**ОСНОВЫ**

**FLASK**

# Что это такое? Простое WSGI-приложение

[Смотреть материал на видео](https://www.youtube.com/watch?v=6jxveKOdyNg&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/flask-chto-eto-takoe-wsgi-prilozhenie>

Начиная с этого занятия мы с вами будем знакомиться с весьма популярным и фреймворком Flask, который позволяет создавать сайты самых разных типов с использованием языка Python. Flask относится к разряду микрофреймворков, то есть, он предоставляет лишь базовый инструментарий для построения сайтов, как говорится, все по минимуму – ничего лишнего. Однако, этого вполне достаточно, чтобы создавать большинство типовых сайтов с поддержкой шаблонов страниц, баз данных и прочими стандартными возможностями. И, кроме того, простота позволяет достаточно быстро разобраться в этом пакете и начать сразу его применять. Наверное, в этом его главная фишка – простота и достаточность функционала для типовых решений.

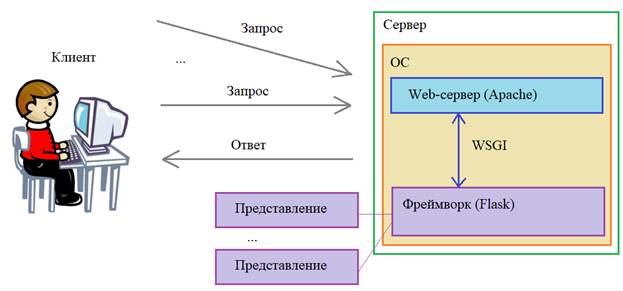
Для тех кто не совсем знаком с принципом взаимодействия между клиентом (браузером) и фреймворком, установленном на сервере, опишу в двух словах этот процесс.

Когда пользователь вводит в браузер строку запроса, например, vk.com, то от браузера отправляется запрос к серверу, где расположен и работает этот сайт. Здесь мы отложим в сторону вопрос маршрутизации и DNS-серверов, сейчас это неважно, главное, что сеть Интернет так устроена, что маршрутизаторы «знают» куда направлять запросы, если они относятся к работающим сайтам.



Сервер постоянно находится в режиме ожидания очередного запроса и как только он приходит, формирует ответ клиенту, как правило, в виде HTML-документа. Этот документ возвращается в браузер и пользователь видит на экране устройства заветную страницу.

Но где же во всей этой схеме фреймворк? В действительности он установлен на сервере. Так как это обычный компьютер (ну может не совсем обычный, но принцип тот же), то на нем установлено соответствующее программное обеспечение. Мы, опять же, не будем здесь глубоко вдаваться в подробности, скажу лишь, что на них часто устанавливают Linux-подобные ОС (благодаря их надежности), затем программу под названием веб-сервер (часто это Apache или Nginx) и уже он отдает обработку запроса конкретному фреймворку:



Здесь WSGI (Web Server Gateway Interface) — стандарт взаимодействия между Python-программой, выполняющейся на стороне сервера, и самим веб-сервером, например Apache. Фактически, это интерпретатор Python, который запускает WSGI-приложение, написанное на Flask.

При поступлении запроса активизируется WSGI-приложение, выполняется определенный обработчик, который еще называется «Представление» и реализованный в виде функции на языке Python. Соответственно, если приходит сразу несколько запросов, то одна и та же функция-обработчик может быть запущена в параллельных потоках. Многопоточность – это норма для фреймворков, поэтому, работая с представлениями во Flask, всегда следует это учитывать.

Конечно, в рамках наших занятий мы не будем использовать удаленный сервер и устанавливать на него данный фреймворк – это отдельная задача. Кстати, современные хостеры предоставляют инструментарий для простой установки и настройки Flask. Поэтому этот процесс не представляет больших сложностей. А для изучения данного пакета на домашнем ПК от вас потребуется только его установить, используя установщик pip:

pip install Flask

Теперь мы можем написать свое первое WSGI-приложение. В самом простом варианте оно выглядит так:

from flask import Flask

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

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

   app.run(debug=True)

Вначале идет импорт класса Flask, который, фактически и формирует это приложение. Далее, мы создаем экземпляр этого класса и первым аргументом должны указать имя нашего приложения. Если вся программа пишется в одном файле, то следует передавать директиву \_\_name\_\_, которая в случае импорта будет содержать имя текущего файла, а в случае самостоятельного запуска – значение «\_\_main\_\_». Для Flask это имеет принципиальное значение, в частности, от этого зависит где искать подкаталоги с шаблонами и статичными документами.

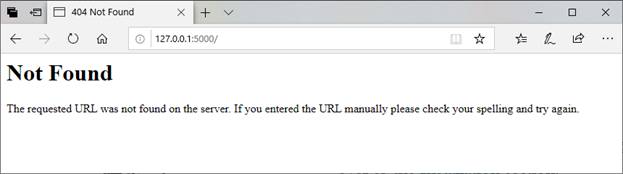
После этого выполняется запуск фреймворка методом run и в качестве параметра указывается debug=True, чтобы мы в браузере видели все ошибки, которые будут возникать при разработке сайта-приложения. Конечно, после его создания, здесь следует прописать debug=False, чтобы случайные ошибки реальный пользователь уже не видел.

И, наконец, условие. Зачем оно? Смотрите, когда мы непосредственно запускаем наш модуль, то директива \_\_name\_\_ будет принимать значение «\_\_main\_\_» и будет запущен локальный веб-сервер для отладки текущего приложения. Если же модуль запускается, например, на удаленном сервере, то нам не нужно запускать еще один сервер. В этом случае директива \_\_name\_\_ будет принимать имя данного модуля и строчка app.run выполнена не будет. То есть, мы это условие прописали с целью запуска приложения непосредственно на локальном устройстве.

Соответственно, как только фреймворк запущен, у нас активизируется локальный веб-сервер и мы можем в браузере создавать запрос, используя вот такой начальный адрес:

http://127.0.0.1:5000/

Давайте сделаем это и посмотрим, что получится. Наберем в браузере указанный запрос и видим, что запрашиваемая страница не найдена:



Все верно, так и должно быть, так как мы в программе не создали еще ни одного представления. Сделаем и это, добавим его:

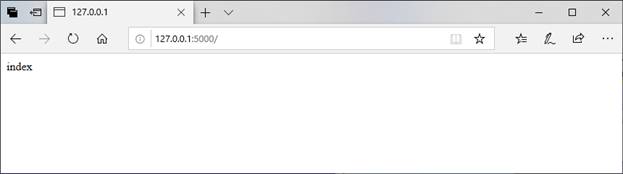
@app.route("/")

def index():

    return "index"

Здесь используется специальный декоратор route, который создает обертку вокруг нашей функции index, которая будет активизироваться при обращении к главной странице сайта, то есть, по запросу http://127.0.0.1:5000/

Запустим программу, обновим страницу и теперь в браузере видим то, что возвратила функция index:



Часто к главной странице обращаются еще по index, то есть:

domain/index

например,

proproprogs.ru/index

Чтобы одну и ту же страницу отобразить по нескольким URL-адресам, следует добавить несколько конструкций route:

@app.route("/index")

@app.route("/")

def index():

    return "index"

Соответственно, для любого другого адреса мы также можем добавить свой отдельный обработчик, прописав еще один декоратор route:

@app.route("/about")

def about():

    return "<h1>О сайте</h1>"

Теперь на нашем сайте как бы две страницы: главная и /about – о сайте. Причем, наши обработчики возвращают HTML-документ и все теги будут соответственно отображаться на странице в браузере.

Итак, на этом занятии мы с вами сделали первые шаги в понимании работы Flask и построили очень простое WSGI-приложение.

# Использование шаблонов страниц сайта

[Смотреть материал на видео](https://www.youtube.com/watch?v=TSsEMFZVr5E&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/ispolzovanie-shablonov>

Продолжаем изучение Flask и вначале хочу обратить ваше внимание, что программа из прошлого занятия возвращала довольно простую информацию. Но реальные страницы имеют довольно внушительную структуру, и даже используя многострочные строки, программа будет не очень читабельной:

from flask import Flask

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

@app.route("/")

def index():

    return '''<!DOCTYPE html>

<html>

<head>

         <title></title>

</head>

<body>

</body>

</html>'''

@app.route("/about")

def about():

    return "<h1>Про Flask</h1>"

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

    app.run(debug=True)

Кроме того структура страницы может меняться и вносить изменения непосредственно в программе – это плохой подход к программированию. Поэтому все HTML-шаблоны хранятся в виде отдельных файлов и загружаются по мере необходимости.

Для работы с шаблонами Flask использует стандартный модуль Jinja (если вы не знаете что это такое и как происходит обработка шаблонов, то смотрите занятия по этой [ссылке](https://www.youtube.com/watch?v=cFJqMXxVNsI&list=PLA0M1Bcd0w8wfmtElObQrBbZjY6XeA06U)).

Чтобы воспользоваться шаблонизатором во Flask нужно импортировать его элемент  render\_template:

from flask import Flask, render\_template

и, затем, в обработчике index вызвать его:

@app.route("/")

def index():

    return render\_template('index.html')

Следующий вопрос: где расположить шаблон 'index.html', чтобы он был найден и загружен модулем Flask. Существует следующее простое соглашение: по умолчанию все шаблоны берутся из подкаталога templates, относительно рабочего каталога программы (или соответствующем подкаталоге пакета). Так мы и сделаем. Разместим в этом подкаталоге файл index.html со следующим содержимым:

<!DOCTYPE html>

<html>

<head>

         <title>Главная страница сайта</title>

</head>

<body>

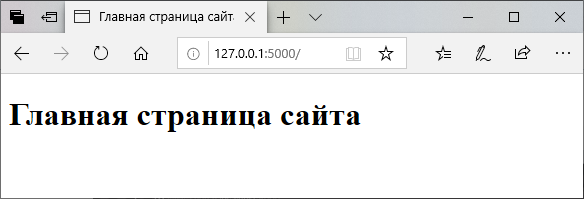
<h1>Главная страница сайта</h1>

</body>

</html>

Обратите внимание, для корректного отображения кириллицы все шаблоны рекомендуется сохранять в кодировке utf-8. Тем более, что сам Python, начиная с версии 3, по умолчанию использует юникод.

Запустим программу и при переходе на главную страницу увидим отображение нашего шаблона в браузере:



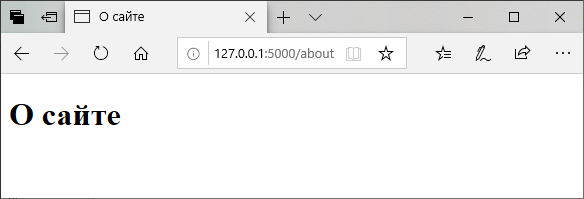
По аналогии создадим такой же шаблон about.html и также будем загружать его при обращении по URL /about:

@app.route("/about")

def about():

    return render\_template('about.html')

Теперь, посещая эту страницу, пользователь увидит:



Вот так легко и просто мы добавили два шаблона для наших двух страниц. Далее, можно их редактировать, не меняя код самой программы, что очень удобно.

## Передача шаблонам параметров

Я много раз произносил слово «шаблон», но что оно означает? Если посмотреть на файлы index.html или about.html, то это просто текст, который загружается и отдается браузеру по соответствующему запросу. Все так, но в этих же файлах можно прописать конструкции для отображения информации, например, из БД. Давайте для начала сделаем так, чтобы на каждой странице был свой заголовок, переданный ей через параметр title. Это можно сделать так:

<!DOCTYPE html>

<html>

<head>

         <title>{{title}}</title>

</head>

<body>

<h1>{{title}}</h1>

</body>

</html>

А в обработчике указать этот параметр:

return render\_template('index.html', title="Про Flask")

Все, теперь вместо title будет подставлена строка «Про Flask». Удобно, правда? Вот в этом и есть роль шаблонов: они описывают структуру страницы, а ее наполнение происходит динамически в самой программе.

Но какие вообще конструкции можно использовать в шаблонах? Как я уже отмечал, модуль Flask использует шаблонизатор Jinja и шаблоны строятся по его правилам. Если вы не знаете как работать с шаблонами, то под этим видео увидите ссылку на этот курс.

Давайте для примера еще добавим в документ простой список, пусть он символизирует наше меню. Шаблон для него можно прописать так (в файле index.html):

<ul>

{% for m in menu %}

<li>{{m}}</li>

{% endfor %}

</ul>

А в программе добавить список и передать его шаблону:

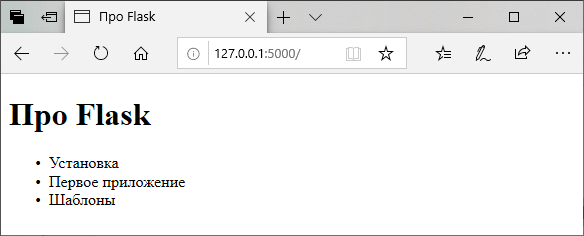
menu = ["Установка", "Первое приложение", "Обратная связь"]

@app.route("/")

def index():

    return render\_template('index.html', title="Про Flask", menu = menu)

При обновлении страницы увидим следующее:



Добавим большей гибкости нашему шаблону и для заголовка пропишем следующую конструкцию:

{% if title %}

         <title>Про Flask - {{title}}</title>

{% else %}

         <title>Про Flask</title>

{% endif %}

И то же самое для тега h1:

{% if title -%}

<h1>{{title}}</h1>

{% else -%}

<h1>Про Flask</h1>

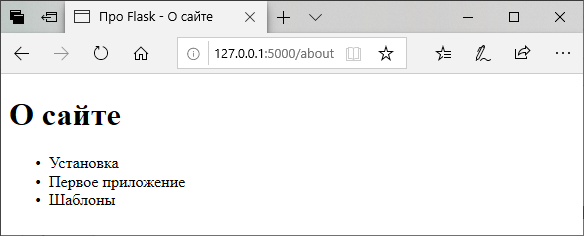
{% endif %}

Теперь в обработчике можно не указывать параметр title, тогда будет отображаться строка «Про Flask», а иначе, подставляться другой заголовок, причем во вкладке браузеры будем видеть «Про Flask - <заколовок>». Например, пропишем такой же шаблон для about.html и в обработчике добавим:

def about():

    return render\_template('about.html', title = "О сайте", menu = menu)

В результате, вид страницы будет такой:



Но, наши созданные шаблоны, мягко говоря, не очень, т.к. они содержат много повторяющегося кода. Лучшим вариантом будет воспользоваться механизмом расширения (наследования) шаблона для создания дочерних страниц сайта. Для начала определим базовый шаблон страницы – ее структуру, следующим образом (файл base.html):

<!DOCTYPE html>

<html>

<head>

{% block title -%}

{% if title %}

         <title>Про Flask - {{title}}</title>

{% else %}

         <title>Про Flask</title>

{% endif %}

{% endblock %}

</head>

<body>

{% block content -%}

         {%- block mainmenu -%}

<ul>

         {% for m in menu -%}

<li>{{m}}</li>

         {% endfor -%}

</ul>

         {% endblock mainmenu -%}

         {% if title -%}

<h1>{{title}}</h1>

         {% else -%}

<h1>Про Flask</h1>

         {% endif -%}

{% endblock -%}

</body>

</html>

А в дочерних расширим этот базовый шаблон:

**- для index.html:**

{% extends 'base.html' %}

{% block content %}

{{ super() }}

Содержимое главной страницы

{% endblock %}

**- для about.html:**

{% extends 'base.html' %}

{% block content %}

{{ super() }}

Содержимое страницы "о сайте"

{% endblock %}

Все, теперь никакого дублирования в наших шаблонах нет и мы можем достаточно просто создавать множество страниц сайта и при необходимости менять их структуру, просто меняя базовый шаблон base.html.

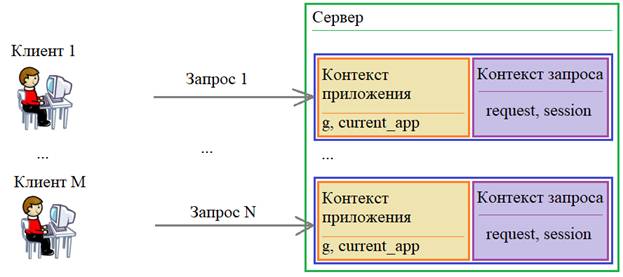
# Контекст приложения и контекст запроса

[Смотреть материал на видео](https://www.youtube.com/watch?v=tP09rxKbNMU&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/kontekst-prilozheniya-i-kontekst-zaprosa>

Продолжаем изучение пакета Flask и чтобы правильно понимать дальнейший материал, нам нужно разобраться с такими понятиями как *контекст приложения* и *контекст запроса*.

Когда сервер получает запрос от клиента, то сначала может быть создан контекст приложения, а затем обязательно создается контекст запроса, который и является фактически представлением (обработчиком) текущего запроса.



Зачем нужны эти контексты и какова их роль? Чаще всего они нужны для следующего. С контекстом приложения связаны две глобальные переменные: g и current\_app. В первой сохраняется общая временная пользовательская информация, необходимая для обработки запросов. Например, мы собираемся из функций обращаться к БД. Тогда было бы правильно при появлении запроса установить соединение с БД и сохранить это соединение в какой-либо переменной, а после обработки запроса – закрыть соединение с БД. Как раз для хранения текущего активного соединения с БД хорошо подходит переменная g контекста приложения. Но, всегда нужно помнить, что как только обработка запроса завершается, данные в объекте g автоматически удаляются.

Вторая переменная (current\_app) ссылается на контекст текущего приложения, которое было активизировано фреймворком Flask в ответ на текущий запрос. Зачем это вообще нужно? Разве мы создаем не одно приложение для одного сайта? В действительности не обязательно одно, Flask обеспечивает поддержку работы сразу нескольких WSGI-приложений. Например, в нашем пакете могут быть модули, в которых встречаются такие строчки:

app1 = Flask("app1")

app2 = Flask("app2")

В результате получаем два приложения и каждое отвечает за свой функционал сайта. Но у каждого приложения могут быть свои настройки и свои пути для хранения шаблонов и прочее. Чтобы встроенные функции Flask могли адекватно работать в условиях нескольких приложений, как раз и используется переменная current\_app, которая ссылается на текущее приложение, активизированное для соответствующего запроса.

Однако, если фреймворк Flask видит, что мы в обработчике не собираемся использовать функционал контекста приложения, то оно не создается, чтобы не тратить процессорное время и ресурсы сервера.

А вот следующий контекст запроса создается всегда и обязательно после контекста приложения (если он применяется). С контекстом запроса связаны две другие глобальные переменные: request и session. Первый глобальный объект содержит данные, связанные с пришедшим запросом, включая текущий URL. Часто – это данные GET-запроса в формате: ключ-значение. Но, могут быть и более сложные, например, POST-запрос при загрузке файлов. Переменная request также доступна во всех функциях в пределах текущего потока (текущего представления), а также в шаблонах. И автоматически исчезает, когда обработка запроса завершается.

Переменная session – это словарь, в котором можно сохранять данные в пределах сессии. В отличие от других переменных, сессия сохраняется между запросами, но уникальна для каждого источника запроса. Например, некий пользователь авторизовался на сайте на странице login.html. Далее, когда он будет переходить на другие страницы сайта, мы должны «знать», что он уже авторизован и показывать ему соответствующую информацию. Вот как раз это можно сделать с помощью сессии: при авторизации сохранить данные, что он авторизован и при новом запросе от него проверять переменную session на предмет этой авторизации.

Разумеется, у любой сессии есть, так называемое, время жизни, спустя которое она очищается. Это время отсчитывается от последнего момента прихода соответствующего запроса. То есть, если наш пользователь долго не заходил на сайт, то ему придется авторизоваться еще раз.

Вот в двух словах, что представляют собой контекст приложения и запроса и для чего они служат. В дальнейшем, при изучении материала нам эта информация очень пригодится. Поэтому, если вы с ней ранее не были знакомы, то постарайтесь разобраться как все это работает.

# Функция url\_for и переменные URL-адреса

[Смотреть материал на видео](https://www.youtube.com/watch?v=oM39KVYsjRs&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/url-for-i-peremennye-adresa>

Следующая важная и очень распространенная функция, с которой мы познакомимся – это url\_for(). Она позволяет генерировать URL-адрес по имени функции-обработчика. Например, рассмотрим вот такую программу:

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

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

menu = ["Установка", "Первое приложение", "Обратная связь"]

@app.route("/")

def index():

    print( url\_for('index') )

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

@app.route("/about")

def about():

    print( url\_for('about') )

    return render\_template('about.html', title = "О сайте", menu = menu)

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

    app.run(debug=True)

Мы здесь дополнительно импортировали функцию url\_for, а затем, вызываем ее в обработчиках index и about. Если теперь посетить данные страницы, то в консоли увидим соответствующие URL:

/ и /about

Но спрашивается: зачем она нам нужна, мы же и так знаем эти URL, поэтому, что нам мешает их явно указать? Если мы так будем делать, то столкнемся с множеством проблем. Во-первых, URL могут измениться, и тогда нам придется везде их менять, а если где-то забудем, то это негативно скажется на отображении страниц сайта. Или же, URL могут генерироваться динамически и тогда мы в принципе не сможем их явно прописать. Наконец, один и тот же шаблон для разных приложений, возможно, должен подставлять разный URL, например, для подключения внешних CSS или JS файлов. И это лишь малая толика проблем, с которыми мы можем столкнуться. Поэтому использовать явную запись URL – это пагубная практика и от нее лучше сразу отказаться, используя взамен функцию url\_for. Ее полный синтаксис, следующий:

url\_for(endpoint, \*\*values)

где values – словарь именованных аргументов.

