**Введение во flask. Обработка HTML-форм**

1. [Инструменты для разработки веб-приложений](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2049/materials/4741#1)
2. [Первая страница на Flask](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2049/materials/4741#2)
3. [Статический контент](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2049/materials/4741#3)
4. [Наведем красоты (чуть-чуть)](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2049/materials/4741#4)
5. [Передача параметров в адресной строке](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2049/materials/4741#5)
6. [Обработка форм](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2049/materials/4741#6)

**Аннотация**

Наконец-то мы добрались до момента, когда мы будем использовать не чужие веб-приложения, а писать свои. И уже очень скоро мы убедимся, что с помощью Python это делать достаточно просто (особенно с использованием дополнительных сторонних библиотек).

**Инструменты для разработки веб-приложений**

В письме про окончание первого курса мы просили вас за лето познакомиться с языком разметки HTML. Вот ссылки на курсы, которые мы рекомендовали:

* [Интерактивные онлайн-курсы HTML Academy](https://htmlacademy.ru/courses)
* [Документация по фреймворку Bootstrap](https://getbootstrap.com/docs/4.2/getting-started/introduction/)
* [Веб-разработка для начинающих: HTML и CSS от Stepik](https://stepik.org/course/38218/promo)
* [ОСНОВЫ Html & CSS от Stepik](https://stepik.org/course/2621/promo)

Если вдруг вы еще этого не сделали, отложите другие дела и посмотрите эти курсы. Дальше без базового знания HTML-разметки вам будет очень сложно разрабатывать собственные веб-приложения.

В настоящее время разрабатывать веб-приложения можно практически на любом языке программирования, для каждого из которых создано не по одному десятку библиотек, облегчающих те или иные аспекты написания веб-приложений.

Многие разработчики мобильных приложений и приложений для десктопа «хитрят» и в целях экономии времени и сил создают веб-приложения просто завернутые в веб-компоненту (встроенный браузер), где вся функциональность реализована, как совокупность веб-страниц.

Одной из сильных областей Python является достаточно простое создание веб-страниц. В принципе, веб-приложения на Python можно писать и без установки дополнительных библиотек, только средствами «из коробки», так как интерпретатор поставляется со встроенным CGI (стандарт интерфейса, применяемого для связи внешней программы с веб-сервером) сервером. Для этого давайте создадим файл server.py и напишем там вот такой код:

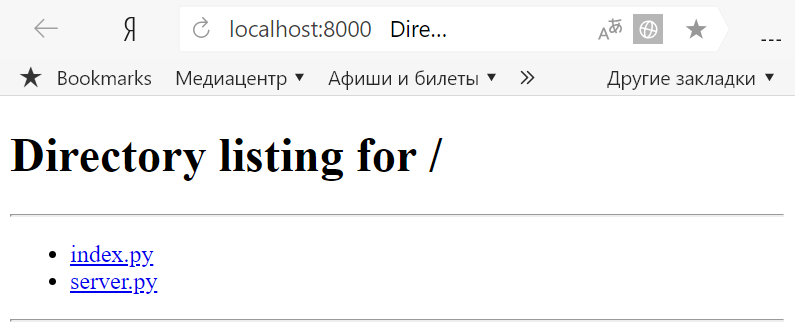
from http.server import HTTPServer, CGIHTTPRequestHandler

server\_address = ("", 8000)

httpd = HTTPServer(server\_address, CGIHTTPRequestHandler)

httpd.serve\_forever()

Этот код создает простейший веб-сервер, и если мы перейдем по адресу localhost:8000 (или 127.0.0.1:8000), то увидим список файлов той директории, в которой запущен наш сервер.



Адрес в командной строке localhost (или 127.0.0.1) — это адрес вашего компьютера для самого этого компьютера, а цифры после двоеточия показывают номер порта. Порт необходим для того, чтобы на одной машине можно было запустить несколько приложений, которые ожидают сообщений. Если утрировать, то ip-адрес — это адрес только до многоквартирного дома, а порт — номер квартиры. Номер порта — целое число от 0 до 65536. Для многих протоколов прикладного уровня есть свои стандартные порты, например для HTTP чаще всего используются 80, 8000 и 8080.

