

-r, --reload monitor Python files for changes (not 100% safe for

production use)

-R, --no-reload do not monitor Python files for changes

--ssl-crt SSL\_CRT Path to ssl certificate

--ssl-key SSL\_KEY Path to ssl key

Самые широко используемые варианты для runserver — это --host и --post. С их помощью можно запустить сервер разработки на конкретном интерфейсе и порте. Например:

(env) gvido@vm:~/flask\_app$ python main2.py runserver --host=127.0.0.2 --port 8000

\* Running on http://127.0.0.2:8000/ (Press CTRL+C to quit)

По умолчанию команда runserver запускает сервер без отладчика. Включить его вручную можно следующим образом:

(env) gvido@vm:~/flask\_app$ python main2.py runserver -d -r

\* Restarting with stat

\* Debugger is active!

\* Debugger PIN: 250-045-653

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

Более простой способ запустить отладчик — выбрать значение True для атрибута debug у экземпляра объекта (app). Для этого нужно открыть main2.py и изменить файл следующим образом:

#...

app = Flask(\_\_name\_\_)

app.debug = True

manager = Manager(app)

#...

Далее о команде shell.

shell запускает консоль Python в [**контексте приложения Flask**](https://pythonru.com/uroki/4-konteksty-vo-flask). Это значит, что все объекты внутри контекстов приложения и запроса будут доступны в консоли без создания дополнительных контекстов. Для запуска консоли нужно ввести следующую команду.

(env) gvido@vm:~/flask\_app$ python main2.py shell

Получим доступ к определенным объектам.

>>>

>>> from flask import current\_app, url\_for, request

>>>

>>> current\_app.name

'main2'

>>>

>>>

>>> url\_for("user\_profile", user\_id=10)

'/user/10/'

>>>

>>> request.path

'/'

>>>

Как и ожидалось, это можно сделать, не создавая контексты приложения и запроса.

## Создание команд

Когда экземпляр Manager создан, можно приступать к созданию собственных команд. Есть два способа:

1. С помощью класса Command
2. С помощью декоратора @command

### Создание команд с помощью класса Command

В файле main2.py добавим класс Faker:

#...

from flask\_script import Manager, Command

#...

manager = Manager(app)

class Faker(Command):

'Команда для добавления поддельных данных в таблицы'

def run(self):

# логика функции

print("Fake data entered")

@app.route('/')

#...

Команда Faker была создана с помощью наследования класса Command. Метод run() вызывается при исполнении команды. Чтобы выполнить команду через командную строку, ее нужно добавить в экземпляр Manager с помощью метода add\_command():

#...

class Faker(Command):

'Команда для добавления поддельных данных в таблицы'

def run(self):

# логика функции

print("Fake data entered")

manager.add\_command("faker", Faker())

#...

Теперь нужно снова вернуться в терминал и запустить файл main2.py:

(env) gvido@vm:~/flask\_app$ python main2.py

usage: main2.py [-?] {faker,shell,runserver} ...

positional arguments:

{faker,shell,runserver}

faker Команда для добавления поддельных данных в таблицы

shell Runs a Python shell inside Flask application context.

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

optional arguments:

-?, --help show this help message and exit

Стоит обратить внимание, что теперь, в дополнение к shell и runserver, есть команда faker. Описание перед самой командой взято из строки документации класса Faker. Для запуска нужно ввести следующую команду:

(env) gvido@vm:~/flask\_app$ python main2.py faker

Fake data entered

### Создание команд с помощью декоратора @command

Создание команд с помощью класса Command достаточно объемно. Как вариант, можно использовать декоратор @command экземпляра класса Manager. Для этого нужно открыть файл main2.py и изменить его следующим образом:

#...

manager.add\_command("faker", Faker())

@manager.command

def foo():

"Это созданная команда"

print("foo command executed")

@app.route('/')

#...

Была создана простая команда foo, которая выводит foo command executed при вызове. Декоратор @command автоматически добавляет команду к существующему экземпляру Manager, так что не нужно вызывать метод add\_command(). Чтобы увидеть, как используются команды, нужно вернуться обратно в терминал и запустить main2.py.

(env) gvido@vm:~/flask\_app$ python main2.py

usage: main2.py [-?] {faker,foo,shell,runserver} ...

positional arguments:

{faker,foo,shell,runserver}

faker Команда для добавления поддельных данных в таблицы

foo Это созданная команда

shell Runs a Python shell inside Flask application context.

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

optional arguments:

-?, --help show this help message and exit

Поскольку команда foo теперь доступна, ее можно исполнить, введя следующую команду.

(env) gvido@vm:~/flask\_app$ python main2.py foo

foo command executed

## Автоматический импорт объектов

Импорт большого количества объектов в командной строке может быть утомительным. С помощью Flask-Script объекты можно сделать видимыми в терминале без явного импорта.

Команда Shell запускает оболочку. Функция конструктора оболочки Shell принимает аргумент-ключевое слово make\_context. Аргумент, передаваемый make\_context должен быть вызываемым и возвращать словарь. По умолчанию вызываемый объект возвращает словарь, содержащий только экземпляр приложения, то есть app. Это значит, что по умолчанию в оболочке можно получить доступ только к экземпляру приложения (app), специально не импортируя его. Чтобы изменить это поведение, нужно назначить новому объекту (функции), поддерживающему вызов, make\_context. Это вернет словарь с объектами, к которым требуется получить доступ внутри оболочки.

Откроем файл main2.py, чтобы добавить следующий код после функции foo().

#...

from flask\_script import Manager, Command, Shell

#...

def shell\_context():

import os, sys

return dict(app=app, os=os, sys=sys)

manager.add\_command("shell", Shell(make\_context=shell\_context))

#...

Здесь вызываемой функции shell\_context() передается аргумент-ключевое слово make\_context. Функция shell\_context возвращает словарь с тремя объектами: app, os и sys. В результате, внутри оболочки теперь есть доступ к этим объектам, хотя их импорт не производился.

(env) gvido@vm:~/flask\_app$ python main2.py shell

>>>

>>> app

<Flask 'main2'>

>>>

>>> os.name

'posix'

>>>

>>> sys.platform

'linux'

>>>

>>>

# #11 Работа с формами во Flask

Формы — важный элемент любого веб-приложения, но, к сожалению, работать с ними достаточно сложно. Сначала нужно подтвердить данные на стороне клиента, затем — на сервере. И даже этого недостаточно, если разработчик приложения озабочен такими проблемами безопасности как CSRF, XSS, SQL Injection и так далее. Все вместе — это масса работы. К счастью, есть отличная библиотека WTForms, выполняет большую часть задач за разработчика. Перед тем как узнать больше о WTForms, следует все-таки разобраться, как работать с формами без библиотек и пакетов.

## Работа с формами — сложный вариант

Для начала создадим шаблон login.html со следующим кодом:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Login</title>

</head>

<body>

{% if message %}

<p>{{ message }}</p>

{% endif %}

<form action="" method="post">

<p>

<label for="username">Username</label>

<input type="text" name="username">

</p>

<p>

<label for="password">Password</label>

<input type="password" name="password">

</p>

<p>

<input type="submit">

</p>

</form>

</body>

</html>

Этот код нужно добавить после функции представления books() в файле main2.py:

from flask import Flask, render\_template, request

#...

@app.route('/login/', methods=['post', 'get'])

def login():

message = ''

if request.method == 'POST':

username = request.form.get('username') # запрос к данным формы

password = request.form.get('password')

if username == 'root' and password == 'pass':

message = "Correct username and password"

else:

message = "Wrong username or password"

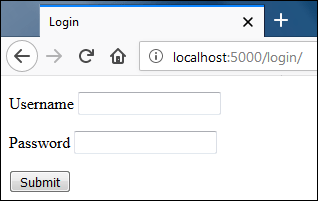
return render\_template('login.html', message=message)

#...

Стоит обратить внимание, что аргумент methods передан декоратору route(). По умолчанию обработчик запросов вызывается только в тех случаях, когда метод request.method — GET или HEAD. Это можно изменить, передав список разрешенных HTTP-методов аргументу-ключевому слову methods. С этого момента функция представления login будет вызываться только тогда, когда запрос к /login/ будет сделан с помощью методов GET, POST или HEAD. Если попробовать получить доступ к URL /login/ другим методом, появится ошибка HTTP 405 Method Not Allowed.

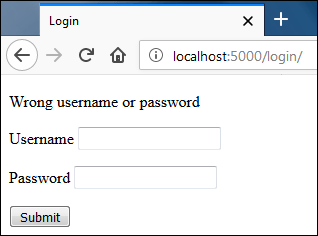
В [**прошлых уроках обсуждалось**](https://pythonru.com/uroki/5-otvet-servera-i-perehvat-zaprosov-vo-flask) то, что объект request предоставляет информацию о текущем веб-запросе. Информация, полученная с помощью формы, хранится в атрибуте form объекта request. request.form — это неизменяемый объект типа словарь, известный как ImmutableMultiDict.

Дальше нужно запустить сервер и зайти на https://localhost:5000/login/. Откроется такая форма.



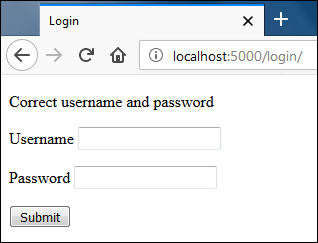
Запрос к странице был сделан с помощью метода GET, поэтому код внутри блока if функции login() пропущен.

Если попробовать отправить форму без ввода данных, страница будет выглядеть следующим образом:



В этот раз страница была отправлена методом POST, поэтому код внутри if оказался исполнен. Внутри этого блока приложение принимает имя пользователя и пароль и устанавливает сообщение для message. Поскольку форма оказалась пустой, отобразилось сообщение об ошибке.

Если заполнить форму с корректными именем пользователям и паролем и нажать Enter, появится приветственное сообщение “Correct username and password”:



Таким образом можно работать с формами во Flask. Теперь же стоит обратить внимание на пакет WTForms.

## WTForms

WTForms – это мощная библиотека, написанная на Python и независимая от фреймворков. Она умеет генерировать формы, проверять их и предварительно заполнять информацией (удобно для редактирования) и многое другое. Также она предлагает защиту от CSRF. Для установки WTForms используется Flask-WTF.

Flask- WTF – это расширение для Flask, которое интегрирует WTForms во Flask. Оно также предлагает дополнительные функции, такие как загрузка файлов, reCAPTCHA, интернационализация (i18n) и другие. Для установки Flask-WTF нужно ввести следующую команду.

(env) gvido@vm:~/flask\_app$ pip install flask-wtf

## Создание класса Form

Начать стоит с определения форм в виде классов Python. Каждая форма должна расширять класс FlaskForm из пакета flask\_wtf. FlaskForm — это обертка, содержащая полезные методы для оригинального класса wtform.Form, который является основной для создания форм. Внутри класса формы, поля формы определяются в виде переменных класса. Поля формы определяются путем создания объекта, ассоциируемого с типом поля. Пакет wtform предлагает несколько классов, представляющих собой следующие поля: StringField, PasswordField, SelectField, TextAreaField, SubmitField и другие.

Для начала нужно создать файл forms.py внутри словаря flask\_app и добавить в него следующий код.

from flask\_wtf import FlaskForm

from wtforms import StringField, SubmitField, TextAreaField

from wtforms.validators import DataRequired, Email

class ContactForm(FlaskForm):

name = StringField("Name: ", validators=[DataRequired()])

email = StringField("Email: ", validators=[Email()])

message = TextAreaField("Message", validators=[DataRequired()])

submit = SubmitField("Submit")

Здесь определен класс формы ContactForm с четырьмя полями: name, email, message и sumbit. Эти переменные будут использоваться, чтобы отрендерить поля формы, а также назначать и получать информацию из них. Эта форма создана с помощью двух StringField, TextAreaField и SumbitField. Каждый раз когда создается объект поля, определенные аргументы передаются его функции-конструктору. Первый аргумент — строка, содержащая метку, которая будет отображаться внутри тега <label> в тот момент, когда поле отрендерится. Второй опциональный аргумент — список валидаторов (элементов системы проверки), которые передаются конструктору в виде аргументов-ключевых слов. Валидаторы — это функции или классы, которые определяют, корректна ли введенная в поле информация. Для каждого поля можно использовать несколько валидаторов, разделив их запятыми (,). Модуль wtforms.validators предлагает базовые валидаторы, но их можно создавать самостоятельно. В этой форме используются два встроенных валидатора: DataRequired и Email.

**DataRequired:** он проверяет, ввел ли пользователь хоть какую-информацию в поле.

**Email:** проверяет, является ли введенный электронный адрес действующим.

Введенные данные не будут приняты до тех пор, пока валидатор не подтвердит соответствие данных.

**Примечание:** это лишь основа полей форм и валидаторов. Полный список доступен по ссылке [**https://wtforms.readthedocs.io**](https://wtforms.readthedocs.io/).

## Установка SECRET\_KEY

По умолчанию Flask-WTF предотвращает любые варианты CSFR-атак. Это делается с помощью встраивания специального токена в скрытый элемент <input> внутри формы. Затем этот токен используется для проверки подлинности запроса. До того как Flask-WTF сможет сгенерировать csrf-токен, необходимо добавить секретный ключ. Установить его в файле main2.py необходимо следующим образом:

#...

app.debug = True

app.config['SECRET\_KEY'] = 'a really really really really long secret key'

manager = Manager(app)

#...

Здесь используется атрибут config объекта Flask. Атрибут config работает как словарь и используется для размещения параметров настройки Flask и расширений Flask, но их можно добавлять и самостоятельно.

Секретный ключ должен быть строкой — такой, которую сложно разгадать и, желательно, длинной. SECRET\_KEY используется не только для создания CSFR-токенов. Он применяется и в других расширениях Flask. Секретный ключ должен быть безопасно сохранен. Вместо того чтобы хранить его в приложении, лучше разместить в переменной окружения. О том как это сделать — будет рассказано в следующих разделах.

## Формы в консоли

Откроем оболочку Python с помощью следующей команды:

(env) gvido@vm:~/flask\_app$ python main2.py shell

Это запустит оболочку Python внутри контекста приложения.

Теперь нужно импортировать класс ContactForm и создать [**экземпляр объекта**](https://pythonru.com/primery/primery-raboty-s-klassami-v-python) новой формы, передав данные формы.

>>>

>>> from forms import ContactForm

>>> from werkzeug.datastructures import MultiDict

>>>

>>>

>>> form1 = ContactForm(MultiDict([('name', 'jerry'),('email', 'jerry@mail.com')]))

>>>

Стоит обратить внимание, что данные передаются в виде объекта MultiDict, потому что функция-конструктор класса wtforms.Form принимает аргумент типа MutiDict. Если данные формы не определены при создании экземпляра объекта формы, а форма отправлена с помощью запроса POST, wtforms.Form использует данные из атрибута request.form. Стоит вспомнить, что request.form возвращает объект типа ImmutableMultiDict. Это то же самое, что и MultiDict, но он неизменяемый.

Метод validate() проверяет форму. Если проверка прошла успешно, он возвращает True, если нет — False.

>>>

>>> form1.validate()

False

>>>

Форма не прошла проверку, потому что обязательному полю message при создании объекта формы не было передано никаких данных. Получить доступ к ошибкам форм можно с помощью атрибута errors объекта формы:

>>>

>>> form1.errors

{'message': ['This field is required.'], 'csrf\_token': ['The CSRF token is missing.']}

>>>

Нужно обратить внимание, что в дополнение к сообщению об ошибке для поля message, вывод также содержит сообщение об ошибке о недостающем csfr-токене. Это из-за того что в данных формы нет запроса POST с csfr-токеном.

Отключить CSFR-защиту можно, передав csfr\_enabled=False при создании экземпляра класса формы. Пример:

>>> form3 = ContactForm(MultiDict([('name', 'spike'),('email', 'spike@mail.com')]), csrf\_enabled=False)

>>>

>>> form3.validate()

False

>>>

>>> form3.errors

{'message': ['This field is required.']}

>>>

>>>

Как и предполагалось, теперь ошибка появляется только для поля message. Теперь можно создать другой объект формы, но в этот раз передать ему информацию для всех полей.

>>>

>>> form4 = ContactForm(MultiDict([('name', 'jerry'), ('email', 'jerry@mail.com'), ('message', "hello tom")]), csrf\_enabled=False)

>>>

>>> form4.validate()

True

>>>

>>> form4.errors

{}

>>>

Проверка формы в этот раз прошла успешно.

Следующий шаг — рендеринг формы.

## Рендеринг формы

Существует два варианта рендеринга:

1. Один за одним.
2. С помощью цикла

### Рендеринг полей один за одним

Поскольку в шаблонах есть доступ к экземпляру формы, можно использовать имена полей, чтобы отрендерить имена, метки и ошибки:

{# выводим название поля #}

{{ form.field\_name.label() }}

{# выводим само поле #}

{{ form.field\_name() }}

{# выводим ошибки валидации, связанные с полем #}

{% for error in form.field\_name.errors %}

{{ error }}

{% endfor %}

Стоит протестировать этот способ в консоли:

>>>

>>> from forms import ContactForm

>>> from jinja2 import Template

>>>

>>> form = ContactForm()

>>>

Здесь экземпляр объекта формы был создан без данных запроса. Так и происходит, когда форма отображается первый раз с помощью запроса GET.

>>>

>>>

>>> Template("{{ form.name.label() }}").render(form=form)

'<label for="name">Name: </label>'

>>>

>>> Template("{{ form.name() }}").render(form=form)

'<input id="name" name="name" type="text" value="">'

>>>

>>>

>>> Template("{{ form.email.label() }}").render(form=form)

'<label for="email">Email: </label>'

>>>

>>> Template("{{ form.email() }}").render(form=form)

'<input id="email" name="email" type="text" value="">'

>>>

>>>

>>> Template("{{ form.message.label() }}").render(form=form)

'<label for="message">Message</label>'

>>>

>>> Template("{{ form.message() }}").render(form=form)

'<textarea id="message" name="message"></textarea>'

>>>

>>>

>>> Template("{{ form.submit() }}").render(form=form)

'<input id="submit" name="submit" type="submit" value="Submit">'

>>>

>>>

Поскольку форма выводится первый раз, у полей не будет ошибок проверки. Следующий код наглядно демонстрирует это:

>>>

>>>

>>> Template("{% for error in form.name.errors %}{{ error }}{% endfor %}").render(form=form)

''

>>>

>>>

>>> Template("{% for error in form.email.errors %}{{ error }}{% endfor %}").render(form=form)

''

>>>

>>>

>>> Template("{% for error in form.message.errors %}{{ error }}{% endfor %}").render(form=form)

''

>>>

>>>

Вместо отображения ошибок проверки для каждого поля можно использовать form.errors, чтобы получить доступ к ошибкам валидации, относящимся к форме. forms.errors используется чтобы отображать ошибки проверки в верхней части формы.

>>>

>>> Template("{% for error in form.errors %}{{ error }}{% endfor %}").render(form=form)

''

>>>

При рендеринге полей и меток можно добавить дополнительные аргументы-ключевые слова, которые окажутся в HTML-коде в виде пар ключей-значений. Например:

>>>

>>> Template('{{ form.name(class="input", id="simple-input") }}').render(form=form)

'<input class="input" id="simple-input" name="name" type="text" value="">'

>>>

>>>

>>> Template('{{ form.name.label(class="lbl") }}').render(form=form)

'<label class="lbl" for="name">Name: </label>'

>>>

>>>

Предположим, форма была отправлена. Теперь можно попробовать отрендерить поля и посмотреть, что получится.

>>>

>>> from werkzeug.datastructures import MultiDict

>>>

>>> form = ContactForm(MultiDict([('name', 'spike'),('email', 'spike@mail.com')]))

>>>

>>> form.validate()

False

>>>

>>>

>>> Template("{{ form.name() }}").render(form=form)

'<input id="name" name="name" type="text" value="spike">'

>>>

>>>

>>> Template("{{ form.email() }}").render(form=form)

'<input id="email" name="email" type="text" value="spike@mail.com">'

>>>

>>>

>>> Template("{{ form.message() }}").render(form=form)

'<textarea id="message" name="message"></textarea>'

>>>

>>>

Стоит обратить внимание, что у атрибута value в полях name и email есть данные. Но элемент <textarea> для поля message пуст, потому что ему данные переданы не были. Получить доступ к ошибке валидации для поля message можно следующим образом:

>>>

>>> Template("{% for error in form.message.errors %}{{ error }}{% endfor %}").render(form=form)

'This field is required.'

>>>

Как вариант, form.errors можно использовать, чтобы перебрать все ошибки валидации за раз.

>>>

>>> s ="""\

... {% for field\_name in form.errors %}\

... {% for error in form.errors[field\_name] %}\

... <li>{{ field\_name }}: {{ error }}</li>

... {% endfor %}\

... {% endfor %}\

... """

>>>

>>> Template(s).render(form=form)

'<li>csrf\_token: The CSRF token is missing.</li>\n

<li>message: This field is required.</li>\n'

>>>

>>>

Стоит обратить внимание, что ошибки csfr-токена нет, потому что запрос был отправлен без токена. Отрендерить поле csfr можно как и любое другое поле:

>>>

>>> Template("{{ form.csrf\_token() }}").render(form=form)

'<input id="csrf\_token" name="csrf\_token" type="hidden" value="IjZjOTBkOWM4ZmQ0MGMzZTY3NDc3ZTNiZDIxZTFjNDAzMGU1YzEwOTYi.DQlFlA.GQ-PrxsCJkQfoJ5k6i5YfZMzC7k">'

>>>

Рендеринг полей один из одним может занять много времени, особенно если их несколько. Для таких случаев используется цикл.