Вызов данной функции непосредственно связан с контекстом запроса и вне его она просто не будет работать. То есть, если выполнить ее за пределами наших обработчиков, например, так:

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

menu = ["Установка", "Первое приложение", "Обратная связь"]

@app.route("/")

def index():

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

@app.route("/about")

def about():

    return render\_template('about.html', title = "О сайте", menu = menu)

print( url\_for('index') )

То возникнет ошибка отсутствия контекста вызова. Как же можно в целях отладки тестировать ее работу? Для этого фреймворк Flask позволяет искусственно создавать контекст запроса без активации веб-сервера. Это делается так:

with app.test\_request\_context():

    print( url\_for('index') )

Теперь все сработает и в консоли мы увидим URL для index.

Также эта функция корректно работает и в случае использования нескольких WSGI-приложений. Она обращается к переменной current\_app и затем, берет URL из активного приложения.



## Способы описания URL

В предыдущих примерах мы видели как с помощью декоратора route происходит привязка функции к URL-адресу. Но эти адреса можно делать и переменными, используя следующий синтаксис:

@app.route("/url/<variable>")

Здесь variable – это некоторая переменная, значение которой определяется  URL-адресом. Например, можно сделать так:

@app.route("/profile/<username>")

def profile(username):

    return f"Пользователь: {username}"

И если в браузере набрать запрос:

http://127.0.0.1:5000/profile/selfedu

то username примет значение selfedu. Или можно создать такой запрос:

http://127.0.0.1:5000/profile/12345678

Соответственно, username = «12345678», то есть все, что указывается в URL, записывается в виде строки.

При необходимости мы можем добавлять конверторы при определении переменных, например, указать, что следует использовать только целые числа:

@app.route("/profile/<int:username>")

Тогда при запросе вида:

http://127.0.0.1:5000/profile/12345678fff

Сервер вернет код страницы 404 – не существующий URL. И все потому, что предполагается запись в поле username из цифр. Если убрать буквы и обновить страницу, то все сработает.

В качестве конверторов можно использовать следующие обозначения:

* int – должны присутствовать только цифры;
* float – можно записывать число с плавающей точкой;
* path – можно использовать любые допустимые символы URL плюс символ слеша ‘/’.

Например, при конверторе path:

@app.route("/profile/<path:username>")

Можно сформировать следующий запрос:

http://127.0.0.1:5000/profile/12345678ddd/fdfgh

и переменная username = «12345678ddd/fdfgh». Но, если убрать это определение:

@app.route("/profile/<username>")

то этот же запрос приведет к ошибке 404, т.к. обратный слеш воспринимается как продолжение URL, не относящееся к полю username.

В заключение этого занятия давайте посмотрим на работу функции url\_for для переменных URL. Для создадим тестовый контекст запроса и в нем вызовем url\_for для разных представлений:

with app.test\_request\_context():

    print(url\_for('index'))

    print(url\_for('about'))

    print(url\_for('profile', username="selfedu"))

В консоли увидим следующие строчки:

/  
/about  
/profile/selfedu

Обратите внимание на последний вызов. Так как для profile реализован переменный URL с полем username, то этот параметр нужно явно прописать, чтобы получить конечный вид URL-адреса. Без этого параметра функция url\_for приведет к ошибке.

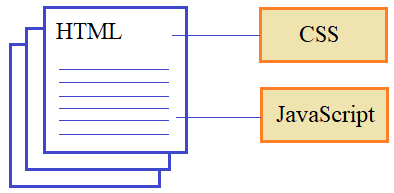
Итак, на этом занятии мы с вами рассмотрели понятия контекста приложения и контекста запроса, познакомились с функцией url\_for и увидели как можно описывать переменные URL-адреса.

# Подключение внешних ресурсов и работа с формами

[Смотреть материал на видео](https://www.youtube.com/watch?v=pgoRiPJkm3g&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/podklyuchenie-vneshnih-resursov-i-rabota-s-formami>

Это занятие начнем с рассмотрения способа подключения внешних ресурсов к шаблону HTML-документа. Наверное, многие из вас знают, что полноценные страницы сайта представляются не только файлом HTML с набором тегов, но и, например, каскадными таблицами стилей (CSS), JavaScript-программами, которые исполняются в браузере клиента и другими дополнительными внешними ресурсами.



И здесь возникает вопрос: где расположить эти вспомогательные файлы, которые, как правило, являются общими для всех страниц сайта, и как прописать к ним путь в шаблоне HTML-страницы. Для этого, как раз хорошо подходит функция

url\_for

о которой мы говорили на предыдущем занятии. Ее также можно вызывать непосредственно в шаблоне и она будет связана с текущим контекстом запроса, по которому берется шаблон. То есть, функция url\_for корректно отработает и корректно возвратит запрашиваемый URL-адрес. Но что следует прописать первым аргументом у этой функции. Как мы говорили, там должно фигурировать имя функции-обработчика. Но здесь же нам нужен не обработчик а путь к внешнему файлу. Для этого во Flask для url\_for зарезервировано специальный параметр

'static'

который означает, что нужно обратиться к подкаталогу 'static' и там взять файл, указанный в именованном параметре

filename='css/styles.css'

Давайте подключим оформление к страницам сайта в шаблоне base.html, используя эту функцию:

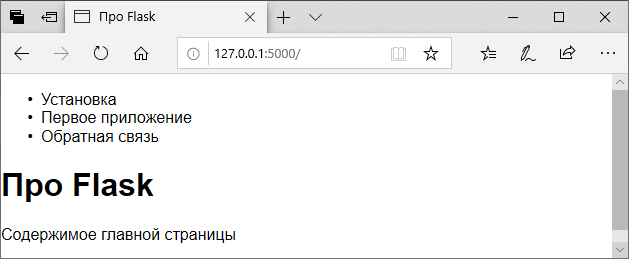
<link type="text/css" href="{{ url\_for('static', filename='css/styles.css')}}" rel="stylesheet" />

В данном случае функция url\_for возвратит путь:

static/css/styles.css

и в шаблон будет подключен этот файл оформления страницы. Содержимое этого файла следующее (см. видео). Я не буду здесь подробно объяснять как работают эти стили, если вы не знакомы с CSS, то посмотрите [это занятие](https://www.youtube.com/watch?v=CkTJXpS7KS4&list=PLA0M1Bcd0w8wRiyGX_9y-fUiBPi1vqaTb&index=9).

Обновим страницу, увидим следующий результат:



Чтобы таблицы стилей были применены к соответствующим элементам HTML-документа, добавим следующие строчки (в base.html):

<ul class="mainmenu">

…

</ul>

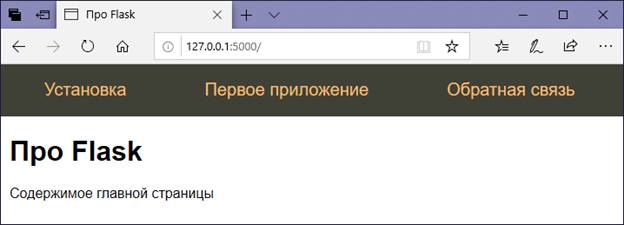
<div class="clear"></div>

<div class="content">

…

</div>

Теперь, при обновлении увидим такую страницу:



Далее, добавим ссылки нашим пунктам меню:

<li><a href="{{m.url}}">{{m.name}}</a></li>

А само меню в программе представим в виде списка словарей:

menu = [{"name": "Установка", "url": "install-flask"},

        {"name": "Первое приложение", "url": "first-app"},

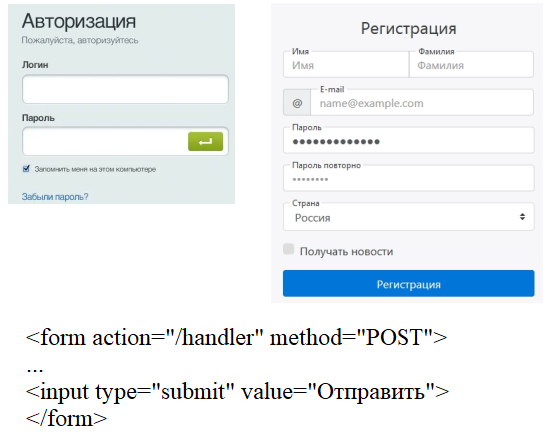
        {"name": "Обратная связь", "url": "contact"}]

Теперь при обновлении страницы мы увидим полноценное меню.

Аналогичным образом можно подключать и другие внешние файлы.

## Работа с формой (form)

Теперь, когда мы с вами разобрались как конструировать шаблоны страниц, пришло время познакомиться с обработкой form, которые есть практически в любом сайте. Что такое формы, я думаю, вы прекрасно знаете, например, очень часто можно встретить формы авторизации и регистрации:



Они описываются на странице с помощью тега form, в параметре action указывается URL, который должен принимать данные от формы, а параметр method определяет способ передачи данных. В основном, используются два вида передачи:

* GET – в виде строки запроса: "/handler?name=Alex&old=18&profit=1000";
* POST – в виде бинарных данных (используется для передачи больших объемов данных: изображений, звуков, документов и т.п., а также закрытых сведений: паролей, логинов и т.п.).

Добавим форму в шаблон contact.html:

{% extends 'base.html' %}

{% block content %}

{{ super() }}

<form action="/contact" method="post" class="form-contact">

<label>Имя: </label> <input type="text" name="username" value="" requied />

<label>Email: </label> <input type="text" name="email" value="" requied />

<label>Сообщение:</label>

<textarea name="message" rows=7 cols=40></textarea>

<input type="submit" value="Отправить" />

</form>

{% endblock %}

Мы здесь указали способ отправки данных в виде POST-запроса и обработчик «/contact», которому будут переданы данные из формы.

Далее, пропишем следующее оформление формы (в файле styles.css):

.form-contact label {

         display: inline-block;

         min-width: 80px;

}

.form-contact p {margin: 10px 0 10px 0;}

.form-contact input[type=submit], textarea {

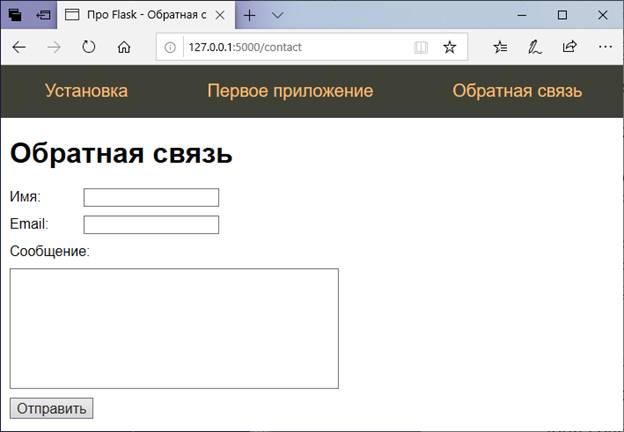
         font-size: 16px;

}

Запустим программу, откроем в браузере страницу

http://127.0.0.1:5000/contact

и увидим следующий результат:



Если сейчас ввести в форму какие-либо данные и нажать на кнопку отправить, то сервер возвратит ошибку:

405 – запрет на прием данных (Method Not Allowed)

Дело в том, что в обработчике мы должны явно указать: может ли он принимать данные методом POST. Для этого нужно прописать параметр methods со значением POST как элемент списка:

@app.route("/contact", methods=["POST"])

Или так, если хотим обрабатывать и POST и GET запросы:

@app.route("/contact", methods=["POST", "GET"])

Далее в обработчике проверим: пришел ли именно POST-запрос, а не какой-либо другой и выведем данные в консоль:

    if request.method == 'POST':

        print(request.form)

А вначале импортируем этот объект:

from flask import Flask, render\_template, request

Мы здесь используем объект request, который связан с данными текущего запроса и существует в пределах контекста запроса, как мы это отмечали на предыдущих занятиях. Далее, все данные формы доступны по свойству form, которое представляет своеобразный словарь, то есть, для доступа к конкретному полю можно использовать следующую запись:

print(request.form['username'])

Мы пока не будем нигде сохранять принятые данные, т.к. для этого следует использовать БД, о которой еще ничего не говорили. Главное, на этом этапе понимать, как происходит прием данных от формы и их представление в объекте request.

На этом завершим это занятие. На следующем продолжим изучать функционал работы с формами.

# Мгновенные сообщения - flash, get\_flashed\_messages

[Смотреть материал на видео](https://www.youtube.com/watch?v=b-Pi5Ggnm2w&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/soobshcheniya-flash-get-flashed-messages>

В современном Web-программировании считается хорошим стилем отправлять пользователю мгновенные сообщения о результатах взаимодействия с сайтом. Например, при отправке данных формы, пользователю можно написать, что данные отправлены успешно, или, наоборот, что произошла ошибка. Этот функционал встроен во фреймворк Flask и реализуется с помощью двух функций:

* flash() – формирование сообщения пользователю;
* get\_flashed\_messages() – обработка сформированных сообщений в шаблоне документа.

Их синтаксис, следующий:

flask.flash(message, category=’message’)

flask.get\_flashed\_messages(with\_categories=False, category\_filter=[])

* message – текст сообщения;
* category – категория сообщения;
* with\_categories – разрешает использование категорий при извлечении сообщений;
* category\_filter – список разрешенных категорий при выборке сообщений.

Чтобы воспользоваться функцией flash ее нужно импортировать из модуля flask:

from flask import Flask, render\_template, request, flash

И, затем, в качестве примера сформируем сообщение условно при успешной проверки принятых данных и при ошибке:

if len(request.form['username']) > 2:

    flash('Сообщение отправлено')

else:

    flash('Ошибка отправки')

Далее, в шаблоне contact.html вызовем функцию get\_flashed\_messages следующим образом:

{% for msg in get\_flashed\_messages() %}

<div class="flash">{{msg}}</div>

{% endfor %}

Обратите внимание, функция get\_flashed\_messages возвращает итератор, содержащий сообщения, то есть, их может быть несколько, если flash в обработчике был вызван несколько раз. Но, в нашем случае, оно будет одно, которое выводится в блоке div.

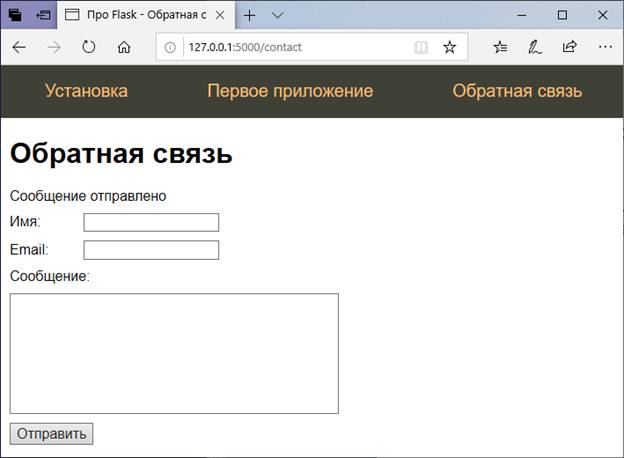
Если мы сейчас попробуем отправить данные формы, то получим ошибку:

RuntimeError: The session is unavailable

Дело в том, что функции flash и get\_flashed\_messages используют механизм сессий, чтобы отображать сообщения только один раз и при обновлении страницы более его не показывать. Чтобы активизировать работу сессий, необходимо в конфигурацию программы прописать секретный ключ. Я сделаю это так:

app.config['SECRET\_KEY'] = 'fdgdfgdfggf786hfg6hfg6h7f'

Это не самый лучший ключ, позже мы еще об этом поговорим, а пока пусть он будет записан в таком виде. Теперь, при обновлении страницы, увидим следующее:



А если обратиться к этой же странице еще раз (без отправки формы), то сообщение пропадет.

Далее, добавим оформление этому сообщению, причем разное для разных типов: успешных и ошибочных. В стилях пропишем следующее:

.flash {padding: 10px;}

.flash.success {

         border: 1px solid *#21DB56;*

         background: *#AEFFC5;*

}

.flash.error {

         border: 1px solid *#FF4343;*

         background: *#FF9C9C;*

}

А в программе вызовем функцию flash с указанием категории сообщения:

if len(request.form['username']) > 2:

    flash('Сообщение отправлено', category='success')

else:

    flash('Ошибка отправки', category='error')

Имя категории у нас и будет именем расширения стиля оформления. В шаблоне пропишем следующее:

{% for cat, msg in get\_flashed\_messages(True) %}

<div class="flash {{cat}}">{{msg}}</div>

{% endfor %}

Здесь параметр True в функции get\_flashed\_messages разрешает оперировать категориями мгновенных сообщений. Все, теперь при успешной отправке увидим сообщение на зеленом фоне, а при ошибке – на красном.

# Декоратор errorhandler, функции redirect и abort

[Смотреть материал на видео](https://www.youtube.com/watch?v=QCQ7GDhr4Tc&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/dekorator-errorhandler-funkcii-redirect-i-abort>

Следующим важным моментом при разработке сайта является отлавливание некоторых ошибок ответа сервера. Например, при отсутствии какой-либо страницы пользователь в браузере увидит следующее:



Это есть стандартный ответ сервера с кодом 404 – страница не найдена. В таких ситуациях лучше иметь обработчик для кода 404, который бы возвращал пользователю более дружественную информацию.

Чтобы создать такой обработчик, следует использовать специальный декоратор errorhandler с указанием в нем кода ответа, с которым будет ассоциирована функция представления:

@app.errorhandler(404)

def pageNotFount(error):

    return render\_template('page404.html', title="Страница не найдена", menu=menu)

Затем, добавим шаблон page404.html, например, такой:

{% extends 'base.html' %}

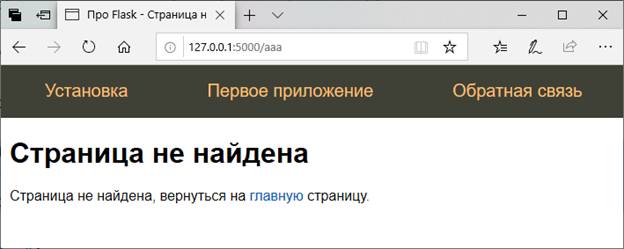
{% block content %}

{{ super() }}

Страница не найдена, вернуться на <a href="/">главную</a> страницу.

{% endblock %}

И при обновлении страницы, увидим более дружественное сообщение. Причем, код, который возвратит сервер, будет равен 200 – все хорошо.



Если все же, требуется при отображении своей собственной страницы возвращать прежний код 404, то это можно реализовать так:

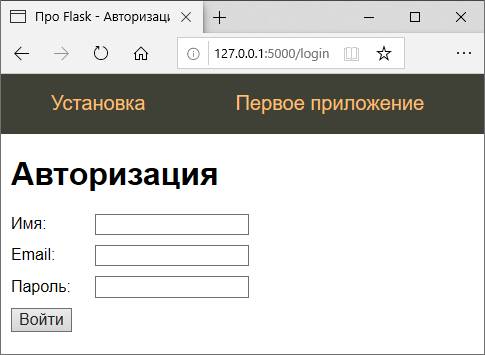
return render\_template('page404.html', title="Страница не найдена", menu=menu), 404

Обновляем страницу и видим в консоли, что сервер возвратил код 404. Однако, при разработке реальных сайтов, лучше в таких случаях возвращать код 200, т.е. то, что идет по умолчанию, т.к. поисковым системам «нравится», когда на сайте страницы отвечают с этим кодом и считается, что это положительно сказывается на ранжировании их в поисковой выдаче.

## Перенаправление запроса

Часто при создании сайта требуется делать перенаправление пользователя на другую страницу. Например, пользователь в браузере вводит адрес страницы с формой авторизации:

http://127.0.0.1:5000/login



И, если он еще не авторизован, то отображается вот такая страница, а иначе – осуществляется переадресация на страницу профайла. Давайте для примера реализуем такой простой функционал. Обработчик представим в следующем виде:

@app.route("/login", methods=["POST", "GET"])

def login():

    if 'userLogged' in session:

        return redirect(url\_for('profile', username=session['userLogged']))

    elif request.form['username'] == "selfedu" and request.form['psw'] == "123":

        session['userLogged'] = request.form['username']

        return redirect(url\_for('profile', username=session['userLogged']))

    return render\_template('login.html', title="Авторизация", menu=menu)

Мы здесь вначале с помощью сессии проверяем: авторизован ли пользователь, то есть, присутствует ли ключ 'userLogged' в сессии. И если это так, то сразу осуществляется переход на страницу профайла. Иначе идет проверка на тот случай, если пользователь уже передал данные для входа. И для username равным selfedu и пароля 123 осуществляется переход на страницу профайла. Если же обе эти проверки оказались ложными, то отображается страница с формой авторизации.

Шаблон формы авторизации можно описать так (файл login.html):

{% extends 'base.html' %}

{% block content %}

{{ super() }}

<form action="/login" method="post" class="form-contact">

<label>Имя: </label> <input type="text" name="username" value="" requied />

<label>Пароль: </label> <input type="password" name="psw" value="" requied />

<input type="submit" value="Войти" />

</form>

{% endblock %}

## Прерывание запроса

Чтобы функционал нашего сайта довести до логического конца, на странице профайла (в обработчике profile) сделаем проверку: если пользователь самостоятельно в браузере набирает путь, например:

http://127.0.0.1:5000/profile/selfedu

то страница должна отображаться только в том случае, если пользователь авторизован. Иначе, завершить вызов ошибкой 401 – доступ запрещен. Это можно сделать так:

@app.route("/profile/<username>")

def profile(username):

    if 'userLogged' not in session or session['userLogged'] != username:

        abort(401)

    return f"Пользователь: {username}"

Смотрите, если в сессии нет ключа 'userLogged', то вызывается функция abort, которая указывает серверу вернуть код ошибки 401. Иначе, отображается профайл пользователя.