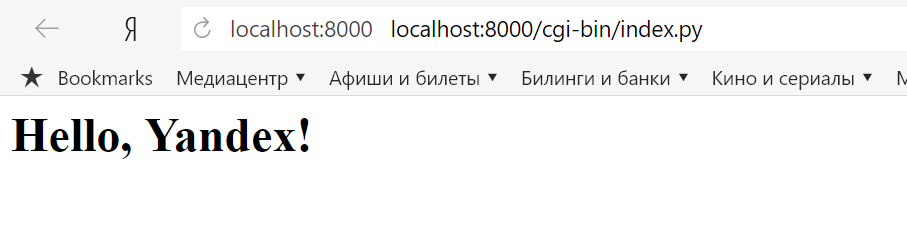
Для создания первой веб-страницы надо сделать еще несколько шагов. Давайте создадим рядом с нашим сервером папку с именем cgi-bin, внутри которой создадим скрипт с именем index.py следующего содержания:

print("Content-type: text/html; charset=utf-8")

print()

print("<h1>Hello, Yandex!</h1>")

Теперь мы можем перейти по адресу localhost:8000/cgi-bin/index.py и увидеть результат выполнения нашего скрипта.



Как вы могли заметить, на деле такой подход к созданию веб-приложений излишен и неудобен, поэтому мы его применять не будем, а будем использовать гораздо более удобные инструменты, которые построены в том числе поверх только что рассмотренного механизма.

Для Python разработано достаточно большое количество библиотек, которые значительно упрощают процесс создания веб-страниц. Некоторые из этих библиотек большие (Django, Twisted), а некоторые (Bottle, Flask) значительно меньше (их еще называют микро-фреймворками).

С чем связан размер библиотек? С тем, насколько сильно библиотека управляет архитектурой вашей системы и поддерживает ее, насколько развиты ее подсистемы, например, интеграция с разными базами данных, настройками, файловыми хранилищами и т. д. Небольшие библиотеки всего лишь скрывают от вас сетевой уровень, давая вам возможность управлять формированием страницы в зависимости от параметров. Остальное вы должны реализовывать сами.

**Первая страница на Flask**

Разумеется, наши веб-приложения не будут огромными, сразу делать корпоративные [интранет](https://ru.wikipedia.org/wiki/Интранет) порталы — довольно-таки непростая задача. Мы начнем с ресурсов с небольшим количеством страниц и ограниченной функциональностью, чтобы рассмотреть лишь базовые механизмы того, как все работает. К счастью, ключевые принципы построения веб-приложений одинаковы вне зависимости от используемого фреймворка.

Для наших целей прекрасно подойдет микрофреймворк Flask. Он популярный, с хорошей документацией, легок в освоении и при этом вполне пригоден для создания больших промышленных веб-приложений. После его освоения вам не составит труда «пощупать» Bottle самостоятельно, так как принципы работы у них очень похожи, а затем перейти к другим библиотекам.

Flask не входит в стандартную библиотеку Python, чтобы установить его необходимо выполнить команду:

pip install flask

После установки библиотеки давайте создадим наше первое веб-приложение на Flask:

from flask import Flask

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

@app.route('/')

@app.route('/index')

def index():

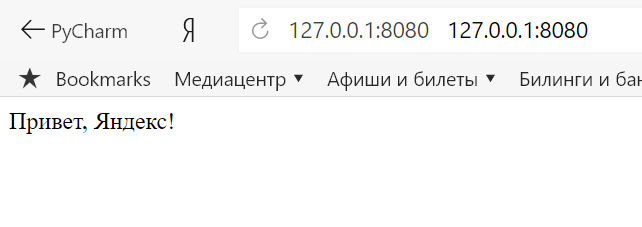
return "Привет, Яндекс!"