### Рендеринг полей с помощью цикла

Следующий код демонстрирует, как можно отрендерить поля с помощью [**цикла for**](https://pythonru.com/osnovy/sintaksis-vozmozhnosti-i-podvodnye-kamni-cikla-for-v-python-3).

>>>

>>> s = """\

... <div>

... {{ form.csrf\_token }}

... </div>

... {% for field in form if field.name != 'csrf\_token' %}

... <div>

... {{ field.label() }}

... {{ field() }}

... {% for error in field.errors %}

... <div class="error">{{ error }}</div>

... {% endfor %}

... </div>

... {% endfor %}

... """

>>>

>>>

>>> print(Template(s).render(form=form))

<div>

<input id="csrf\_token" name="csrf\_token" type="hidden" value="IjZjOTBkOWM4ZmQ0MGMzZTY3NDc3ZTNiZDIxZTFjNDAzMGU1YzEwOTYi.DQlFlA.GQ-PrxsCJkQfoJ5k6i5YfZMzC7k">

</div>

<div>

<label for="name">Name: </label>

<input id="name" name="name" type="text" value="spike">

</div>

<div>

<label for="email">Email: </label>

<input id="email" name="email" type="text" value="spike@mail.com">

</div>

<div>

<label for="message">Message</label>

<textarea id="message" name="message"></textarea>

<div class="error">This field is required.</div>

</div>

<div>

<label for="submit">Submit</label>

<input id="submit" name="submit" type="submit" value="Submit">

</div>

>>>

>>>

Важно заметить, что вне зависимости от используемого метода нужно вручную добавлять тег <form>, чтобы обернуть поля формы.

Теперь, зная как создавать, поверять и рендерить формы, можно использовать полученные знания для создания реальных форм.

Вначале нужно создать шаблон contact.html со следующим кодом:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<form action="" method="post">

{{ form.csrf\_token() }}

{% for field in form if field.name != "csrf\_token" %}

<p>{{ field.label() }}</p>

<p>{{ field }}

{% for error in field.errors %}

{{ error }}

{% endfor %}

</p>

{% endfor %}

</form>

</body>

</html>

Единственный недостающий кусочек пазла — функция представления, которая будет создана далее.

## Работа с подтверждением формы

Откроем main2.py, чтобы добавить следующий код после функции представления login().

from flask import Flask, render\_template, request, redirect, url\_for

from flask\_script import Manager, Command, Shell

from forms import ContactForm

#...

@app.route('/contact/', methods=['get', 'post'])

def contact():

form = ContactForm()

if form.validate\_on\_submit():

name = form.name.data

email = form.email.data

message = form.message.data

print(name)

print(email)

print(message)

# здесь логика базы данных

print("\nData received. Now redirecting ...")

return redirect(url\_for('contact'))

return render\_template('contact.html', form=form)

#...

В 7 строке создается объект формы. На 8 строке проверяется значение, которое вернул метод validate\_on\_submit() для исполнения кода внутри инструкции if.

Почему используется validate\_on\_sumbit(), а не validate(), как это было в консоли?

validate() всего лишь проверяет, корректны ли данные формы. Он не проверяет, был ли запрос отправлен с помощью метода POST. Это значит, что если использовать метод validate(), тогда запрос GET к /contact/ запустит форму проверки, а пользователь увидит ошибки валидации. Вообще процедура проверки запускается только в том случае, если данные были отправлены с помощью метода POST. В противном случае вернется False. Метод validate\_on\_submit() вызывает метод validate() внутри себя. Также нужно обратить внимание, что при создании экземпляра объекта формы данные не передаются, потому что когда форма отправляется с помощью запроса POST, WTForm считывает данные формы из атрибута request.form.

Поля формы, определенные в классе формы становятся атрибутами объекта формы. Чтобы получить доступ к данным поля используется атрибут data поля формы:

form.name.data # доступ к данным в поле name.

form.email.data # доступ к данным в поле email.

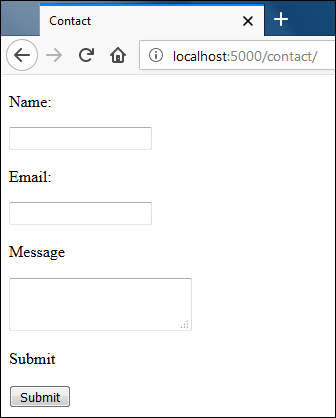
Чтобы получить доступ ко всем данные формы сразу нужно использовать атрибут data к объекту формы:

form.data # доступ ко всем данным

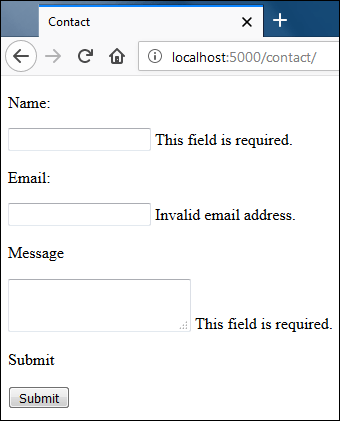
Если использовать запрос GET при посещении /contact/, метод validate\_on\_sumbit() вернет False. Код внутри if будет пропущен, а пользователь получит пустую HTML-форму.

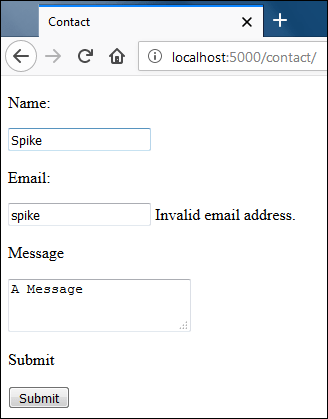
Когда форма отправляется с помощью запроса POST, validate\_on\_sumbit() возвращает True, предполагая, что данные верны. Вызовы print() внутри блока if выведут данные, введенные пользователем, а функция redirect() перенаправит пользователя на страницу /contact/. С другой стороны, если validate\_on\_sumbit() вернет False, исполнение инструкций внутри тела if будет пропущено, и появится сообщение об ошибке валидации.

Если сервер не запущен, его нужно запустить и открыть https://localhost:5000/contact/. Появится следующая контактная форма:



Если попробовать нажать Submit, не вводя данных, появятся следующие сообщения об ошибках валидации:



Теперь можно ввести определенные данные в поля Name и Message и некорректные данные в поле Email, и попробовать отправить форму снова.  


Нужно обратить внимание, что все поля содержат данные из прошлого запроса.

Теперь можно ввести корректный email в поле Email и нажать Submit. Теперь проверка пройдет успешно, а в оболочке появится следующий вывод:

Spike

spike@gmail.com

A Message

Data received. Now redirecting ...

После отображения принятых данных в оболочке функция представления перенаправит пользователя по адресу /contact/. В этот момент должна отображаться пустая форма без ошибок валидации так, будто пользователь впервые открыл /contact/ с помощью запроса GET.

Рекомендуется отображать обратную связь пользователю после успешной отправки. Во Flask это делается с помощью всплывающих сообщений.

## Всплывающие сообщения

Всплывающие сообщения — еще одна из тех функций, которые зависят от секретного ключа. Он необходим, потому что сообщения хранятся в сессиях. Сессиям во Flask будет посвящен отдельный урок. Поскольку в этом уроке секретный ключ уже был настроен, можно двигаться дальше.

Для отображения сообщения используется функция flash() из пакета flask. Функция flash() принимает два аргумента: сообщение и категория (опционально). Категория указывает на тип сообщения: \_success\_, \_error\_, \_warning\_ и так далее. Категория может быть использована в шаблоне, чтобы определить тип сообщения.

Снова откроем main2.py, чтобы добавить flash(“Message Received”, “success”) прямо перед вызовом redirect() в функции представления contact():

from flask import Flask, render\_template, request, redirect, url\_for, flash

#...

# здесь логика базы данных

print("\nData received. Now redirecting ...")

flash("Message Received", "success")

return redirect(url\_for('contact'))

return render\_template('contact.html', form=form)

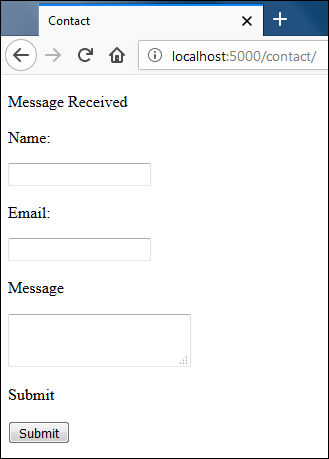
Сообщение, заданное с помощью функции flash(), будет доступно только последующему запросу, а потом удалится.

Это только настройка сообщения. Для его отображения нужно поменять также шаблон.

Для этого нужно открыть файл contact.html и изменить его следующим образом:

[**Jinja**](https://pythonru.com/uroki/7-osnovy-shablonizatora-jinja) предлагает функцию get\_flashed\_messages(), которая возвращает список активных сообщений без категории. Чтобы получить их вместе с категорией нужно передать with\_category=True при вызове get\_flashed\_messages(). Когда значение with\_categories – True, get\_flashed\_messages() вернет список кортежей формы (category, message).

После этих изменений следует открыть https://localhost:5000/contact снова. Заполнить форму и нажать Submit. Сообщение об успешной отправке отобразится в верхней части формы.



# #12 Куки во Flask

До этого момента все созданные в уроках страницы были очень простыми. Браузер отправляет запрос на сервер, сервер отвечает HTML-страницей, и это все. HTTP — это протокол, который не сохраняет свое состояние. Это значит, что в HTTP нет встроенных способов сообщить серверу, что оба запроса поступили от одного и того же пользователя. В результате сервер не знает, пытается ли пользователь получить доступ к странице впервые или в тысячный раз. Он обслуживает каждого так, будто бы это первое обращение к странице.

Если попробовать зайти в любой интернет-магазин и поискать определенные товары, то при следующем посещении сайт будет предлагать рекомендации, основанные на примерах прошлых поисков. Как же получается, что сайт узнает конкретного пользователя?

Ответ — куки и сессии.

Этот урок посвящен куки, а о сессиях речь пойдет в следующем.

## Что такое куки?

Куки — это всего лишь фрагмент данных, которые сервер устанавливает в браузере. Вот как это работает:

1. Браузер отправляет запрос на получение веб-страницы от сервера.
2. [**Сервер отвечает на запрос**](https://pythonru.com/uroki/5-otvet-servera-i-perehvat-zaprosov-vo-flask), отправляя запрошенную страницу вместе с одним или несколькими куки.
3. При получении ответа браузер [**рендерит страницу**](https://pythonru.com/uroki/6-shablony-vo-flask) и сохраняет куки на компьютере пользователя.
4. Последующий запрос на сервер будет включать информацию из куки в заголовке Cookie. Так будет продолжаться, пока не истечет сроки куки. Как только это происходит, куки удаляется из браузера.

## Настройка куки

Во Flask для настройки куки используется метод объекта ответа set\_cookie(). Синтаксис set\_cookie() следующий:

set\_cookie(key, value="", max\_age=None)

key — обязательный аргумент, это название куки. value — данные, которые нужно сохранить в куки. По умолчанию это пустая строка. max\_age — это срок действия куки в секундах. Если не указать срок, срок истечет при закрытии браузера пользователем.

Откроем main2.py, чтобы добавить следующий код после функции представления contact():

from flask import Flask, render\_template, request, redirect, url\_for, flash, make\_response

#...

@app.route('/cookie/')

def cookie():

res = make\_response("Setting a cookie")

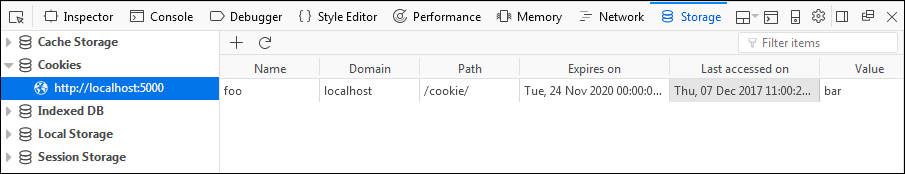
res.set\_cookie('foo', 'bar', max\_age=60\*60\*24\*365\*2)

return res

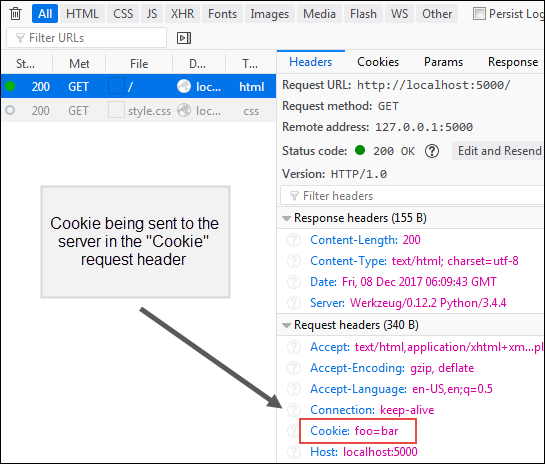
#...

Это пример создание куки под названием foo со значением bar, срок которых — 2 года.

Нужно запустить сервер и зайти на https://localhost:5000/cookie/. В качестве ответа откроется страница “Setting a cookie”. Чтобы посмотреть куки, настроенные сервером, нужно открыть инспектор хранилища в Firefox, нажав Shift+F9. Новое окно откроется в нижней части браузера. С левой стороны необходимо выбрать тип хранилища “Cookies” и нажать https://localhost:5000/, чтобы посмотреть все куки, настроенные сервером для https://localhost:5000/.



С этого момента куки foo будут отправляться вместе с запросом на сервер https://localhost:5000/. Убедиться в этом можно с помощью сетевого монитора в Firefox. Он открывается сочетанием Ctrl+Shift+E. В мониторе нужно открыть https://localhost:5000/. В списке запросов с левой стороны — выбрать первый запрос, чтобы в правой панели отобразились его подробности:



Стоит отметить, что когда куки настроены, последующие запросы к https://localhost:5000/cookie/ будут обновлять срок куки.

## Доступ к куки

Для доступа к куки используется атрибут cookie объекта request. cookie — это атрибут типа словарь, содержащий все куки, отправленные браузером. Снова откроем main2.py, чтобы изменить функцию представления cookie():

#...

@app.route('/cookie/')

def cookie():

if not request.cookies.get('foo'):

res = make\_response("Setting a cookie")

res.set\_cookie('foo', 'bar', max\_age=60\*60\*24\*365\*2)

else:

res = make\_response("Value of cookie foo is {}".format(request.cookies.get('foo')))

return res

#...

Функция представления изменена таким образом, чтобы страница показывала значение куки, если они есть. Если нет — они будут настроены автоматически.

Если открыть https://localhost:5000/cookie/ сейчас, отобразится страница со следующим содержимым.



Объект request также доступен внутри шаблона. Это значит, что доступ к куки можно получить и с помощью кода Python. Подробнее об в этом в одном из следующих разделов.

## Удаление куки

Чтобы удалить куки, нужно вызвать метод set\_cookie() с названием куки, любым значением и указать срок max\_age=0. В файле main2.py это можно сделать, добавив следующий код после функции представления cookie().

#...

@app.route('/delete-cookie/')

def delete\_cookie():

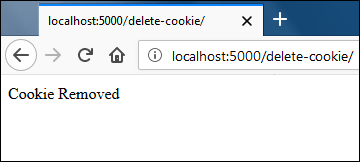
res = make\_response("Cookie Removed")

res.set\_cookie('foo', 'bar', max\_age=0)

return res

#...

Если сейчас зайти на https://localhost:5000/delete-cookie/, отобразится следующий ответ:



Теперь, понимая как работают куки, можно изучить следующие примеры кода и получить реальные практические примеры того, как настраивать куки для сохранения пользовательских предпочтений.

Добавим следующий код после функции представления delete\_cookie() в файле main2.py.

#...

@app.route('/article/', methods=['POST', 'GET'])

def article():

if request.method == 'POST':

print(request.form)

res = make\_response("")

res.set\_cookie("font", request.form.get('font'), 60\*60\*24\*15)

res.headers['location'] = url\_for('article')

return res, 302

return render\_template('article.html')

#...

Далее нужно создать новый шаблон article.html со следующим кодом:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Article</title>

</head>

<body style="{% if request.cookies.get('font') %}font-family:{{ request.cookies.get('font') }}{% endif %}">

Select Font Preference: <br>

<form action="" method="post">

<select name="font" onchange="submit()">

<option value="">----</option>

<option value="consolas" {% if request.cookies.get('font') == 'consolas' %}selected{% endif %}>consolas</option>

<option value="arial" {% if request.cookies.get('font') == 'arial' %}selected{% endif %}>arial</option>

<option value="verdana" {% if request.cookies.get('font')== 'verdana' %}selected{% endif %}>verdana</option>

</select>

</form>

<h1>Festus, superbus toruss diligenter tractare de brevis, dexter olla.</h1>

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Aperiam blanditiis debitis doloribus eos magni minus odit, provident tempora. Expedita fugiat harum in incidunt minus nam nesciunt voluptate. Facilis nesciunt, similique!

</p>

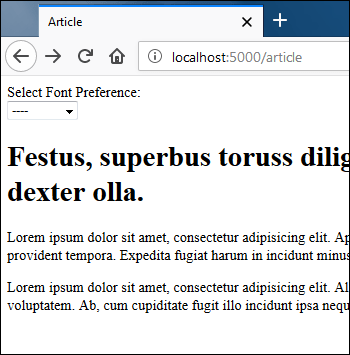
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Alias amet animi aperiam inventore molestiae quos, reiciendis voluptatem. Ab, cum cupiditate fugit illo incidunt ipsa neque quam, qui quidem vel voluptatum.</p>

</body>

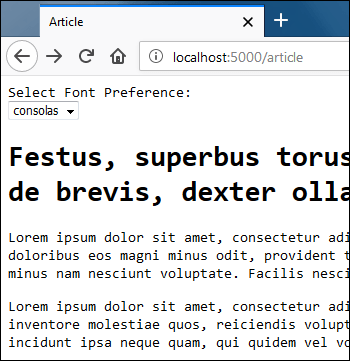
</html>

При первом посещении https://localhost:5000/article страница отобразится со шрифтом по умолчанию. Если пользователь поменяет шрифт с помощью выпадающего меню, будет отправлена форма. Значение условия if request.method == 'POST' станет истинным, и будут настроены куки font со значением выбранного шрифта, срок которых истечет через 15 дней, а пользователь будет перенаправлен на страницу https://localhost:5000/article, которая отобразится с новым выбранным шрифтом.

При посещении https://localhost:5000/article станица отобразится со шрифтом по умолчанию.



Но если выбрать новый шрифт из выпадающего меню, шрифт страницы поменяется на выбранный ранее.



## Недостатки куки

Перед использованием куки в реальном проекте нужно знать об их недостатках.

1. Куки небезопасны. Данные, которые в них хранятся, видны всем, поэтому в куки нельзя хранить пароли, данные банковских карт и так далее.
2. Куки можно отключить. Большинство браузеров дают пользователям возможность отключить куки. Если это происходит, никакого предупреждения нет. Чтобы бороться с этой проблемой, можно использовать, например, такой простейший JS-код, чтобы уведомить пользователя о том, что куки должны быть включены для корректной работы.

<script>

document.cookie = "foo=bar;";

if (!document.cookie)

{

alert("This website requires cookies to function properly");

}

</script>

1. Каждая порция куки хранит не более 4 КБ данных. Помимо этого браузеры накладывают ограничения на то, сколько куки может установить сайт. Лимиты варьируются от 30 до 50.
2. Куки отправляются с каждым запросом к серверу. Если предположить, что у сайта 20 куки размером 4 КБ, то каждый запрос будет давать дополнительную нагрузку в размере 80 КБ.

Некоторые из этих проблем можно решить с помощью сессий, речь о которых пойдет в следующем уроке.

# #13 Сессии во Flask

Сессии — еще один способ хранить данные конкретных пользователей между запросами. Они работают по похожему на куки принципу. Для использования сессии нужно сперва настроить секретный ключ. Объект session из пакета flask используется для настройки и получения данных сессии. Объект session работает как словарь, но он также может отслеживать изменения.

При использовании сессий данные хранятся в браузере как куки. Куки, используемые для хранения данных сессии — это куки сессии. Тем не менее в отличие от обычных [**куки Flask**](https://pythonru.com/uroki/12-kuki-vo-flask) криптографически отмечает куки сессии. Это значит, что каждый может видеть содержимое куки, но не может их менять, не имея секретного ключа для подписи. Как только куки сессии настроены, каждый последующий запрос к серверу подтверждает подлинность куки с помощью такого же секретного ключа. Если Flask не удается это сделать, тогда его контент отклоняется, а браузер получает новые куки сессии.

Знакомые с сессиями из языка PHP заметят, что сессии во Flask немного отличаются. В PHP куки сессии не хранят данные о сессии, а только id сессии. Это уникальная строка, которую PHP создает для ассоциации данных сессии с куки. Данные сессии хранятся на сервере в виде файла. При получении запроса от пользователя PHP использует id сессии, чтобы найти данные сессии и отобразить их в коде. Такой тип сессий известен как серверный, а те, которые используются во Flask, называется клиентскими.