Таким образом, пользователь может смотреть только свой профайл и даже если попытается в запросе браузера указать логин другого человека, то получит ошибку доступа.

**Использование БД,**

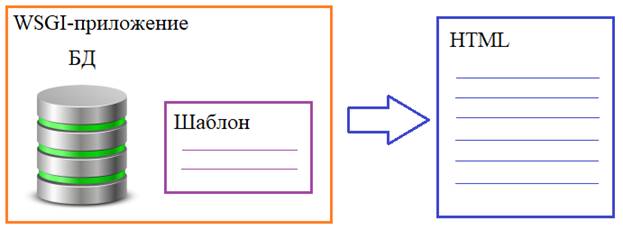
**cookies и сессии**

# Создание БД, установление и разрыв соединения при запросах

[Смотреть материал на видео](https://www.youtube.com/watch?v=aHWQkbk3xVA&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/sozdanie-bd-ustanovlenie-i-razryv-soedineniya-pri-zaprosah>

Продолжаем изучать фреймворк Flask и на этом занятии поговорим об использовании БД в WSGI-приложениях. Зачем они нужны, я думаю вы хорошо себе представляете. В двух словах, в БД сохраняется вся изменяемая информация, которая, затем, используется при формировании ответов на запросы пользователей. Например, мы можем сохранить в БД меню сайта и при формировании HTML-страниц, брать оттуда информацию и представлять ее в виде меню страницы.



В рамках наших занятий мы будем использовать СУБД SQLite, которая поставляется вместе с Python3. Но для реальных сайтов – это не лучший выбор. Гораздо продуктивнее будет использовать функционал модуля

SQLAlchemy

или СУБД

PostgreSQL

Однако, в качестве первого шага, вполне подойдет и SQLite. Если вы не знаете как работать с этой СУБД, то смотрите занятие [по этой теме](https://www.youtube.com/watch?v=TwnCXdCa8qg&list=PLA0M1Bcd0w8x4Inr5oYttMK6J47vxgv6J). Итак, первым делом в программе помимо Flask выполним импорт дополнительных пакетов:

import sqlite3

import os

from flask import Flask, render\_template, request

Затем, нам понадобится выполнить конфигурацию нашего WSGI-приложения. Во Flask принято соглашение: все переменные, записанные заглавными буквами, относятся к конфигурационной информации. Мы этим воспользуемся и определим следующие вспомогательные значения:

*# конфигурация*

DATABASE = '/tmp/flsite.db'

DEBUG = True

SECRET\_KEY = 'fdgfh78@#5?>gfhf89dx,v06k'

USERNAME = 'admin'

PASSWORD = '123'

Здесь все параметры, в принципе, понятны, кроме, может быть, одного –  SECRET\_KEY. Он необходим для безопасной работы сессий на стороне клиента. С помощью этого секретного ключа выполняется шифрование данных, которые, затем, сохраняются в куках браузера. Поэтому, даже глядя на информацию, сохраненную в браузере, пользователь не сможет понять, что она означает. Разумеется, все шифрование Flask выполняет автоматически. От нас требуется только прописать этот ключ с как можно более непонятным набором символов.

Далее создаем само приложение и загружаем конфигурацию из текущего модуля:

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

app.config.from\_object(\_\_name\_\_)

Здесь последняя строчка как раз и выполняет начальную инициализацию приложения, формируя конфигурацию из вышеопределенных переменных модуля flsite.py.

Далее, мы переопределим в конфигурации значение DATABASE, расположив БД в текущем каталоге приложения:

app.config.update(dict(DATABASE=os.path.join(app.root\_path,'flsite.db')))

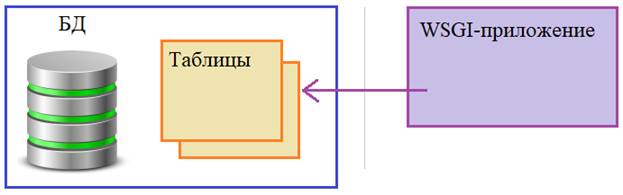
Возможно, у вас возникнет вопрос: почему нельзя было сразу вначале определить DATABASE как нужно и не выполнять эту последнюю команду? Дело в том, что изначально у нас не было созданного приложения app и, соответственно, не могли обратиться к его свойству root\_path, для определения корневого каталога. Конечно, можно было бы воспользоваться вот такой конструкцией для определения рабочего каталога:

os.path.dirname(os.path.abspath(\_\_file\_\_))

Однако, в случае использования нескольких WSGI-приложений рабочий каталог и каталог текущего приложения могут различаться. По этой причине в программе используется свойство root\_path.

## Создание БД

Итак, конфигурацию нашего приложения определили. Теперь пришло время создать БД. Общая концепция такая. Сначала отдельно создается БД с набором таблиц (без запуска веб-сервера). А уже потом, обработчики запросов обращаются к созданным таблицам БД, записывают и считывают из них информацию.



Как это сделать? В самом простом случае мы можем поступить так. Создадим общую функцию для установления соединения с БД:

def connect\_db():

    conn = sqlite3.connect(app.config['DATABASE'])

    conn.row\_factory = sqlite3.Row

    return conn

И, затем, объявим вспомогательную функцию, которая и будет создавать начальную БД с набором необходимых таблиц:

def create\_db():

    """Вспомогательная функция для создания таблиц БД"""

    db = connect\_db()

    with app.open\_resource('sq\_db.sql', mode='r') as f:

        db.cursor().executescript(f.read())

    db.commit()

    db.close()

Здесь мы используем метод open\_resource, который открывает файл 'sq\_db.sql' на чтение, расположенный в рабочем каталоге нашего приложения. Затем, для открытой БД выполняется скрипт, записанный в файле 'sq\_db.sql'. В конце вызывается метод commit, чтобы изменения применились к текущей БД, и метод close закрывает установленное соединение.

Теперь, давайте посмотрим на содержимое файла 'sq\_db.sql':

create table if not exists mainmenu (

id integer primary key autoincrement,

title text not null,

url text not null

);

Здесь записан SQL-запрос для формирования таблицы mainmenu с тремя полями: id, title и url. Их назначение вполне очевидно.

Как теперь воспользоваться этой функцией для создания начальной БД? Можно поступить так. Перейти в консоль Python и выполнить импорт функции create\_db из файла flsite.py:

from flsite import create\_db

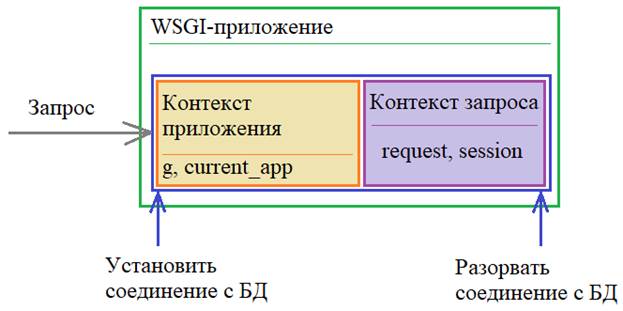
И, затем, просто вызвать эту функцию:

create\_db()

Все, в результате, в рабочем каталоге нашего приложения появится файл flsite.db с таблицей mainmenu. Мы в этом можем легко убедиться, если откроем эту БД с помощью специальной программы DB Browser SQLite.

## Подключение к БД в запросах

После создания БД мы можем ее использовать при обработке запросов. И для оптимизации работы делают так: в момент прихода запроса, устанавливается соединение с БД, а в момент завершения обработки – разрывается соединение с БД.



Момент поступления запроса мы можем «поймать» непосредственно в обработчике. Добавим обработчик главной страницы:

@app.route("/")

def index():

    db = get\_db()

    return render\_template('index.html', menu = [])

Здесь вначале вызывается вспомогательная функция get\_db, которая возвращает активное соединение с БД. Ее реализация, следующая:

def get\_db():

    '''Соединение с БД, если оно еще не установлено'''

    if not hasattr(g, 'link\_db'):

        g.link\_db = connect\_db()

    return g.link\_db

Смотрите, мы здесь используем объект g контекста приложения, которое создается в момент поступления запроса. В этом объекте можно сохранять любую пользовательскую информацию, которая будет доступна в любой функции и шаблонах, в пределах этого запроса. Причем, для разных запросов, объект g будет разным, то есть, он уникален в пределах текущего запроса.

Далее, мы проверяем: было ли соединение уже установлено (существует ли атрибут link\_db, который мы создаем в момент соединения с БД. Если соединения еще нет, то устанавливаем его и, затем, возвращаем. Как вы уже догадались, если где-либо в функциях (или шаблонах) будет повторное обращение к этой функции, то она просто возвратит ранее установленное соединение, что очень удобно.

Отлично, с БД связь установлена. Но как нам ее теперь разорвать в момент завершения запроса? Для в Flask есть специальный декоратор teardown\_appcontext, который позволяет определять функцию, вызываемую в момент уничтожения контекста приложения. А это, обычно, происходит в момент завершения обработки запроса. В итоге получаем следующий обработчик для завершения соединения с БД:

@app.teardown\_appcontext

def close\_db(error):

    '''Закрываем соединение с БД, если оно было установлено'''

    if hasattr(g, 'link\_db'):

        g.link\_db.close()

Все, теперь в нашей программе существует БД, к которой можно получить доступ в момент возникновения запроса и автоматическое завершение соединения при окончании работы с запросом.

На следующем занятии мы продолжим работу с БД и посмотрим как можно добавлять и брать данные из таблицы для их отображения на страницах сайта.

# Добавление и отображение статей из БД

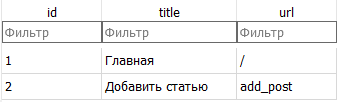
[Смотреть материал на видео](https://www.youtube.com/watch?v=Crxq-d9t_uc&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/dobavlenie-i-otobrazhenie-statey-iz-bd>

На этом занятии продолжим изучать работу с БД во Flask и давайте для начала наполним таблицу mainmenu, которую создали на прошлом занятии. Для этого воспользуемся вспомогательной программой

DB Browser for SQLite

И введем несколько записей:



Затем, в программе в обработчике главной страницы обратимся к этой БД и сформируем меню:

@app.route("/")

def index():

    db = get\_db()

    dbase = FDataBase(db)

    return render\_template('index.html', menu = dbase.getMenu())

Смотрите, я здесь для удобства создал вспомогательный класс FDataBase, который запоминает ссылку на БД и, затем, с помощью метода getMenu возвращает список для отображения меню в шаблоне index.html. Сама реализация класса, следующая. В новом файле FDataBase.py, запишем следующие строчки:

import sqlite3

class FDataBase:

    def \_\_init\_\_(self, db):

        self.\_\_db = db

        self.\_\_cur = db.cursor()

    def getMenu(self):

     sql = '''SELECT \* FROM mainmenu'''

     try:

         self.\_\_cur.execute(sql)

         res = self.\_\_cur.fetchall()

         if res: return res

     except:

         print("Ошибка чтения из БД")

     return []

Здесь в конструкторе запоминается ссылка на БД и ссылка на класс курсор, через который осуществляется взаимодействие с таблицами этой БД. Далее, идет метод getMenu и в блоке try/except осуществляется выборка всех записей из таблицы mainmenu. Если операция прошла успешно, то возвращается список словарей из записей, а иначе – пустой список.

Чтобы воспользоваться этим классом в файле flsite.py, его нужно импортировать:

from FDataBase import FDataBase

Все, теперь при запуске программы и отображения главной страницы мы увидим меню, взятое из нашей БД.

## Добавление статьи

Далее реализуем функционал по добавлению статей в БД. Для начала нам нужно создать таблицу, которая будет их хранить. Добавим в файл sq\_db.sql следующий SQL-запрос:

CREATE TABLE IF NOT EXISTS posts (

id integer PRIMARY KEY AUTOINCREMENT,

title text NOT NULL,

text text NOT NULL,

time integer NOT NULL

);

Мы здесь создаем таблицу posts с четырьмя вполне очевидными полями. Затем, в программе добавим обработчик для адреса /add\_post:

@app.route("/add\_post", methods=["POST", "GET"])

def addPost():

    db = get\_db()

    dbase = FDataBase(db)

    if request.method == "POST":

        if len(request.form['name']) > 4 and len(request.form['post']) > 10:

            res = dbase.addPost(request.form['name'], request.form['post'])

        if not res:

            flash('Ошибка добавления статьи', category = 'error')

        else:

            flash('Статья добавлена успешно', category='success')

    else:

        flash('Ошибка добавления статьи', category='error')

    return render\_template('add\_post.html', menu = dbase.getMenu(), title="Добавление статьи")

Вначале идет подключение к БД, и после этого проверка: если были переданы данные от формы методом POST, то нужно осуществить добавление статьи в таблицу posts. Для этого вначале проверяем наличие данных в полях name и post и, если все нормально, то вызываем метод addPost класса FDataBase (позже мы его пропишем). Кроме того, формируются мгновенные сообщения об успешности или ошибке при добавлении статьи. В конце возвращается шаблон 'add\_post.html', который выглядит так:

{% extends 'base.html' %}

{% block content %}

{{ super() }}

{% for cat, msg in get\_flashed\_messages(True) %}

<div class="flash {{cat}}">{{msg}}</div>

{% endfor %}

<form action="{{url\_for('addPost')}}" method="post" class="form-contact">

<label>Название статьи: </label> <input type="text" name="name" value="" requied />

<label>Текст статьи:</label>

<textarea name="post" rows=7 cols=40></textarea>

<input type="submit" value="Добавить" />

</form>

{% endblock %}

Здесь все вам уже знакомо по предыдущим занятиям. Последнее, что нам нужно сделать – это добавить метод addPost в класс FDataBase:

    def addPost(self, title, text):

        try:

            tm = math.floor(time.time())

            self.\_\_cur.execute("INSERT INTO posts VALUES(NULL, ?, ?, ?)", (title, text, tm))

            self.\_\_db.commit()

        except sqlite3.Error as e:

            print("Ошибка добавления статьи в БД "+str(e))

            return False

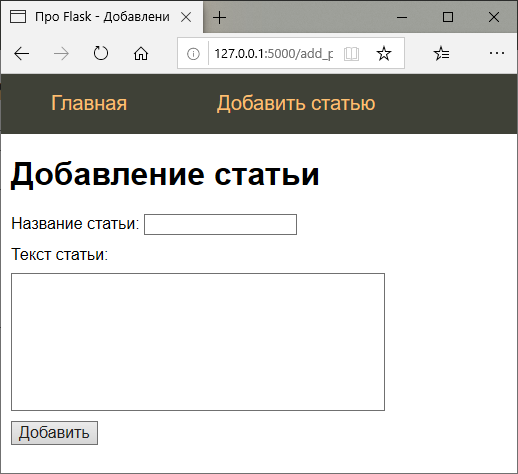
        return True

Ему передаются два аргумента: title и text. Затем, вычисляется текущее время добавления статьи (в секундах) и выполняется запрос на добавление переданных данных. После этого обязательно нужно вызвать метод commit для физического сохранения изменений в БД. Также этот метод использует два вспомогательных модуля:

import time

import math

для вычисления текущего времени. Все, теперь наша страница готова и выглядит следующим образом:



При нажатии на кнопку «Добавить» данные будут переданы обработчику /add\_post и при успешной проверке принятых значений, статья будет помещена в таблицу posts.

## Отображение статьи

После того, как статьи добавлены в БД, их можно отобразить. Как это сделать? Для начала запишем обработчик для следующего URL-адреса:

@app.route("/post/<int:id\_post>")

def showPost(id\_post):

    db = get\_db()

    dbase = FDataBase(db)

    title, post = dbase.getPost(id\_post)

    if not title:

        abort(404)

    return render\_template('post.html', menu=dbase.getMenu(), title=title, post=post)

С его помощью будут отображаться статьи с указанным id\_post. Вначале также устанавливается соединение с БД, затем, вызывается метод getPost класса FDataBase, который возвращает заголовок статьи и ее текст, а, иначе, при ошибке считывания данных из таблицы posts, формируется ответ сервера 404 – страница не найдена. В конце возвращается шаблон 'post.html' с содержимым статьи. Шаблон имеет вот такой простой вид:

{% extends 'base.html' %}

{% block content %}

{{ super() }}

{{ post }}

{% endblock %}

Наконец, метод getPost, имеет следующую реализацию:

    def getPost(self, postId):

        try:

            self.\_\_cur.execute(f"SELECT title, text FROM posts WHERE id = {postId} LIMIT 1")

        res = self.\_\_cur.fetchone()

        if res:

            return res

    except sqlite3.Error as e:

        print("Ошибка получения статьи из БД "+str(e))

        return (False, False)

Здесь все вполне очевидно. Сначала выбираются поля title и text для статьи, у которой id равен posted. Если метод fetchone возвращает не None, то есть, статья была найдена в БД, то возвращается кортеж из ее названия и текста. Иначе, возвращается кортеж из значений False.

Теперь можно перейти в браузер и набрать что-то вроде:

http://127.0.0.1:5000/post/1

Будет произведена попытка отобразить статью с id равным 1.

## Отображение списка статей

Последнее, что мы сделаем на этом занятии – это отобразим список статей на главной странице сайта. Для этого, в ее обработчике запишем следующий код:

@app.route("/")

def index():

    db = get\_db()

    dbase = FDataBase(db)

    return render\_template('index.html', menu = dbase.getMenu(), posts=dbase.getPostsAnonce())

То есть, мы здесь добавили один параметр posts, который будет ссылаться на список кортежей статей. Сам метод getPostsAnonce, имеет вид:

    def getPostsAnonce(self):

        try:

            self.\_\_cur.execute(f"SELECT id, title, text FROM posts ORDER BY time DESC")

        res = self.\_\_cur.fetchall()

        if res: return res

    except sqlite3.Error as e:

        print("Ошибка получения статьи из БД "+str(e))

        return []

Мы здесь выбираем все записи из таблицы posts и сортируем их по новизне: сначала самые свежие, затем, более позние. После этого выбираем все записи с помощью метода fetchall и при успешности этой операции, возвращаем список. Иначе, возвращается пустой список.

Осталось прописать шаблон главной страницы для отображения этого списка. Он выглядит так:

{% extends 'base.html' %}

{% block content %}

{{ super() }}

<hr>

<h2>Список статей</h2>

<ul class="list-posts">

{% for p in posts %}

<li>

<p class="title"><a href="{{ url\_for('showPost', id\_post=p.id)}}">{{p.title}}</a></p>

<p class="annonce">{{ p.text[:50]}}</p>

</li>

{% endfor %}

</ul>

{% endblock %}

Мы здесь в блоке for перебираем список posts и формируем теги li с названием статьи и ее анонсом. Чтобы все это выглядело более-менее красиво, добавим в файл styles.css следующие стили:

ul.list-posts {

     list-style: none;

     margin: 0;

     padding 0;

     max-width: 600px;

}

ul.list-posts li {

     margin: 20px 0 0 0;

     border: 1px solid *#eee;*

}

ul.list-posts .title {

     margin: 0;

     padding: 5px;

     background: *#eee;*

}

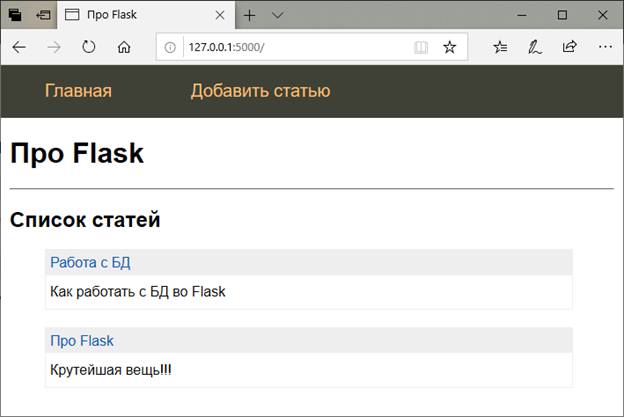
ul.list-posts .annonce {

    margin: 0;

    padding: 10px 5px 10px 5px;

}

Все, теперь, переходя на главную страницу сайта, увидим что-то вроде следующего:



Вот так в самом простом случае можно записывать данные в таблицы БД и выполнять их считывание.

# Представление полноценных HTML-страниц на сервере

[Смотреть материал на видео](https://www.youtube.com/watch?v=M5oKwDB0Hdk&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/predstavlenie-polnocennyh-html-stranic-na-servere>

На предыдущем занятии мы рассмотрели общий принцип добавления и считывания публикаций из БД. И прописали меню непосредственно в шаблоне base.html. Я верну его считывание из списка:

{% for p in menu %}

<li><a href="{{p.url}}">{{p.title}}</a></li>

{% endfor %}

И здесь у нас возникала ошибка: при отображении страницы:

http://127.0.0.1:5000/post/1

ссылка «Добавить статью» становилась следующей:

http://127.0.0.1:5000/post/add\_post

Я здесь вначале сделаю небольшую работу над ошибками и поправлю этот баг при работе с БД. Исправляется все очень просто: в таблице в поле url вместо «add\_post» нужно прописать «/add\_post». После сохранения изменений и обновления страницы сайта, все стало работать как и должно быть, ссылка стала:

http://127.0.0.1:5000/ add\_post

А теперь о главном. Все наши прежние статьи были очень простыми: заголовок и обычный текст. Реальные HTML-страницы содержат гораздо более сложную информацию: различные теги, ссылки на изображения и прочее. Как их хранить на сервере, добавлять на сайт и отдавать пользователю по определенному запросу? Вот об этом и пойдет речь на этом занятии.