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=8080, host='127.0.0.1')

Наш скрипт создает объект приложения как экземпляр класса *Flask*, который мы предварительно импортировали из пакета flask. Декораторы app.route над функцией *index* используются для регистрации нашей функции, как функции обратного вызова для определенных событий. В нашем случае создается связь между адресом (URL) в браузере ('/' и '/index') и функцией *index*. Это означает, что когда веб-браузер запрашивает один из этих двух URL-адресов, Flask будет вызывать эту функцию и передать возвращаемое значение обратно в браузер в качестве ответа.

Давайте запустим наш скрипт и перейдем на страницу http://127.0.0.1:8080, что соответствует URL '/', или на страницу http://127.0.0.1:8080/index, чтобы увидеть результат выполнения функции *index*.



С помощью декоратора app.route мы можем создавать сколько угодно страниц со своими собственными адресами. Давайте добавим еще одну страницу с обратным отсчетом:

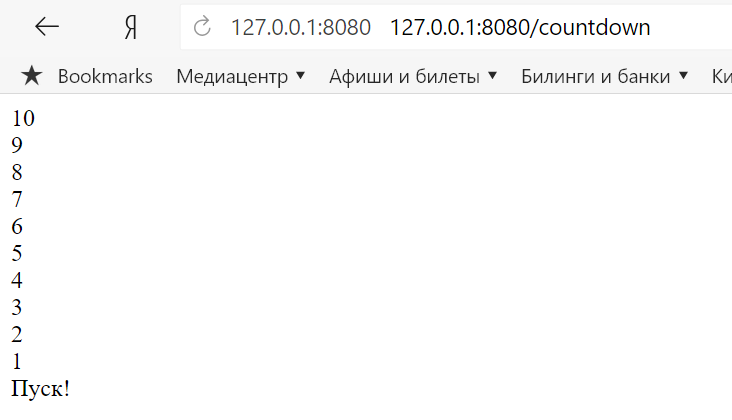
@app.route('/countdown')

def countdown():

countdown\_list = [str(x) for x in range(10, 0, -1)]

countdown\_list.append('Пуск!')

return '</br>'.join(countdown\_list)



**Статический контент**

Практически любой веб-сайт содержит большое количество разнообразного статического контента, к которому могут относиться следующие виды информации:

* Изображения
* Таблицы стилей CSS
* Шрифты
* Файлы для скачивания
* Файлы скриптов JavaScript
* Музыка, видео
* И т. д.

Всю подобную информацию Flask по умолчанию ищет в директории static, поэтому давайте ее создадим. Для более аккуратной организации файлов рекомендуется создавать подпапки в static для каждого типа статического контента, который у вас есть, например, static/img, static/fonts и т. д.

**Важное замечание:** если тут и далее после изменения что-то не отображается, возможно, вы не перезапустили ваше веб-приложение; запустили новую версию параллельно со старой и браузер посылает запросы именно предыдущей версии; или браузер закешировал какие-то данные (сохранил к себе, чтобы отдавать пользователю без запроса к серверу для ускорения работы). Чтобы это поправить, обновите страницу с очисткой кеша — Shift + F5.

Давайте разместим хорошо известную нам сову Риану на отдельной странице.



Для этого в папке static создадим подпапку img, разместим там наше изображение, импортируем из модуля flask функцию *url\_for*. И добавим в наше веб-приложение следующий код:

@app.route('/image\_sample')

def image():

return f'''<img src="{url\_for('static', filename='img/riana.jpg')}"

alt="здесь должна была быть картинка, но не нашлась">'''

Если бы название файла с картинкой было на русском языке, оно перекодировалось бы в коды символов Unicode, что можно увидеть, посмотрев код страницы в браузере.