По умолчанию различий между куки и клиентскими сессиями во Flask не так много. В итоге клиентские сессии страдают от тех же недостатков, что и обычные куки:

* Не могут хранить конфиденциальную информацию, такую как пароли.
* Дают лишнюю нагрузку при каждом запросе.
* Не способны хранить больше 4 КБ.
* Ограничены в общем количестве куки для одного сайта

и так далее.

Единственное реальное различие между куки и клиентскими сессиями — Flask гарантирует, что содержимое куки сессии не может быть изменено пользователям (только если у него нет секретного ключа).

Для использования клиентских сессий во Flask можно или написать собственный интерфейс сессии или использовать [**расширения**](https://pythonru.com/uroki/10-rasshirenie-vozmozhnostej-flask-s-pomoshhju-flask-script), такие как Flask-Session или Flask-KVSession.

## Как читать, записывать и удалять данные сессии

Следующий код демонстрирует, как можно читать, записывать и удалять данные сессии. Откроем файл main2.py, чтобы добавить следующий код после функции представления article():

from flask import Flask, render\_template, request, redirect, url\_for, flash, make\_response, session

#...

@app.route('/visits-counter/')

def visits():

if 'visits' in session:

session['visits'] = session.get('visits') + 1 # чтение и обновление данных сессии

else:

session['visits'] = 1 # настройка данных сессии

return "Total visits: {}".format(session.get('visits'))

@app.route('/delete-visits/')

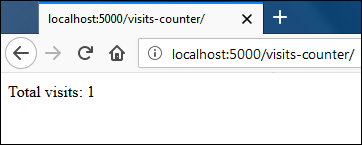
def delete\_visits():

session.pop('visits', None) # удаление данных о посещениях

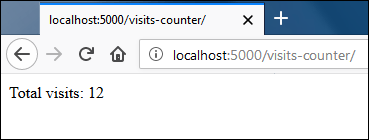
return 'Visits deleted'

#...

Стоит обратить внимание, что объект session используется как обычный словарь. Если сервер не запущен, нужно его запустить и зайти на https://localhost:5000/visits-counter/. На странице будет счетчик посещений:

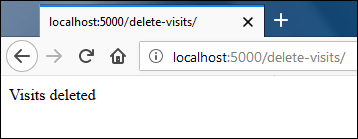


Чтобы увеличить его, нужно несколько раз обновить страницу.

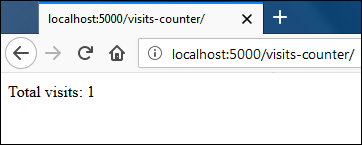


Flask отправляет куки сессии клиенту только при создании новой сессии или изменении существующей. При первом посещении https://localhost:5000/visits-counter/ будет исполнено тело else в функции представления visits(), в результате чего будет создана новая сессия. При создании новой сессии Flask отправит куки сессии клиенту. Последующие запросы к https://localhost:5000/visits-counter приведут к исполнению кода в блоке if, в котором обновляется значение счетчика visits сессии. При изменении сессии будет создан новый файл куки, поэтому Flask отправит новые куки сессии клиенту.

Чтобы удалить данные сессии нужно зайти на https://localhost:5000/delete-visits/.



Если сейчас открыть https://localhost:5000/visits-counter, счетчик посещений снова будет показывать 1.



По умолчанию куки сессии существуют до тех пор, пока не закроется браузер. Чтобы продлить жизнь куки сессии, нужно установить значение True для атрибута permanent объекта session. Когда значение permanent равно True, срок куки сессии будет равен permanent\_session\_lifetime. permanent\_session\_lifetime — это атрибут datetime.timedelta объекта Flask. Его значение по умолчанию равно 31 дню. Изменить его можно, выбрав новое значение для атрибута permanent\_session\_lifetime, используя ключ настройки PERMANENT\_SESSION\_LIFETIME.

import datetime

app = Flask(\_\_name\_\_)

app.permanent\_session\_lifetime = datetime.timedelta(days=365)

# app.config['PERMANENT\_SESSION\_LIFETIME'] = datetime.timedelta(days=365)

## Изменение данных сессии

**Примечание:** перед тем как следовать инструкции, нужно удалить куки, установленные локальным хостом.

Большую часть времени объект session автоматически подхватывает изменения. Но бывают случаи, например изменение структуры изменяемых данных, которые не подхватываются автоматически. Для таких ситуаций нужно установить значение True для атрибута modified объекта session. Если этого не сделать, Flask не будет отправлять обновленные куки клиенту. Следующий код показывает, как использовать атрибут modified объекта session. Откроем файл main2.py, чтобы добавить следующий код перед функцией представления delete\_visitis().

#...

@app.route('/session/')

def updating\_session():

res = str(session.items())

cart\_item = {'pineapples': '10', 'apples': '20', 'mangoes': '30'}

if 'cart\_item' in session:

session['cart\_item']['pineapples'] = '100'

session.modified = True

else:

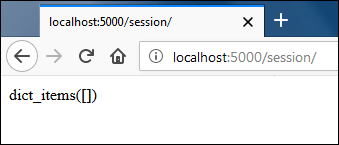
session['cart\_item'] = cart\_item

return res

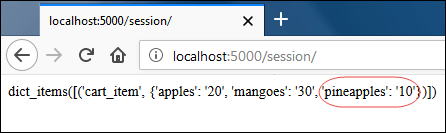
#...

При первом посещении https://localhost:5000/session/ код в блоке else будет исполнен. Он создаст новую сессию, где данные сессии будут в виде словаря. Последующий запрос к https://localhost:5000/session/ обновляет данные сессии, установив количество «ананасов» на значении 100. В следующей строке атрибут modified получает значение True, потому что без него Flask не будет отправлять обновленные куки сессии клиенту.

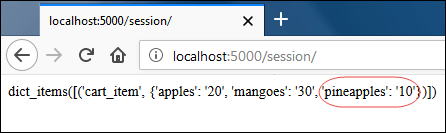
Если сервер не запущен, его следует запустить и зайти на https://localhost:5000/session/. Отобразится пустой словарь session, потому что у браузера еще нет куки сессии, которые он мог бы отправить серверу:



Если страницу перезагрузить, в словаре session будет уже «10 ананасов»:



Перезагрузив страницу в третий раз, можно увидеть, что словарь session имеет значение «ананасов» равное 100, а не 10:



Объект сессии подхватил изменение благодаря атрибуту modified. Удостовериться в этом можно, удалив куки сессии и закомментировав строку, где для атрибута modified устанавливается значение True. Теперь после первого запроса значение словаря сессии будет равно «10 ананасам».

Это все, что нужно знать о сессиях во Flask. И важно не забывать, что по умолчанию сессии во Flask являются клиентскими.

# #14 Создание баз данных во Flask

В этом уроке речь пойдет о взаимодействии с базой данных. Сегодня существуют две конкурирующих системы баз данных:

1. Реляционные базы данных.
2. Нереляционные или NoSQL базы данных.

Реляционные базы по традиции используются в веб-приложения. Многие крупные игроки на рынке веб-программирования все еще используют их. Например, Facebook. Реляционные базы данных хранят данные в таблицах и колонках и используют внешний ключ для создания связи между несколькими таблицами. Реляционные базы данных также поддерживают транзакции. Это значит, что можно исполнить набор SQL-операторов, которые должны быть атомарными (atomic). Под atomic подразумеваются все операторы, которые исполняются по принципу «все или ничего».

В последние годы выросла популярность баз данных NoSQL. Такие базы данных не хранят данные в таблицах и колонках, а вместо них используют такие структуры, как документные хранилища, хранилища ключей и значений, графы и так далее. Большинство NoSQL баз данных не поддерживают транзакции, но предлагают более высокую скорость работы.

Реляционные базы данных намного старше NoSQL. Они доказали свою надежность и безопасность во многих отраслях. Следовательно, оставшаяся часть урока будет посвящена описанию принципов использования реляционных баз данных во Flask. Это не значит, что NoSQL не используются. Есть случаи, когда в NoSQL-базах даже больше смысла, но сейчас речь пойдет только о реляционных базах данных.

## SQLAlchemy и Flask-SQLAchemy

SQLAlchemy – это фреймворк для работы, который на практике используется для работы с реляционными [**базами данных в Python**](https://pythonru.com/osnovy/sqlite-v-python). Он был создан Майком Байером в 2005 году. SQLAlchemy поддерживает следующие базы данных: MySQL, PostgreSQL, Oracle, MS-SQL, SQLite и другие.

SQLAchemy поставляется с мощным ORM (технология объектно-реляционного отображения), который позволяет работать с разными базами данных с помощью объектно-ориентированного кода, а не сырого SQL (языка структурированных запросов). Конечно, это не обязывает использовать только ORM. В любой момент можно задействовать возможности SQL.

Flask-SQLAlchemy – это расширение, которое интегрирует [**SQLAlchemy во фреймворк Flask**](https://pythonru.com/biblioteki/sqlalchemy-v-flask). Он также предлагает дополнительные методы, благодаря которым работать с SQLAlchemy становится немного проще. Установить Flask-SQLAlchemy вместе с дополнительными модулями можно с помощью следующей команды:

(env) gvido@vm:~/flask\_app$ pip install flask-sqlalchemy

Для использования Flask-SQLAlchemy нужно импортировать класс SQLAlchemy из пакета flask\_sqlalchemy и создать экземпляр объекта SQLAlchemy, передав ему экземпляр приложения. Откроем файл main2.py, чтобы изменить код следующим образом:

#...

from forms import ContactForm

from flask\_sqlalchemy import SQLAlchemy

app = Flask(\_\_name\_\_)

app.debug = True

app.config['SECRET\_KEY'] = 'a really really really really long secret key'

manager = Manager(app)

db = SQLAlchemy(app)

class Faker(Command):

#...

Экземпляр db объекта SQLAlchemy предоставляет доступ к функциям SQLAlchemy.

Дальше нужно сообщить SQLAlchemy местоположение базы данных в виде URI. Формат URI базы данных следующий:

dialect+driver://username:password@host:port/database

dialect ссылается на имя базы данных, такое как mysql, mssql, postgresql и так далее.

driver ссылается на DBAPI, который он использует, чтобы соединяться с базой данных. По умолчанию SQLAlchemy работает только с SQLite без дополнительных драйверов. Чтобы работать с другими базами данных, нужно установить конкретный драйвер для базы данных, совместимый с DBAPI.

Что такое DBAPI?

DBAPI – это всего лишь стандарт, определяющий API Python для доступа к базам данных от разных производителей.

Следующая таблица содержит некоторые базы данных и драйвера для них, совместимые с DBAPI:

| **База данных** | **Драйвер DBAPI** |
| --- | --- |
| MySQL | PyMysql |
| PostgreSQL | Psycopg 2 |
| MS-SQL | pyodbc |
| Oracle | cx\_Oracle |

Username и password указываются только при необходимости. Если указаны, они будут использоваться для авторизации в базе данных.

host — местоположение сервера базы данных.

port — порт сервера базы данных.

database — имя базы данных.

Вот некоторые примеры URL баз данных для самых популярных типов:

# URL базы данных для MySQL с использованием драйвера PyMysql

'mysql+pymysql://root:pass@localhost/my\_db'

# URL базы данных для PostgreSQL с использованием psycopg2

'postgresql+psycopg2://root:pass@localhost/my\_db'

# URL базы данных для MS-SQL с использованием драйвера pyodbc

'mssql+pyodbc://root:pass@localhost/my\_db'

# URL базы данных для Oracle с использованием драйвера cx\_Oracle

'oracle+cx\_oracle://root:pass@localhost/my\_db'

Формат URL базы данных для SQLite слегка отличается. Поскольку SQLite – это база данных, основанная на файле, и она не требует имени пользователя и пароля, в URL базы данных указывается только путь к файлу базы.

# Для Unix / Mac мы используем 4 слеша

sqlite:////absolute/path/to/my\_db.db

# Для Windows мы используем 3 слеша

sqlite:///c:/absolute/path/to/mysql.db

Flask-SQLAlchemy использует конфигурационный ключ SQLALCHEMY\_DATABASE\_URI для определения URI базы данных. Откроем main2.py, чтобы добавить SQLALCHEMY\_DATABASE\_URI :

#...

app = Flask(\_\_name\_\_)

app.debug = True

app.config['SECRET\_KEY'] = 'a really really really really long secret key'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql+pymysql://root:pass@localhost/flask\_app\_db'

manager = Manager(app)

db = SQLAlchemy(app)

#...

В этом курсе будет использоваться база данных MySQL. Поэтому прежде чем переходить к следующему разделу, нужно убедиться, что MySQL работает на компьютере.

## Создание моделей

Модель — это [**класс в Python**](https://pythonru.com/osnovy/klass-i-obekt-v-python), который представляет собой таблицу базы данных. Ее атрибуты сопоставляются со столбцами таблицы. Класс модели наследуется из db.Mobel и определяет колонки как экземпляры класса db.Column. Откроем main2.py, чтобы добавить следующий класс перед функцией представления updating\_session():

#...

from flask\_sqlalchemy import SQLAlchemy

from datetime import datetime

#...

class Post(db.Model):

\_\_tablename\_\_ = 'posts'

id = db.Column(db.Integer(), primary\_key=True)

title = db.Column(db.String(255), nullable=False)

slug = db.Column(db.String(255), nullable=False)

content = db.Column(db.Text(), nullable=False)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

updated\_on = db.Column(db.DateTime(), default=datetime.utcnow, onupdate=datetime.utcnow)

def \_\_repr\_\_(self):

return "<{}:{}>".format(self.id, self.title[:10])

Здесь создается модель Post с 5 переменными класса. Каждая переменная класса, кроме \_\_tablename\_\_, — это экземпляр класса db.Column. \_\_tablename\_\_ — это специальная переменная класса, используемая для определения имени таблицы базы данных. По умолчанию SQLAlchemy не следует соглашению о создании имен во множественном числе, поэтому название таблицы здесь — это название модели. Если на хочется опираться на такое поведение, следует использовать переменную \_\_tablename\_\_, чтобы явно указать название таблицы.

Первый аргумент конструктора db.Column() — это тип колонки, которая создается. SQLAlchemy предлагает большое количество типов колонок, а если их недостаточно, то можно создать свои. Следующая таблица описывает основные типы колонок в SQLAlchemy и их соответствующие типы в Python и SQL.

| **SQLAlchemy** | **Python** | **SQL** |
| --- | --- | --- |
| BigInteger | int | BIGINT |
| Boolean | bool | BOOLEAN или SMALLINT |
| Date | datetime.date | DATE |
| DateTime | datetime.date | DATETIME |
| Integer | int | INTEGER |
| Float | float | FLOAT или REAL |
| Numeric | decimal.Decimal | NUMERIC |
| Text | str | TEXT |

Также можно задать дополнительные ограничения для колонки, передав их в виде аргументов-ключевых слов конструктору db.Column. Следующая таблица включает некоторые широко используемые ограничения:

| **Ограничение** | **Описание** |
| --- | --- |
| nullable | Когда значение равно False, делает колонку обязательной. Значение по умолчанию — True. |
| default | Создает значение по умолчанию для колонки. |
| index | Логический атрибут. Если True, создает индексированную колонку. |
| onupdate | Создает значение по умолчанию для колонки при обновлении записи. |
| primary\_key | Логический атрибут. Если True, отмечает колонку основным ключом таблицы. |
| unique | Логический атрибут. Если True, каждая колонка должна быть уникальной. |

В строках 16-17 был определен метод \_\_repr\_\_(). Он не необходим, но если есть, то создает строчное представление объекта.

Можно было заметить, что значениями по умолчанию для created\_on и updated\_on выбрано название метода (datetime.utcnow), а не его вызов (datetime.utcnow()). Так сделано, потому что при исполнении кода вызывать метод datetime.utcnow() нет необходимости. Вместо этого его стоит вызывать, когда запись добавляется или обновляется.

## Определение отношений (связей)

В прошлом разделе была создана модель Post с парой полей. На практике классы моделей существуют сами по себе. Большую часть времени они связаны с другими моделями различными типами отношений: один-к-одному, один-ко-многим, многие-ко-многим.

Стоит дальше поработать над аналогией блога. Обычно, пост в блоге относится к одной категории и имеет один или несколько тегов. Другими словами, есть отношение один-к-одному между категорией и постом и отношение многие-ко-многим между постом и тегом.

Откроем main2.py, чтобы добавить модели Category и Tag:

#...

def updating\_session():

#...

return res

class Category(db.Model):

\_\_tablename\_\_ = 'categories'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(255), nullable=False)

slug = db.Column(db.String(255), nullable=False)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

def \_\_repr\_\_(self):

return "<{}:{}>".format(id, self.name)

class Posts(db.Model):

# ...

class Tag(db.Model):

\_\_tablename\_\_ = 'tags'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(255), nullable=False)

slug = db.Column(db.String(255), nullable=False)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

def \_\_repr\_\_(self):

return "<{}:{}>".format(id, self.name)

#...

### Отношение один-ко-многим

Для создания отношения один-ко-многим нужно разместить внешний ключ в дочерней таблице. Это самый распространенный тип отношений. Для создания отношения один-ко-многим в SQLAlchemy нужно выполнить следующие шаги:

1. Создать новый экземпляр db.Column с помощью ограничения db.ForeignKey в дочернем классе.
2. Определить новое свойство с помощью инструкции db.relationship в родительском классе. Это свойство будет использоваться для получения доступа к связанным объектам.

Откроем main2.py, чтобы изменить модели Post и Catеgory:

#...

class Category(db.Model):

# ...

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

posts = db.relationship('Post', backref='category')

class Post(db.Model):

# ...

updated\_on = db.Column(db.DateTime(), default=datetime.utcnow, onupdate=datetime.utcnow)

category\_id = db.Column(db.Integer(), db.ForeignKey('categories.id'))

#...

Здесь для модели Post в Category были добавлены два новых атрибута: posts и category\_id.

db.ForeignKey() принимает имя столбца, внешний ключ которого используется. Здесь значение categories.id передается исключению db.ForeignKey(). Это значит, что атрибут category\_id у Post может принимать значение только у колонки id таблицы categories.

Далее в модели Catagory имеется атрибут posts, определенный инструкцией db.relationship(). db.relationship() используется для добавления двунаправленной связи. Другими словами, она добавляет [**атрибут классу**](https://pythonru.com/primery/primery-raboty-s-klassami-v-python) модели для доступа к связанным объектам. Простыми словами, она принимает как минимум один позиционный аргумент, который является именем класса на другой стороне отношений.

class Category(db.Model):

# ...

posts = db.relationship('Post')

Например, если есть объект Category (скажем, c), тогда доступ ко всем постам можно получить с помощью c.posts. А что, если нужно получить данные с другой стороны, то есть, получить категорию у объекта поста? Для этого используется backref. Так, код:

posts = db.relationship('Post', backref='category')

добавляет атрибут category объекту Post. Это значит, что если есть объект Post (например, p), тогда доступ к категории можно получать с помощью p.category.

Атрибуты category и posts у объектов Post и Category существуют только для удобства. Они не являются реальными колонками в таблице.

Стоит отметить, что в отличие от атрибута, представленного внешним ключом (который должен быть определен на стороне «много» в отношениях), db.relationship() можно определять с любой стороны.

### Отношение один-к-одному

Создание отношения один-к-одному в SQLAlchemy – это почти то же самое, что и отношение один-ко-многим. Единственное отличие — то, что инструкции db.relationship() передается дополнительный аргумент uselist=False. Например:

class Employee(db.Model):

\_\_tablename\_\_ = 'employees'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(255), nullable=False)

designation = db.Column(db.String(255), nullable=False)

doj = db.Column(db.Date(), nullable=False)

dl = db.relationship('DriverLicense', backref='employee', uselist=False)

class DriverLicense(db.Model):

\_\_tablename\_\_ = 'driverlicense'

id = db.Column(db.Integer(), primary\_key=True)

license\_number = db.Column(db.String(255), nullable=False)

renewed\_on = db.Column(db.Date(), nullable=False)

expiry\_date = db.Column(db.Date(), nullable=False)

employee\_id = db.Column(db.Integer(), db.ForeignKey('employees.id')) # Foreign key

**Примечание**: в этих класса предполагается, что у сотрудника (employee) не может быть большого одного водительского удостоверения (driver license). Поэтому отношения между сотрудником и правами — один-к-одному.

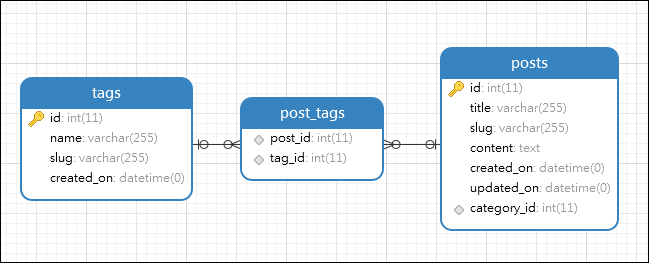
С объектом Employee можно использовать e.dl, чтоб вернуть объект DriverLicense. Если не передать инструкции db.relationship() значение uselist=False, тогда между Employee и DriverLicense будет установлено отношение один-ко-многим, и e.dl вернет список объектов DriverLicense, вместо одного объекта. При этом аргумент uselist=False не повлияет на атрибут employee объекта DriverLicense. Как и обычно, он вернет один объект.

### Отношение многие-ко-многим

Отношение многие-ко-многим требует дополнительной ассоциативной таблицы. В качестве примера можно взять блог.