В качестве примера я подготовил страницу

framework-flask-intro.htm

которая содержит множество различных изображений, расположенных в отдельном каталоге framework-flask-intro.files. И первый вопрос: где на сервере разместить эти изображения? Мы их расположим в каталоге static и в нем создадим специальный подкаталог images\_html для хранения изображений наших HTML-страниц. И в этот подкаталог скопируем каталог framework-flask-intro.files с нашими изображениями. Почему мы их разместили именно здесь? Об этом чуть позже, а пока давайте реализуем функционал для добавления самого HTML-документа в БД.

Для начала заметим, что отображать HTML-страницу по номеру ее id не лучшая практика, так как поисковые системы лучше ранжируют страницы, у которых URL имеет более читабельный вид. Например, для добавляемой страницы URL может быть такой:

домен/post/framework-flask-intro

Как видите, это более понятный для человека адрес. Так вот, чтобы наше приложение «знало» что подставлять после /post/, дополнительно в таблицу posts добавим поле url и структура таблицы будет следующая:

CREATE TABLE IF NOT EXISTS posts (

id integer PRIMARY KEY AUTOINCREMENT,

title text NOT NULL,

text text NOT NULL,

url text NOT NULL,

time integer NOT NULL

);

Обновим нашу БД, чтобы таблицы posts приняла такой вид.

## Добавление статьи в БД

Далее, в форме добавления статьи (в add\_post.html) пропишем еще одно поле ввода для url:

<form action="{{url\_for('addPost')}}" method="post" class="form-contact">

<label>Название статьи: </label> <input type="text" name="name" value="" requied />

<label>URL статьи: </label> <input type="text" name="url" value="" requied />

<label>Текст статьи:</label>

<textarea name="post" rows=7 cols=40></textarea>

<input type="submit" value="Добавить" />

</form>

А в функции-представления addPost в метод dbase.addPost передадим этот третий параметр с url:

res = dbase.addPost(request.form['name'], request.form['post'], request.form['url'])

Соответственно, сам метод в классе FDataBase поправим, следующим образом:

    def addPost(self, title, text, url):

        try:

            self.\_\_cur.execute("SELECT COUNT() as `count` FROM posts WHERE url LIKE ?", (url,))

    res = self.\_\_cur.fetchone()

    if res['count'] > 0:

        print("Статья с таким url уже существует")

        return False

            tm = math.floor(time.time())

            self.\_\_cur.execute("INSERT INTO posts VALUES(NULL, ?, ?, ?, ?)", (title, text, url, tm))

            self.\_\_db.commit()

        except sqlite3.Error as e:

            print("Ошибка добавления статьи в БД "+str(e))

            return False

        return True

Смотрите, мы здесь вначале проверяем: существует ли такой url в таблице posts, и если существует, то статья не добавляется, т.к. все статьи должны иметь уникальный url. После этой проверки происходит добавление записи.

Все статья добавляется. Давайте, перейдем на страницу добавления и пропишем там заголовок, url и скопируем HTML-текст в тело статьи. После нажатия на кнопку добавить, первая статья будет добавлена в БД.

## Отображение списка статей

Далее, нам нужно отобразить список статей. Для начала в методе getPostsAnonce добавим выборку еще по url:

self.\_\_cur.execute(f"SELECT id, title, text, url FROM posts ORDER BY time DESC")

А в шаблоне index.html, укажем его:

<p class="title"><a href="{{ url\_for('showPost', alias=p.url)}}">{{p.title}}</a></p>

Дополнительно, чтобы не отображать в анонсе теги HTML-документа, пропишем фильтр striptags:

<p class="annonce">{{ p.text[:50] | striptags }}</p>

Все, теперь при обновлении главной страницы, увидим нашу добавленную статью с анонсом.

## Отображение статьи

Осталось отобразить статью. Для начала перепишем обработчик showPost, следующим образом:

@app.route("/post/<alias>")

def showPost(alias):

    db = get\_db()

    dbase = FDataBase(db)

    title, post = dbase.getPost(alias)

    if not title:

        abort(404)

    return render\_template('post.html', menu=dbase.getMenu(), title=title, post=post)

Мы теперь после post/ указываем переменную alias, которая может принимать строковые значения. Затем, в метод getPost передаем alias из URL и по нему уже отбираем статью:

    def getPost(self, alias):

        try:

        self.\_\_cur.execute("SELECT title, text FROM posts WHERE url LIKE ? LIMIT 1", (alias,))

        res = self.\_\_cur.fetchone()

        if res:

            return res

    except sqlite3.Error as e:

        print("Ошибка получения статьи из БД "+str(e))

        return (False, False)

Если сейчас попробовать отобразить страницу, то получим также и отображение тегов, т.к. по умолчанию шаблон преобразовывает наши данные и теги заменяются специальными символами. Чтобы этого не происходило и теги страницы передавались браузеру в чистом виде, необходимо в шаблоне post.html указать фильтр safe:

{{ post | safe}}

Теперь, обновляя страницу, увидим разметку тегами. Но изображения не отображаются, т.к. к ним указан неверный путь. Поправим это.

## Прописываем пути к изображениям

В самом простом варианте мы можем поступить следующим образом. В методе getPost добавить следующую обработку HTML-страницы:

res = self.\_\_cur.fetchone()

if res:

     base = url\_for('static', filename='images\_html')

     text = re.sub(r"(?P<tag><img**\s**+[^>]\*src=)(?P<quote>[**\"**'])(?P<url>.+?)(?P=quote)>",

        "**\\**g<tag>"+base+"/**\\**g<url>>",

        res['text'])

     return (res['title'], text)

Мы здесь с помощью регулярного выражения в тексте выделяем все URL тегов img и корректируем их с поправкой на папку static, полученной с помощью функции

base = url\_for('static', filename='images\_html')

Чтобы воспользоваться модулем регулярных выражений и функцией url\_for, выполним их импорт в начале файла:

import re

from flask import url\_for

Все, теперь, при запуске мы увидим изображения в нашем HTML-документе.

Если вы не знаете как работают регулярные выражения, то смотрите [плейлист с занятиями по этой теме](https://www.youtube.com/watch?v=1SWGdyVwN3E&list=PLA0M1Bcd0w8w8gtWzf9YkfAxFCgDb09pA).

## Модификация HTML-страницы перед ее добавлением в БД

У приведенной реализации есть один серьезный недостаток: выполнять регулярное выражение для каждой отдаваемой по запросу страницы – это ресурсоемкая операция, которая негативно скажется на нагрузке сервера. Лучше это преобразование выполнять при добавлении поста в БД. Сделаем это. В методе addPost пропишем следующие строчки:

    def addPost(self, title, text, url):

        try:

            self.\_\_cur.execute("SELECT COUNT() as `count` FROM posts WHERE url LIKE ?", (url,))

     res = self.\_\_cur.fetchone()

     if res['count'] > 0:

         print("Статья с таким url уже существует")

         return False

            base = url\_for('static', filename='images\_html')

            text = re.sub(r"(?P<tag><img**\s**+[^>]\*src=)(?P<quote>[**\"**'])(?P<url>.+?)(?P=quote)>",

            "**\\**g<tag>"+base+"/**\\**g<url>>",

         text)

            tm = math.floor(time.time())

            self.\_\_cur.execute("INSERT INTO posts VALUES(NULL, ?, ?, ?, ?)", (title, text, url, tm))

            self.\_\_db.commit()

    except sqlite3.Error as e:

        print("Ошибка добавления статьи в БД "+str(e))

        return False

    return True

Смотрите, мы здесь перед добавлением текста в БД, выполняем регулярное выражение и корректируем URL-адреса для изображений. Теперь, наш документ полностью готов для отправки пользователю по запросу. Соответственно, в методе getPost регулярное выражением можно просто убрать.

Давайте для примера добавим еще одну статью и посмотрим как это будет работать.

У этого подхода есть только один существенный недостаток: если в будущем потребуется изменить путь хранения изображений, то придется переделывать все HTML-документы. Но, как правило, это редкая ситуация и в целом страницы сайта можно представлять по описанной схеме.

# Формирование ответа сервера, декораторы перехвата запроса

[Смотреть материал на видео](https://www.youtube.com/watch?v=1BZTZVqfZwc&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/formirovanie-otveta-servera-dekoratory-perehvata-zaprosa>

Прежде чем идти дальше, нам нужно разобраться со способами формирования ответа сервера во Flask. Что это вообще такое? Когда пользователь запрашивает ту или иную страницу, то браузер получает данные в виде:

<заголовок ответа><HTML-документ>

В этом заголовке находится «служебная» информация, сообщающая браузеру, как интерпретировать принятые данные. Например, там указывается код ответа:

* 200 – все нормально;
* 404 – страница не найдена;
* 301 – выполнено перенаправление с другого URL;
* 401 – доступ запрещен

и так далее. Кроме того, в заголовке в параметре content-type прописывается тип данных:

* text/html
* text/plain
* image/jpeg
* audio/mp4
* multipart/form-data

Все это имеет большое значение при разработке сайтов. На этом занятии мы как раз и поговорим о работе с этим заголовком.

Обратите внимание, заголовок ответа сервера не имеет ничего общего с заголовком HTML-страницы, то есть, с содержимым тега

<head></head>

это разные вещи. Итак, во Flask заголовок в ответ на запрос можно формировать тремя способами:

1. В обработчике возвратить строку, тогда автоматически ответом будет content-type=text/html и код 200.
2. Сформировать ответ с помощью функции make\_response()
3. Возвратить кортеж формата: (response, status, headers) или (response, headers)

Первый способ используется наиболее часто, например, вот в такой упрощенной программе:

from flask import Flask, render\_template

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

menu = [{"title": "Главная", "url": "/"},

        {"title": "Добавить статью", "url": "/add\_post"}]

@app.route("/")

def index():

    return render\_template('index.html', menu=menu, posts=[])

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

    app.run(debug=True)

Функция представления главной страницы возвращает текст (после обработки шаблона) и сервер (Flask) автоматически формирует ответ с параметрами:

* content-type=text/html
* code = 200

## Функция make\_response

Однако, если мы хотим самостоятельно определить некоторые параметры заголовка, то можно воспользоваться специальной функцией

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

которая возвращает ссылку на объект ответа. Здесь:

* res\_body – передаваемое содержимое (контент);
* status\_code – код ответа сервера (по умолчанию 200).

Предположим, что мы хотим отправить браузеру страницу в виде обычного текста. Для этого создадим следующий ответ:

@app.route("/")

def index():

    content = render\_template('index.html', menu=menu, posts=[])

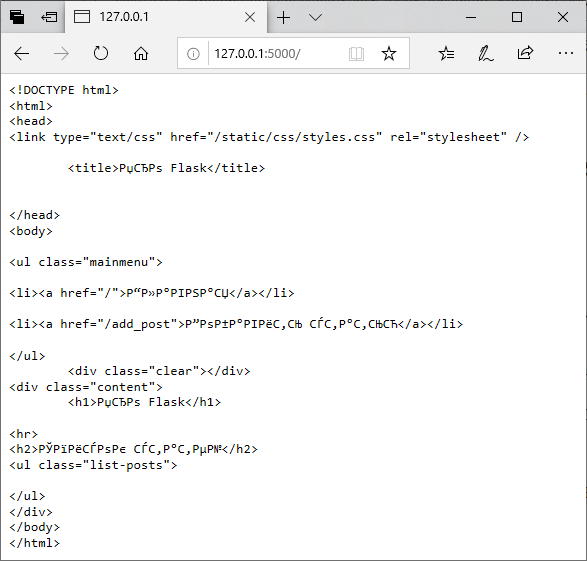
    res = make\_response(content)

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

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

    return res

Смотрите, здесь параметр 'Content-Type' принимает значение 'text/plain', которое указывает браузеру отображать данные в виде обычного текста. После обновления главной страницы, увидим следующее:



Вся страница отображена в виде текста. Или, можно сделать так:

@app.route("/")

def index():

    img = None

    with app.open\_resource( app.root\_path + "/static/images/ava.png", mode="rb") as f:

       img = f.read()

    if img is None:

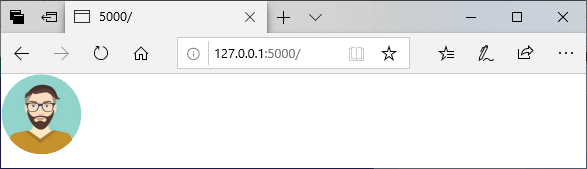
        return "None image"

    res = make\_response(img)

    res.headers['Content-Type'] = 'image/png'

    return res

Мы здесь загружаем изображение из подкаталога static/images и отдаем его браузеру, указывая, что принятые данные следует представлять в виде png изображения. И, действительно, при обновлении страницы, увидим загруженное изображение:



Или, можно вернуть страницу с определенным кодом, отличным от 200, например:

@app.route("/")

def index():

    res = make\_response("<h1>Ошибка сервера</h1>", 500)

    return res

Здесь генерируется ошибка 500 – внешняя ошибка сервера с заголовком h1. Вот так заголовок кардинально влияет на отображение данных в окне браузера.

С помощью функции make\_response можно также передавать информацию для cookies браузера, то есть, информация, которая будет храниться в браузере клиента и передаваться серверу при каждом очередном запросе к нашему сайту. Но подробнее об этом мы поговорим на следующем занятии.

## Создание ответа с помощью кортежей

Последний способ создать ответ – использовать кортежи 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. Например, так:

@app.errorhandler(404)

def pageNot(error):

    return ("Страница не найдена", 404)

Мы здесь формирует ответ и указываем код ответа – 404 страница не найдена. Или, так:

@app.route("/")

def index():

    return "<h1>Main Page</h1>", 200, {'Content-Type': 'text/plain'}

Здесь после оператора return записан тоже кортеж только без круглых скобок. Последний параметр headers – словарь из набора параметров заголовка.

## Создание 301 и 302 редиректов

Очень часто при развитии сайта некоторые его страницы переносятся на другой URL-адрес. И, чтобы не потерять позиции этих страниц в поисковой выдаче, поисковым системам нужно явно указать, что страница перемещена либо временно, либо постоянно на новый URL. Это делается с помощью перенаправления с кодами:

* 301 – страница перемещена на другой постоянный URL-адрес;
* 302 – страница перемещена временно на другой URL-адрес.

Чтобы во Flask выполнить перенаправление с прежнего URL на новый, можно использовать функцию

redirect(location, status)

о которой мы уже говорили. В данном случае ее можно применить так:

@app.route('/transfer')

def transfer():

    return redirect(url\_for('index'), 301)

## Перехват запросов

Последнее, о чем мы поговорим на этом обзорном занятии – это перехват запросов. Что это такое? В веб-приложениях часто нужно исполнить определенный код до или после запроса. И, как мы видели, в частности, это используется для установления соединения с БД. Нам в каждой функции представления приходилось осуществлять подключение к БД, следующим образом:

@app.route("/")

def index():

    db = get\_db()               *# функции*

    dbase = FDataBase(db)       *# подключения к БД*

    return render\_template('index.html', menu=dbase.getMenu(), posts=dbase.getPostsAnonce())

И это не самое лучшее решение, оно нарушает один из основополагающих принципов программирования:

DRY – Don’t Repeat Yourself (не повторяйся)

Так вот, чтобы вынести за скобки этот общий код, можно воспользоваться специальными декораторами для перехвата запросов:

* before\_first\_request – выполняет функцию до обработки первого запроса;
* before\_request – выполняет функцию до обработки текущего запроса;
* after\_request – выполняет функцию после обработки запроса (такая функция не вызывается при возникновении исключений в обработчике запросов);
* teardown\_request (похож на after\_request) – вызванная функция всегда будет выполняться вне зависимости от того, возвращает ли обработчик исключение (ошибку) или нет.

Давайте для примера добавим эти обработчики в нашу программу и посмотрим последовательность их вызовов:

@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.teardown\_request

def teardown\_request(response):

    print("teardown\_request() called")

    return response

Обратите внимание, первые две функции только выполняют определенные действия, не возвращая никакого значения, а последние две – принимают объект response и возвращают его. Запустим эту программу и при переходе на главную страницу сайта увидим следующие строки в консоли приложения:

before\_first\_request() called

before\_request() called

after\_request() called

teardown\_request() called

Обновим страницу и теперь увидим только три вызова:

before\_request() called

after\_request() called

teardown\_request() called

При последующих вызовах (любых страниц) функция before\_first\_request вызвана уже не будет.

В последующих занятиях, развивая наш тестовый сайт, мы воспользуемся декоратором before\_request, чтобы настраивать связь с БД до выполнения самого запроса.

# Порядок работы с cookies

[Смотреть материал на видео](https://www.youtube.com/watch?v=H0TxIqb9rts&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/poryadok-raboty-s-cookies>

Продолжим изучение возможностей фреймворка Flask, которые позже понадобятся для развития нашего тестового сайта. И следующее, о чем мы поговорим – это возможность передавать и сохранять информацию в браузере клиента в, так называемых, cookies (куках).

## Сохранение информации в cookies браузера

Итак, с помощью функции make\_response, которую мы рассмотрели на прошлом занятии, можно передавать информацию для cookies браузера, то есть, информация, которая будет храниться в браузере клиента и передаваться серверу при каждом очередном запросе к нашему сайту. Cookies очень часто применяются при создании сайтов. Например, в формах авторизации очень часто имеется галочка «Запомнить», которая позволяет запомнить авторизацию текущего пользователя:



Как раз это делается с помощью cookies: в них сохраняется хеш авторизации и, затем, при отправке запроса, на стороне сервера можно посмотреть наличие и корректность этого хэша. В случае успешной проверки пользователю не придется постоянно выполнять авторизацию.

Итак, чтобы работать с cookies через объект ответа, используется функция

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

* key – название куки;
* value – данные, которые сохраняются в cookies под указанным ключом;
* max\_age – необязательный аргумент, указывающий предельное время хранения данных cookies в барузере клиента (в секундах). Если время не указывается, то куки пропадут при закрытии браузера.

Например, этой функцией можно воспользоваться так:

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

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

menu = [{"title": "Главная", "url": "/"},

        {"title": "Добавить статью", "url": "/add\_post"}]

@app.route("/")

def index():

    return "<h1>Main Page</h1>"

@app.route("/login")

def login():

    log = ""

    if request.cookies.get('logged'):

        log = request.cookies.get('logged')

    res = make\_response(f"<h1>Форма авторизации</h1>logged: {log}")

    res.set\_cookie("logged", "yes")

    return res

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

    app.run(debug=True)

Мы здесь вначале проверяем в функции представления login: переданы ли cookies вместе с запросом к странице и если это так, то переменная log будет ссылаться на значение с ключом 'logged'. Далее, формируется объект ответа и, затем, вызывается метод set\_cookie этого объекта. Через него устанавливается ключ logged со значение yes. В результате, при первом запросе формы в браузер клиента будет записано значение yes и, при повторном обращении, объект reques будет ссылаться на cookies с этим ключом.

Если же требуется сохранять куки в пределах, например, месяца, то это указывается третьим параметром:

res.set\_cookie("logged", "yes", 30\*24\*3600)

Теперь для нашего сайта при отправке каждого запроса браузер автоматически будет добавлять информацию из cookies, которая станет доступной на сервере.

## Удаление cookies

Если требуется удалить определенную информацию из cookies, то для этого в методе set\_cookie достаточно установить параметр

max\_age = 0

Например, так:

@app.route("/logout")

def logout():

    res = make\_response("Вы больше не авторизованы!</p>")

    res.set\_cookie("logged", "", 0)

    return res

## Ограничения cookies

Перед использованием cookies в реальном проекте нужно знать об их особенностях. Во-первых, куки небезопасны и все данные, которые в них попадают, видны всем, поэтому в них нельзя хранить пароли, данные банковских карт и так далее. Во-вторых, в большинстве браузеров cookies можно отключать и тогда сохранять в них данные уже не получится. Причем, сервер об этом знать не будет, метод set\_cookie отработает штатно, просто данные не будут размещены в браузере и сайт потеряет часть своего функционала.

Для решения этой проблемы, часто в HTML-страницах размещают JavaScript-код, который проверяет работоспособность cookies и при их отключении сообщает пользователю, что сайт будет некорректно работать в этих условиях:

<script>

    document.cookie = "ex=1;";

    if (!document.cookie)

    {

         alert("Это сайт требует включенных cookies для своей корректной работы ");

    }

</script>

Также следует знать, что на cookies накладывается ограничение по размеру в 4 Кб на каждый ключ и количеству ключей в расчете на один сайт. Обычно оно составляет от 30 до 50.

Разумеется, размеры cookies нужно делать минимальными еще и по той причине, что они добавляются в каждый запрос к серверу. Представьте, если каждый раз из-за cookies будет отправляться дополнительно:

4∙25 = 100 Кб

информации. Это не самая лучшая реализация. Нужно стараться минимизировать эту информацию.