В принципе, никто нам не запрещает написать путь к файлу вот так:

return '''<img src="/static/img/riana.jpg" alt="здесь должна была быть картинка, но не нашлась">'''

Однако при создании даже небольшого проекта вы можете столкнуться с тем, что ваши пути перестанут работать, так как кто-то переопределил у микро-фреймворка параметр, отвечающий за местонахождение директории со статическими файлами, а наш первоначальный вариант сохранит работоспособность.

Точно по такому же принципу работает и остальной статический контент. Давайте рассмотрим еще один пример: сделаем обработчик *return\_sample\_page*, который вернет пользователю нашу первую HTML-страницу, сверстанную по всем правилам:

@app.route('/sample\_page')

def return\_sample\_page():

return """<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Привет, Яндекс!</title>

</head>

<body>

<h1>Первая HTML-страница</h1>

</body>

</html>"""

А теперь давайте добавим css-файл, который заменит цвет текста на красный. Для этого создадим в папке css внутри папки со статическим контентом файл style.css со следующим текстом:

h1 {

color: #d22e3a

}

Заменим возвращаемое значение на:

f"""<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

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

<title>Привет, Яндекс!</title>

</head>

<body>

<h1>Первая HTML-страница</h1>

</body>

</html>"""

Перезапустим приложение, обновим страницу и убедимся, что цвет текста изменился на указанный.

**Наведем красоты (чуть-чуть)**

Наш курс ориентирован на изучение технологий, которые используются при создании промышленных приложений, и, к сожалению, создание стилей и верстка HTML не входит в цели нашего курса (да и просто времени не хватит, так как это отдельная большая тема). Но, согласитесь, хочется как-то относительно просто делать веб-приложения, которые выглядят симпатично, поэтому давайте обратимся к такому набору инструментов, как Bootstrap.

Bootstrap — это свободный набор инструментов для создания сайтов и веб-приложений, который включает в себя скрипты, стили, иконки и многое другое. Набор расширяемый и достаточно гибкий. Кроме того, у него огромное сообщество, которое предлагает большое количество уже готовых тем и компонентов для Bootstrap, большая часть которых либо свободно распространяемая, либо бесплатная, либо стоит не запредельно много даже по меркам бюджета школьного/студенческого проекта.

В своем первоначальном виде Bootstrap частенько используется программистами как отправная точка для создания веб-приложения без участия дизайнеров и верстальщиков. Инструкцию по подключению и использованию Bootstrap можно найти на [официальном сайте](https://getbootstrap.com/). Само подключение состоит из нескольких частей:

* Подключение стилей
* Подключение JavaScript

Для нашего следующего примера достаточно подключить только стили, но для использования всех возможностей Bootstrap необходимо выполнить оба шага.

Давайте сделаем функцию, которая будет нам отдавать простую страничку с подключением Bootstrap и несколькими элементами на ней. Примерно вот так:

@app.route('/bootstrap\_sample')

def bootstrap():

return '''<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<link rel="stylesheet"

href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta1/dist/css/bootstrap.min.css"

integrity="sha384-giJF6kkoqNQ00vy+HMDP7azOuL0xtbfIcaT9wjKHr8RbDVddVHyTfAAsrekwKmP1"

crossorigin="anonymous">

<title>Привет, Яндекс!</title>

</head>

<body>

<h1>Привет, Яндекс!</h1>

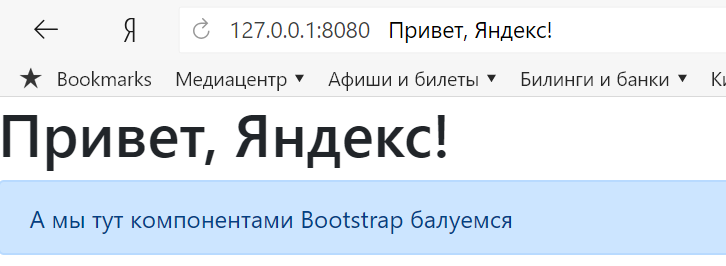
<div class="alert alert-primary" role="alert">

А мы тут компонентами Bootstrap балуемся

</div>

</body>

</html>'''



Как вы могли заметить, даже создание простых страниц напрямую из кода выглядит громоздко, а при небольшом увеличении сложности страницы трудоемкость написания и поддержки возрастает значительно. Поэтому вариант, который мы сегодня рассматривали, годится только для того, чтобы попрактиковаться с библиотекой и языком разметки HTML. На следующем уроке мы рассмотрим, как решается эта проблема.