Пост в блоге обычно имеет один или несколько тегов. Аналогичным образом один тег может ассоциироваться с одним или несколькими постами. Так образовывается отношение между posts и tags. Недостаточно добавить внешний ключ, ссылающийся на id постов, потому что у тега может быть один или несколько постов.

В качестве решения нужно создать новую таблицу ассоциаций, определив 2 внешних ключа, ссылающихся на колонки post.id и tag.id.



Как видно на изображении, отношение многие-ко-многим между постом и тегом создается с помощью двух отношений один-к-одному. Первое такое отношение установлено между таблицами posts и post\_tags, второе — между tags и post\_tags. Следующий код демонстрирует, как создать отношение многие-ко-многим в SQLAlchemy. Откроем файл main2.py, чтобы добавить следующий код.

# ...

class Category(db.Model):

# ...

def \_\_repr\_\_(self):

return "<{}:{}>".format(id, self.name)

post\_tags = db.Table('post\_tags',

db.Column('post\_id', db.Integer, db.ForeignKey('posts.id')),

db.Column('tag\_id', db.Integer, db.ForeignKey('tags.id'))

)

class Post(db.Model):

# ...

class Tag(db.Model):

# ...

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

posts = db.relationship('Post', secondary=post\_tags, backref='tags')

#...

На строках 7-10 таблица ассоциаций определяется в виде объекта db.Table(). Первый аргумент таблицы db.Table() — имя таблицы, а дополнительные аргументы — это колонки, представленные экземплярами db.Column(). Синтаксис для создания таблицы ассоциаций может показаться странным, если сравнивать с процессом создания класса модели. Это потому что таблица ассоциаций создается с помощью SQLAlchemy Core – еще одного элемента SQLAlchemy.

Дальше нужно сообщить классу модели о таблице ассоциаций, которая будет использоваться. За это отвечает аргумент-ключевое слово secondary. На 18 строке db.relationship() вызывается с аргументом secondary, значение которого — post\_tags. Хотя отношение было определено в модели Tag, его можно так же просто определить в модели Post.

Если есть, например, объект p класса Post, тогда доступ ко всем его тегам можно получить с помощью p.tags. С помощью объекта класса Tag (t), доступ к постам можно получить командой t.posts.

Пришло время создать базу данных и таблицы.

## Создание таблиц

Чтобы выполнить все шаги урока, нужно убедиться, что MySQL установлен на компьютере.

Стоит напомнить, что по умолчанию SQLAlchemy работает только с базой данных SQLite. Для работы с другими базами данных нужно установить драйвер, совместимый с DBAPI. Для использования MySQL подойдет драйвер PyMySql.

(env) gvido@vm:~/flask\_app$ pip install pymysql

После этого необходимо авторизоваться на сервере MySQL и создать базу данных flask\_app\_db с помощью следующей команды:

(env) gvido@vm:~/flask\_app$ mysql -u root -p

mysql>

mysql> CREATE DATABASE flask\_app\_db CHARACTER SET utf8mb4 COLLATE utf8mb4\_unicode\_ci;

Query OK, 1 row affected (0.26 sec)

mysql> \q

Bye

(env) gvido@vm:~/flask\_app$

Эта команда создает базу данных flask\_app\_db с полной поддержкой Unicode.

Для создания необходимых таблицы нужно запустить метод create\_all() объекта SQLAlchemy — db. Далее нужно запустить оболочку Python и выполнить следующую команду:

(env) gvido@vm:~/flask\_app$ python main2.py shell

>>>

>>> from main2 import db

>>>

>>> db.create\_all()

>>>

Метод create\_all() создает таблицы только в том случае, если их нет в базе данных. Поэтому запускать его можно несколько раз. Также этот метод не берет во внимание изменения моделей при создании таблиц. Это значит, что если запустить метод create\_all() после изменения его метода, когда таблица уже создана, то он не поменяет схему таблицы. Чтобы сделать это, нужно воспользоваться инструментом переноса Alembic. О том, как переносить базы данных с помощью Alembic, будет рассказано в отдельном уроке «Перенос базы данных с помощью Alembic».

Чтобы посмотреть созданные таблицы, нужно авторизоваться на сервере MySQL и выполнить следующую команду:

mysql>

mysql> use flask\_app\_db

Database changed

mysql>

mysql> show tables;

+------------------------+

| Tables\_in\_flask\_app\_db |

+------------------------+

| categories |

| post\_tags |

| posts |

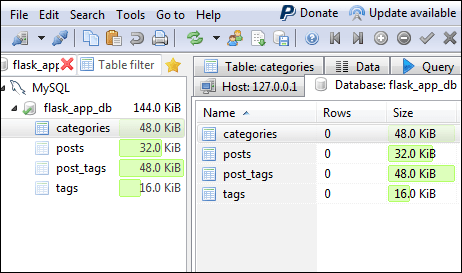
| tags |

+------------------------+

4 rows in set (0.02 sec)

mysql>

Еще один способ посмотреть таблицы — использовать инструмент администрирования базы данных, такой как HeidiSQL. HeidiSQL – это кроссплатформенное ПО с открытым исходным кодом для управления базами данных MySQL, MS-SQL и PostgreSQL. Оно позволяет просматривать и редактировать данные, смотреть схему, менять таблицу и делать многое другое без единой строчки SQL. Скачать HeidiSQL можно [**отсюда**](https://www.heidisql.com/download.php).

Установив HeidiSQL поверх базы данных flask\_app\_db, можно получить приблизительно следующий список таблиц:  


База данных flask\_app\_db имеет 4 таблицы. Таблицы с названиями categories, posts и tags созданы прямо из моделей, а post\_tags — это таблица ассоциаций, которая представляет собой отношение многие-ко-многим между моделями Post и Tag.

Класс SQLAlchemy также определяет метод drop\_all(), который используется для удаления всех таблиц в базе данных. Стоит помнить, что метод drop\_all() не учитывает, есть ли данные в таблице или нет. Он удаляет все данные, поэтому использовать его нужно с умом.

Все таблицы на месте. Пора добавить в них какие-то данные.

# #15 Основы ORM SQLAlchemy

## Добавление данных

Чтобы создать новую запись с данными с помощью SQLAlchemy, нужно выполнить следующие шаги:

1. Создать объект
2. Добавить объект в сессию
3. Загрузить (сделать коммит) сессию

В SAQLAlchemy взаимодействие с базой данных происходит с помощью [**сессии**](https://pythonru.com/uroki/13-sessii-vo-flask). К счастью, ее не нужно создавать вручную. Это делает Flask-SQLAlchemy. Доступ к объекту сессии можно получить с помощью db.session. Это объект сессии, которые отвечает за подключение к базе данных. Он же отвечает за процесс транзакции. По умолчанию транзакция запускается и остается открытой до тех пор, пока выполняются коммиты и откаты.

Запустим оболочку Python для создания некоторых объектов модели:

(env) gvido@vm:~/flask\_app$ python main2.py shell

>>>

>>> from main2 import db, Post, Tag, Category

>>>

>>>

>>> c1 = Category(name='Python', slug='python')

>>> c2 = Category(name='Java', slug='java')

>>>

Были созданы два объекта Category. Получить доступ к их атрибутам можно с помощью оператора точки (.):

>>>

>>> c1.name, c1.slug

('Python', 'python')

>>>

>>> c2.name, c2.slug

('Java', 'java')

>>>

Дальше необходимо добавить объекты в сессию.

>>>

>>> db.session.add(c1)

>>> db.session.add(c2)

>>>

Добавление объектов не записывает их в базу данных , этот процесс лишь готовит их для сохранения при следующей коммите. Удостовериться в этом можно, проверив первичный ключ объектов.

>>>

>>> print(c1.id)

None

>>>

>>> print(c2.id)

None

>>>

Значение атрибута id обоих объектов — None. Это значит, что объекты не сохранены в базе данных.

Вместо добавления по одному объекту в сессию каждый раз, можно использовать метод add\_all(). Метод add\_all() принимает список объектов, которые нужно добавить в сессию.

>>>

>>> db.session.add\_all([c1, c1])

>>>

Если попытаться добавить объект в сессию несколько раз, ошибок не возникнет. В любой момент можно посмотреть все объекты сессии с помощью db.session.new.

>>>

>>> db.session.new

IdentitySet([<None:Python>, <None:java>])

>>>

Наконец, для сохранения объектов в базе данных нужно вызвать метод commit():

>>>

>>> db.session.commit()

>>>

Если обратиться к атрибуту id объекта Category сейчас, то он вернет первичный ключ, а не None.

>>>

>>> print(c1.id)

1

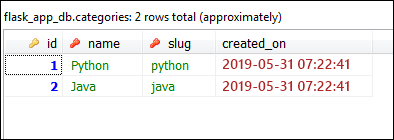
>>>

>>> print(c2.id)

2

>>>

На этом этапе таблица categories в HeidiSQL должна выглядеть примерно так:



Новые категории пока не связаны с постами. Поэтому c1.posts и c2.posts вернут пустой список.

>>>

>>> c1.posts

[]

>>>

>>> c2.posts

[]

>>>

Стоит попробовать создать несколько постов.

>>>

>>> p1 = Post(title='Post 1', slug='post-1', content='Post 1', category=c1)

>>> p2 = Post(title='Post 2', slug='post-2', content='Post 2', category=c1)

>>> p3 = Post(title='Post 3', slug='post-3', content='Post 3', category=c2)

>>>

Вместо того чтобы передавать категорию при создании объекта Post, можно выполнить следующую команду:

>>>

>>> p1.category = c1

>>>

Дальше нужно добавить объекты в сессию и сделать коммит.

>>>

>>> db.session.add\_all([p1, p2, p3])

>>> db.session.commit()

>>>

Если сейчас попробовать получить доступ к атрибуту posts объекта Category, то он вернет не-пустой список:

>>>

>>> c1.posts

[<1:Post 1>, <2:Post 2>]

>>>

>>> c2.posts

[<3:Post 3>]

>>>

С другой стороны отношения, можно получить доступ к объекту Category, к которому относится пост, с помощью атрибута category у объекта Post.

>>>

>>> p1.category

<1:Python>

>>>

>>> p2.category

<1:Python>

>>>

>>> p3.category

<2:Java>

>>>

Стоит напомнить, что все это возможно благодаря инструкции relationship() в модели Category. Сейчас в базе данных есть три поста, но ни один из них не связан с тегами.

>>>

>>> p1.tags, p2.tags, p3.tags

([], [], [])

>>>

Пришло время создать теги. Это можно сделать в оболочке следующим образом:

>>>

>>> t1 = Tag(name="refactoring", slug="refactoring")

>>> t2 = Tag(name="snippet", slug="snippet")

>>> t3 = Tag(name="analytics", slug="analytics")

>>>

>>> db.session.add\_all([t1, t2, t3])

>>> db.session.commit()

>>>

Этот код создает три объекта тегов и делает их коммит в базу данных. Посты все еще не привязаны к тегам. Вот как можно связать объект Post с объектом Tag.

>>>

>>> p1.tags.append(t1)

>>> p1.tags.extend([t2, t3])

>>> p2.tags.append(t2)

>>> p3.tags.append(t3)

>>>

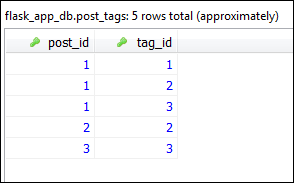
>>> db.session.add\_all([p1, p2, p3])

>>>

>>> db.session.commit()

>>>

Этот коммит добавляет следующие пять записей в таблицу post\_tags.



Посты теперь связаны с одним или большим количеством тегов:

>>>

>>> p1.tags

[<1:refactoring>, <2:snippet>, <3:analytics>]

>>>

>>> p2.tags

[<2:snippet>]

>>>

>>> p3.tags

[<3:analytics>]

>>>

С другой стороны можно получить доступ к постам, которые относятся к конкретному тегу:

>>>

>>> t1.posts

[<1:Post 1>]

>>>

>>> t2.posts

[<1:Post 1>, <2:Post 2>]

>>>

>>> t3.posts

[<1:Post 1>, <3:Post 3>]

>>>

>>>

Важно отметить, что вместо изначального коммита объектов Tag и последующей их связи с объектами Post, все это можно сделать и таким способом:

>>>

>>> t1 = Tag(name="refactoring", slug="refactoring")

>>> t2 = Tag(name="snippet", slug="snippet")

>>> t3 = Tag(name="analytics", slug="analytics")

>>>

>>> p1.tags.append(t1)

>>> p1.tags.extend([t2, t3])

>>> p2.tags.append(t2)

>>> p3.tags.append(t3)

>>>

>>> db.session.add(p1)

>>> db.session.add(p2)

>>> db.session.add(p3)

>>>

>>> db.session.commit()

>>>

Важно обратить внимание, что на строках 11-13 в сессию добавляются только объекты Post. Объекты Tag и Post связаны отношением многие-ко-многим. В результате, добавление объекта Post в сессию влечет за собой добавление связанных с ним объектов Tag. Но даже если сейчас вручную добавить объекты Tag в сессию, ошибки не будет.

## Обновление данных

Для обновления объекта нужно всего лишь передать его атрибуту новое значение, добавить объект в сессию и сделать коммит.

>>>

>>> p1.content # начальное значение

'Post 1'

>>>

>>> p1.content = "This is content for post 1" # задаем новое значение

>>> db.session.add(p1)

>>>

>>> db.session.commit()

>>>

>>> p1.content # обновленное значение

'This is content for post 1'

>>>

## Удаление данных

Для удаления объекта нужно использовать метод delete() объекта сессии. Он принимает объект и отмечает, что тот подлежит удалению при следующем коммите.

Создадим новый временный тег seo и свяжем его с постами p1 и p2:

>>>

>>> tmp = Tag(name='seo', slug='seo') # создание временного объекта Tag

>>>

>>> p1.tags.append(tmp)

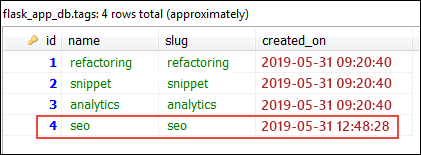
>>> p2.tags.append(tmp)

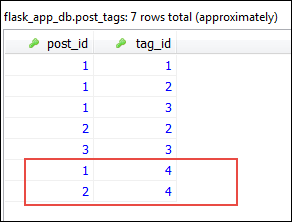
>>>

>>> db.session.add\_all([p1, p2])

>>> db.session.commit()

>>>

Этот коммит добавляет всего 3 строки: одну в таблицу table и еще две — в таблицу post\_tags. В базе данных эти три строки выглядят следующим образом:  




Теперь нужно удалить тег seo:

>>>

>>> db.session.delete(tmp)

>>> db.session.commit()

>>>

Этот коммит удаляет все три строки, добавленные в предыдущем шаге. Тем не менее он не удаляет пост, с которым тег был связан.

По умолчанию при удалении объекта в родительской таблице (например, categories) значение внешнего ключа объекта, который с ним связан в дочерней таблице (например, posts) становится NULL. Следующий код демонстрирует это поведение на примере создания нового объекта категории и объекта поста, который с ней связан, и дальнейшим удалением объекта категории:

>>>

>>> c4 = Category(name='css', slug='css')

>>> p4 = Post(title='Post 4', slug='post-4', content='Post 4', category=c4)

>>>

>>> db.session.add(c4)

>>>

>>> db.session.new

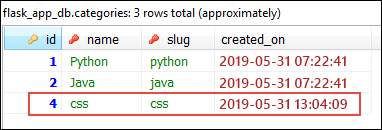
IdentitySet([<None:css>, <None:Post 4>])

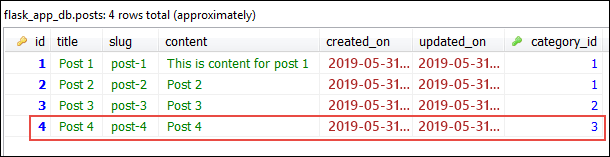
>>>

>>> db.session.commit()

>>>

Этот коммит добавляет две строки. Одну в таблицу categories, и еще одну — в таблицу posts.





Теперь нужно посмотреть, что происходит при удалении объекта Category.

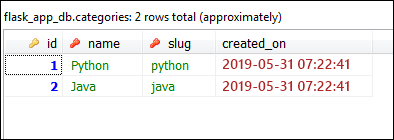
>>>

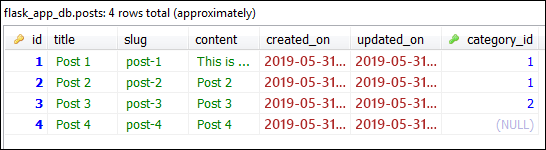
>>> db.session.delete(c4)

>>> db.session.commit()

>>>

Этот коммит удаляет категорию css из таблицы categories и устанавливает значение внешнего ключа (category\_id) для поста, который с ней связан, на NULL.





В некоторых случаях может возникнуть необходимость удалить все дочерние записи при том, что родительские записи уже удалены. Это можно сделать, передав cascade=’all,delete-orphan’ инструкции db.relationship(). Откроем main2.py, чтобы изменить инструкцию db.relationship() в модели Catagory:

#...

class Category(db.Model):

#...

posts = db.relationship('Post', backref='category', cascade='all,delete-orphan')

#...

С этого момента удаление категории повлечет за собой удаление постов, которые с ней связаны. Чтобы это начало работать, нужно перезапустить оболочку. Далее импортируем нужные объекты и создаем категорию вместе с постом:

(env) gvido@vm:~/flask\_app$ python main2.py shell

>>>

>>> from main2 import db, Post, Tag, Category

>>>

>>> c5 = Category(name='css', slug='css')

>>> p5 = Post(title='Post 5', slug='post-5', content='Post 5', category=c5)

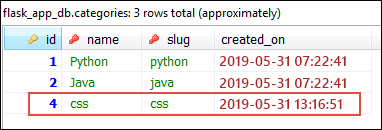
>>>

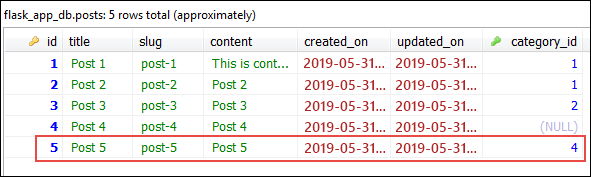
>>> db.session.add(c5)

>>> db.session.commit()

>>>

Вот как база данных выглядит после этого коммита.





Удалим категорию.

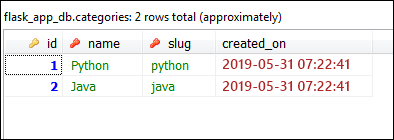
>>>

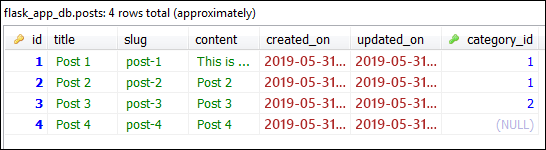
>>> db.session.delete(c5)

>>> db.session.commit()

>>>

После этого коммита база данных выглядит вот так:





## Запрос данных

Чтобы выполнить запрос к базе данных, используется метод query() объекта session. Метод query() возвращает объект flask\_sqlalchemy.BaseQuery, который является расширением оригинального объекта sqlalchemy.orm.query.Query. Объект flask\_sqlalchemy.BaseQuery представляет собой оператор SELECT, который будет использоваться для осуществления запросов к базе данных. В этой таблице перечислены основные [**методы класса**](https://pythonru.com/primery/primery-raboty-s-klassami-v-python) flask\_sqlalchemy.BaseQuery.

| **Метод** | **Описание** |
| --- | --- |
| all() | Возвращает результат запроса (представленный flask\_sqlalchemy.BaseQuery) в виде списка. |
| count() | Возвращает количество записей в запросе. |
| first() | Возвращает первый результат запроса или None, если в нем нет строк. |
| first\_or\_404() | Возвращает первый результат запроса или ошибку 404, если в нем нет строк. |
| get(pk) | Возвращает объект, который соответствует данному первичному ключу или None, если объект не найден. |
| get\_or\_404(pk) | Возвращает объект, который соответствует данному первичному ключу или ошибку 404, если объект не найден. |
| filter(\*criterion) | Возвращает новый экземпляр flask\_sqlalchemy.BaseQuery с оператором WHERE. |
| limit(limit) | Возвращает новый экземпляр flask\_sqlalchemy.BaseQuery с оператором LIMIT. |
| offset(offset) | Возвращает новый экземпляр flask\_sqlalchemy.BaseQuery с оператором OFFSET. |
| order\_by(\*criterion) | Возвращает новый экземпляр flask\_sqlalchemy.BaseQuery с оператором OFFSET. |
| join() | Возвращает новый экземпляр flask\_sqlalchemy.BaseQuery после создания SQL JOIN. |

### Метод all()

В своей простейшей форме метод query() принимает в качестве аргументов один или больше классов модели или колонки. Следующий код вернет все записи из таблицы posts.

>>>

>>> db.session.query(Post).all()

[<1:Post 1>, <2:Post 2>, <3:Post 3>, <4:Post 4>]

>>>

Похожим образом следующий код вернет все записи из таблиц categories и tags.