На этом мы завершим данное занятие по кукам. На следующем продолжим эту тему и поговорим об еще одном похожем хранилище – сессиях.

# Порядок работы с сессиями (session)

[Смотреть материал на видео](https://www.youtube.com/watch?v=Bx5PRzS9p44&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/poryadok-raboty-s-session>

На этом занятии мы поближе познакомимся с механизмом сессий, который уже затрагивали, когда говорили о мгновенных сообщениях. Так вот, сессии работают по похожему с cookies принципу: они также хранятся в браузере в виде особых кук сессий, но дополнительно шифруются с помощью ключа, который можно задать в WSGI-приложении, следующим образом:

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

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

app.config['SECRET\_KEY'] = 'cb02820a3e94d72c9f950ee10ef7e3f7a35b3f5b'

Для надежности этот ключ должен содержать самые разные символы и один из хороших способов его сгенерировать – это воспользоваться следующей командой пакета os:

os.urandom(20).hex()

То есть, мы можем ее выполнить, например, в консоли Python и результат вставить в секретный ключ. Соответственно, не зная этот ключ, пользователь не сможет понять и изменить сессионную информацию в браузере. Фактически, только этим сессии и отличаются от cookies, о которых мы говорили на предыдущем занятии. Во всем остальном они практически не отличимы: также передаются в запросах браузеру и обратно – из браузера к серверу. Имеют ограничение в 4 Кб хранимой информации и перестают работать, если пользователь отключит cookies в своем браузере. Но, вместе с тем, почти ни один серьезный сайт без них не обходится.

## Объект session

Для работы с сессиями во Flask имеется специальный объект

session

который доступен и в приложении и в шаблонах. Данный объект работает по подобию словаря, следующим образом:

@app.route("/")

def index():

    if 'visits' in session:

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

    else:

        session['visits'] = 1  *# запись данных в сессию*

    return f"<h1>Main Page</h1>Число просмотров: {session['visits']}"

Смотрите, мы здесь обращаемся к данным сессии по ключу 'visits', если они там есть. Иначе, создаем этот ключ со значением 1. Далее, при каждом обновлении страницы значение 'visits' будет увеличиваться на 1 и мы увидим увеличение числа просмотров главной страницы сайта. Это лишь простой пример, демонстрирующий порядок работы с объектом session во Flask. Шифрование, передача и прием данных через запросы скрыта и происходит в автоматическом режиме, что очень удобно.

Но при работе с сессиями есть одна существенная особенность, которую следует знать и хорошо понимать: передача сессионных данных браузеру происходит только при изменении состояния объекта session. Чтобы лучше понять о чем речь, рассмотрим такой отвлеченный, но простой пример:

data = [1,2,3,4]

@app.route("/session")

def session\_data():

    if 'data' not in session:

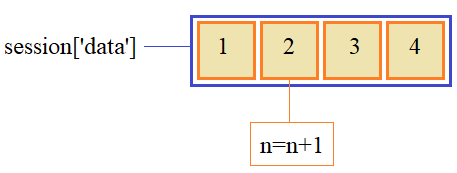
        session['data'] = data

    else:

        session['data'][1] += 1

    return f"session['data']: {session['data']}"

Мы здесь при первом запросе формируем поле data, которое ссылается на список data. А при последующих запросах происходит увеличение второго элемента этого списка на 1. По идее, при обновлении страницы, мы должны видеть постоянное увеличение второго значения на 1. Давайте посмотрим, обновляем, но ничего не происходит. Почему? Как раз по той причине, что объект session в этом случае никак не меняется: он как ссылался на список, так и ссылается (адрес списка остается прежним). Меняется лишь элемент в самом списке и это изменение не влияет на изменение session. А раз так, то Flask решает, что сессия не поменялась и нет смысла нагружать канал связи и дополнительно ее отправлять браузеру.



Но это можно поправить, прописав следующую строчку:

session.modified = True

Этим мы явно указываем Flask, что состояние сессии изменилось и ее нужно обновить в браузере. Теперь, при обновлении страницы, второй элемент списка будет увеличиваться каждый раз на единицу.

## Время жизни сессии

По умолчанию куки сессии существуют, пока пользователь не закроет браузер. Но это поведение можно изменить и явно указать время жизни сессий в куках. Для этого в функции представления нужно указать:

session.permanent = True

Тогда время жизни сессии устанавливается равным параметру

app.permanent\_session\_lifetime

который по умолчанию составляет 31 день. Но, можно определить любое другое значение, например, так:

app.permanent\_session\_lifetime = datetime.timedelta(days=10)

Теперь сессии будут максимум храниться 10 дней в браузере клиента. Давайте посмотрим на работу этого функционала. Если сейчас произвести обновления страницы

127.0.0.1:5000/session

то увидим увеличение второго значения списка. Выйдем из браузера, снова зайдем и увидим последнее установленное значения, т.е. сессия сохраняется в браузере. Далее, изменим значение:

session.permanent = False

также обновим страницу, выйдем из браузера и при возвращении увидим начальные значения списка 1, 2, 3, 4, т.е. сессия пропала при закрытии окна браузера. Вот так это работает.

# Регистрация пользователей и шифрование паролей

[Смотреть материал на видео](https://www.youtube.com/watch?v=Q0ztepEQ6b8&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/registraciya-polzovateley-i-shifrovanie-paroley>

После некоторого отступления на прошлых занятиях, посвященных обработке, перехвату запросов, а также кукам и сессиям, я возвращаюсь к нашему тестовому сайту и продолжу его совершенствование.

У нас с вами были сформированы функции представления для главной страницы, а также для добавления статьи и ее отображения. Во всех этих функциях имеется общий код соединения с БД, который следует вынести как бы за скобки и описать в функции декоратора before\_request:

dbase = None

@app.before\_request

def before\_request():

    """Установление соединения с БД перед выполнением запроса"""

   global dbase

    db = get\_db()

    dbase = FDataBase(db)

А в функциях обработчиках запросов это соединение просто использовать.

## Подготовка форм авторизации и регистрации

Теперь предположим, что на нашем сайте мы хотели бы выполнять регистрацию и авторизацию пользователей. С чего здесь следует начать? Для начала добавим пункт в главное меню «Авторизация», т.е. в БД таблицы mainmenu следует создать запись:

Авторизация, /login

И, затем, добавим функцию представления для URL /login:

@app.route("/login")

def login():

    return render\_template("login.html", menu=dbase.getMenu(), title="Авторизация")

Создадим шаблон login.html:

{% extends 'base.html' %}

{% block content %}

{{ super() }}

{% for cat, msg in get\_flashed\_messages(True) %}

<div class="flash {{cat}}">{{msg}}</div>

{% endfor %}

<form action="/login" method="post" class="form-contact">

<label>Email: </label> <input type="text" name="email" value="" requied />

<label>Пароль: </label> <input type="password" name="psw" value="" requied />

<input type="checkbox" name="remainme" /> Запомнить меня

<input type="submit" value="Войти" />

<hr align=left width="300px">

<a href="{{url\_for('register')}}">Регистрация</a>

</form>

{% endblock %}

Здесь у нас простая форма авторизации по Email и паролю, а также имеется ссылка на страницу регистрации, если пользователь еще не зарегистрирован.

Соответственно, в программе нужно добавить обработчик для регистрации:

@app.route("/register")

def register():

    return render\_template("register.html", menu=dbase.getMenu(), title="Регистрация")

И прописать шаблон register.html:

{% extends 'base.html' %}

{% block content %}

{{ super() }}

{% for cat, msg in get\_flashed\_messages(True) %}

<div class="flash {{cat}}">{{msg}}</div>

{% endfor %}

<form action="/register" method="post" class="form-contact">

<label>Имя: </label> <input type="text" name="name" value="" requied />

<label>Email: </label> <input type="text" name="email" value="" requied />

<label>Пароль: </label> <input type="password" name="psw" value="" requied />

<label>Повтор пароля: </label> <input type="password" name="psw2" value="" requied />

<input type="submit" value="Регистрация" />

</form>

{% endblock %}

Здесь все очень похоже на форму авторизации, только прописаны другие поля и обработка данных по запросу /register.

После всего этого подготовительного этапа, пришло время создать таблицу в БД, которая бы хранила данные о зарегистрированных пользователях. Мы ее сделаем простой: назовем users и определим следующие поля:

CREATE TABLE IF NOT EXISTS users (

id integer PRIMARY KEY AUTOINCREMENT,

name text NOT NULL,

email text NOT NULL,

psw text NOT NULL,

time integer NOT NULL

);

Создадим ее в БД, выполним функцию create\_db в консоли Python:

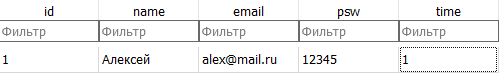
from flsite import create\_db

create\_db()

Если кто не помнит, мы эту функцию создавали ранее, когда говорили об использовании БД во Flask. Я не буду здесь останавливаться на этом моменте, если есть необходимость, просто посмотрите это видео.

## Шифрование паролей в БД

Итак, мы создали таблицу users в БД flsite и пришла пора ее наполнить, т.е. написать функционал, который бы добавлял нового пользователя в эту таблицу. Как это правильно сделать? Если мы будем создавать записи один в один с введенными значениями в форме регистрации, то в поле psw пароли будут храниться в открытом (не защищенном) виде:



Такие данные быстро утекут, чего бы вы не делали: вашему системному администратору сделают предложение, от которого он не сможет отказаться и передаст всю информацию злоумышленнику. Или он сам ими воспользуется. Короче, хранить приватные данные в открытом виде в БД – это верх безрассудства, их нужно зашифровать. Благо для этого была придумано масса математических алгоритмов и во Flask можно использовать один из них, определяемый стандартом

PBKDF2 (Password-Based Key Derivation Function)

который доступен из пакета

Werkzeug

устанавливаемый совместно с Flask.

Общий принцип шифрования информации у всех алгоритмов един: на выходе формируется на первый взгляд случайная последовательность символов, которая уникальная для исходных кодируемых данных. Например, для пароля 12345 получим следующую последовательность по стандарту PBKDF2:

pbkdf2:sha256:150000$Xi7Spouc$60425441c622d897c2ae4dce97ee1a5a3f522c0bbb4f72c87c15f3d0fcc227b1

В шифровании это называется хешем закодированных данных. И глядя на этот хеш, совершенно непонятно что в нем записано. Причем, нет обратного алгоритма декодирования, который бы из этой строки получал пароль 12345. Поэтому, у злоумышленников здесь только один вариант – полный перебор. То есть, перебирать все возможные варианты паролей, кодировать их и сравнивать полученный хеш с хешем в БД. Но это очень долгий и сложный путь. Поэтому так важно им знать приватную информацию в чистом виде. И мы им такой радости не доставим, все нужное будем кодировать! А как мы тогда на сервере проверим, например, корректность ввода пароля? Для этого, пароль, указанный пользователем подвергается шифрованию и, затем, сравниваются между собой их хеши. Если они равны, значит, пароль верный.

Итак, как во Flask выполнять кодирование и последующую проверку пароля. Для этого из пакета Werkzeug нужно импортировать следующие функции:

from werkzeug.security import generate\_password\_hash, check\_password\_hash

* generate\_password\_hash() – выполняет кодирование строки данных по стандарту PBKDF2;
* check\_password\_hash() – выполняет проверку указанных данных на соответствие хеша.

Например, чтобы закодировать пароль, достаточно выполнить:

hash = generate\_password\_hash("12345")

а для проверки:

check\_password\_hash(hash, "12345")

Последняя функция вернет True, если хеши совпадают и False в противном случае.

## Добавление пользователя в БД

Теперь, когда мы знаем как шифровать данные, запишем в обработчик функционал для добавления пользователя в таблицу users. Вначале нужно импортировать функции для шифрования и проверки паролей:

from werkzeug.security import generate\_password\_hash, check\_password\_hash

А затем, реализовать обработчик формы. Я не буду здесь усложнять программу и представлю простой обработчик для данных формы регистрации:

@app.route("/register", methods=["POST", "GET"])

def register():

    if request.method == "POST":

        session.pop('\_flashes', None)

        if len(request.form['name']) > 4 and len(request.form['email']) > 4 \

            and len(request.form['psw']) > 4 and request.form['psw'] == request.form['psw2']:

            hash = generate\_password\_hash(request.form['psw'])

            res = dbase.addUser(request.form['name'], request.form['email'], hash)

            if res:

                flash("Вы успешно зарегистрированы", "success")

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

            else:

                flash("Ошибка при добавлении в БД", "error")

        else:

            flash("Неверно заполнены поля", "error")

    return render\_template("register.html", menu=dbase.getMenu(), title="Регистрация")

Здесь все достаточно очевидно и нам нужно еще прописать метод addUser во вспомогательном классе FDataBase:

    def addUser(self, name, email, hpsw):

        try:

            self.\_\_cur.execute(f"SELECT COUNT() as `count` FROM users WHERE email LIKE '{email}'")

            res = self.\_\_cur.fetchone()

            if res['count'] > 0:

                print("Пользователь с таким email уже существует")

                return False

            tm = math.floor(time.time())

            self.\_\_cur.execute("INSERT INTO users VALUES(NULL, ?, ?, ?, ?)", (name, email, hpsw, tm))

            self.\_\_db.commit()

        except sqlite3.Error as e:

            print("Ошибка добавления пользователя в БД "+str(e))

            return False

        return True

Сначала идет проверка на уникальность email для пользователя и если указанного email не существует, то пользователь добавляется в БД.

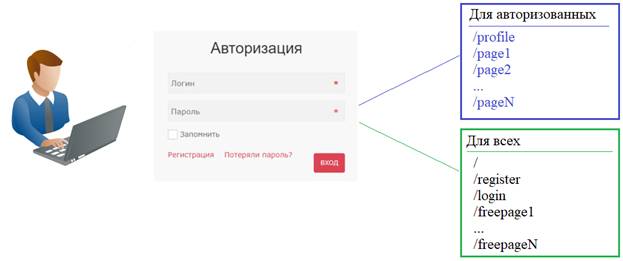
# Авторизация пользователей на сайте через Flask-Login

[Смотреть материал на видео](https://www.youtube.com/watch?v=24vVFtwuBWs&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/avtorizaciya-polzovateley-na-sayte-cherez-flask-login>

Файл проекта: <https://github.com/selfedu-rus/flasksite-16>

На предыдущем занятии мы с вами добавили регистрацию пользователей. Теперь пришло время сделать их авторизацию. Общий смысл авторизации заключается в том, что на сайте имеются страницы, которые доступны как всем пользователям, так и те, которые доступны только авторизованным пользователям.



Конечно, мы можем самостоятельно реализовать весь этот функционал, используя сессии и куки, о работе которых вы уже знаете. Но это долгий путь. И, так как задача типовая, то для Flask был разработан специальный модуль

Flask-Login

который значительно упрощает эту задачу. Им мы и воспользуемся. Сначала его нужно установить. Это делается с помощью команды:

pip install flask-login

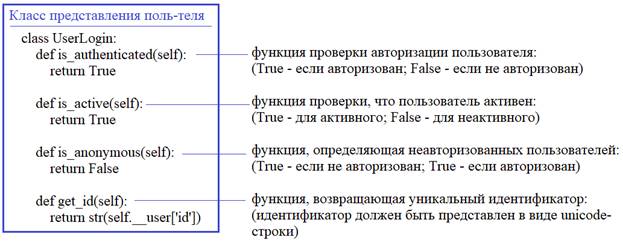
И, далее, импортировать его, а точнее, класс LoginManager, который управляет процессом авторизации:

from flask\_login import LoginManager

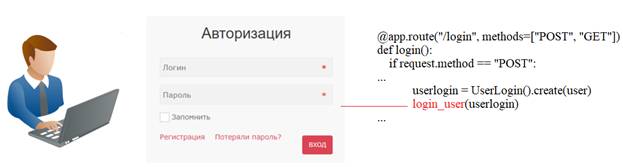
Ну и, затем, создать экземпляр этого класса, например, так:

login\_manager = LoginManager(app)

Чтобы было понятнее дальнейшее изложение, я в двух словах опишу общую концепцию работы этого расширения. Для его работы необходимо создать отдельный класс, который будет описывать состояние текущего пользователя: статус авторизации, активности и способ определения уникального идентификатора. В самом простом варианте этот класс можно объявить, например, так:



Далее, в обработчике авторизации пользователя создается объект класса UserLogin и передается специальной функции login\_user, которая определена в модуле Flask-Login:



Данная функция заносит в сессию информацию о зарегистрированном пользователе, используя определенные в классе методы. После этого сессионная информация будет присутствовать во всех запросах к серверу. Для ее обработки во Flask-Login определен специальный декоратор:

@login\_manager.user\_loader

def load\_user(user\_id):

    print("load\_user")

    return UserLogin().fromDB(user\_id, dbase)

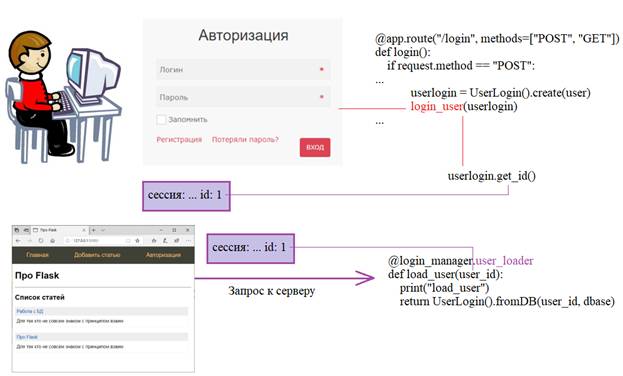
Причем функции load\_user этого декоратора будет передан идентификатор пользователя, который присутствует в сессии запроса. Фактически, это будет user\_id, который возвращается методом get\_id класса UserLogin.

Зная этот идентификатор, мы можем обратиться к БД, прочитать данные этого пользователя и создать экземпляр класса UserLogin на основе прочитанных данных. Благодаря этому, у нас при каждом запросе в системе гарантированно будет объект класса UserLogin, с которым мы сможем в дальнейшем спокойно работать.

Также следует иметь в виду, что функция декоратора user\_loader вызывается после функции декоратора

before\_query

это гарантирует нам наличие подключения к БД в функции load\_user, в которой используется переменная dbase.



Когда весь этот механизм реализован, мы очень легко можем ограничивать доступ к разным страницам сайта. Например, сделаем так, чтобы статьи можно было просматривать только авторизованным пользователям. Для этого перед функцией представления добавляется декоратор login\_required:

@app.route("/post/<alias>")

@login\_required

def showPost(alias):

…

И теперь страница доступна только авторизованным посетителям сайта.

## Начальная реализация авторизации

Давайте теперь в нашей программе реализуем эту концепцию. После создания экземпляра класса LoginManager добавим в наш проект еще один файл UserLogin.py, в котором пропишем вспомогательный класс UserLogin:

class UserLogin:

    def fromDB(self, user\_id, db):

        self.\_\_user = db.getUser(user\_id)

        return self

    def create(self, user):

        self.\_\_user = user

        return self

    def is\_authenticated(self):

        return True

    def is\_active(self):

        return True

    def is\_anonymous(self):

        return False

    def get\_id(self):

        return str(self.\_\_user['id'])

Здесь первый метод fromDB используется при создании объекта в декораторе user\_loader. Он по user\_id выполняет загрузку пользовательских данных из БД и сохраняет в частном свойстве \_\_user. Второй метод create используется при создании объекта в момент авторизации пользователя. Вся информация о нем уже известна и мы ее просто передаем по ссылке user и также сохраняем в частной переменной \_\_user. Эта информация потом пригодится в методе get\_id, который возвращает id текущего пользователя.

Далее, нам нужно сразу же определить метод getUser в классе FDataBase. Он будет следующий:

    def getUser(self, user\_id):

        try:

            self.\_\_cur.execute(f"SELECT \* FROM users WHERE id = {user\_id} LIMIT 1")

            res = self.\_\_cur.fetchone()

            if not res:

                print("Пользователь не найден")

                return False

            return res

        except sqlite3.Error as e:

            print("Ошибка получения данных из БД "+str(e))

        return False

Все предельно просто. Мы извлекаем из таблицы users все поля для указанного id пользователя и, затем, возвращаем прочитанные данные. Если данные прочитать не удалось, то метод возвращает значение False.

После объявления класса UserLogin, мы можем его использовать в нашей основной программе. Сначала сделаем его импорт:

from UserLogin import UserLogin

а, затем, пропишем декоратор

@login\_manager.user\_loader

def load\_user(user\_id):

    print("load\_user")

    return UserLogin().fromDB(user\_id, dbase)

Как мы отмечали, здесь будет создаваться объект UserLogin при каждом запросе, если пользователь авторизован. Если не авторизован, то функция вызываться не будет.

Ну и, наконец, самое главное – прописать обработчик для авторизации пользователя. Я его приведу в таком упрощенном виде:

@app.route("/login", methods=["POST", "GET"])

def login():

    if request.method == "POST":

        user = dbase.getUserByEmail(request.form['email'])

        if user and check\_password\_hash(user['psw'], request.form['psw']):

            userlogin = Use

rLogin().create(user)

            login\_user(userlogin)

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

        flash("Неверная пара логин/пароль", "error")

    return render\_template("login.html", menu=dbase.getMenu(), title="Авторизация")

Здесь сначала идет проверка: если данные были переданы из формы авторизации, то мы обращаемся к БД и считываем информацию о пользователе, опираясь на указанный в форме email. Если данные из БД были прочитаны успешно и верно введен пароль, то формируется объект класса UserLogin и вызывается функция login\_user. С этого момента пользователь считается авторизованным. Далее идет перенаправление на главную страницу сайта. Если какие-либо проверки не прошли, то формируется мгновенное сообщение и снова отображается форма авторизации.