Сейчас же мы добавим в наше приложение некоторую динамику путем передачи информации от пользователя на сервер. В общем-то, мы можем влиять на состояние сервера и сейчас, например, изменяя значение глобальной переменной:

from flask import Flask, url\_for

i = 0

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

@app.route('/i')

def show\_i():

global i

i += 1

return str(i)

Но это плохая практика, не делайте так, если от этого не зависит ваша жизнь. Единственное, на что можно обратить внимание в этом примере, это тот факт, что если мы попробуем возвратить этой функцией просто число, то получим страницу с ошибкой следующего содержания:

TypeError: 'int' object is not callable

The view function did not return a valid response.

The return type must be a string, tuple, Response instance,

or WSGI callable, but it was a int.

Данное сообщение недвусмысленно намекнет нам на то, каким может быть возвращаемое значение у функций, украшенных декоратором app.route.

**Передача параметров в адресной строке**

Статичные страницы делать не интересно, поэтому давайте рассмотрим то, как мы можем взаимодействовать с пользователем нашего веб-приложения. Таких способов несколько, давайте начнем с передачи параметров в адресной строке.

Декоратор app.route умеет принимать на вход значение одного или нескольких параметров, которые пишется после завершающего слэша в пути внутри треугольных скобок <>, вот так:

@app.route('/greeting/<username>')

def greeting(username):

return f'''<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<link rel="stylesheet"

href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta1/dist/css/bootstrap.min.css"

integrity="sha384-giJF6kkoqNQ00vy+HMDP7azOuL0xtbfIcaT9wjKHr8RbDVddVHyTfAAsrekwKmP1"

crossorigin="anonymous">

<title>Привет, {username}</title>

</head>

<body>

<h1>Привет, {username}!</h1>

</body>

</html>'''

Обратите внимание, что в данном случае меняется и сигнатура функции, которая теперь принимает на вход одноименный параметр. Кроме того, этот параметр — обязательный.

Flask позволяет указывать несколько параметров, а также явно определять их тип с помощью конвертера. Рассмотрим обе этих возможности в следующем примере:

@app.route('/two\_params/<username>/<int:number>')

def two\_params(username, number):

return f'''<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<link rel="stylesheet"

href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta1/dist/css/bootstrap.min.css"

integrity="sha384-giJF6kkoqNQ00vy+HMDP7azOuL0xtbfIcaT9wjKHr8RbDVddVHyTfAAsrekwKmP1"

crossorigin="anonymous">

<title>Пример с несколькими параметрами</title>

</head>

<body>

<h2>{username}</h2>

<div>Это первый параметр и его тип: {str(type(username))[1:-1]}</div>

<h2>{number}</h2>

<div>Это второй параметр и его тип: {str(type(number))[1:-1]}</div>

</body>

</html>'''

Всего таких конвертеров пять:

| **Имя конвертера** | **Описание** |
| --- | --- |
| string | (по умолчанию) любой текст без слешей |
| int | положительное целое число |
| float | положительное дробное число |
| path | строка, но может содержать слеши, для передачи некоторого URL-пути |
| uuid | стандарт строк-идентификаторов из 16 байт в шестнадцатеричном представлении. Выглядит примерно так: 550e8400-e29b-41d4-a716-446655440000 |

Передача параметров в адресе командной строки используется достаточно часто для динамического формирования однотипных страниц. Например, на сайте pypi.org такой способ применяется для формирования страниц с информацией о каждом конкретном модуле.