>>>

>>> db.session.query(Category).all()

[<1:Python>, <2:Java>]

>>>

>>>

>>> db.session.query(Tag).all()

[<1:refactoring>, <2:snippet>, <3:analytics>]

>>>

Чтобы получить чистый SQL, использованный для запроса к базе данных, нужно просто вывести объект flask\_sqlalchemy.BaseQuery:

>>>

>>> print(db.session.query(Post))

SELECT

posts.id AS posts\_id,

posts.title AS posts\_title,

posts.slug AS posts\_slug,

posts.content AS posts\_content,

posts.created\_on AS posts\_created\_on,

posts.u pdated\_on AS posts\_updated\_on,

posts.category\_id AS posts\_category\_id

FROM

posts

>>>

В предыдущих примерах данные возвращались со всех колонок таблицы. Это можно поменять, передав методу query() названия колонок:

>>>

>>> db.session.query(Post.id, Post.title).all()

[(1, 'Post 1'), (2, 'Post 2'), (3, 'Post 3'), (4, 'Post 4')]

>>>

### Метод count()

Метод count() возвращает количество результатов в запросе.

>>>

>>> db.session.query(Post).count() # получить общее количество записей в таблице Post

4

>>> db.session.query(Category).count() # получить общее количество записей в таблице Category

2

>>> db.session.query(Tag).count() # получить общее количество записей в таблице Tag

3

>>>

### Метод first()

Метод first() вернет только первый запрос из запроса или None, если в запросе нет результатов.

>>>

>>> db.session.query(Post).first()

<1:Post 1>

>>>

>>> db.session.query(Category).first()

<1:Python>

>>>

>>> db.session.query(Tag).first()

<1:refactoring>

>>>

### Метод get()

Метод get() вернет экземпляр объекта с соответствующим первичным ключом или None, если такой объект не был найден.

>>>

>>> db.session.query(Post).get(2)

<2:Post 2>

>>>

>>> db.session.query(Category).get(1)

<1:Python>

>>>

>>> print(db.session.query(Category).get(10)) # ничего не найдено по первичному ключу 10

None

>>>

### Метод get\_or\_404()

То же самое, что и метод get(), но вместо None вернет ошибку 404, если объект не найден.

>>>

>>> db.session.query(Post).get\_or\_404(1)

<1:Post 1>

>>>

>>>

>>> db.session.query(Post).get\_or\_404(100)

Traceback (most recent call last):

...

werkzeug.exceptions.NotFound: 404 Not Found: The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

>>>

### Метод filter()

Метод filter() позволяет отсортировать результатов с помощью оператора WHERE, примененного к запросу. Он принимает колонку, оператор или значение. Например:

>>>

>>> db.session.query(Post).filter(Post.title == 'Post 1').all()

[<1:Post 1>]

>>>

Запрос вернет все посты с заголовком "Post 1". SQL-эквивалент запроса следующий:

>>>

>>> print(db.session.query(Post).filter(Post.title == 'Post 1'))

SELECT

posts.id AS posts\_id,

posts.title AS posts\_title,

posts.slug AS posts\_slug,

posts.content AS posts\_content,

posts.created\_on AS posts\_created\_on,

posts.u pdated\_on AS posts\_updated\_on,

posts.category\_id AS posts\_category\_id

FROM

posts

WHERE

posts.title = % (title\_1) s

>>>

>>>

Строка % (title\_1) s в условии WHERE — это заполнитель. На ее месте будет реальное значение при выполнении запроса.

Методу filter() можно передать несколько значений и они будут объединены оператором AND в SQL. Например:

>>>

>>> db.session.query(Post).filter(Post.id >= 1, Post.id <= 2).all()

[<1:Post 1>, <2:Post 2>]

>>>

>>>

Этот запрос вернет все посты, первичный ключ которых больше 1, но меньше 2. SQL-эквивалент:

>>>

>>> print(db.session.query(Post).filter(Post.id >= 1, Post.id <= 2))

SELECT

posts.id AS posts\_id,

posts.title AS posts\_title,

posts.slug AS posts\_slug,

posts.content AS posts\_content,

posts.created\_on AS posts\_created\_on,

posts.u pdated\_on AS posts\_updated\_on,

posts.category\_id AS posts\_category\_id

FROM

posts

WHERE

posts.id >= % (id\_1) s

AND posts.id <= % (id\_2) s

>>>

### Метод first\_or\_404()

Делает то же самое, что и метод first(), но вместо None возвращает ошибку 404, если запрос без результата.

>>>

>>> db.session.query(Post).filter(Post.id > 1).first\_or\_404()

<2:Post 2>

>>>

>>> db.session.query(Post).filter(Post.id > 10).first\_or\_404().all()

Traceback (most recent call last):

...

werkzeug.exceptions.NotFound: 404 Not Found: The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

>>>

### Метод limit()

Метод limit() добавляет оператор LIMIT к запросу. Он принимает количество строк, которые нужно вернуть с запросом.

>>>

>>> db.session.query(Post).limit(2).all()

[<1:Post 1>, <2:Post 2>]

>>>

>>> db.session.query(Post).filter(Post.id >= 2).limit(1).all()

[<2:Post 2>]

>>>

SQL-эквивалент:

>>>

>>> print(db.session.query(Post).limit(2))

SELECT

posts.id AS posts\_id,

posts.title AS posts\_title,

posts.slug AS posts\_slug,

posts.content AS posts\_content,

posts.created\_on AS posts\_created\_on,

posts.u pdated\_on AS posts\_updated\_on,

posts.category\_id AS posts\_category\_id

FROM

posts

LIMIT % (param\_1) s

>>>

>>>

>>> print(db.session.query(Post).filter(Post.id >= 2).limit(1))

SELECT

posts.id AS posts\_id,

posts.title AS posts\_title,

posts.slug AS posts\_slug,

posts.content AS posts\_content,

posts.created\_on AS posts\_created\_on,

posts.u pdated\_on AS posts\_updated\_on,

posts.category\_id AS posts\_category\_id

FROM

posts

WHERE

posts.id >= % (id\_1) s

LIMIT % (param\_1) s

>>>

>>>

### Метод offset()

Метод offset() добавляет условие OFFSET в запрос. В качестве аргумента он принимает смещение. Часто используется вместе с limit().

>>>

>>> db.session.query(Post).filter(Post.id > 1).limit(3).offset(1).all()

[<3:Post 3>, <4:Post 4>]

>>>

SQL-эквивалент:

>>>

>>> print(db.session.query(Post).filter(Post.id > 1).limit(3).offset(1))

SELECT

posts.id AS posts\_id,

posts.title AS posts\_title,

posts.slug AS posts\_slug,

posts.content AS posts\_content,

posts.created\_on AS posts\_created\_on,

posts.u pdated\_on AS posts\_updated\_on,

posts.category\_id AS posts\_category\_id

FROM

posts

WHERE

posts.id > % (id\_1) s

LIMIT % (param\_1) s, % (param\_2) s

>>>

Строки % (param\_1) s и % (param\_2) — заполнители для смещения и ограничения вывода, соответственно.

### Метод order\_by()

Метод order\_by() используется, чтобы упорядочить результат, добавив к запросу оператор ORDER BY. Он принимает количество колонок, для которых нужно установить порядок. По умолчанию сортирует в порядке возрастания.

>>>

>>> db.session.query(Tag).all()

[<1:refactoring>, <2:snippet>, <3:analytics>]

>>>

>>> db.session.query(Tag).order\_by(Tag.name).all()

[<3:analytics>, <1:refactoring>, <2:snippet>]

>>>

Для сортировки по убыванию нужно использовать функцию db.desc():

>>>

>>> db.session.query(Tag).order\_by(db.desc(Tag.name)).all()

[<2:snippet>, <1:refactoring>, <3:analytics>]

>>>

### Метод join()

Метод join() используется для создания JOIN в SQL. Он принимает имя таблицы, для которой нужно создать JOIN.

>>>

>>> db.session.query(Post).join(Category).all()

[<1:Post 1>, <2:Post 2>, <3:Post 3>]

>>>

SQL-эквивалент:

>>>

>>> print(db.session.query(Post).join(Category))

SELECT

posts.id AS posts\_id,

posts.title AS posts\_title,

posts.slug AS posts\_slug,

posts.content AS posts\_content,

posts.created\_on AS posts\_created\_on,

posts.u pdated\_on AS posts\_updated\_on,

posts.category\_id AS posts\_category\_id

FROM

posts

Метод join() широко используется, чтобы получить данные из одной или большего количества таблиц одним запросом. Например:

>>>

>>> db.session.query(Post.title, Category.name).join(Category).all()

[('Post 1', 'Python'), ('Post 2', 'Python'), ('Post 3', 'Java')]

>>>

Можно создать JOIN для большее чем двух таблиц с помощью цепочки методов join():

db.session.query(Table1).join(Table2).join(Table3).join(Table4).all()

Закончить урок можно завершением контактной формы.

Стоит напомнить, что в уроке «[**Работа с формами во Flask**](https://pythonru.com/uroki/11-rabota-s-formami-vo-flask)» была создана контактная форма для получения обратной связи от пользователей. Пока что функция представления contact() не сохраняет отправленные данные. Она только выводит их в консоли. Для сохранения полученной информации сначала нужно создать новую таблицу. Откроем main2.py, чтобы добавить модель Feedback следом за моделью Tag:

#...

class Feedback(db.Model):

\_\_tablename\_\_ = 'feedbacks'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(1000), nullable=False)

email = db.Column(db.String(100), nullable=False)

message = db.Column(db.Text(), nullable=False)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

def \_\_repr\_\_(self):

return "<{}:{}>".format(self.id, self.name)

#...

Дальше нужно перезапустить оболочку Python и вызвать метод create\_all() объекта db для создания таблицы feedbacks:

(env) gvido@vm:~/flask\_app$ python main2.py shell

>>>

>>> from main2 import db

>>>

>>> db.create\_all()

>>>

Также нужно изменить функция представления contact():

#...

@app.route('/contact/', methods=['get', 'post'])

def contact():

form = ContactForm()

if form.validate\_on\_submit():

name = form.name.data

email = form.email.data

message = form.message.data

print(name)

print(Post)

print(email)

print(message)

# здесь логика базы данных

feedback = Feedback(name=name, email=email, message=message)

db.session.add(feedback)

db.session.commit()

print("\nData received. Now redirecting ...")

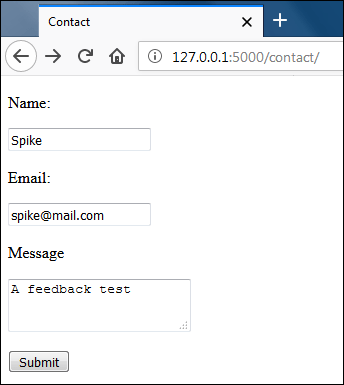
flash("Message Received", "success")

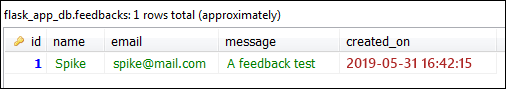
return redirect(url\_for('contact'))

return render\_template('contact.html', form=form)

#...

Запустим сервер и зайдем на https://127.0.0.1:5000/contact/, чтобы заполнить и отправить форму.



Отправленная запись в HeidiSQL будет выглядеть следующим образом:  


# #16 Миграции базы данных с помощью Alembic

Alembic — это инструмент для миграции базы данных, используемый в [**SQLAlchemy**](https://pythonru.com/uroki/15-osnovy-orm-sqlalchemy). Миграция базы данных — это что-то похожее на систему контроля версий для баз данных. Стоит напомнить, что метод create\_all() в SQLAlchemy лишь создает недостающие таблицы из моделей. Когда таблица уже создана, он не меняет ее схему, основываясь на изменениях в модели.

При разработке приложения распространена практика изменения схемы таблицы. Здесь и приходит на помощью Alembic. Он, как и другие подобные инструменты, позволяет менять схему базы данных при развитии приложения. Он также следит за изменениями самой базы, так что можно двигаться туда и обратно. Если не использовать Alembic, то за всеми изменениями придется следить вручную и менять схему с помощью Alter.

Flask-Migrate — это расширение, которое интегрирует Alembic в приложение Flask. Установить его можно с помощью следующей команды.

(env) gvido@vm:~/flask\_app$ pip install flask-migrate

Для интеграции Flask-Migrate с приложением нужно импортировать классы Migrate и MigrateCommand из пакета flask\_package, а также создать экземпляр класса Migrate, передав экземпляр приложения (app) и объект SQLAlchemy (db):

#...

from flask\_migrate import Migrate, MigrateCommand

app = Flask(\_\_name\_\_)

app.debug = True

app.config['SECRET\_KEY'] = 'a really really really really long secret key'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql+pymysql://root:pass@localhost/flask\_app\_db'

manager = Manager(app)

db = SQLAlchemy(app)

migrate = Migrate(app, db)

manager.add\_command('db', MigrateCommand)

#...

Класс MigrateCommand определяет некоторые команды миграции [**базы данных**](https://pythonru.com/uroki/14-sozdanie-baz-dannyh-vo-flask), доступные во Flask-Script. На 12 строке эти команды выводятся с помощью аргумента командной строки db. Чтобы посмотреть созданные команды, нужно вернуться обратно в терминал и ввести следующую команду:

(env) gvido@vm:~/flask\_app$ python main2.py

positional arguments:

{db,faker,foo,shell,runserver}

db Perform database migrations

faker A command to add fake data to the tables

foo Just a simple command

shell Runs a Python shell inside Flask application context.

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

optional arguments:

-?, --help show this help message and exit

(env) gvido@vm:~/flask\_app$

Так, можно видеть, что новая команда db используется для миграций базы данных. Чтобы посмотреть полный список подкоманд для dv, нужно ввести следующее:

(env) gvido@vm:~/flask\_app$ python main2.py db -?

Perform database migrations

positional arguments:

{init,revision,migrate,edit,merge,upgrade,downgrade,show,history,heads,branche

s,current,stamp}

init Creates a new migration repository

revision Create a new revision file.

migrate Alias for 'revision --autogenerate'

edit Edit current revision.

merge Merge two revisions together. Creates a new migration

file

upgrade Upgrade to a later version

downgrade Revert to a previous version

show Show the revision denoted by the given symbol.

history List changeset scripts in chronological order.

heads Show current available heads in the script directory

branches Show current branch points

current Display the current revision for each database.

stamp 'stamp' the revision table with the given revision;

don't run any migrations

optional arguments:

-?, --help show this help message and exit

Это реальные команды, которые будут использоваться для миграций базы данных.

Перед тем как Alembic начнет отслеживать изменения, нужно установить репозиторий миграции. Репозиторий миграции — это всего лишь папка, которая содержит настройки Alembic и скрипты миграции. Для создания репозитория нужно исполнить команду init:

(env) gvido@vm:~/flask\_app$ python main2.py db init

Creating directory /home/gvido/flask\_app/migrations ... done

Creating directory /home/gvido/flask\_app/migrations/versions ... done

Generating /home/gvido/flask\_app/migrations/README ... done

Generating /home/gvido/flask\_app/migrations/env.py ... done

Generating /home/gvido/flask\_app/migrations/alembic.ini ... done

Generating /home/gvido/flask\_app/migrations/script.py.mako ... done

Please edit configuration/connection/logging settings in

'/home/gvido/flask\_app/migrations/alembic.ini' before proceeding.

(env) gvido@vm:~/flask\_app$

Эта команда создаст папку “migrations” внутри папки flask\_app. Структура папки migrations следующая:

migrations

├── alembic.ini

├── env.py

├── README

├── script.py.mako

└── versions

Краткое описание каждой папки и файла:

* alembic.ini — файл с настройки Alembic.
* env.py — файл Python, который запускается каждый раз, когда вызывается Alembic. Он соединяется с базой данных, запускает транзакцию и вызывает движок миграции.
* README — файл README.
* script.py.mako — файл шаблона Mako, который будет использоваться для создания скриптов миграции.
* version — папка для хранения скриптов миграции.

## Создание скрипта миграции

Alembic хранит миграции базы данных в скриптах миграции, которые представляют собой обычные [**файлы Python**](https://pythonru.com/osnovy/fajly-v-python-vvod-vyvod). Скрипт миграции определяет две функции: upgrade() и downgrade(). Задача upgrade() — применить изменения к базе данных, а downgrade() — откатить их обратно. Когда миграция применяется, вызывается функция upgrade(). При возврате обратно — downgrade().

Alembic предлагает два варианта создания миграций:

1. Вручную с помощью команды revision.
2. Автоматически с помощью команды migrate.

## Ручная миграция

Ручная или пустая миграция создает скрипт миграции с пустыми функциями upgrade() и downgrade(). Задача — заполнить их с помощью инструкций Alembic, которые и будет применять изменения к базе данных. Ручная миграция используется тогда, когда нужен полный контроль над процессом миграции. Для создания пустой миграции нужно ввести следующую команду:

(env) gvido@vm:~/flask\_app$ python main2.py db revision -m "Initial migration"

Эта команда создаст новый скрипт миграции в папке migrations/version. Название файла должно быть в формате someid\_initial\_migration.py. Файл должен выглядеть вот так:

"""Initial migration

Revision ID: 945fc7313080

Revises:

Create Date: 2019-06-03 14:39:27.854291

"""

from alembic import op

import sqlalchemy as sa

# идентификаторы изменений, используемые Alembic.

revision = '945fc7313080'

down\_revision = None

branch\_labels = None

depends\_on = None

def upgrade():

pass

def downgrade():

pass

Он начинается с закомментированной части, которая содержит сообщение, заданное с помощью метки -m, ID изменения и времени, когда файл был создан. Следующая важная часть — идентификаторы изменения. Каждый скрипт миграции получает уникальный ID изменения, который хранится в переменной revision. На следующей строке есть переменная down\_revision со значением None. Alembic использует переменную down\_revision, чтобы определить, какую миграцию запускать и в каком порядке. Переменная down\_revision указывает на идентификатор изменения родительской миграции. В этом случае его значение — None, потому что это только первый скрипт миграции. В конце файла есть пустые функции upgrade() и downgrade().

Теперь нужно отредактировать файл миграции, чтобы добавить операции создания и удаления таблицы для функций upgrade() и downgrade(), соответственно.

В функции upgrade() используется инструкция create\_table() Alembic. Инструкция create\_table() использует оператор CREATE TABLE.

В функции downgrade() инструкция drop\_table() задействует оператор DROP TABLE.

При первом запуске миграции будет создана таблица users, а при откате — эта же миграция удалит таблицу users.

Теперь можно выполнить первую миграцию. Для этого нужно ввести следующую команду:

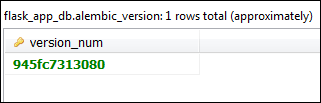
(env) gvido@vm:~/flask\_app$ python main2.py db upgrade

]  
Эта команда исполнит функцию upgrade() скрипта миграции. Команда db upgrade вернет базу данных к последней миграции. Стоит заметить, что db upgrade не только запускает последнюю миграции, но все, которые еще не были запущены. Это значит, что если миграций было создано несколько, то db upgrade запустит их все вместе в порядке создания.

Вместо запуска последней миграции можно также передать идентификатор изменения нужной миграции. В таком случае db upgrade остановится после запуска конкретной миграции и не будет выполнять последующие.

(env) gvido@vm:~/flask\_app$ python main2.py db upgrade 945fc7313080

Поскольку миграция запускается первый раз, Alembic также создаст таблицу alembic\_version. Она состоит из одной колонки version\_num, которая хранит идентификатор изменения последней запущенной миграции. Именно так Alembic знает текущее состояние миграции, и откуда ее нужно исполнять. Сейчас таблица alembic\_version выглядит вот так:



Определить последнюю примененную миграцию можно с помощью команды db current. Она вернет идентификатор изменения последней миграции. Если таковой не было, то ничего не вернется.

(env) gvido@vm:~/flask\_app$ python main2.py db current

INFO [alembic.runtime.migration] Context impl MySQLImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

945fc7313080 (head)

(env) gvido@vm:~/flask\_app$

Вывод показывает, что текущая миграция — 945fc7313080. Также нужно обратить внимание на строку (head) после идентификатора изменения, которая указывает на то, что 945fc7313080 — последняя миграция.

Создадим еще одну пустую миграцию с помощью команды db revision:

(env) gvido@vm:~/flask\_app$ python main2.py db revision -m "Second migration"

Дальше нужно снова запустить команду db current. В этот раз идентификатор изменения будет отображаться без строки (head), потому что миграция 945fc7313080 — не последняя.

(env) gvido@vm:~/flask\_app$ python main2.py db current

INFO [alembic.runtime.migration] Context impl MySQLImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

945fc7313080

(env) gvido@vm:~/flask\_app$

Чтобы посмотреть полный список миграций (запущенных и нет), нужно использовать команду db history. Она вернет список миграций в обратном хронологическом порядке (последняя миграция будет отображаться первой).

(env) gvido@vm:~/flask\_app$ python main2.py db history

945fc7313080 -> b0c1f3d3617c (head), Second migration

<base> -> 945fc7313080, Initial migration

(env) gvido@vm:~/flask\_app$

Вывод показывает, что 945fc7313080 — первая миграция, а следом за ней идет b0c1f3d3617 — последняя миграция. Как и обычно, (head) указывает на последнюю миграцию.

Таблица users был создана исключительно в целях тестирования. Вернуть базу данных к исходному состоянию, которое было до исполнения команды db upgrade, можно с помощью отката миграции. Чтобы откатиться к последней миграции, используется команда db downgrade.