И последнее, что мы сделаем перед пробным запуском, это ограничим доступ к страницам статей неавторизованным посетителям:

@app.route("/post/<alias>")

@login\_required

def showPost(alias):

    title, post = dbase.getPost(alias)

    if not title:

        abort(404)

    return render\_template('post.html', menu=dbase.getMenu(), title=title, post=post)

Все, теперь при запуске мы можем авторизовываться и просматривать статьи только в этом состоянии.

# Улучшение процесса авторизации (Flask-Login)

[Смотреть материал на видео](https://www.youtube.com/watch?v=L_o0wRaZJdg&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/uluchshenie-processa-avtorizacii-flask-login>

Файл проекта: <https://github.com/selfedu-rus/flasksite-16>

Давайте теперь немного улучшим наш функционал и добавим добавим страницу профайла пользователя. Чтобы сейчас не грузиться информацией, я ее сделаю примитивной:

@app.route('/profile')

@login\_required

def profile():

    return f"""<a href="{url\_for('logout')}">Выйти из профиля</a>

                user info: {current\_user.get\_id()}"""

Мы здесь используем специальную своего рода глобальную переменную current\_user (для простоты ее можно так воспринимать), через которую можно обращаться к методам класса UserLogin. В частности, используется метод get\_id для получения идентификатора авторизованного пользователя. И также добавлена ссылка для выхода из профиля. Чтобы она заработала, необходимо прописать функцию представления logout:

@app.route('/logout')

@login\_required

def logout():

    logout\_user()

    flash("Вы вышли из аккаунта", "success")

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

В этой функции мы вызываем специальный метод logout\_user(), который очищает сессию и делает пользователя не авторизованным. Затем, формируется мгновенное сообщение и перенаправление на форму авторизации.

Далее, реализуем возможность оставаться в системе после закрытия окна браузера. Для этого нужно прочитать данные для checkbox’a «Запомнить меня»:

rm = True if request.form.get('remainme') else False

и при вызове функции login\_user передать это значение параметру remember:

login\_user(userlogin, remember=rm)

Все, теперь пользователь браузер будет запоминать пользователя.

## Следующий шаг улучшения

Предположим, что мы не авторизованы и пытаемся посетить закрытую страницу. В этом случае сервер возвратит код 401 – страница недоступна. Но это не лучшее поведение сайта. Более дружественный интерфейс должен перенаправлять пользователя, например, на форму авторизации. Это делается следующей строчкой:

login\_manager.login\_view = 'login'

то есть, мы здесь атрибуту login\_view присваиваем имя функции представления для формы авторизации. И, далее, пропишем еще две такие строки:

login\_manager.login\_message = "Авторизуйтесь для доступа к закрытым страницам"

login\_manager.login\_message\_category = "success"

Они будут формировать мгновенное сообщение для таких ситуаций.

Наконец, последний важный штрих. Если авторизованный пользователь перейдет по адресу /login, то он увидит форму авторизации, что как-то неестественно – он же уже вошел на сайт! Поэтому в обработчике login в самом начале можно сделать такую проверку:

    if current\_user.is\_authenticated:

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

Снова обращаемся к глобальному объекту current\_user и проверяем: является ли пользователь авторизованным. Если это так, то делаем автоматическое перенаправление на страницу профайла.

Но, что если неавторизованный пользователь попытается просмотреть материал статьи? Тогда мы тоже попадем на страницу авторизации, но после этого произойдет перенаправление в профайл. Но лучше делать перенаправление на исходную страницу – со статьей. Сделаем это.

Смотрите, когда происходит перенаправление, Flask в строке запроса автоматически добавляет специальный параметр next, который ссылается на предыдущую страницу, с которой ушли. Воспользуемся им и в шаблоне login.html пропишем вот такие строчки:

<form action="" method="post" class="form-contact">

Здесь происходит дублирование обратного URL-адреса, который был в браузере. Далее, в обработчике /login сделаем перенаправление с учетом наличия этого параметра:

return redirect(request.args.get("next") or url\_for("profile"))

Мы перейдем либо на предыдущую страницу, либо в профайл, если параметр next отсутствует. Все, вот таким образом, в самом простом варианте мы реализовали авторизацию пользователя на нашем сайте. Но, несмотря на эту простоту, здесь продемонстрированы основные подходы и возможности модуля Flask-Login. И теперь вы сможете использовать его на своих сайтах.

## Самый последний штрих – UserMixin

И последний, момент, который я все-таки решил вставить в это занятие – это вспомогательный класс UserMixin, который по умолчанию реализует методы: is\_authenticated, is\_active, is\_anonymous и, потому, наш класс UserLogin можно записать в несколько более кратком виде:

from flask\_login import UserMixin

class UserLogin(UserMixin):

    def fromDB(self, user\_id, db):

        self.\_\_user = db.getUser(user\_id)

        return self

    def create(self, user):

        self.\_\_user = user

        return self

    def get\_id(self):

        return str(self.\_\_user['id'])

При этом весь функционал сохраняется без изменений.

Разумеется, учесть и рассказать обо всех нюансах авторизации с помощью модуля Flask-Login в видеоуроках просто нереально. Я привел общий принцип пользования этим пакетом. Дополнительную информацию можно посмотреть на сайте официальной документации:

<https://flask-login.readthedocs.io/en/latest/>

# Загрузка файлов на сервер и сохранение в БД

[Смотреть материал на видео](https://www.youtube.com/watch?v=ICKN_R0wGiI&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/zagruzka-faylov-na-server-i-sohranenie-v-bd>

Файл проекта: <https://github.com/selfedu-rus/flasksite-17>

На этом занятии рассмотрим порядок загрузки файлов на сервер. Когда это бывает необходимо? Например, пользователь в своем профайле хочет сменить аватарку. Для этого он должен выбрать изображение на своем устройстве и загрузить его на сервер. Далее, на стороне сервера происходит обработка принятого файла и связывается с аккаунтом текущего пользователя.



Здесь может возникнуть вопрос: зачем нам создавать отдельный обработчик upload для загрузки файла? Почему бы его не реализовать сразу в profile? Дело в том, что если не разнести эти обработчики, то при обновлении страницы profile автоматически произойдет повторная загрузка этого же изображения, т.к. браузеры при refresh страницы дублируют всю информацию, в том числе и отправку данных формы. Чтобы этого не происходило, как раз и создается отдельный загрузчик, данные из формы направляются к нему, а затем, осуществляется автоматическое перенаправление в профайл. В этом случае обновление profile приведет просто к обновлению страницы без отправки данных формы.

Добавим этот функционал на наш сайт. Загруженную аватарку пользователя будем хранить непосредственно в БД. Это обычная практика в веб-программировании. Поэтому в таблице users создадим специальное поле avatar с типом BLOB:

CREATE TABLE IF NOT EXISTS users (

id integer PRIMARY KEY AUTOINCREMENT,

name text NOT NULL,

email text NOT NULL,

psw text NOT NULL,

avatar BLOB DEFAULT NULL,

time integer NOT NULL

);

И, затем, с помощью известной уже вам функции create\_db создадим эту таблицу в БД. Но, предварительно удалим ее (иначе она не будет создана).

Затем, в основном программном модуле добавим специальную константу:

MAX\_CONTENT\_LENGTH = 1024 \* 1024

которая будет ограничивать максимальный размер загружаемого файла, в данном случае 1 Мб.

После этого нам нужно поправить метод addUser в классе FDataBase, чтобы пользователь создавался с аватаром по умолчанию (значение NULL):

    def addUser(self, name, email, hpsw):

…

            self.\_\_cur.execute("INSERT INTO users VALUES(NULL, ?, ?, ?, NULL, ?)", (name, email, hpsw, tm))

…

Далее, создадим шаблон отображения профайла (profile.html). Он будет следующий:

{% extends 'base.html' %}

{% block content %}

{{ super() }}

{% for cat, msg in get\_flashed\_messages(True) %}

<div class="flash {{cat}}">{{msg}}</div>

{% endfor %}

<table border="0" class="profile-table">

         <tr><td valign="top">

                   <div class="profile-ava"><img src="{{ url\_for('userava') }}"></div>

                   <div class="profile-load">

                   <form action="{{url\_for('upload')}}" method="POST" enctype="multipart/form-data">

                            <input type="file" name="file">

                            <input type="submit" value="Загрузить">

                   </form>

                   </div>

         </td>

         <td valign="top" class="profile-panel">

                   <a href="{{url\_for('logout')}}">Выйти из профиля</a>

                   <ul class="profile-info">

                   <li>Имя: {{ current\_user.getName() }}</li>

                   <li>Email: {{ current\_user.getEmail() }}</li>

                   </ul>

         </td></tr>

</table>

{% endblock %}

Мы здесь используем прокси-переменную current\_user для обращения к авторизованному пользователю. И через нее вызываем методы класса UserLogin. Добавим этим методы:

    def getName(self):

        return self.\_\_user['name'] if self.\_\_user else "Без имени"

    def getEmail(self):

        return self.\_\_user['email'] if self.\_\_user else "Без email"

Далее, чтобы профиль выглядел более-менее прилично, пропишем следующие стили оформления шаблона profile.html:

.profile-ava {

         width: 130px;

         text-align: center;

         overflow: hidden;

         background: *#eee;*

         padding: 10px;

}

.profile-ava img {max-width: 150px; height: auto;}

.profile-load {

         margin-top: 10px;

         overflow: hidden;

         max-width: 150px;

}

.profile-load input[type=submit], input[type=file] {

         width: 100%;

         font-size: 18px;

}

.profile-load p {

         padding: 0;

         margin: 5px 0 0 0;

}

.profile-panel {padding: 0 0 0 10px;}

ul.profile-info {

         list-style: none;

         margin: 10px 0 0 0;

         padding: 0;

         color: *#7E652F;*

}

ul.profile-info li {margin-top: 10px;}

## Отображение аватара

Для отображения аватара создадим отдельный обработчик userava:

@app.route('/userava')

@login\_required

def userava():

    img = current\_user.getAvatar(app)

    if not img:

        return ""

    h = make\_response(img)

    h.headers['Content-Type'] = 'image/png'

    return h

Смотрите, мы здесь загружаем аватар пользователя с помощью метода getAvatar, который пропишем в классе UserLogin. Если данные не были получены, то возвращается пустая строка, а иначе, создается объект ответа с параметром 'Content-Type' равным 'image/png', т.к. мы для простоты будем полагать, что все загруженные аватары будут представлены в формате PNG. В результате, при отображении изображения через тег img, в профиле будем видеть соответствующий аватар текущего пользователя.

Теперь перейдем в класс UserLogin и там добавим метод получения аватара:

    def getAvatar(self, app):

        img = None

        if not self.\_\_user['avatar']:

            try:

                with app.open\_resource(app.root\_path + url\_for('static', filename='images/default.png'), "rb") as f:

                    img = f.read()

            except FileNotFoundError as e:

                print("Не найден аватар по умолчанию: "+str(e))

        else:

            img = self.\_\_user['avatar']

        return img

Мы здесь вначале проверяем: если в БД аватар не был загружен, то берется изображение по умолчанию:

static/images/default.png

А, иначе, берем аватар из БД. В конце метод возвращает прочитанные данные изображения.

## Загрузка аватара

Следующим шагом нам нужно реализовать загрузку изображения пользователя в профайле. Как мы говорили вначале, для этого необходим отдельный обработчик, который можно представить в следующем виде:

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

@login\_required

def upload():

    if request.method == 'POST':

        file = request.files['file']

        if file andcurrent\_user.verifyExt(file.filename):

            try:

                img = file.read()

                res = dbase.updateUserAvatar(img, current\_user.get\_id())

                if not res:

                    flash("Ошибка обновления аватара", "error")

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

                flash("Аватар обновлен", "success")

            except FileNotFoundError as e:

                flash("Ошибка чтения файла", "error")

        else:

            flash("Ошибка обновления аватара", "error")

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

Здесь вначале проверяется, что пришли данные по POST-запросу и берется поле file из объекта request. Это поле ассоциировано с загруженным на сервер файлом, который был выбран через форму в профайле. Далее, идет проверка, что файл был успешно загружен и что его расширение PNG. Затем, происходит чтение данных из файла и обновляется аватар пользователя в БД.

Чтобы этот обработчик работал, нужно добавить два метода. Первый verifyExt в классе UserLogin:

    def verifyExt(self, filename):

        ext = filename.rsplit('.', 1)[1]

        if ext == "png" or ext == "PNG":

            return True

        return False

И второй – updateUserAvatar в классе FDataBase:

    def updateUserAvatar(self, avatar, user\_id):

        if not avatar:

            return False

        try:

            binary = sqlite3.Binary(avatar)

            self.\_\_cur.execute(f"UPDATE users SET avatar = ? WHERE id = ?", (binary, user\_id))

            self.\_\_db.commit()

        except sqlite3.Error as e:

            print("Ошибка обновления аватара в БД: "+str(e))

            return False

        return True

Все, теперь наш функционал полностью готов и пользователи могут загружать свои аватарки в профайл.

**Blueprint и полезные расширения**

# Применение WTForms для работы с формами сайта

[Смотреть материал на видео](https://www.youtube.com/watch?v=oba6GGprvKc&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/primenenie-wtforms-dlya-raboty-s-formami-sayta>

Файл проекта: <https://github.com/selfedu-rus/flasksite-18>

Во всех наших предыдущих занятиях мы создавали формы вручную, прописывая их целиком в шаблонах. И если так делать, то за веб-программистом остаются задачи:

* проверить принимаемые данные на стороне сервера;
* сформировать ошибки для формы (если что-то пошло не так);
* продумать безопасное сохранение и представление данных на сервере

и так далее. Это необходимая и не всегда тривиальная работа. Но, к счастью, есть расширения для Flask, которые значительно облегчают реализацию этих и подобных им типовых задач. В частности, библиотека

WTForms

позволяет достаточно просто оперировать формами и выполняет большую часть задач за разработчика. На этом занятии мы, как раз, и познакомимся с основами ее функционала на примере нашего тестового сайта.

Вообще, WTForms – это библиотека, написанная на Python и независимая от фреймворков. Она способна генерировать формы, проверять их, наполнять начальной информацией, работать с reCaptcha и многое другое. Кроме того, в нее встроена защита от CSRF:

CSRF (Cross-Sire Request Forgery) – межсайтовая подделка запросов.

Это атака, при которой происходит имитация запроса пользователя к стороннему сайту со страницы другого сайта. То есть, злоумышленник создает некую страницу на своем сайте, жертва заходит на нее и из формы запроса отправляется запрос на сайт, в котором посетитель авторизован. Если на сайте нет защиты от CSRF-атаки, злоумышленник, от имени пользователя получает доступ к стороннему ресурсу и творит свои «черные дела».

Так вот, такие атаки будут нипочем, при использовании WTForms. И первым делом нужно установить это расширение. Для Flask оно называется

Flask-WTF

и устанавливается с помощью команды:

pip install flask\_wtf

Концепция создания форм здесь состоит в расширении базового класса

FlaskForm

А все поля формы описываются переменными этого класса и ссылаются на соответствующие объекты, которые могут быть образованы из следующих встроенных классов:

* StringField – для работы с полем ввода;
* PasswordField – для работы с полем ввода пароля;
* BooleanField – для checkbox полей;
* TextAreaField – для работы с вводом текста;
* SelectField – для работы со списком;
* SubmitField – для кнопки submit.

Это лишь часть классов. Полную документацию можно посмотреть на [официальном сайте](https://wtforms.readthedocs.io/).

## Создание класса формы

Давайте для примера создадим в нашем проекте вспомогательный файл forms.py, в котором будем определять все классы форм и начнем с класса LoginForm:

from flask\_wtf import FlaskForm

from wtforms import StringField, SubmitField, BooleanField, PasswordField

from wtforms.validators import DataRequired, Email, Length

class LoginForm(FlaskForm):

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

    psw = PasswordField("Пароль: ", validators=[DataRequired(), Length(min=4, max=100)])

    remember = BooleanField("Запомнить", default = False)

    submit = SubmitField("Войти")

Смотрите, мы прописали переменные: email, psw, remember и submit, которые ссылаются на соответствующие объекты. У каждого объекта вначале указана строка, которую, затем, можно будет отобразить рядом с полем ввода и параметр validators. Этот параметр содержит список валидаторов, с помощью которых выполняется проверка корректности введенных данных. Например:

* DataRequired – валидатор, требующий ввода каких-либо данных;
* Email – проверяет корректность введенного email-адреса;
* Length – проверяет количество введенных символов.

Конечно, это не все валидаторы, которые есть в WTForms. Полный их список и набор параметров можно посмотреть на странице [официальной документации](https://wtforms.readthedocs.io/en/2.3.x/validators/).

И, обратите внимание, валидатор Email может требовать отдельной дополнительной установки:

pip install email-validator

Поэтому, если при запуске программы будет отображаться ошибка с указанием класса Email, то просто выполните его установку.

## Создание шаблона формы

Итак, класс определен. Как им теперь пользоваться? Для начала в основном модуле программы выполним импорт:

from forms import LoginForm

И, далее, в функции представления login создадим его экземпляр и передадим шаблону login.html:

@app.route("/login", methods=["POST", "GET"])

def login():

    form = LoginForm()

    return render\_template("login.html", menu=dbase.getMenu(), title="Авторизация", form=form)

То есть, в шаблоне будет доступ к переменным этого класса через параметр form:

{% extends 'base.html' %}

{% block content %}

{{ super() }}

{% for cat, msg in get\_flashed\_messages(True) %}

<div class="flash {{cat}}">{{msg}}</div>

{% endfor %}

<form action="" method="post" class="form-contact">

{{ form.hidden\_tag() }}

{{ form.email.label() }} {{ form.email() }}

{{ form.psw.label() }} {{ form.psw() }}

{{ form.remember.label() }} {{ form.remember() }}

{{ form.submit() }}

<hr align=left width="300px">

<a href="{{url\_for('register')}}">Регистрация</a>

</form>

{% endblock %}

Смотрите, здесь в самом начале идет вызов метода:

form.hidden\_tag()

который создает скрытое поле, содержащее токен, используемый для защиты формы от CSRF-атак. Это все, что от нас требуется, остальное Flask-WTF сделает автоматически. Как видите, все просто и удобно.

Далее, мы вызываем методы label, которые вставляют в форму тег:

<label>Название</label>

А методы email, psw, remember, submit – создают соответствующие теги полей ввода, кнопок, чекбоксов и так далее.

Давайте запустим программу и посмотрим как будет выглядеть эта форма и что представлять на уровне HTML-документа.

Отлично, это сделано, теперь в обработчике login выполним обработку элементов этой формы и свяжем ее с ранее созданным функционалом:

@app.route("/login", methods=["POST", "GET"])

def login():

    if current\_user.is\_authenticated:

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

    form = LoginForm()

    if form.validate\_on\_submit():

        user = dbase.getUserByEmail(form.email.data)

        if user and check\_password\_hash(user['psw'], form.psw.data):

            userlogin = UserLogin().create(user)

            rm = form.remember.data

            login\_user(userlogin, remember=rm)

            return redirect(request.args.get("next") or url\_for("profile"))

        flash("Неверная пара логин/пароль", "error")

    return render\_template("login.html", menu=dbase.getMenu(), title="Авторизация", form=form)

Смотрите, здесь вначале идет проверка validate\_on\_submit() корректности переданных данных по POST-запросу и правильность заполненных полей формы. И это очень удобно, т.к. программа пойдет дальше только в случае верных данных. В этом случае мы дополнительно проверяем корректность ввода пароля и, затем, авторизовываем пользователя и перенаправляем его на соответствующий URL. Иначе, снова отображается форма авторизации.

# Обработка ошибок во Flask-WTF

[Смотреть материал на видео](https://www.youtube.com/watch?v=Pn5RxiAFuoc&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

Файл проекта: <https://github.com/selfedu-rus/flasksite-18>

Продолжаем предыдущее занятие и если мы сейчас введем на странице неверные данные, например, некорректный пароль, то страница просто обновится без указания каких-либо ошибок. Вместе с тем, ошибки автоматически генерируются и доступны в шаблоне из коллекции:

form.<переменная поля>.errors

Добавим их вывод:

{{ form.email.label() }}

{% if form.email.errors %}

         {{ form.email(class="invalid") }}

<span class="invalid-feedback">

         {% for e in form.email.errors %}

         {{ e }}

         {% endfor %}

</span>

{% else %}

         {{ form.email() }}

{% endif %}

{{ form.psw.label() }}

{% if form.psw.errors %}

         {{ form.psw(class="invalid") }}

<span class="invalid-feedback">

         {% for e in form.psw.errors %}

         {{ e }}

         {% endfor %}

</span>

{% else %}

         {{ form.psw() }}

{% endif %}

Мы здесь после отображения имени поля проверяем: существуют ли ошибки и если да, то указываем класс стилей для отображения ошибочных полей а, затем, выводим список ошибок (их может быть несколько). Если же ошибок нет, то поле отображается в первоначальном виде.