**Обработка форм**

Передача параметра в адресной строке — это скорее не инструмент взаимодействия с пользователем, а динамическое формирование для него ссылок на разные объекты внутри веб-приложения. Для непосредственного общения с пользователем такой способ неудобен, так как в общем случае параметров может быть много, а предлагать пользователю заносить их в адресную строку руками — не лучшая идея.

Для взаимодействия с пользователем применяется другой механизм — формы. HTML-разметка позволяет создавать элементы для ввода данных разных типов, с которыми Flask, конечно, умеет работать. Давайте вспомним, какие типы элементов ввода поддерживает HTML.

Вообще говоря, разных типов полей ввода довольно много, и периодически в новые версии языка разметки добавляются дополнительные. Вот некоторые из таких типов:

* button — кнопка
* checkbox — множественный выбор
* color — поле выбора цвета
* date, datetime, datetime-local, month, time, week — ввод даты и времени
* email — поле для ввода адреса электронной почты
* file — поле для выбора файла
* number — поле для ввода числовой информации
* password — поле для ввода пароля
* radio — выбор одного из нескольких вариантов
* range — ползунок (как в музыкальном или видео-плеере)
* submit — кнопка для отправки формы
* tel — поле для ввода телефона
* text — поле для ввода текста
* url — поле для ввода адреса в Интернете

Давайте сделаем форму с несколькими самыми распространенными типами полей ввода, для их стилизации используем Bootstrap. Чтобы увидеть, в каком виде информация из этих полей придет на сервер нашего веб-приложения, напишем такой код (предварительно импортировав request из Flask):

@app.route('/form\_sample', methods=['POST', 'GET'])

def form\_sample():

if request.method == 'GET':

return f'''<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<link rel="stylesheet"

href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta1/dist/css/bootstrap.min.css"

integrity="sha384-giJF6kkoqNQ00vy+HMDP7azOuL0xtbfIcaT9wjKHr8RbDVddVHyTfAAsrekwKmP1"

crossorigin="anonymous">

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

<title>Пример формы</title>

</head>

<body>

<h1>Форма для регистрации в суперсекретной системе</h1>

<div>

<form class="login\_form" method="post">

<input type="email" class="form-control" id="email" aria-describedby="emailHelp" placeholder="Введите адрес почты" name="email">

<input type="password" class="form-control" id="password" placeholder="Введите пароль" name="password">

<div class="form-group">

<label for="classSelect">В каком вы классе</label>

<select class="form-control" id="classSelect" name="class">

<option>7</option>

<option>8</option>

<option>9</option>

<option>10</option>

<option>11</option>

</select>

</div>

<div class="form-group">

<label for="about">Немного о себе</label>

<textarea class="form-control" id="about" rows="3" name="about"></textarea>

</div>

<div class="form-group">

<label for="photo">Приложите фотографию</label>

<input type="file" class="form-control-file" id="photo" name="file">

</div>

<div class="form-group">

<label for="form-check">Укажите пол</label>

<div class="form-check">

<input class="form-check-input" type="radio" name="sex" id="male" value="male" checked>

<label class="form-check-label" for="male">

Мужской

</label>

</div>

<div class="form-check">

<input class="form-check-input" type="radio" name="sex" id="female" value="female">

<label class="form-check-label" for="female">

Женский

</label>

</div>

</div>

<div class="form-group form-check">

<input type="checkbox" class="form-check-input" id="acceptRules" name="accept">

<label class="form-check-label" for="acceptRules">Готов быть добровольцем</label>

</div>

<button type="submit" class="btn btn-primary">Записаться</button>

</form>

</div>

</body>

</html>'''

elif request.method == 'POST':

print(request.form['email'])

print(request.form['password'])

print(request.form['class'])

print(request.form['file'])

print(request.form['about'])

print(request.form['accept'])

print(request.form['sex'])

return "Форма отправлена"