(env) gvido@vm:~/flask\_app$ python main2.py db downgrade

INFO [alembic.runtime.migration] Context impl MySQLImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

INFO [alembic.runtime.migration] Running downgrade 945fc7313080 -> , Initial mi

gration

(env) gvido@vm:~/flask\_app$

Она выполнит метод downgrade() миграции 945fc7313080, которая удалит таблицу users из базы данных. Как и в случае с командой db upgrade, можно передать идентификатор изменения миграции, к которому нужно откатиться. Например, чтобы откатиться к миграции 645fc5113912, нужно использовать следующую команду.

(env) gvido@vm:~/flask\_app$ python main2.py db downgrade 645fc5113912

Чтобы вернуть все принятые миграции, нужно использовать следующую команду:

(env) gvido@vm:~/flask\_app$ python main2.py db downgrade base

Сейчас к базе данных не применено ни единой миграции. Убедиться в этом можно, запустив команду db current:

(env) gvido@vm:~/flask\_app$ python main2.py db current

INFO [alembic.runtime.migration] Context impl MySQLImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

(env) gvido@vm:~/flask\_app$

Как видно, вывод не возвращает идентификатор изменения. Стоит обратить внимание, что откат миграции лишь отменяет изменения базы данных, но не удаляет сам скрипт миграции. В результате команда db history покажет два скрипта миграции.

(env) gvido@vm:~/flask\_app$ python main2.py db history

945fc7313080 -> b0c1f3d3617c (head), Second migration

<base> -> 945fc7313080, Initial migration

(env) gvido@vm:~/flask\_app$

Что будет, если сейчас запустить команду db upgrade?

Команда db upgrade в первую очередь запустит миграцию 945fc7313080, а следом за ней — b0c1f3d3617.

База данных снова в изначальном состоянии, а поскольку изменения в скриптах миграции не требуются, их можно удалить.

## Автоматическая миграция

**Примечание:** перед тем как двигаться дальше, нужно убедиться, что миграции из прошлого раздела удалены.

Автоматическая миграция создает код для функций upgrade() и downgrade() после сравнения моделей с текущей версией базы данных. Для создания автоматической миграции используется команда migrate, которая по сути повторяет то, что делает revision --autogenerate. В терминале нужно ввести команду migrate:

Важно обратить внимание, что на последней строчке вывода написано ”No changes in schema detected.”. Это значит, что модели синхронизированы с базой данных.

Откроем main2.py, чтобы добавить модель Employee после модели Feedback:

#...

class Employee(db.Model):

\_\_tablename\_\_ = 'employees'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(255), nullable=False)

designation = db.Column(db.String(255), nullable=False)

doj = db.Column(db.Date(), nullable=False)

#...

Дальше нужно снова запустить команду db migrate. В этот раз Alembic определит, что была добавлена новая таблица employees и создаст скрипт миграции с функциями для последующего создания и удаления таблицы employees.

(env) gvido@vm:~/flask\_app$ python main2.py db migrate -m "Adding employees table"

Скрипт миграции, созданный с помощью предыдущей команды, должен выглядеть вот так:

"""Adding employees table

Revision ID: 6e059688f04e

Revises:

Create Date: 2019-06-03 16:01:28.030320

"""

from alembic import op

import sqlalchemy as sa

# идентификаторы изменений, используемые Alembic.

revision = '6e059688f04e'

down\_revision = None

branch\_labels = None

depends\_on = None

def upgrade():

# ### автоматически генерируемые команды Alembic - пожалуйста, настройте! ###

op.create\_table('employees',

sa.Column('id', sa.Integer(), nullable=False),

sa.Column('name', sa.String(length=255), nullable=False),

sa.Column('designation', sa.String(length=255), nullable=False),

sa.Column('doj', sa.Date(), nullable=False),

sa.PrimaryKeyConstraint('id')

)

# ### конец команд Alembic ###

def downgrade():

# ### автоматически генерируемые команды Alembic - пожалуйста, настройте! ###

op.drop\_table('employees')

# ### конец команд Alembic ###

Ничего нового здесь нет. Функция upgrade() использует инструкцию create\_table для создания таблицы, а функция downgrade() — инструкцию drop\_table для ее удаления.

Запустим миграцию с помощью команды db upgrade:

(env) gvido@vm:~/flask\_app$ python main2.py db upgrade

INFO [alembic.runtime.migration] Context impl MySQLImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

INFO [alembic.runtime.migration] Running upgrade -> 6e059688f04e, Adding emplo

yees table

(env) gvido@vm:~/flask\_app$

Это добавит таблицу employees в базу данных.

## Ограничения автоматической миграции

Автоматическая миграция не идеальна. Она не определяет все возможные изменения.

Операции, которые Alembic умеет выявлять:

* Добавление и удаление таблиц
* Добавление и удаление колонок
* Изменения во внешних ключах
* Изменения в типах колонок
* Изменения в индексах и использованных уникальных ограничениях

Изменения, которые Alembic не определяет:

* Имени таблицы
* Имени колонки
* Ограничения с отдельно указанными именами

Для создания скриптов миграции для операций, которые Alembic не умеет выявлять, нужно создать пустой скрипт миграции и заполнить функции upgrade() и downgrade() соответствующим образом.

# #17 Отправка email во Flask

Веб-приложения отправляют электронные письма постоянно, и в этом уроке речь пойдет о том, как добавить инструмент для отправки email в приложение Flask.

В стандартной библиотеке Python есть модуль smtplib, который можно использовать для отправки сообщений. Хотя сам модуль smtplib не является сложным, он все равно требует кое-какой работы. Для облегчения процесса работы с ним было создано расширение Flask-Mail. Flask-Mail построен на основе модуля Python smtplib и предоставляет простой интерфейс для отправки электронных писем. Он также предоставляет возможности по массовой рассылке и прикрепленным к сообщениям файлам. Установить Flask-Mail можно с помощью следующей команды:

(env) gvido@vm:~/flask\_app$ pip install flask-mail

Чтобы запустить расширение, нужно импортировать класс Mail из пакета flask\_mail и создать экземпляр класса Mail:

#...

from flask\_mail import Mail, Message

app = Flask(\_\_name\_\_)

app.debug = True

app.config['SECRET\_KEY'] = 'a really really really really long secret key'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql+pymysql://root:pass@localhost/flask\_app\_db'

manager = Manager(app)

manager.add\_command('db', MigrateCommand)

db = SQLAlchemy(app)

migrate = Migrate(app, db)

mail = Mail(app)

#...

Дальше нужно указать некоторые параметры настройки, чтобы Flask-Mail знал, к какому SMTP-серверу подключаться. Для этого в файл main2.py нужно добавить следующий код:

#...

app.config['SECRET\_KEY'] = 'a really really really really long secret key'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql+pymysql://root:pass@localhost/flask\_app\_db'

app.config['MAIL\_SERVER'] = 'smtp.googlemail.com'

app.config['MAIL\_PORT'] = 587

app.config['MAIL\_USE\_TLS'] = True

app.config['MAIL\_USERNAME'] = 'test@gmail.com' # введите свой адрес электронной почты здесь

app.config['MAIL\_DEFAULT\_SENDER'] = 'test@gmail.com' # и здесь

app.config['MAIL\_PASSWORD'] = 'password' # введите пароль

manager = Manager(app)

manager.add\_command('db', MigrateCommand)

db = SQLAlchemy(app)

mail = Mail(app)

#...

В данном случае используется SMTP-сервер Google. Стоит отметить, что Gmail позволяет отправлять только 100-150 сообщений в день. Если этого недостаточно, то стоит обратить внимание на альтернативы: SendGrid или MailChimp.

Вместо того чтобы напрямую указывать email и пароль в приложении, как это было сделано ранее, лучше хранить их в переменных среды. В таком случае, если почта или пароль поменяются, не будет необходимости обновлять код. О том, как это сделать, будет рассказано в следующих уроках.

## Основы Flask-Mail

Для составления электронного письма, нужно создать экземпляр класса Message:

msg = Message("Subject", sender="sender@example.com", recipients=['recipient\_1@example.com'])

Если при настройке параметров конфигурации MAIL\_DEFAULT\_SENDER был указан, то при создании экземпляра Message передавать значение sender не обязательно.

msg = Message("Subject", recipients=['recipient@example.com'])

Для указания тела письма необходимо использовать атрибут body экземпляра Message:

msg.body = "Mail body"

Если оно состоит из HTML, передавать его следует атрибуту html.

msg.html = "<p>Mail body</p>"

Наконец, отправить сообщение можно, передав экземпляр Message метод send() экземпляра Mail:

mail.send(msg)

Пришло время проверить настройки, отправив email с помощью командной строки.

## Отправка тестового сообщения

Откроем терминал, чтобы ввести следующие команды:

(env) overiq@vm:~/flask\_app$ python main2.py shell

>>>

>>> from main2 import mail, Message

>>> # введите свою почту

>>> msg = Message("Subject", recipients=["you@mail.com"])

>>> msg.html = "<h2>Email Heading</h2>\n<p>Email Body</p>"

>>>

>>> mail.send(msg)

>>>

Если операция прошла успешно, то на почту должно прийти следующее сообщение с темой “Subject”:

Email Heading

Email Body

Стоит заметить, что отправка через SMTP-сервер Gmail не сработает, если не отключить двухфакторную аутентификацию и не разрешить небезопасным приложениям получать доступ к аккаунту.

## Интеграция email в приложение

Сейчас когда пользователь отправляет обратную связь, она сохраняется в базу данных, сам пользователь получает уведомление о том, что его сообщение было отправлено, и на этом все. Но в идеале приложение должно уведомлять администраторов о полученной обратной связи. Это можно сделать. Откроем main2.py, чтобы изменить функцию представления contact() так, чтобы она отправляла сообщения:

#...

@app.route('/contact/', methods=['get', 'post'])

def contact():

#...

db.session.commit()

msg = Message("Feedback", recipients=[app.config['MAIL\_USERNAME']])

msg.body = "You have received a new feedback from {} <{}>.".format(name, email)

mail.send(msg)

print("\nData received. Now redirecting ...")

#...

Дальше нужно запустить сервер и зайти на https://localhost:5000/contact/. Заполним и отправим [**форму**](https://pythonru.com/uroki/11-rabota-s-formami-vo-flask). Если все прошло успешно, должен прийти email.

Можно было обратить внимание на задержку между отправкой обратной связи и появлением уведомления о том, что она была отправлена успешно. Проблема в том, что метод mail.send() блокирует исполнение функции представления на несколько секунд. В результате, код с перенаправлением страницы не будет исполнен до тех пор, пока не вернется метод mail.send(). Решить это можно с помощью потоков (threads).

Также прямо сейчас можно слегка изменить код отправки сообщений. На данный момент если email потребуется отправить в любом другом месте кода, нужно будет копировать и вставлять те самые строки. Но можно сохранить несколько строк, заключив логику отправки сообщений в функцию.

Откроем main2.py, чтобы добавить следующий код перед index:

#...

from threading import Thread

#...

def shell\_context():

import os, sys

return dict(app=app, os=os, sys=sys)

manager.add\_command("shell", Shell(make\_context=shell\_context))

def async\_send\_mail(app, msg):

with app.app\_context():

mail.send(msg)

def send\_mail(subject, recipient, template, \*\*kwargs):

msg = Message(subject, sender=app.config['MAIL\_DEFAULT\_SENDER'], recipients=[recipient])

msg.html = render\_template(template, \*\*kwargs)

thr = Thread(target=async\_send\_mail, args=[app, msg])

thr.start()

return thr

@app.route('/')

def index():

return render\_template('index.html', name='Jerry')

#...

Было сделано несколько изменений. Функция send\_mail() теперь включает в себя всю логику отправки email. Она принимает тему письма, получателя и шаблон сообщения. Ей также можно передать дополнительные аргументы в виде аргументов-ключевых слов. Почему именно так? Дополнительные аргументы представляют собой данные, которые нужно передать шаблону. На 17 строке рендерится шаблон, а его результат передается атрибуту msg.html. На строке 18 создается объект Thread. Это делается с помощью передачи названия функции и аргументов функции, с которыми она должна быть вызвана. Следующая строка запускает потоки. Когда поток запускается, вызывается async\_send\_mail(). Теперь самое интересное. Впервые в коде работа происходит вне приложения (то есть, вне функции представления) в новом потоке. with app.app\_context(): создает контекст приложения, а mail.send() отправляет email.

Дальше нужно создать шаблон для сообщения обратной связи. В папке templates необходимо создать папку mail. Она будет хранить шаблоны для электронных писем. Внутри папки необходимо создать шаблон feedback.html со следующим кодом:

<p>You have received a new feedback from {{ name }} &lt;{{ email }}&gt; </p>

Теперь нужно изменить функцию представления contact(), чтобы использовать функцию send\_mail():

После этого нужно снова зайти на https://localhost:5000/contact, заполнить форму и отправить ее. В этот раз задержки не будет.

# #18 Аутентификация во Flask

Аутентификация — один из самых важных элементов веб-приложений. Этот процесс предотвращает попадание неавторизованных пользователей на непредназначенные для них страницы. Собственную систему аутентификации можно создать с помощью куки и хэширования паролей. Такой миниатюрный проект станет отличной проверкой полученных навыков.

Как можно было догадаться, уже существует расширение, которое может значительно облегчить жизнь. Flask-Login — это расширение, позволяющее легко интегрировать систему аутентификации в приложение Flask. Установить его можно с помощью следующей команды:

(env) gvido@vm:~/flask\_app$ pip install flask-login

## Создание модели пользователя

Сейчас информация о пользователях, которые являются администраторами или редакторами сайта, нигде не хранится. Первая задача — создать модель User для хранения пользовательских данных. Откроем main2.py, чтобы добавить модель User после модели Employee:

#..

class User(db.Model):

\_\_tablename\_\_ = 'users'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(100))

username = db.Column(db.String(50), nullable=False, unique=True)

email = db.Column(db.String(100), nullable=False, unique=True)

password\_hash = db.Column(db.String(100), nullable=False)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

updated\_on = db.Column(db.DateTime(), default=datetime.utcnow, onupdate=datetime.utcnow)

def \_\_repr\_\_(self):

return "<{}:{}>".format(self.id, self.username)

#...

Для обновления базы данных нужно создать новую миграцию. В терминале для создания нового скрипта миграции необходимо ввести следующую команду:

(env) gvido@vm:~/flask\_app$ python main2.py db migrate -m "Adding users table"

Запустить миграцию необходимо с помощью команды upgrade:

(env) gvido@vm:~/flask\_app$ python main2.py db upgrade

INFO [alembic.runtime.migration] Context impl MySQLImpl.

INFO [alembic.runtime.migration] Will assume non-transactional DDL.

INFO [alembic.runtime.migration] Running upgrade 6e059688f04e -> 0f0002bf91cc,

Adding users table

(env) gvido@vm:~/flask\_app$

Это создаст таблицу users в базе данных.

## Хэширование паролей

Пароли никогда не должны храниться в виде чистого текста в [**базе данных**](https://pythonru.com/uroki/14-sozdanie-baz-dannyh-vo-flask). Если так делать, злоумышленник, способный взломать базу данных, получит возможность узнать и пароли, и электронные адреса. Известно, что люди используют один и тот же пароль для разных сайтов, а это значит, что одна комбинация откроет злоумышленнику доступ к остальным аккаунтам пользователей.

Вместо хранения паролей прямо в базе данных, нужно сохранять их хэши. Хэш — это строка символов, которые смотрятся так, будто бы были подобраны случайно.

pbkdf2:sha256:50000$Otfe3YgZ$4fc9f1d2de2b6beb0b888278f21a8c0777e8ff980016e043f3eacea9f48f6dea

Хэш создается с помощью односторонней функции хэширования. Она принимает длину переменной и возвращает вывод фиксированной длины, которую мы и называем хэшем. Безопасным хэш делает тот факт, что его нельзя использовать для получения изначальной строки (поэтому функция и называется односторонней). Тем не менее для одного ввода односторонняя функция хэширования будет возвращать один и тот же результат.

Вот процессы, которые задействованы при создании хэша пароля:

Когда пользователь передает пароль (на этапе регистрации), необходимо его хэшировать и сохранить хэш в базу данных. Когда пользователь будет снова авторизоваться, функция повторно создаст хэш и сравнит его с тем, что хранится в базе данных. Если они совпадают, пользователь получит доступ к аккаунту. В противном случае, возникнет ошибка.

Flask поставляется с пакетом Werkzeug, в котором есть две вспомогательные функции для хэширования паролей.

| **Метод** | **Описание** |
| --- | --- |
| generate\_password\_hash(password) | Принимает пароль и возвращает хэш. По умолчанию использует одностороннюю функцию pbkdf2 для создания хэша. |
| check\_password\_hash(password\_hash, password) | Принимает хэш и пароль в чистом виде, затем сравнивает password и password\_hash. Если они одинаковые, возвращает True. |

Следующий код демонстрирует, как работать с этими функциями:

>>>

>>> from werkzeug.security import generate\_password\_hash, check\_password\_hash

>>>

>>> hash = generate\_password\_hash("secret password")

>>>

>>> hash

'pbkdf2:sha256:50000$zB51O5L3$8a43788bc902bca96e01a1eea95a650d9d5320753a2fbd16bea984215cdf97ee'

>>>

>>> check\_password\_hash(hash, "secret password")

True

>>>

>>> check\_password\_hash(hash, "pass")

False

>>>

>>>

Стоит обратить внимание, что когда check\_password\_hash() вызывается с правильными паролем (“secret password”), возвращается True, а если с неправильными — False.

Дальше нужно обновить модель User, и добавить в нее хэширование паролей:

#...

from werkzeug.security import generate\_password\_hash, check\_password\_hash

#...

#...

class User(db.Model):

#...

updated\_on = db.Column(db.DateTime(), default=datetime.utcnow, onupdate=datetime.utcnow)

def \_\_repr\_\_(self):

return "<{}:{}>".format(self.id, self.username)

def set\_password(self, password):

self.password\_hash = generate\_password\_hash(password)

def check\_password(self, password):

return check\_password\_hash(self.password\_hash, password)

#...

Создадим пользователей, чтобы проверить хэширование паролей.

(env) gvido@vm:~/flask\_app$ python main2.py shell

>>>

>>> from main2 import db, User

>>>

>>> u1 = User(username='spike', email='spike@example.com')

>>> u1.set\_password("spike")

>>>

>>> u2 = User(username='tyke', email='tyke@example.com')

>>> u2.set\_password("tyke")

>>>

>>> db.session.add\_all([u1, u2])

>>> db.session.commit()

>>>

>>> u1, u2

(<1:spike>, <2:tyke>)

>>>

>>>

>>> u1.check\_password("pass")

False

>>> u1.check\_password("spike")

True

>>>

>>> u2.check\_password("foo")

False

>>> u2.check\_password("tyke")

True

>>>

>>>

Вывод демонстрирует, что все работает как нужно, и в базе данных теперь есть два пользователя.

## Интеграция Flask-Login

Для запуска Flask-Login нужно импортировать класс LoginManager из пакета flask\_login и создать новый экземпляр LoginManager:

#...

from werkzeug.security import generate\_password\_hash, check\_password\_hash

from flask\_login import LoginManager

app = Flask(\_\_name\_\_)

app.debug = True

app.config['SECRET\_KEY'] = 'a really really really really long secret key'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql+pymysql://root:pass@localhost/flask\_app\_db'

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

app.config['MAIL\_SERVER'] = 'smtp.googlemail.com'

app.config['MAIL\_PORT'] = 587

app.config['MAIL\_USE\_TLS'] = True

app.config['MAIL\_USERNAME'] = 'youmail@gmail.com'

app.config['MAIL\_DEFAULT\_SENDER'] = 'youmail@gmail.com'

app.config['MAIL\_PASSWORD'] = 'password'

manager = Manager(app)

manager.add\_command('db', MigrateCommand)

db = SQLAlchemy(app)

migrate = Migrate(app, db)

mail = Mail(app)

login\_manager = LoginManager(app)

#...

Для проверки пользователей Flask-Login требует добавления нескольких методов в класс User. Эти методы перечислены в следующей таблице:

| **Метод** | **Описание** |
| --- | --- |
| is\_authenticated() | Возвращает True, если пользователь проверен (то есть, зашел с корректным паролем). В противном случае — False. |
| is\_active() | Возвращает True, если действие аккаунта не приостановлено. |
| is\_anonymous() | Возвращает True для неавторизованных пользователей. |
| get\_id() | Возвращает уникальный идентификатор объекта User. |

Flask-Login предлагает реализацию этих методов по умолчанию с помощью класса UserMixin. Так, вместо определения их вручную, можно настроить их наследование из класса UserMixin. Откроем main2.py, чтобы изменить заголовок модели User:

#...

from flask\_login import LoginManager, UserMixin

#...

class User(db.Model, UserMixin):

\_\_tablename\_\_ = 'users'

#...

Осталось только добавить обратный вызов user\_loader. Соответствующий метод можно добавить над моделью User.

#...

@login\_manager.user\_loader

def load\_user(user\_id):

return db.session.query(User).get(user\_id)

#...

Функция, принимающая в качестве аргумента декоратор user\_loader, будет вызываться с каждым запросом к серверу. Она загружает пользователя из идентификатора пользователя в куки сессии. Flask-Login делает загруженного пользователя доступным с помощью прокси current\_user. Для использования current\_user его нужно импортировать из пакета flask\_login. Он ведет себя как глобальная переменная и доступен как в функциях представления, так и в шаблонах. В любой момент времени current\_user ссылается либо на вошедшего в систему, либо на анонимного пользователя. Различать их можно с помощью атрибута is\_authenticated прокси current\_user. Для анонимных пользователей is\_authenticated вернет False. В противном случае — True.