Чтобы все выглядело относительно прилично, добавим стили для отображения ошибочных состояний:

.form-contact .invalid {

         display: inline-block;

         background: *#FF9898;*

}

.form-contact .invalid-feedback {

         color: *#CC0000;*

}

Перейдем в браузер, наберем неверный email и увидим сообщение об ошибке:

Invalid email address.

По умолчанию в WTForms ошибки показываются на английском языке. Чтобы прописать свои собственные, в классе LoginForm нужно их явно прописать:

class LoginForm(FlaskForm):

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

    psw = PasswordField("Пароль: ", validators=[DataRequired(),

                                                Length(min=4, max=100, message="Пароль должен быть от 4 до 100 символов")])

    remember = BooleanField("Запомнить", default = False)

    submit = SubmitField("Войти")

Теперь, при неверном вводе информации мы будем видеть указанные нами сообщения.

## Формирование полей в шаблоне через цикл

Если вернуться к нашей форме авторизации, то увидим повторяющиеся части текста, что не очень хорошо. Чтобы этого не было, мы можем создавать однотипные поля формы через цикл, используя параметр form как итерируемый объект:

{% for field in form if field.name not in ['csrf\_token', 'remember', 'submit'] -%}

         {{ field.label() }}

         {% if field.errors %}

                   {{ field(class="invalid") }}

         <span class="invalid-feedback">

                   {% for e in field.errors %}

                   {{ e }}

                   {% endfor %}

         </span>

         {% else %}

                   {{ field() }}

         {% endif %}

{% endfor %}

Смотрите, мы здесь выбираем все поля (field) из формы, кроме полей с именами 'csrf\_token', 'remember' и 'submit'. Первое поле – это специальный токен, служащий для защиты от CSRF-атак, а последние два – это флажок «Запомнить» и кнопка «Войти». Мы их отображаем на форме без указания списка ошибок.

Если теперь перейти в браузер, то при обновлении страницы увидим ту же самую форму авторизации со всеми полями. Причем, порядок полей тот же, что и порядок переменных в классе LoginForm. Мало того, если будет добавлена еще какая-либо типовая переменная поля, то она автоматически добавится в форму и будет соответствующим образом обрабатываться, что очень удобно.

## Форма регистрации

Давайте заменим в нашем сайте еще одну форму для регистрации пользователей. Ее класс можно прописать следующим образом:

class RegisterForm(FlaskForm):

    name = StringField("Имя: ", validators=[Length(min=4, max=100, message="Имя должно быть от 4 до 100 символов")])

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

    psw = PasswordField("Пароль: ", validators=[DataRequired(),

                                                Length(min=4, max=100, message="Пароль должен быть от 4 до 100 символов")])

    psw2 = PasswordField("Повтор пароля: ", validators=[DataRequired(), EqualTo('psw', message="Пароли не совпадают")])

    submit = SubmitField("Регистрация")

Мы здесь используем еще один валидатор EqualTo для проверки совпадения паролей. Все остальное очень похоже на форму авторизации.

Далее, шаблон register.html запишем в виде:

{% extends 'base.html' %}

{% block content %}

{{ super() }}

{% for cat, msg in get\_flashed\_messages(True) %}

<div class="flash {{cat}}">{{msg}}</div>

{% endfor %}

<form action="{{ url\_for('register') }}" method="post" class="form-contact">

{{ form.hidden\_tag() }}

{% for field in form if field.name not in ['csrf\_token', 'submit'] -%}

         {{ field.label() }}

         {% if field.errors %}

                   {{ field(class="invalid") }}

         <span class="invalid-feedback">

                   {% for e in field.errors %}

                   {{ e }}

                   {% endfor %}

         </span>

         {% else %}

                   {{ field() }}

         {% endif %}

{% endfor %}

{{ form.submit() }}

</form>

{% endblock %}

И обработчик register:

@app.route("/register", methods=["POST", "GET"])

def register():

    form = RegisterForm()

    if form.validate\_on\_submit():

            hash = generate\_password\_hash(request.form['psw'])

            res = dbase.addUser(form.name.data, form.email.data, hash)

            if res:

                flash("Вы успешно зарегистрированы", "success")

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

            else:

                flash("Ошибка при добавлении в БД", "error")

    return render\_template("register.html", menu=dbase.getMenu(), title="Регистрация", form=form)

Как видите, все достаточно просто и при этом получаем доступ к богатому функционалу модуля WTForms.

Конечно, мы здесь рассмотрели лишь общий принцип использования расширения WTForms. Этих знаний вполне достаточно для простых реализаций. А для более глубокого погружения в эту тему советую почитать [официальную документацию](https://wtforms.readthedocs.io/) по данному модулю.

# Blueprint - что это такое, где и как использовать

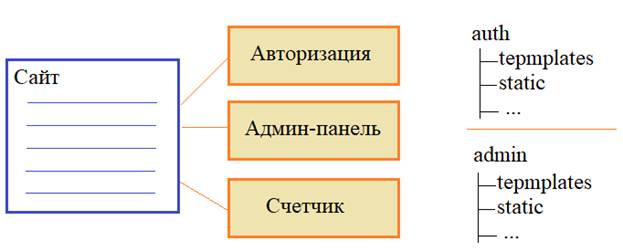
[Смотреть материал на видео](https://www.youtube.com/watch?v=leUbqB2A1HI&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/blueprint-chto-eto-takoe-gde-i-kak-ispolzovat>

Файл проекта: <https://github.com/selfedu-rus/flasksite-21>

К этому моменту мы с вами создали проект (WSGI-приложение), состоящий из нескольких файлов и реализующий некоторый типовой функционал: отображение статей, авторизацию и регистрацию пользователей. Но даже в этом простом приложении становится непросто ориентироваться, вносить изменения, не говоря уже о тестировании отдельных его компонент, например, авторизации пользователей. Наверное, главной проблемой разрастающегося проекта, является все большая мешанина программного кода, усложнение его понимания, редактирования и переноса части функционала в какой-либо другой подобный проект.

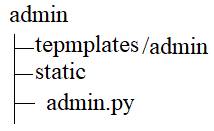
Так вот, все эти слова я, конечно, говорю не просто так. Фреймворк Flask позволяет разделять большой, сложный проект на набор независимых модулей. Например, на сайте могут быть модули авторизации, админ-панели, подсчета числа просмотров страниц и так далее. Причем, каждый модуль может иметь свои шаблоны, стили оформления, наборы изображений. Все это хранится в отдельном подкаталоге, что делает фрагмент приложения совершенно независимым. И при необходимости, этот каталог можно просто скопировать в другой проект и подключить к другому сайту.



Такие модули в терминологии Flask называются Blueprint (эскизами) и о них пойдет речь на этом занятии. Чтобы не копаться в уже написанном коде и не усложнять занятие, я покажу пример реализации Blueprint на примере создания простейшей админ-панели.

## Структура подкаталога admin

Вся наша админ-панель будет реализована в отдельном подкаталоге admin со следующей структурой:



В корне этого каталога будет лежать файл admin.py, в котором и реализуем админ-панель с использованием Blueprint. В подкаталоге templates/admin будут располагаться файлы шаблонов, а в static – файлы оформления, js, изображений и другие статические данные, подключаемые к HTML-страницам в панели администратора. Таким образом, мы строго разделяем данные, относящиеся к нашему модулю admin от всех остальных файлов приложения. В дальнейшем, при необходимости, можно будет просто скопировать этот подкаталог в другой проект и в нем реализовать похожую админ-панель.

Здесь может возникнуть вопрос: зачем в подкаталоге templates создавать еще один каталог admin? Фактически, здесь используется Django’вский подход, когда шаблоны в каждом модуле помещаются в дополнительный подкаталог с тем же именем. Это необходимо, т.к. при компиляции проекта все шаблоны собираются в одну кучу и может возникнуть конфликт имен, когда в разных модулях будут файлы с одинаковыми именами. Чтобы этого избежать, как раз и создается дополнительный подкаталог. В этом случае в момент выполнения, шаблоны будут отделяться от других этим подкаталогом.

## Создание и регистрация Blueprint

Итак, теперь у нас все готово, чтобы создать Blueprint в модуле admin.py. В начале выполним импорт класса Blueprint:

from flask import Blueprint

и ниже создадим экземпляр этого класса:

admin = Blueprint('admin', \_\_name\_\_, template\_folder='templates', static\_folder='static')

* 'admin' – имя Blueprint, которое будет суффиксом ко всем именам методов, данного модуля;
* \_\_name\_\_ – имя исполняемого модуля, относительно которого будет искаться папка admin и соответствующие подкаталоги;
* template\_folder – подкаталог для шаблонов данного Blueprint (необязательный параметр, при его отсутствии берется подкаталог шаблонов приложения);
* static\_folder – подкаталог для статических файлов (необязательный параметр, при его отсутствии берется подкаталог static приложения).

После создания эскиза его нужно зарегистрировать в основном приложении. Перейдем в файл flsite.py и выполним импорт переменной admin:

from admin.admin import admin

Обратите внимание, мы импортируем именно переменную, а не класс или функцию. Далее, ниже выполним непосредственно регистрацию Blueprint:

app.register\_blueprint(admin, url\_prefix='/admin')

Здесь admin – ссылка на созданный Blueprint; url\_prefix – префикс для всех URL модуля admin. Это необязательный параметр. Без него все URL внутри Blueprint будут записываться непосредственно после домена сайта. Но это не лучшая практика, так как, подключая несколько таких модулей, можно опять же столкнуться с проблемой дублирования URL. Поэтому лучше использовать префик, по которому они будут четко разделяться.

Итак, мы создали Blueprint и зарегистрировали его в приложении. Если теперь перейти по URL:

http://127.0.0.1:5000/admin

то получим ошибку 404 – страница не найдена, так как внутри эскиза не создано ни одного представления. Давайте его добавим.

## Маршрутизация в Blueprint

Перейдем в модуль admin.py и пропишем декоратор route:

@admin.route('/')

def index():

    return "admin"

Смотрите, мы вызываем route для admin, а не app, как это делали в основном приложении. Тем самым указываем, что корневая (главная) страница – это страница Blueprint, а не приложения app. Причем, адрес этого URL определяется по правилу:

домен/<url\_prefix>/<URL-blueprint>

и в нашем случае будет выглядеть так:

http://127.0.0.1:5000/admin/

то есть, это главная страница админ-панели, а не сайта. И благодаря префиксу, указанному при регистрации этого эскиза, мы можем не беспокоиться о дублировании URL в нашем модуле.

## Авторизация в админ-панели

Давайте теперь добавим возможность авторизации в нашей тестовой панели администратора. Для этого пропишем еще одну функцию представления login, следующим образом:

@admin.route('/login', methods=["POST", "GET"])

def login():

    if request.method == "POST":

        if request.form['user'] == "admin" and request.form['psw'] == "12345":

            login\_admin()

            return redirect(url\_for('.index'))

        else:

            flash("Неверная пара логин/пароль", "error")

    return render\_template('admin/login.html', title='Админ-панель')

Здесь все достаточно просто и очевидно. Сначала проверяем, что пришли данные по POST-запросу, затем, проверяем правильность логина и пароля и при истинности условий, выполняем авторизацию с помощью функции login\_admin, которую пропишем чуть позже. Далее, делается перенаправление на главную страницу админ-панели, а иначе – формируется мгновенное сообщение «Неверная пара логин/пароль». В конце возврашается шаблон 'admin/login.html' с заголовком 'Админ-панель'.

Обратите внимание, как здесь записан параметр в функции

url\_for('.index')

Перед index указана точка. Эта точка означает, что функцию-представления index следует брать для текущего Blueprint, а не глобальную из приложения. Если убрать точку, то будет возвращен URL-адрес главной страницы сайта, а не панели администратора. Как вариант, функцию url\_for можно еще вызвать и так:

url\_for('admin.index’)

Здесь admin – это имя Blueprint, а не название файла admin.py. Например, если изменить имя эскиза на bp, то придется уже прописывать:

url\_for(bp.index’)

Но, в данном случае, это избыточная запись и, кроме того, идет жесткая привязка к имени Blueprint. Поэтому, я в дальнейшем буду писать просто точку.

Итак, возвращаясь к обработчику login, добавим функцию login\_admin в модуль admin.py:

def login\_admin():

    session['admin\_logged'] = 1

Мы здесь просто в сессии создаем и сохраняем запись 'admin\_logged' со значением 1. И в дальнейшем будем полагать, если она существует, то пользователь зашел в админ-панель.

Некоторые из вас могут задаться вопросом: почему бы нам здесь не использовать рассмотренный ранее модуль Flask-Login? Дело в том, что нельзя создать еще один его экземпляр в рамках одного приложения. А Blueprint – это лишь дополнение, расширение, но не самостоятельная программа. Конечно, мы могли бы передать ссылку на Flask-Login в наш модуль admin и как то его использовать, но тогда теряется концепция независимости и модульности Blueprint. И наша реализация будет ничем не лучше обычного дополнительного вспомогательного класса, записанного в отдельном файле проекта. Поэтому, я авторизацию сделал через сессии.

Давайте здесь же рядом объявим еще две вспомогательные функции:

def isLogged():

    return True if session.get('admin\_logged') else False

def logout\_admin():

    session.pop('admin\_logged', None)

Первая проверяет: авторизован ли администратор, а вторая – удаляет из сессии запись об авторизации и будет использоваться при выходе из админ-панели. И сразу пропишем функцию представления logout:

@admin.route('/logout', methods=["POST", "GET"])

def logout():

    if not isLogged():

        return redirect(url\_for('.login'))

    logout\_admin()

    return redirect(url\_for('.login'))

## Шаблоны для страницы админ-панели и авторизации

Теперь давайте добавим первые шаблоны для нашей панели администратора. Я специально для демонстрации сделаю другое оформление и пропишу новый базовый шаблон. В каталоге templates/admin добавлю файл base\_admin.html, который будет иметь следующий вид:

<!DOCTYPE html>

<html>

<head>

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

<title>{{ title }}</title>

</head>

<body>

{% if menu -%}

         <ul class="mainmenu">

         {% for p in menu %}

         <li><a href="{{ url\_for(p.url) }}">{{p.title}}</a></li>

         {% endfor %}

         </ul>

         <div class="clear"></div>

{%- endif -%}

<div class="content">

{% block content -%}

{% endblock %}

</div>

</body>

</html>

Здесь все, опять же, достаточно очевидно. Обратите внимание, для обращения к каталогу static первый параметр функции

url\_for('.static', filename='css/styles.css')

записан с точкой вначале. Эта точка будет указывать брать каталог static из подкаталога admin, то есть, сформируется следующий путь:

admin/static/css/styles.css

Далее, в шаблоне идет отображение меню, если оно передается в шаблон и записан именованный блок content для добавления информации в дочерних шаблонах. И первый такой шаблон будет login.html со следующим содержимым:

{% extends 'admin/base\_admin.html' %}

{% block content %}

{{ super() }}

<div id="login" class="wnd\_dlg\_back login\_wnd">

<div class="login">

         <p class="title">Админ-панель</p>

{% for cat, msg in get\_flashed\_messages(True) %}

<div class="flash {{cat}}">{{msg}}</div>

{% endfor %}

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

                   <label class="form-label">Логин: </label><input type="text" name="user" />

                   <label class="form-label">Пароль: </label><input type="password" name="psw" />

                   <p align="center"><input class="login\_button" type="submit" value="Войти">

         </form>

</div>

</div>

{% endblock %}

Мы здесь отображаем форму авторизации, где пользователь вводит логин/пароль.

А шаблон index.html будет пока пустой:

{% extends 'admin/base\_admin.html' %}

{% block content %}

{{ super() }}

{% endblock %}

Далее, нам нужно прописать стили оформления. Они представлены в файле

admin/static/css/styles.css

(подробнее см. в файле проекта).

И перед первым пробным тестированием, изменим обработчик главной страницы админ-панели:

@admin.route('/')

def index():

    if not isLogged():

        return redirect(url\_for('.login'))

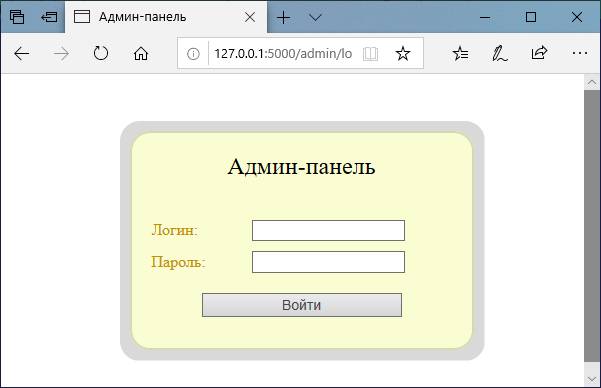
    return render\_template('admin/index.html', menu=menu, title='Админ-панель')

Вначале идет проверка: если пользователь не авторизован, то он перенаправляется на страницу авторизации. Иначе, будет отображена панель администратора. Здесь в шаблон 'admin/index.html' передаются два параметра: menu и title. И для menu пропишем следующую коллекцию:

menu = [{'url': '.index', 'title': 'Панель'},

        {'url': '.logout', 'title': 'Выйти'}]

Все, теперь при запуске, мы увидим окно авторизации и, вводя admin/12345, перейдем в панель администратора.



# Blueprint - подключение к БД и работа с ней

[Смотреть материал на видео](https://www.youtube.com/watch?v=WSJ8l_RYMAo&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/blueprint-podklyuchenie-k-bd-i-rabota-s-ney>

Файл проекта: <https://github.com/selfedu-rus/flasksite-21>

Обычно, админ-панель использует БД приложения и выполняет определенные действия. Давайте в качестве примера сделаем вывод списка статей и пользователей, зарегистрированных на сайте. Но для начала нам нужно в Blueprint выполнить соединение с БД. Как это сделать, чтобы максимально сохранить модульность приложения? Хорошей практикой является использование декораторов:

* before\_request – перед выполнением запроса;
* teardown\_request – после выполнения запроса.

В первом происходит установление связи с БД, а во втором – закрытие соединения. В нашем тестовом сайте это можно сделать так (в модуле admin.py):

db = None

@admin.before\_request

def before\_request():

    """Установление соединения с БД перед выполнением запроса"""

    global db

    db = g.get('link\_db')

@admin.teardown\_request

def teardown\_request(request):

    global db

    db = None

    return request

Мы здесь в функции before\_request обращаемся к глобальной переменной g контекста приложения и берем оттуда значение 'link\_db', которое связано со ссылкой на соединение с БД. Эта связь, в свою очередь, устанавливается в основном приложении, когда вызывается функция декоратора:

@app.before\_request

Причем, декораторы уровня приложения как бы «обертывают» выполнение декораторов в Blueprint. То есть, их последовательность вызовов будет такой:

@app.before\_request  
@admin.before\_request  
@admin.teardown\_request  
@app.teardown\_appcontext

Таким образом, мы просто берем соединение с БД, которое устанавливается основным приложением. Конечно, если архитектура программы изменится и связь с БД будет настраиваться как-то по-другому, то это повлечет необходимость изменения и в модуле Blueprint. Но, ничего не поделаешь: абсолютной независимости не бывает.

Также, для обращения к приложению можно использовать специальную глобальную переменную:

current\_app

В некоторых случаях через нее обращаются к объекту конфигурации, чтобы получить ту или иную константу:

current\_app.config['DATABASE']

Но здесь, всегда следует помнить правило: чем меньше связи с внешними модулями, тем лучше. Например, можно добавить ссылку на админ-панель. Разместим ее для простоты на главной странице сайта, т.е. в шаблоне index.html пропишем:

<a href="{{ url\_for('admin.index') }}">Админ-панель</a></p>

Обратите внимание на префикс ‘admin’ перед именем функции-представления index. Этот префикс соответствует имени Blueprint, к которому мы обращаемся, чтобы уже там взять функцию index и определить ее URL. В данном случае будет возвращен адрес:

http://127.0.0.1:5000/admin/

Вот так следует формировать маршрутизацию из внешнего модуля, используя функции-представления Blueprint.

## Отображение списка статей и пользователей

Теперь, когда у нас есть соединение с БД, мы можем в панели администратора вывести список статей и зарегистрированных пользователей. Начнем со статей. Создадим еще одно представление listpubs со следующим содержимым:

@admin.route('/list-pubs')

def listpubs():

    if not isLogged():

        return redirect(url\_for('.login'))

    list = []

    if db:

        try:

            cur = db.cursor()

            cur.execute(f"SELECT title, text, url FROM posts")

            list = cur.fetchall()

        except sqlite3.Error as e:

            print("Ошибка получения статей из БД " + str(e))

    return render\_template('admin/listpubs.html', title='Список статей', menu=menu, list=list)

Вначале идет проверка: если пользователь не вошел в админ-панель, то он перенаправляется на страницу авторизации. Далее, проверяем: если соединение с БД установлено, то из таблицы posts выбираются все записи с полями: title, text, url. Затем, читаются все значения и сохраняются в переменной list. В конце возвращается страница из шаблона 'admin/listpubs.html', который выглядит так:

{% extends 'admin/base\_admin.html' %}

{% block content %}

{{ super() }}

<h1>{{title}}</h1>

<ul class="list-posts">

{% for p in list %}

<li>

<p class="title"><a href="{{ url\_for('showPost', alias=p.url)}}">{{p.title}}</a></p>

<p class="annonce">{{ p.text[:50] | striptags  }}</p>

</li>

{% endfor %}

</ul>

{% endblock %}

Здесь все достаточно очевидно, происходит расширение базового шаблона и формируется список из коллекции list.