Заодно давайте настроим нашу форму в style.css, который подключим после стилей Bootstrap:

form.login\_form {

margin-left: auto;

margin-right: auto;

max-width: 450px;

background-color: #ffcc00;

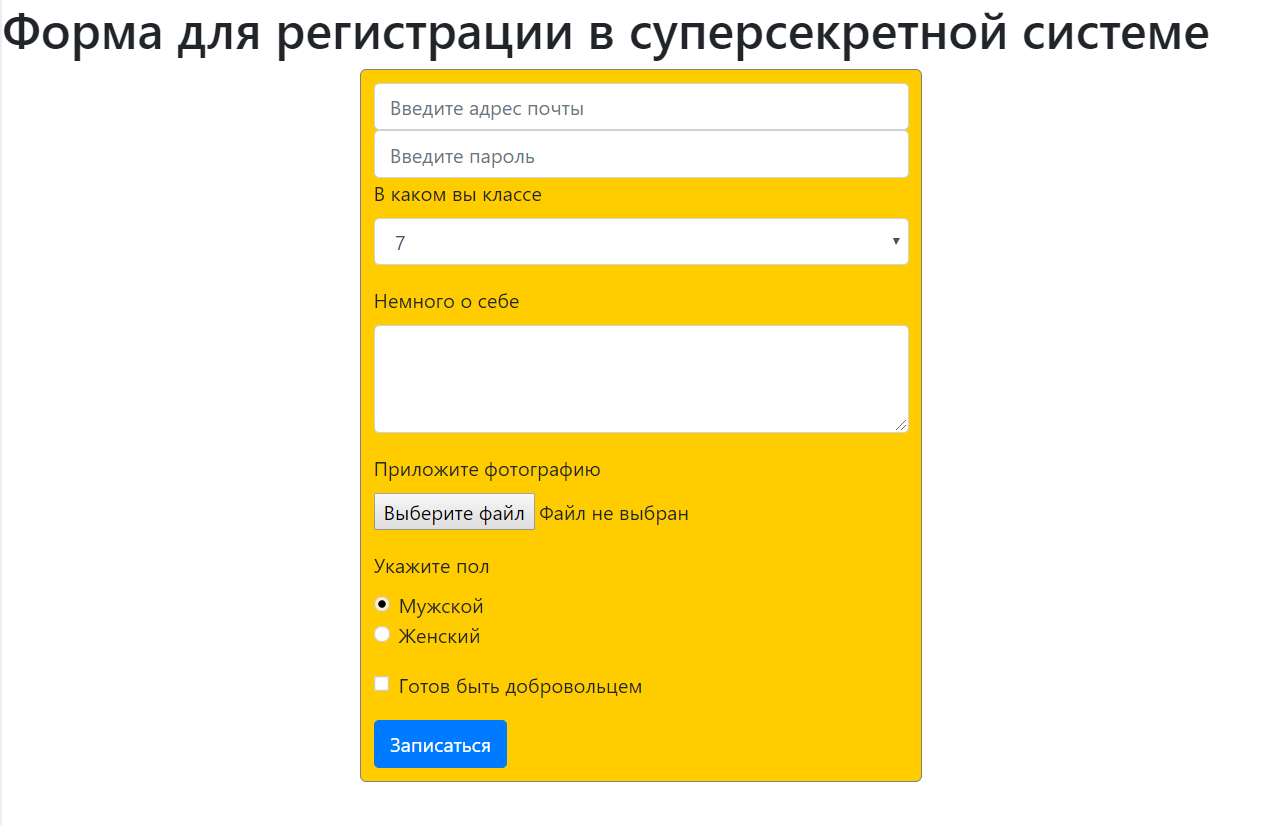
border: 1px solid gray;

border-radius: 5px;

padding: 10px;

}

Если мы все сделали правильно, увидим вот такой впечатляющий результат.