## Ограничение доступа к просмотру

Пока что на сайте нет никакой административной панели. В этом уроке она будет представлена обычной страницей. Чтобы не допустить неавторизованных пользователей к защищенным страница у Flask-Login есть декоратор login\_required. Добавим следующий код в файле main2.py сразу за функцией представления updating\_session():

#...

from flask\_login import LoginManager, UserMixin, login\_required

#...

@app.route('/admin/')

@login\_required

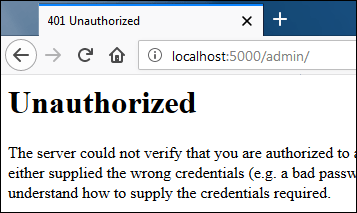
def admin():

return render\_template('admin.html')

#...

Декоратор login\_required гарантирует, что функция представления admin() вызовется только в том случае, если пользователь авторизован. По умолчанию, если анонимный пользователь попытается зайти на защищенную страницу, он получит ошибку 401 «Не авторизован».

Необходимо запустить сервер и зайти на https://localhost:5000/login, чтобы проверить, как это работает. Откроется такая страница:



Вместо того чтобы показывать пользователю ошибку 401, лучше перенаправить его на страницу авторизации. Чтобы сделать это, нужно передать атрибуту login\_view экземпляра LoginManager значение функции представления login():

#...

migrate = Migrate(app, db)

mail = Mail(app)

login\_manager = LoginManager(app)

login\_manager.login\_view = 'login'

class Faker(Command):

'A command to add fake data to the tables'

#...

Сейчас функция login() определена следующим образом (но ее нужно будет поменять):

#...

@app.route('/login/', methods=['post', 'get'])

def login():

message = ''

if request.method == 'POST':

print(request.form)

username = request.form.get('username')

password = request.form.get('password')

if username == 'root' and password == 'pass':

message = "Correct username and password"

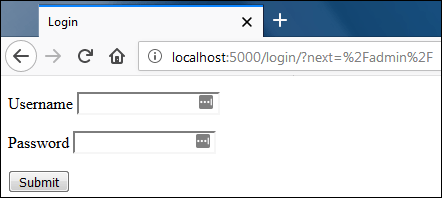
else:

message = "Wrong username or password"

return render\_template('login.html', message=message)

#...

Если теперь зайти на https://localhost:5000/admin/, произойдет перенаправление на страницу авторизации:



Flask-Login также настраивает всплывающее сообщение, когда пользователя перенаправляют на страницу авторизации, но сейчас никакого сообщения нет, потому что шаблон авторизации (template/login.html) не отображает никаких сообщений. Нужно открыть login.html и добавить следующий код перед тегом <form>:

#...

{% endif %}

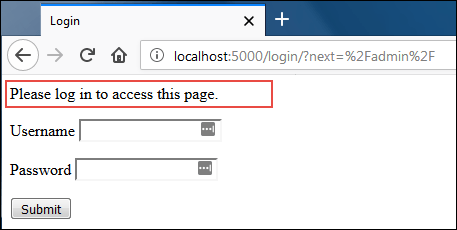
{% for category, message in get\_flashed\_messages(with\_categories=true) %}

<spam class="{{ category }}">{{ message }}</spam>

{% endfor %}

<form action="" method="post">

#...

Если снова зайти на https://localhost:5000/admin/, на странице отобразится сообщение.  


Чтобы изменить содержание сообщения, нужно передать новый текст атрибуту login\_message экземпляра LoginManager.

Заодно почему бы не создать шаблон для функции представления admin(). Создадим новый шаблон admin.html со следующим кодом:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<h2>Logged in User details</h2>

<ul>

<li>Username: {{ current\_user.username }}</li>

<li>Email: {{ current\_user.email }}</li>

<li>Created on: {{ current\_user.created\_on }}</li>

<li>Updated on: {{ current\_user.updated\_on }}</li>

</ul>

</body>

</html>

Здесь используется переменная current\_user для отображения подробностей о авторизованном пользователе.

## Создание формы авторизации

Перед авторизацией нужно создать форму. В ней будет три поля: имя пользователя, пароль и запомнить меня. Откроем forms.py, чтобы добавить класс LoginForm под классом ContactForm:

#...

from wtforms import StringField, SubmitField, TextAreaField, BooleanField, PasswordField

#...

#...

class LoginForm(FlaskForm):

username = StringField("Username", validators=[DataRequired()])

password = PasswordField("Password", validators=[DataRequired()])

remember = BooleanField("Remember Me")

submit = SubmitField()

## Авторизация пользователей

Для авторизации пользователя Flask-Login предоставляет функцию login\_user(). Она принимает объект пользователя. В случае успеха возвращает True и устанавливает сессию. В противном случае — False. По умолчанию сессия, установленная login\_user(), заканчивается при закрытии браузера. Чтобы позволить пользователям оставаться авторизованными на дольше, нужно передать remember=True функции login\_user() при авторизации пользователя. Откроем main2.py, чтобы изменить функцию представления login():

#...

from forms import ContactForm, LoginForm

#...

from flask\_login import LoginManager, UserMixin, login\_required, login\_user, current\_user

#...

@app.route('/login/', methods=['post', 'get'])

def login():

form = LoginForm()

if form.validate\_on\_submit():

user = db.session.query(User).filter(User.username == form.username.data).first()

if user and user.check\_password(form.password.data):

login\_user(user, remember=form.remember.data)

return redirect(url\_for('admin'))

flash("Invalid username/password", 'error')

return redirect(url\_for('login'))

return render\_template('login.html', form=form)

#...

Дальше нужно обновить login.html, чтобы использовать класс LoginForm(). Нужно добавить в файл следующие изменения:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Login</title>

</head>

<body>

{% for category, message in get\_flashed\_messages(with\_categories=true) %}

<spam class="{{ category }}">{{ message }}</spam>

{% endfor %}

<form action="" method="post">

{{ form.csrf\_token }}

<p>

{{ form.username.label() }}

{{ form.username() }}

{% if form.username.errors %}

{% for error in form.username.errors %}

{{ error }}

{% endfor %}

{% endif %}

</p>

<p>

{{ form.password.label() }}

{{ form.password() }}

{% if form.password.errors %}

{% for error in form.password.errors %}

{{ error }}

{% endfor %}

{% endif %}

</p>

<p>

{{ form.remember.label() }}

{{ form.remember() }}

</p>

<p>

{{ form.submit() }}

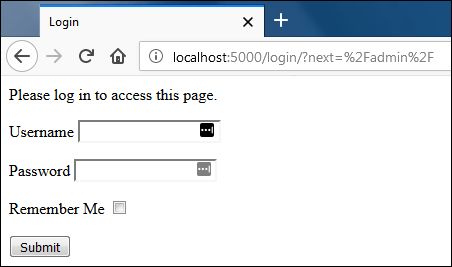
</p>

</form>

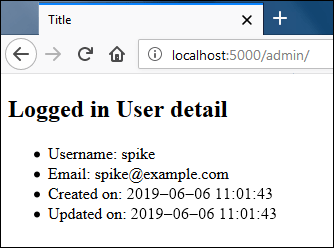
</body>

</html>

Теперь можно авторизоваться. Если зайти https://localhost:5000/admin, произойдет перенаправление на страницу авторизации.

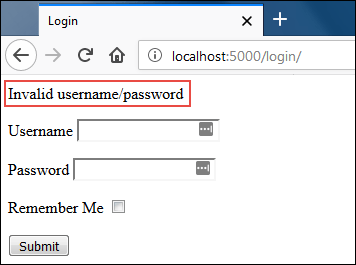


Необходимо ввести корректное имя пользователя и пароль и нажать sumbit. Произойдет перенаправление на страницу администратора, которая должна выглядеть следующим образом.



Если не кликнуть “Remember Me” при авторизации, после закрытия браузера сайт выйдет из аккаунта. Если кликнуть, то логин останется.

Если ввести неправильные имя пользователя или пароль, произойдет перенаправление на страницу авторизации со всплывающим сообщением:



## Завершение сеансов пользователей (выход из аккаунтов)

Функция logout\_user() во Flask-Login завершает сеанс пользователя, удаляя его идентификатор из сессии. В файле main2.py нужно добавить следующий код под функцией представления login():

#...

from flask\_login import LoginManager, UserMixin, login\_required, login\_user, current\_user, logout\_user

#...

@app.route('/logout/')

@login\_required

def logout():

logout\_user()

flash("You have been logged out.")

return redirect(url\_for('login'))

#...

Далее необходимо обновить шаблон admin.html, чтобы добавить ссылку на маршрут logout:

#...

<ul>

<li>Username: {{ current\_user.username }}</li>

<li>Email: {{ current\_user.email }}</li>

<li>Created on: {{ current\_user.created\_on }}</li>

<li>Updated on: {{ current\_user.updated\_on }}</li>

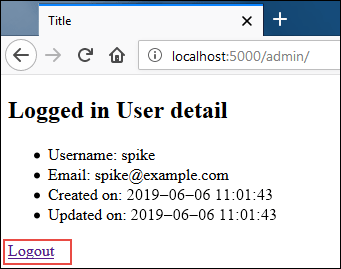
</ul>

<p><a href="{{ url\_for('logout') }}">Logout</a></p>

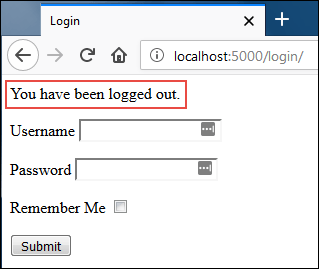
</body>

</html>

Если сейчас зайти на https://localhost:5000/admin/ (будучи авторизованным), то в нижней части страницы должны быть ссылка для выхода из аккаунта.



Если ее нажать, произойдет перенаправление на страницу авторизации.



## Финальные штрихи

Есть одна маленькая проблема со страницей авторизации. Сейчас если авторизованный пользователь зайдет на https://localhost:5000/login/, то он снова увидит страницу авторизации. Нет смысла в демонстрации формы авторизованному пользователю. Для разрешения этой проблемы нужно добавить следующие изменения в функцию представления login():

#...

@app.route('/login/', methods=['post', 'get'])

def login():

if current\_user.is\_authenticated:

return redirect(url\_for('admin'))

form = LoginForm()

if form.validate\_on\_submit():

#...

После этих изменений если авторизованный пользователь зайдет на страницу авторизации, он будет перенаправлен на страницу администратора.

# #19 Структура и эскиз приложения Flask

До этого момента все приложение хранилось в одном файле main2.py. Это нормально для маленьких программ, но когда масштабы растут, ими становится сложно управлять. Если разбить крупный файл на несколько, код в каждом из них становится читабельнее и предсказуемее.

У Flask нет никаких ограничений в плане структурирования приложений. Тем не менее существуют советы (гайдлайны) о том, как делать их модульными.

Мы будем использовать следующую структуру приложения.

/app\_dir

/app

\_\_init\_\_.py

/static

/templates

views.py

config.py

runner.py

Ниже описание каждого файла и папки:

| **Файл** | **Описание** |
| --- | --- |
| app\_dir | Корневая папка проекта |
| app | Пакет Python с файлами представления, шаблонами и статическими файлами |
| \_\_init\_\_.py | Этот файл сообщает Python, что папка app — пакет Python |
| static | Папка со статичными файлами проекта |
| templates | Папка с шаблонами |
| views.py | Маршруты и функции представления |
| config.py | Настройки приложения |
| runner.py | Точка входа приложения |

Оставшаяся часть урока будет посвящена преобразованию проекта к такой структуре. Начнем с создания config.py.

## Настройки на основе классов

Проект по созданию ПО обычно работает в трех разных средах:

1. Разработка
2. Тестирование
3. Рабочий режим

При развитии проекта понадобится определить разные параметры для разных сред. Некоторые также будут оставаться неизменными вне зависимости от среды. Внедрить такую систему конфигурации можно с помощью классов.

Начать стоит с определения настроек по умолчанию в базовом классе и только потом — создавать классы для отдельных сред, которые будут наследовать параметры из базового. Специальные классы могут перезаписывать или дополнять настройки, необходимые для конкретной среды.

Создадим файл config.py внутри папки flask\_app и добавим следующий код:

import os

app\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_))

class BaseConfig:

SECRET\_KEY = os.environ.get('SECRET\_KEY') or 'A SECRET KEY'

SQLALCHEMY\_TRACK\_MODIFICATIONS = False

##### настройка Flask-Mail #####

MAIL\_SERVER = 'smtp.googlemail.com'

MAIL\_PORT = 587

MAIL\_USE\_TLS = True

MAIL\_USERNAME = os.environ.get('MAIL\_USERNAME') or 'YOU\_MAIL@gmail.com'

MAIL\_PASSWORD = os.environ.get('MAIL\_PASSWORD') or 'password'

MAIL\_DEFAULT\_SENDER = MAIL\_USERNAME

class DevelopementConfig(BaseConfig):

DEBUG = True

SQLALCHEMY\_DATABASE\_URI = os.environ.get('DEVELOPMENT\_DATABASE\_URI') or \

'mysql+pymysql://root:pass@localhost/flask\_app\_db'

class TestingConfig(BaseConfig):

DEBUG = True

SQLALCHEMY\_DATABASE\_URI = os.environ.get('TESTING\_DATABASE\_URI') or \

'mysql+pymysql://root:pass@localhost/flask\_app\_db'

class ProductionConfig(BaseConfig):

DEBUG = False

SQLALCHEMY\_DATABASE\_URI = os.environ.get('PRODUCTION\_DATABASE\_URI') or \

'mysql+pymysql://root:pass@localhost/flask\_app\_db'

Стоит обратить внимание, что в этом коде значения некоторых настроек впервые берутся у переменных среды. Также здесь есть некоторые значения по умолчанию, если таковые для сред не будут указаны. Этот метод особенно полезен, когда имеется конфиденциальная информацию, и ее не хочется вписывать напрямую в приложение.

Считывать настройки из класса будет метод from\_object():

app.config.from\_object('config.Create')

### Создание пакета приложения

В папке flask\_app нужно создать новую папку под названием app и переместить все файлы и папки в нее (за исключением env и migrations, а также созданного только что файла config.py). Внутри папки app нужно создать файл \_\_init\_\_.py со следующим кодом:

from flask import Flask

from flask\_migrate import Migrate, MigrateCommand

from flask\_mail import Mail, Message

from flask\_sqlalchemy import SQLAlchemy

from flask\_script import Manager, Command, Shell

from flask\_login import LoginManager

import os, config

# создание экземпляра приложения

app = Flask(\_\_name\_\_)

app.config.from\_object(os.environ.get('FLASK\_ENV') or 'config.DevelopementConfig')

# инициализирует расширения

db = SQLAlchemy(app)

mail = Mail(app)

migrate = Migrate(app, db)

login\_manager = LoginManager(app)

login\_manager.login\_view = 'login'

# import views

from . import views

# from . import forum\_views

# from . import admin\_views

\_\_init\_\_.py создает экземпляр приложения и запускает расширения. Если переменная среды FLASK\_ENV не задана, приложение запустится в режиме отладки (то есть, app.debug = True). Чтобы перевести приложение в рабочий режим, нужно установить для переменной среды FLASK\_ENV значение config.ProductionConfig.

После запуска расширений инструкция import на 21 строке импортирует все функции представления. Это нужно, что подключить экземпляр приложение к этим функциям, иначе Flask не будет о них знать.

Необходимо переименовать файл main2.py на views.py и обновить его так, чтобы он содержал только маршруты и функции представления. Это полный код обновленного файла views.py.

from app import app

from flask import render\_template, request, redirect, url\_for, flash, make\_response, session

from flask\_login import login\_required, login\_user,current\_user, logout\_user

from .models import User, Post, Category, Feedback, db

from .forms import ContactForm, LoginForm

from .utils import send\_mail

@app.route('/')

def index():

return render\_template('index.html', name='Jerry')

@app.route('/user/<int:user\_id>/')

def user\_profile(user\_id):

return "Profile page of user #{}".format(user\_id)

@app.route('/books/<genre>/')

def books(genre):

return "All Books in {} category".format(genre)

@app.route('/login/', methods=['post', 'get'])

def login():

if current\_user.is\_authenticated:

return redirect(url\_for('admin'))

form = LoginForm()

if form.validate\_on\_submit():

user = db.session.query(User).filter(User.username == form.username.data).first()

if user and user.check\_password(form.password.data):

login\_user(user, remember=form.remember.data)

return redirect(url\_for('admin'))

flash("Invalid username/password", 'error')

return redirect(url\_for('login'))

return render\_template('login.html', form=form)

@app.route('/logout/')

@login\_required

def logout():

logout\_user()

flash("You have been logged out.")

return redirect(url\_for('login'))

@app.route('/contact/', methods=['get', 'post'])

def contact():

form = ContactForm()

if form.validate\_on\_submit():

name = form.name.data

email = form.email.data

message = form.message.data

# здесь логика БД

feedback = Feedback(name=name, email=email, message=message)

db.session.add(feedback)

db.session.commit()

send\_mail("New Feedback", app.config['MAIL\_DEFAULT\_SENDER'], 'mail/feedback.html',

name=name, email=email)

flash("Message Received", "success")

return redirect(url\_for('contact'))

return render\_template('contact.html', form=form)

@app.route('/cookie/')

def cookie():

if not request.cookies.get('foo'):

res = make\_response("Setting a cookie")

res.set\_cookie('foo', 'bar', max\_age=60\*60\*24\*365\*2)

else:

res = make\_response("Value of cookie foo is {}".format(request.cookies.get('foo')))

return res

@app.route('/delete-cookie/')

def delete\_cookie():

res = make\_response("Cookie Removed")

res.set\_cookie('foo', 'bar', max\_age=0)

return res

@app.route('/article', methods=['POST', 'GET'])

def article():

if request.method == 'POST':

res = make\_response("")

res.set\_cookie("font", request.form.get('font'), 60\*60\*24\*15)

res.headers['location'] = url\_for('article')

return res, 302

return render\_template('article.html')

@app.route('/visits-counter/')

def visits():

if 'visits' in session:

session['visits'] = session.get('visits') + 1

else:

session['visits'] = 1

return "Total visits: {}".format(session.get('visits'))

@app.route('/delete-visits/')

def delete\_visits():

session.pop('visits', None) # удаление посещений

return 'Visits deleted'

@app.route('/session/')

def updating\_session():

res = str(session.items())

cart\_item = {'pineapples': '10', 'apples': '20', 'mangoes': '30'}

if 'cart\_item' in session:

session['cart\_item']['pineapples'] = '100'

session.modified = True

else:

session['cart\_item'] = cart\_item

return res

@app.route('/admin/')

@login\_required

def admin():

return render\_template('admin.html')

views.py содержит не только функции представления. Сюда перемещен код моделей, классов форм и другие функции для соответствующих файлов:

models.py

from app import db, login\_manager

from datetime import datetime

from flask\_login import (LoginManager, UserMixin, login\_required,

login\_user, current\_user, logout\_user)

from werkzeug.security import generate\_password\_hash, check\_password\_hash

class Category(db.Model):

\_\_tablename\_\_ = 'categories'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(255), nullable=False, unique=True)

slug = db.Column(db.String(255), nullable=False, unique=True)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

posts = db.relationship('Post', backref='category', cascade='all,delete-orphan')

def \_\_repr\_\_(self):

return "<{}:{}>".format(self.id, self.name)

post\_tags = db.Table('post\_tags',

db.Column('post\_id', db.Integer, db.ForeignKey('posts.id')),

db.Column('tag\_id', db.Integer, db.ForeignKey('tags.id'))

)

class Post(db.Model):

\_\_tablename\_\_ = 'posts'

id = db.Column(db.Integer(), primary\_key=True)

title = db.Column(db.String(255), nullable=False)

slug = db.Column(db.String(255), nullable=False)

content = db.Column(db.Text(), nullable=False)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

updated\_on = db.Column(db.DateTime(), default=datetime.utcnow, onudate=datetime.utcnow)

category\_id = db.Column(db.Integer(), db.ForeignKey('categories.id'))

def \_\_repr\_\_(self):

return "<{}:{}>".format(self.id, self.title[:10])

class Tag(db.Model):

\_\_tablename\_\_ = 'tags'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(255), nullable=False)

slug = db.Column(db.String(255), nullable=False)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

posts = db.relationship('Post', secondary=post\_tags, backref='tags')

def \_\_repr\_\_(self):

return "<{}:{}>".format(self.id, self.name)

class Feedback(db.Model):

\_\_tablename\_\_ = 'feedbacks'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(1000), nullable=False)

email = db.Column(db.String(100), nullable=False)

message = db.Column(db.Text(), nullable=False)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

def \_\_repr\_\_(self):

return "<{}:{}>".format(self.id, self.name)

class Employee(db.Model):

\_\_tablename\_\_ = 'employees'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(255), nullable=False)

designation = db.Column(db.String(255), nullable=False)

doj = db.Column(db.Date(), nullable=False)

@login\_manager.user\_loader

def load\_user(user\_id):

return db.session.query(User).get(user\_id)

class User(db.Model, UserMixin):

\_\_tablename\_\_ = 'users'

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(100))

username = db.Column(db.String(50), nullable=False, unique=True)

email = db.Column(db.String(100), nullable=False, unique=True)

password\_hash = db.Column(db.String(100), nullable=False)

created\_on = db.Column(db.DateTime(), default=datetime.utcnow)

updated\_on = db.Column(db.DateTime(), default=datetime.utcnow, onupdate=datetime.utcnow)

def \_\_repr\_\_(self):

return "<{}:{}>".format(self.id, self.username)

def set\_password(self, password):

self.password\_hash = generate\_password\_hash(password)

def check\_password(self, password):

return check\_password\_hash(self.password\_hash, password)

forms.py

from flask\_wtf import FlaskForm

from wtforms import Form, ValidationError

from wtforms import StringField, SubmitField, TextAreaField, BooleanField

from wtforms.validators import DataRequired, Email

class ContactForm(FlaskForm):

name = StringField("Name: ", validators=[DataRequired()])

email = StringField("Email: ", validators=[Email()])

message = TextAreaField("Message", validators=[DataRequired()])

submit = SubmitField()

class LoginForm(FlaskForm):

username = StringField("Username", validators=[DataRequired()])

password = StringField("Password", validators=[DataRequired()])

remember = BooleanField("Remember Me")

submit = SubmitField()

utils.py

from . import mail, db

from flask import render\_template

from threading import Thread

from app import app

from flask\_mail import Message

def async\_send\_mail(app, msg):

with app.app\_context():

mail.send(msg)

def send\_mail(subject, recipient, template, \*\*kwargs):

msg = Message(subject, sender=app.config['MAIL\_DEFAULT\_SENDER'], recipients=[recipient])

msg.html = render\_template(template, \*\*kwargs)

thrd = Thread(target=async\_send\_mail, args=[app, msg])

thrd.start()

return thrd

Наконец, для запуска приложения нужно добавить следующий код в файл runner.py:

import os

from app import app, db

from app.models import User, Post, Tag, Category, Employee, Feedback

from flask\_script import Manager, Shell

from flask\_migrate import MigrateCommand

manager = Manager(app)

# эти переменные доступны внутри оболочки без явного импорта

def make\_shell\_context():

return dict(app=app, db=db, User=User, Post=Post, Tag=Tag, Category=Category, Employee=Employee, Feedback=Feedback)

manager.add\_command('shell', Shell(make\_context=make\_shell\_context))

manager.add\_command('db', MigrateCommand)

if \_\_name\_\_ == '\_\_main\_\_':

manager.run()

runner.py — это точка входа проекта. В первую очередь файл создает [**экземпляр объекта**](https://pythonru.com/primery/primery-raboty-s-klassami-v-python) Manager(). Затем он определяет функцию make\_shell\_context(). Объекты, которые вернет make\_shell\_context(), будут доступны в оболочке без импорта соответствующих инструкций. Наконец, метод run() экземпляра Manager будет вызван для запуска сервера.