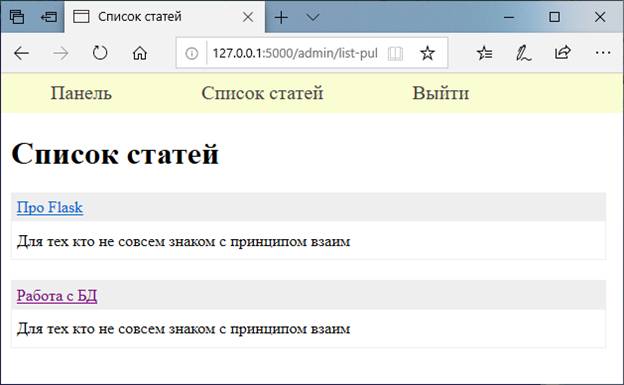
Протестируем эту реализацию. Добавим в меню новый пункт:

menu = [{'url': '.index', 'title': 'Панель'},

        {'url': '.listpubs', 'title': 'Список статей'},

        {'url': '.logout', 'title': 'Выйти'}]

И после запуска увидим следующее:



По аналогии сделаем отображение списка пользователей. Добавим обработчик:

@admin.route('/list-users')

def listusers():

    if not isLogged():

        return redirect(url\_for('.login'))

    list = []

    if db:

        try:

            cur = db.cursor()

            cur.execute(f"SELECT name, email FROM users ORDER BY time DESC")

            list = cur.fetchall()

        except sqlite3.Error as e:

            print("Ошибка получения статей из БД " + str(e))

    return render\_template('admin/listusers.html', title='Список пользователей', menu=menu, list=list)

Здесь все, практически, то же самое, только из БД выбираются пользователи с полями: name и email. И, затем, возвращается страница по шаблону listusers.html:

{% extends 'admin/base\_admin.html' %}

{% block content %}

{{ super() }}

<h1>{{title}}</h1>

<ul class="list-posts">

{% for p in list %}

<li>

<p class="title">{{p.name}}</p>

<p class="annonce">{{ p.email }}</p>

</li>

{% endfor %}

</ul>

{% endblock %}

Добавим в меню ссылку для этого обработчика:

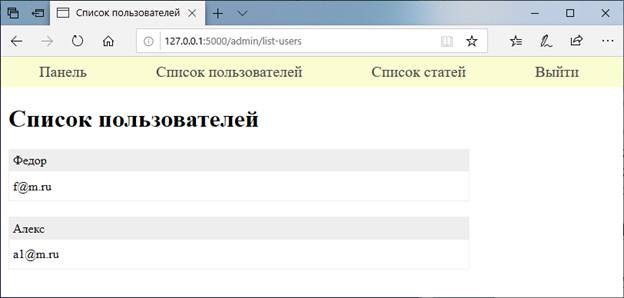
menu = [{'url': '.index', 'title': 'Панель'},

        {'url': '.listusers', 'title': 'Список пользователей'},

        {'url': '.listpubs', 'title': 'Список статей'},

        {'url': '.logout', 'title': 'Выйти'}]

И, окончательно, получим следующую админ-панель:



Конечно, это лишь пример использования механизма Blueprint, позволяющий создавать во Flask наборы отдельных модулей по принципу, похожему на плагины. И, тем самым, расширять приложение без излишнего усложнения программного кода. За более детальной информацией можно обратиться к официальной документации:

<https://flask.palletsprojects.com/en/1.1.x/blueprints/>

# Flask-SQLAlchemy: установка, создание таблиц, добавление записей

[Смотреть материал на видео](https://www.youtube.com/watch?v=JqZbeMtwthY&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

<https://proproprogs.ru/flask/flask-sqlalchemy-ustanovka-sozdanie-tablic-dobavlenie-zapisey>

Файл проекта: <https://github.com/selfedu-rus/flsite_sqlalchemy-23>

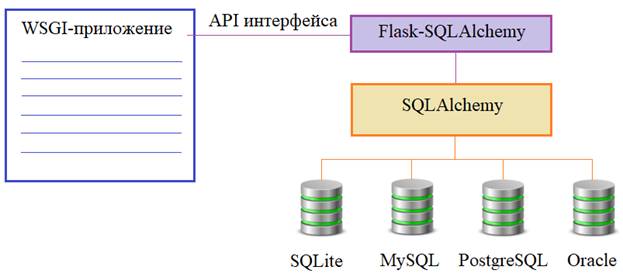
На всех наших занятиях этого курса по Flask мы с вами использовали БД SQLite: непосредственно подключались к ней и работали с таблицами через SQL-запросы. Но это не лучшее решение. Для реальных сайтов следует выбирать более функциональные СУБД, рассчитанные на большую нагрузку, когда множество пользователей одновременно могут записывать и читать данные из разных таблиц. Например:

MySQL, PostgreSQL, Oracle и другие.

Причем, разработчик сайта наперед может не знать с какой именно СУБД будет работать WSGI-приложение. Да и потом, уже в процессе работы, выбор может пасть на другой тип БД. Как в этом случае построить универсальную программу не привязанную к конкретной СУБД? В Python часто для этого используют пакет

SQLAlchemy

который является интерфейсом, построенным по технологии ORM (Object Role Model), позволяющий оперировать таблицами БД как объектами языка Python. Причем этот интерфейс универсален и на уровне WSGI-приложения не привязан к конкретной СУБД.



Так вот, для фреймворка Flask разработано специальное расширение:

Flask-SQLAlchemy

которое значительно упрощает настройку и использование самого SQLAlchemy. И о нем пойдет речь на этом занятии.

Вначале, как всегда, его нужно установить. Для этого достаточно выполнить команду:

pip install Flask-SQLAlchemy

и он будет готов к использованию. Далее, я создам новый проект, чтобы нам не разбираться в прежнем коде и не тратить на это время. И, к тому же, так будет проще воспринимать этот материал.

## Настройка Flask-SQLAlchemy

Первым делом нужно подключить Flask-SQLAlchemy к нашей программе. Для этого выполним его импорт:

from flask\_sqlalchemy import SQLAlchemy

from flask import Flask

и, затем, после создания экземпляра приложения, создадим в конфигурации специальную константу 'SQLALCHEMY\_DATABASE\_URI':

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

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///blog.db'

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

    app.run(debug=True)

Эта константа определяет вид используемой СУБД (в данном случае SQLite) и местоположение файла БД (в данном случае – корневой каталог программы). Именно определение этой константы связывает SQLAlchemy с той или иной СУБД. То есть, если в будущем потребуется другая БД, то мы можем просто переопределить эту константу, например, так:

* postgresql://user:password@localhost/mydatabase
* mysql://user:password@localhost/mydatabase
* oracle://user:password@127.0.0.1:1521/mydatabase

и автоматически к программе подключится указанная СУБД. Как видите, все достаточно просто и удобно.

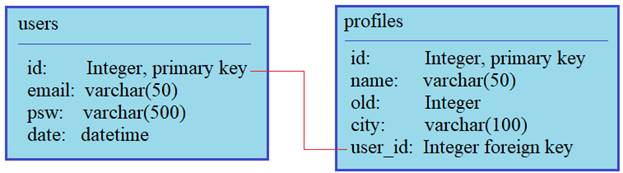
После настройки конфигурации необходимо создать экземпляр класса SQLAlchemy, через который и осуществляется работа с БД. В самом простом варианте это делается так:

db = SQLAlchemy(app)

Здесь при создании класса передается ссылка на текущее приложение, с которым он будет связан.

## Создание таблиц во Flask-SQLAlchemy

После начальной настройки SQLAlchemy нам нужно определить набор таблиц, с которыми мы в дальнейшем будем работать. И давайте в качестве примера будем выполнять регистрацию пользователей и сохранять информацию о них в двух таблицах со следующими структурами:



Так вот, концепция SQLAlchemy заключается в отображении таблиц с помощью Python-классов. То есть, нам достаточно в программе прописать два класса с именами

Users и Profiles

которые бы расширяли базовый класс модели таблиц:

SQLAlchemy.Model

Сначала определим класс Users (порядок не имеет значения, просто иду по рисунку):

class Users(db.Model):

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

    email = db.Column(db.String(50), unique=True)

    psw = db.Column(db.String(500), nullable=True)

    date = db.Column(db.DateTime, default=datetime.utcnow)

    def \_\_repr\_\_(self):

        return f"<users {self.id}>"

Смотрите, мы здесь указываем базовый класс Model, который и превращает класс Users в модель таблицы для SQLAlchemy. А поля таблицы прописываются как обычные переменные, которые ссылаются на специальный класс Column. Этот класс как раз и указывает SQLAlchemy воспринимать эти переменные как поля таблицы. Причем, имя таблицы будет соответствовать имени класса, записанное малыми буквами, а имена переменных – именам полей в этой таблице. Далее, в конструкторе класса Column мы указываем тип поля и дополнительные параметры. Например, поле id будет создаваться как целочисленное и представлять собой главный ключ (то есть, оно будет принимать уникальное значение для каждой записи). Далее, тип String(50) – это строка максимальной длиной в 50 символов, а параметр unique=True означает, что оно должно быть уникальным среди всех записей данной таблицы. Следующее поле psw также является строкой и параметр nullable=True указывает, что оно должно обязательно содержать данные, т.е. пустым быть не может. Наконец, последнее поле date имеет тип datetime для хранения даты и по умолчанию ему присваивается текущая дата, взятая с помощью функции datetime.utcnow. Выше в программе мы ее импортируем:

from datetime import datetime

В конце описания класса идет «магическая» функция \_\_repr\_\_, которая определяет способ отображения класса в консоли. С ее помощью мы будем выводить класс в виде строки формата:

<users идентификатор>

Она здесь прописана для удобства и к функционированию таблиц не имеет отношения.

По аналогии создается вторая таблица Profiles:

class Profiles(db.Model):

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

    name = db.Column(db.String(50), nullable=True)

    old = db.Column(db.Integer)

    city = db.Column(db.String(100))

    user\_id = db.Column(db.Integer, db.ForeignKey('users.id'))

    def \_\_repr\_\_(self):

        return f"<profiles {self.id}>"

Здесь все достаточно очевидно, только дополнительно прописано целочисленное поле user\_id, определенное как внешний ключ. Как раз через него можно будет связывать эти две таблицы, то есть, конкретного пользователя с его профайлом.

Вообще, наиболее распространенными типами данными в SQLAlchemy, являются следующие:

* Integer – целочисленный;
* String(size) – строка максимальной длиной size;
* Text – текст (в формате Unicode);
* DateTime – дата и время представленные в формате объекта datetime;
* Float – число с плавающей точкой (вещественное);
* Boolean – логическое значение;
* LargeBinary – для больших произвольных бинарных данных (например, изображений).

Итак, таблицы определены и теперь можно их создать в БД. Для этого файл (или файлы) БД нужно перенести в какое-либо другое место (или удалить) и, затем, в консоли Python выполнить следующие команды:

from app import db

db.create\_all()

Здесь app – это название файла, в котором инициализирована переменная db. А метод create\_all как раз и создает таблицы, используя классы Users и Profiles.

## Добавление записей

Пришло время создать в нашей программе первую функцию представления для регистрации пользователей, где мы могли бы добавлять новые записи в таблицы БД. Вначале пропишем обработчик:

@app.route("/register", methods=("POST", "GET"))

def register():

    return render\_template("register.html", title="Регистрация")

И определим шаблон register.html на основе базового шаблона layout.html:

**- layout.html:**

<!DOCTYPE html>

<html>

<head>

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

         <title>{{title}}</title>

</head>

<body>

<ul class="mainmenu">

<li><a href="{{ url\_for('index') }}">Главная</a></li>

<li><a href="{{ url\_for('register') }}">Регистрация</a></li>

</ul>

<div class="content">

{% block content %}

{% endblock %}

</div>

</body>

</html>

**- register.html:**

{% extends 'layout.html' %}

{% block content %}

{% for cat, msg in get\_flashed\_messages(True) %}

<div class="flash {{cat}}">{{msg}}</div>

{% endfor %}

<form action="" method="post" class="form-contact">

<label>Имя: </label> <input type="text" name="name" value="" requied />

<label>Возраст: </label> <input type="text" name="old" value="" requied />

<label>Город: </label> <input type="text" name="city" value="" requied />

<label>Email: </label> <input type="text" name="email" value="" requied />

<label>Пароль: </label> <input type="password" name="psw" value="" requied />

<label>Повтор пароля: </label> <input type="password" name="psw2" value="" requied />

<input type="submit" value="Регистрация" />

</form>

{% endblock %}

Здесь, в принципе, должно быть все вам знакомо: мы описали простую форму с набором полей и кнопкой «Регистрация». Осталось в самом обработчике выполнить добавление записей на основе данных формы:

    if request.method == "POST":

*# здесь должна быть проверка корректности введенных данных*

        try:

            hash = generate\_password\_hash(request.form['psw'])

            u = Users(email=request.form['email'], psw=hash)

            db.session.add(u)

            db.session.flush()

            p = Profiles(name=request.form['name'], old=request.form['old'],

                         city=request.form['city'], user\_id = u.id)

            db.session.add(p)

            db.session.commit()

        except:

            db.session.rollback()

            print("Ошибка добавления в БД")

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

Сначала, конечно, идет проверка, что данные переданы методом POST, затем, по идее должна быть проверка корректности заполнения полей формы. Но мы пропустим этот шаг, чтобы не перегружать занятие лишней информацией. И после этого, в блоке try осуществляем работу с БД.

В первую очередь мы создаем экземпляр класса Users и через именованные параметры передаем данные из формы. Причем, именованные параметры должны совпадать с именами переменных, объявленных нами ранее. В результате создается объект с данными по email и паролю, который и представляет собой будущую запись в таблице users.

Далее, чтобы ее добавить, происходит обращение к специальному объекту session – сессии БД и в нее добавляется запись с помощью метода add. В качестве параметра этот метод принимает ссылку на объект класса Users. Но эта запись пока еще хранится в сессии – памяти устройства. Физически в таблицу она не занесена. Поэтому, далее, выполняется метод flush, который из сессии перемещает запись в таблицу. Но пока все изменения происходят в памяти устройства и, физически, файл БД остается прежним. Это необходимо, на случай возникновения исключений (ошибок) и «отката» состояния БД к исходному состоянию с помощью метода rollback.

Если ошибок не возникает, то формируется следующий экземпляр класса Profiles с набором данных из формы и дополнительно берется значение u.id, которое сформировалось после метода flush при помещении записи в таблицу users. Вот именно поэтому мы его и вызывали. Далее, запись помещается в сессию и вызывается метод commit, который уже физически меняет файлы БД и сохраняет изменения в таблицах.

В итоге всех этих действий, мы добавили в таблицы users и profiles по одной записи для нового зарегистрированного пользователя, причем, внешний ключ user\_id будет ссылаться на id записи из таблицы users.

# Операции с таблицами через Flask-SQLAlchemy

[Смотреть материал на видео](https://www.youtube.com/watch?v=EUqo8TzDotI&list=PLA0M1Bcd0w8yrxtwgqBvT6OM4HkOU3xYn" \t "_blank)

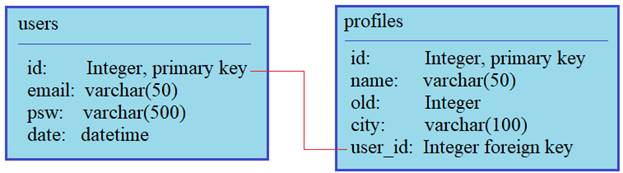
<https://proproprogs.ru/flask/operacii-s-tablicami-cherez-flask-sqlalchemy>

Файл проекта: <https://github.com/selfedu-rus/flsite_sqlalchemy-23>

На предыдущем занятии мы с вами создали две таблицы, используя механизм SQLAlchemy, и объявили функцию-представления для регистрации новых пользователей. И теперь пришло время разобраться с чтением данных из таблиц.

## Выборка записей из таблиц

Как вы помните, у нас в БД две таблицы, причем, они связаны между собой через внешний ключ user\_id таблицы profiles:



Наша задача сделать выборку по пользователям, в которой бы фигурировали данные из обеих таблиц. Но для начала посмотрим, как вообще осуществляется получение данных. Для демонстрации я перейду в консоль Python и выполню команду:

from app import db, Users, Profiles

то есть, из нашего текущего модуля app импортируем переменную db и классы Users, Profiles. Далее, чтобы выбрать все записи, например, из таблицы users, следует выполнить метод all объекта query:

Users.query.all()

и на выходе получим список объектов, которые отображаются в соответствии с определением магического метода \_\_repr\_\_ в классе Users:

[<users 1>, <users 2>, <users 3>]

Здесь объект query берется из базового класса db.Model, от которого образованы классы Users и Profiles. Благодаря концепции наследования в ООП, мы автоматически получаем полный функционал для работы с таблицами БД.

Давайте теперь сохраним возвращаемый список в переменной res:

res = Users.query.all()

и посмотрим на ее содержимое. Мы видим, что это коллекция объектов и каждый объект содержит атрибуты: id, email, psw, date. Как раз те, что мы определяли в классе Users и те, что были прочитаны из соответствующей таблицы БД. То есть, мы можем обратиться к любому элементу и вывести нужное нам свойство, например, так:

res[0].email

Получим email из первой записи.

По аналогии работает метод first, только он возвращает первую запись, соответствующего запроса (или значение None, если ничего нет):

f = Users.query.first()

f.id

Далее, для выбора записей по определенному критерию можно воспользоваться методы filter\_by и filter:

Users.query.filter\_by(id = 1).all()

Users.query.filter(Users.id == 1).all()

Разница между этими методами в том, что в filter\_by передаются именованные параметры, а в filter прописывается логическое выражение. Поэтому последний метод обладает большей гибкостью, например, можно вывести все записи с id>1:

Users.query.filter(Users.id > 1).all()

Также можно делать ограничение на максимальное число возвращаемых записей:

Users.query.limit(2).all()

Выполнять сортировку по определенному полю:

Users.query.order\_by(Users.email).all()

Users.query.order\_by(Users.email.desc()).all()

Или, просто получать пользователя по значению первичного ключа:

Users.query.get(2)

Разумеется, все эти методы можно комбинировать и создавать более сложные запросы.

## Выборка из нескольких таблиц

Ну хорошо, мы увидели как можно выбирать записи из одной конкретной таблицы. Но как объединить данные, например, из двух наших таблиц и сформировать одну общую выборку? Для этого нужно соединить записи таблиц по внешнему ключу user\_id, следующим образом:

res = db.session.query(Users, Profiles).join(Profiles, Users.id == Profiles.user\_id).all()

Здесь вначале в методе query указываются таблицы, формирующие выборку. Затем, используется метод join, в котором прописывается условие связывания записей этих двух таблиц. И в конце, метод all возвращает все записи, удовлетворяющие запросу.

На выходе переменная res будет ссылаться на список, содержащий выбранные данные. К ним можно обратиться, используя следующую конструкцию:

res[0].Users.email

или

res[0].Profiles.name

Однако, SQLAlchemy предоставляет нам еще один довольно удобный механизм связывания таблиц. Если мы наперед знаем, что необходимо выбирать для каждого пользователя информацию из таблиц users и profiles, то в классе Users, как таблицы с «первичными данными», к которой подбираются соответствующие записи из «вторичной таблицы» profiles, можно прописать специальную переменную:

pr = db.relationship('Profiles', backref='users', uselist=False)

Через эту переменную будет устанавливаться связь с таблицей Profiles по внешнему ключу user\_id. Параметр backref указывает таблицу, к которой присоединять записи из таблицы profiles. Последнее значение uselist=False указывает, что одной записи из users должна соответствовать одна запись из profiles, что, в общем-то, и должно быть.

Теперь, выполняя простую команду:

res = Users.query.all()

в объектах списка будет присутствовать атрибут pr, который ссылается на объект Profiles с соответствующими данными:

res[0].pr

res[0].pr.name

Как видите, все довольно удобно. Мы воспользуемся этим механизмом и отобразим на главной странице сайта список зарегистрированных пользователей:

@app.route("/")

def index():

    info = []

    try:

        info = Users.query.all()

    except:

        print("Ошибка чтения из БД")

    return render\_template("index.html", title="Главная", list=info)

И модифицируем шаблон:

{% extends 'layout.html' %}

{% block content %}

<ul>

{% for u in list %}

<li>id: {{u.id}}, email: {{u.email}}</p>

         <ul>

         <li>Имя: {{u.pr.name}}</li>

         <li>Возраст: {{u.pr.old}}</li>

         <li>Город: {{u.pr.city}}</li>

         </ul>

</li>

{% endfor %}

</ul>

{% endblock %}

Все, теперь на главной странице видим информацию о пользователях из обеих таблиц.

## Заключение

На этом мы завершим серию занятий по микрофреймворку Flask. Конечно, коснуться всех деталей в рамках видеоуроков просто нереально. В частности, модуль SQLAlchemy нами был рассмотрен лишь обзорно, чтобы дать основные представления об этом весьма полезном расширении, которое повсеместно используется при работе с БД. И, если вы задумали создать сайт на Flask, то обязательно используйте его (или какой-либо подобный) для работы с таблицами БД. Это избавит вас в будущем от большого количества проблем, и, кроме того, при трудоустройстве по этому профилю знание SQLAlchemy будет весьма кстати. Хорошей отправной точкой в его изучении будет страница документации на русском языке:

<https://flask-sqlalchemy-russian.readthedocs.io/ru/latest/index.html>