### Порядок выполнения

В этом уроке было создано немало файлов, и достаточно легко запутаться в том, за что отвечает каждый из них, а также в порядке, в котором они запускаются. Этот раздел создан для разъяснения всего процесса.

Все начинается с исполнения файла runner.py. Вторая строка в файле runner.py импортирует app и db из пакета app. Когда интерпретатор Python доходит до этой строки, управление программой передается файлу \_\_init\_\_.py, который в этот момент начинает исполняться. На 7 строке \_\_init\_\_.py импортирует модуль config, который передает управление config.py. Когда исполнение config.py завершается, управление снова возвращается к \_\_init\_\_.py. На 21 строке \_\_init\_\_.py импортирует модуль views, который передает управление views.py. Первая строка views.py снова импортирует экземпляр приложения app из пакета app. Экземпляр приложения app уже в памяти, поэтому снова он не будет импортирован. На строках 4, 5 и 6 views.py импортирует модели, формы и функцию send\_mail и временно передает управление соответствующим файлам. Когда исполнение views.py завершается, управление программой возвращается к \_\_init\_\_.py. Это завершает исполнение \_\_init\_\_.py. Управление возвращается к runner.py и начинается исполнения инструкции на строке 3.

Третья строка runner.py импортирует классы, определенные в модуле models.py. Поскольку модели уже доступны в файле views.py, файл models.py не будет исполняться.

Поскольку runner.py работает как основной модуль, условие на 17 строке выполнено, и manager.run() запускает приложение.

### Запуск проекта

Теперь можно запускать проект. В терминале для запуска сервера нужно ввести следующую команду.

(env) gvido@vm:~/flask\_app$ python runner.py runserver

\* Restarting with stat

\* Debugger is active!

\* Debugger PIN: 391-587-440

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

Если переменная среды FLASK\_ENV не установлена, предыдущая команда запустит приложение в режиме отладки. Если зайти на https://127.0.0.1:5000/, откроется домашняя страница со следующим содержанием:

Name: Jerry

Также необходимо проверить остальные страницы приложения, чтобы убедиться, что все работает.

Приложение теперь является гибким. Оно может получить совсем иной набор настроек с помощью всего лишь одной переменной среды. Предположим, нужно перевести приложение в рабочий режим. Для нужно всего лишь создать переменную среды FLASK\_ENV со значением config.ProductionConfig.

В терминале нужно вести следующую команду для создания переменной среды FLASK\_ENV:

(env) gvido@vm:~/flask\_app$ export FLASK\_ENV=config.ProductionConfig

Эта команда создает переменную среды в Linux и macOS. Пользователи Windows могут использовать следующую команду:

(env) C:\Users\gvido\flask\_app>set FLASK\_ENV=config.ProductionConfig

Снова запустим приложение.

(env) gvido@vm:~/flask\_app$ python runner.py runserver

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

Теперь приложение работает в рабочем режиме. Если сейчас Python вызовет исключения, то вместо трассировки стека отобразится ошибка 500.

Поскольку сейчас все еще этап разработки, переменную среды FLASK\_ENV следует удалить. Она будет удалена автоматически при закрытии терминала. Чтобы сделать это вручную, нужно ввести следующую команду:

(env) gvido@vm:~/flask\_app$ unset FLASK\_ENV

Пользователи Windows могут использовать следующую команду:

(env) C:\Users\gvido\flask\_app>set FLASK\_ENV=

Проект теперь в лучшей форме. Его элементы организованы более логично. Использованный здесь подход подойдет для маленьких и средних по масштабу проектов. Тем не менее у Flask есть еще несколько козырей для тех, кто хочет быть еще продуктивнее.

## Blueprints (чертежы/схемы/планы/эскизы)

Эскизы — еще один способ организации приложений. Они предполагают разделение на уровне представления. Как и приложение Flask, эскиз может иметь собственные функции представления, шаблоны и статические файлы. Для них даже можно выбрать собственные URI. Предположим, ведется работа над блогом и административной панелью. Чертеж для блога будет включать функцию представления, шаблоны и статические файлы, необходимые только блогу. В то же время эскиз административной панели будет содержать файлы, которые нужны ему. Их можно использовать в виде модулей или пакетов.

Пришло время добавить эскиз к проекту.

### Создание эскиза

Сначала нужно создать папку main в папке flask\_app/app и переместить туда views.py и forms.py. Внутри папки main необходимо создать файл \_\_init\_\_.py со следующим кодом:

from flask import Blueprint

main = Blueprint('main', \_\_name\_\_)

from . import views

Здесь создается объект эскиза с помощью класса Blueprint. Конструктор Blueprint() принимает два аргумента: имя эскиза и имя пакета, где он расположен; для большинства приложений достаточно будет передать \_\_name\_\_.

По умолчанию функции представления в эскизе будут искать [**шаблоны**](https://pythonru.com/uroki/6-shablony-vo-flask) и статические файлы в папках приложения templates и static.

Изменить это можно, задав местоположение шаблонов и статических файлов при создании объекта Blueprint:

main = Blueprint('main', \_\_name\_\_,

template\_folder='templates\_dir')

static\_folder='static\_dir')

В этом случае [**Flask будет искать шаблоны**](https://pythonru.com/uroki/6-shablony-vo-flask) и статические файлы в папках templates\_dir и static\_dir, которые находятся в папке эскиза.

Путь шаблона, добавленный в эскизе, имеет более низкий приоритет по сравнению с папкой шаблонов приложения. Это значит, что если есть два шаблона с одинаковыми именами в папках templates\_dir и templates, Flask использует шаблон из папки templates.

Вот некоторые вещи, которые важно помнить, когда речь заходит о эскизах:

1. При использовании эскизов маршруты определяется с помощью декоратора route, а не экземпляра приложения (app).
2. Для[**создания URL**](https://pythonru.com/uroki/8-sozdanie-url-vo-flask) при использовании эскизов нужно в качестве префикса указать название эскиза, а через оператор точки (.) — конечную точку. Это необходимо для создания URL и в Python, и в шаблонах. Например:
3. url\_for("main.index")

Этот код вернет URL маршрута index эскиза main.  
Можно не указывать название эскиза, если работа ведется в том же эскизе, для которого создается URL. Например:

url\_for(".index")

Этот код вернет URL маршрута index для эскиза main в том случае, если код редактируется в функции представления или шаблоне эскиза main.

Чтобы приспособить изменения, нужно обновить инструкции import, вызовы url\_for() и маршруты во views.py. Это обновленная версия файла views.py.

from app import app, db

from . import main

from flask import Flask, request, render\_template, redirect, url\_for, flash, make\_response, session

from flask\_login import login\_required, login\_user, current\_user, logout\_user

from app.models import User, Post, Category, Feedback, db

from .forms import ContactForm, LoginForm

from app.utils import send\_mail

@main.route('/')

def index():

return render\_template('index.html', name='Jerry')

@main.route('/user/<int:user\_id>/')

def user\_profile(user\_id):

return "Profile page of user #{}".format(user\_id)

@main.route('/books/<genre>/')

def books(genre):

return "All Books in {} category".format(genre)

@main.route('/login/', methods=['post', 'get'])

def login():

if current\_user.is\_authenticated:

return redirect(url\_for('.admin'))

form = LoginForm()

if form.validate\_on\_submit():

user = db.session.query(User).filter(User.username == form.username.data).first()

if user and user.check\_password(form.password.data):

login\_user(user, remember=form.remember.data)

return redirect(url\_for('.admin'))

flash("Invalid username/password", 'error')

return redirect(url\_for('.login'))

return render\_template('login.html', form=form)

@main.route('/logout/')

@login\_required

def logout():

logout\_user()

flash("You have been logged out.")

return redirect(url\_for('.login'))

@main.route('/contact/', methods=['get', 'post'])

def contact():

form = ContactForm()

if form.validate\_on\_submit():

name = form.name.data

email = form.email.data

message = form.message.data

print(name)

print(email)

print(message)

# здесь логика БД

feedback = Feedback(name=name, email=email, message=message)

db.session.add(feedback)

db.session.commit()

send\_mail("New Feedback", app.config['MAIL\_DEFAULT\_SENDER'], 'mail/feedback.html',

name=name, email=email)

print("\nData received. Now redirecting ...")

flash("Message Received", "success")

return redirect(url\_for('.contact'))

return render\_template('contact.html', form=form)

@main.route('/cookie/')

def cookie():

if not request.cookies.get('foo'):

res = make\_response("Setting a cookie")

res.set\_cookie('foo', 'bar', max\_age=60\*60\*24\*365\*2)

else:

res = make\_response("Value of cookie foo is {}".format(request.cookies.get('foo')))

return res

@main.route('/delete-cookie/')

def delete\_cookie():

res = make\_response("Cookie Removed")

res.set\_cookie('foo', 'bar', max\_age=0)

return res

@main.route('/article/', methods=['POST', 'GET'])

def article():

if request.method == 'POST':

print(request.form)

res = make\_response("")

res.set\_cookie("font", request.form.get('font'), 60\*60\*24\*15)

res.headers['location'] = url\_for('.article')

return res, 302

return render\_template('article.html')

@main.route('/visits-counter/')

def visits():

if 'visits' in session:

session['visits'] = session.get('visits') + 1 # чтение и обновление данных сессии

else:

session['visits'] = 1 # настройка данных сессии

return "Total visits: {}".format(session.get('visits'))

@main.route('/delete-visits/')

def delete\_visits():

session.pop('visits', None) # удаление посещений

return 'Visits deleted'

@main.route('/session/')

def updating\_session():

res = str(session.items())

cart\_item = {'pineapples': '10', 'apples': '20', 'mangoes': '30'}

if 'cart\_item' in session:

session['cart\_item']['pineapples'] = '100'

session.modified = True

else:

session['cart\_item'] = cart\_item

return res

@main.route('/admin/')

@login\_required

def admin():

return render\_template('admin.html')

Стоит обратить внимание, что в файле views.py URL создаются без определения названия эскиза, потому что работа ведется в этом же эскизе.

Также нужно следующим образом обновить вызов url\_for() в admin.html:

#...

<p><a href="{{ url\_for('.logout') }}">Logout</a></p>

#...

Функции представления во views.py теперь ассоциируются с эскизом main. Дальше нужно зарегистрировать эскизы в приложении Flask. Необходимо открыть app/\_\_init\_\_.py и изменить его следующим образом:

#...

# создать экземпляр приложения

app = Flask(\_\_name\_\_)

app.config.from\_object(os.environ.get('FLASK\_ENV') or 'config.DevelopementConfig')

# инициализирует расширения

db = SQLAlchemy(app)

mail = Mail(app)

migrate = Migrate(app, db)

login\_manager = LoginManager(app)

login\_manager.login\_view = 'main.login'

# регистрация blueprints

from .main import main as main\_blueprint

app.register\_blueprint(main\_blueprint)

#from .admin import main as admin\_blueprint

#app.register\_blueprint(admin\_blueprint)

Метод register\_blueprint() экземпляра приложения используется для регистрации эскиза. Можно зарегистрировать несколько эскизов, вызвав register\_bluebrint() для каждого. Важно обратить внимание, что на 11 строке login\_manager.login\_view присваивается main.login. В этом случае важно указать, о каком эскизе идет речь, иначе Flask не сможет понять, на какой эскиз ссылается код.

Сейчас структура приложения выглядит так:

├── flask\_app/

├── app/

│ ├── \_\_init\_\_.py

│ ├── main/

│ │ ├── forms.py

│ │ ├── \_\_init\_\_.py

│ │ └── views.py

│ ├── models.py

│ ├── static/

│ │ └── style.css

│ ├── templates/

│ │ ├── admin.html

│ │ ├── article.html

│ │ ├── contact.html

│ │ ├── index.html

│ │ ├── login.html

│ │ └── mail/

│ │ └── feedback.html

│ └── utils.py

├── migrations/

│ ├── alembic.ini

│ ├── env.py

│ ├── README

│ ├── script.py.mako

│ └── versions/

│ ├── 0f0002bf91cc\_adding\_users\_table.py

│ ├── 6e059688f04e\_adding\_employees\_table.py

├── runner.py

├── config.py

├── env/

## Фабрика приложения

В приложении уже используются пакеты и эскизы (blueprints). Его можно улучшать и дальше, передав функцию создания экземпляров приложения Фабрике приложения. Это всего лишь функция, создающая объект.

Что это даст:

1. Упростит процесс тестирования, потому что можно будет создать экземпляр приложения с разными настройками
2. Можно будет запускать несколько экземпляров одного приложения в одном и том же процессе. Это удобно, когда используется балансировка нагрузки трафика между разными серверами.

Для внедрения фабрики приложения нужно обновить app/\_\_init\_\_.py:

from flask import Flask

from flask\_migrate import Migrate, MigrateCommand

from flask\_mail import Mail, Message

from flask\_sqlalchemy import SQLAlchemy

from flask\_script import Manager, Command, Shell

from flask\_login import LoginManager

import os, config

db = SQLAlchemy()

mail = Mail()

migrate = Migrate()

login\_manager = LoginManager()

login\_manager.login\_view = 'main.login'

# Фабрика приложения

def create\_app(config):

# создание экземпляра приложения

app = Flask(\_\_name\_\_)

app.config.from\_object(config)

db.init\_app(app)

mail.init\_app(app)

migrate.init\_app(app, db)

login\_manager.init\_app(app)

from .main import main as main\_blueprint

app.register\_blueprint(main\_blueprint)

#from .admin import main as admin\_blueprint

#app.register\_blueprint(admin\_blueprint)

return app

Теперь за создание экземпляров приложения ответственна функция create\_app. Она принимает один аргумент config и возвращает экземпляр приложения.

Фабрика приложений разделяет процесс создания экземпляров расширений и их настройки. Создание экземпляров происходит до того, как create\_app() вызывается, а настройка происходит внутри функции create\_app() с помощью метода init\_app().

Дальше нужно обновить runner.py для фабрики приложения:

import os

from app import db, create\_app

from app.models import User, Post, Tag, Category, Employee, Feedback

from flask\_script import Manager, Shell

from flask\_migrate import MigrateCommand

app = create\_app(os.getenv('FLASK\_ENV') or 'config.DevelopementConfig')

manager = Manager(app)

def make\_shell\_context():

return dict(app=app, db=db, User=User, Post=Post, Tag=Tag, Category=Category,

Employee=Employee, Feedback=Feedback)

manager.add\_command('shell', Shell(make\_context=make\_shell\_context))

manager.add\_command('db', MigrateCommand)

if \_\_name\_\_ == '\_\_main\_\_':

manager.run()

Стоит заметить, что при использовании фабрик приложения пропадает доступ к экземпляру приложения в эскизе во время импорта. Для получения доступа к экземплярам в эскизе нужно использовать прокси current\_app из пакета flask. Необходимо обновить проект для использования переменной current\_app:

from app import db

from . import main

from flask import (render\_template, request, redirect, url\_for, flash,

make\_response, session, current\_app)

from flask\_login import login\_required, login\_user, current\_user, logout\_user

from app.models import User, Feedback

from app.utils import send\_mail

from .forms import ContactForm, LoginForm

@main.route('/')

def index():

return render\_template('index.html', name='Jerry')

@main.route('/user/<int:user\_id>/')

def user\_profile(user\_id):

return "Profile page of user #{}".format(user\_id)

@main.route('/books/<genre>/')

def books(genre):

return "All Books in {} category".format(genre)

@main.route('/login/', methods=['post', 'get'])

def login():

if current\_user.is\_authenticated:

return redirect(url\_for('.admin'))

form = LoginForm()

if form.validate\_on\_submit():

user = db.session.query(User).filter(User.username == form.username.data).first()

if user and user.check\_password(form.password.data):

login\_user(user, remember=form.remember.data)

return redirect(url\_for('.admin'))

flash("Invalid username/password", 'error')

return redirect(url\_for('.login'))

return render\_template('login.html', form=form)

@main.route('/logout/')

@login\_required

def logout():

logout\_user()

flash("You have been logged out.")

return redirect(url\_for('.login'))

@main.route('/contact/', methods=['get', 'post'])

def contact():

form = ContactForm()

if form.validate\_on\_submit():

name = form.name.data

email = form.email.data

message = form.message.data

# логика БД здесь

feedback = Feedback(name=name, email=email, message=message)

db.session.add(feedback)

db.session.commit()

send\_mail("New Feedback", current\_app.config['MAIL\_DEFAULT\_SENDER'], 'mail/feedback.html',

name=name, email=email)

flash("Message Received", "success")

return redirect(url\_for('.contact'))

return render\_template('contact.html', form=form)

@main.route('/cookie/')

def cookie():

if not request.cookies.get('foo'):

res = make\_response("Setting a cookie")

res.set\_cookie('foo', 'bar', max\_age=60\*60\*24\*365\*2)

else:

res = make\_response("Value of cookie foo is {}".format(request.cookies.get('foo')))

return res

@main.route('/delete-cookie/')

def delete\_cookie():

res = make\_response("Cookie Removed")

res.set\_cookie('foo', 'bar', max\_age=0)

return res

@main.route('/article', methods=['POST', 'GET'])

def article():

if request.method == 'POST':

res = make\_response("")

res.set\_cookie("font", request.form.get('font'), 60\*60\*24\*15)

res.headers['location'] = url\_for('.article')

return res, 302

return render\_template('article.html')

@main.route('/visits-counter/')

def visits():

if 'visits' in session:

session['visits'] = session.get('visits') + 1

else:

session['visits'] = 1

return "Total visits: {}".format(session.get('visits'))

@main.route('/delete-visits/')

def delete\_visits():

session.pop('visits', None) # удаление посещений

return 'Visits deleted'

@main.route('/session/')

def updating\_session():

res = str(session.items())

cart\_item = {'pineapples': '10', 'apples': '20', 'mangoes': '30'}

if 'cart\_item' in session:

session['cart\_item']['pineapples'] = '100'

session.modified = True

else:

session['cart\_item'] = cart\_item

return res

@main.route('/admin/')

@login\_required

def admin():

return render\_template('admin.html')

utils.py

from . import mail, db

from flask import render\_template, current\_app

from threading import Thread

from flask\_mail import Message

def async\_send\_mail(app, msg):

with app.app\_context():

mail.send(msg)

def send\_mail(subject, recipient, template, \*\*kwargs):

msg = Message(subject, sender=current\_app.config['MAIL\_DEFAULT\_SENDER'], recipients=[recipient])

msg.html = render\_template(template, \*\*kwargs)

thr = Thread(target=async\_send\_mail, args=[current\_app.\_get\_current\_object(), msg])

thr.start()

return thr

В [**этих уроках**](https://pythonru.com/tag/uroki-po-flask-na-russkom) речь шла о многих вещах, которые дают необходимые знания о Flask, его составляющих и том, как они сочетаются между собой.