Мы дополнили наш декоратор app.route новым параметром — списком методов протокола HTTP, с которыми он работает. Всего методов довольно много, но мы будем говорить только о пяти из них:

1. GET — запрашивает данные, не меняя состояния сервера («прочитать»).
2. POST — отправляет данные на сервер («отправить»).
3. PUT — заменяет все текущие данные сервера данными запроса («заменить»).
4. DELETE — удаляет указанные данные («удалить»).
5. PATCH — используется для частичного изменения данных («изменить»).

Таким образом, наша функция *form\_sample* работает с двумя методами. Если мы хотим получить данные с сервера, тогда отрабатывает ветка условия, в которой мы отправляем пользователю форму для заполнения. Когда пользователь заполнил форму и нажал на кнопку «Записаться», данные формы заворачиваются в специальный аналог словаря в сущности *request*, которая хранит всю информацию о пользовательском запросе.

Обратите внимание: поскольку данные формы заворачиваются в аналог словаря, может случиться ситуация, когда по ключу мы не сможем найти значение (например, если пользователь не отметит чекбокс), и тогда выбросится исключение. Чтобы этого избежать, лучше обращаться к данным не напрямую по ключу, а через метод *get*. Таким образом, более корректная обработка значения чекбокса будет выглядеть так:

request.form.get('accept')

Как вы могли заметить, произошла небольшая странность с приложенным файлом: мы достали только его название, а не содержимое. Это произошло потому, что содержимое файла хранится в другом месте. Давайте напишем еще один пример:

@app.route('/sample\_file\_upload', methods=['POST', 'GET'])

def sample\_file\_upload():

if request.method == 'GET':

return f'''<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<link rel="stylesheet"

href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta1/dist/css/bootstrap.min.css"

integrity="sha384-giJF6kkoqNQ00vy+HMDP7azOuL0xtbfIcaT9wjKHr8RbDVddVHyTfAAsrekwKmP1"

crossorigin="anonymous">

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

<title>Пример загрузки файла</title>

</head>

<body>

<h1>Загрузим файл</h1>

<form method="post" enctype="multipart/form-data">

<div class="form-group">

<label for="photo">Выберите файл</label>

<input type="file" class="form-control-file" id="photo" name="file">

</div>

<button type="submit" class="btn btn-primary">Отправить</button>

</form>

</body>

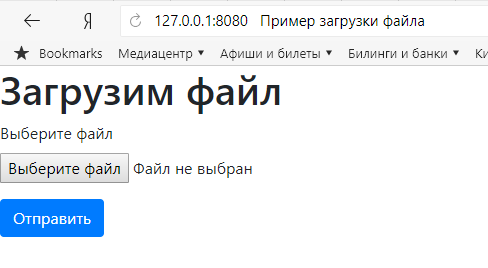
</html>'''

elif request.method == 'POST':

f = request.files['file']

print(f.read())

return "Форма отправлена"



Обратите внимание: кроме доступа к файлу в *request.files* мы немного изменили саму разметку формы, добавив в нее параметр

enctype="multipart/form-data"

Иначе форма так и продолжит отправлять только имена файлов, а при попытке доступа к самому файлу мы получим ошибку.

Все достаточно просто, не так ли? Пока самая большая сложность, которая нас преследует — это громоздкая и неудобная генерация HTML-разметки внутри функции, от этого недостатка мы избавимся уже на следующем уроке.

**Шаблоны. flask-wtf**

1. [Шаблоны](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2050/materials/4773#1)
2. [Условия в шаблонах](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2050/materials/4773#2)
3. [Циклы в шаблонах](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2050/materials/4773#3)
4. [Создание переменных в шаблоне](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2050/materials/4773#4)
5. [Наследование шаблонов](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2050/materials/4773#5)
6. [Знакомство с Flask-WTF](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2050/materials/4773#6)

**Аннотация**

На этом уроке избавимся от необходимости создавать HTML-разметку непосредственно в коде, а также рассмотрим возможность создания форм с использованием объектного подхода с помощью библиотеки flask-wtf.

**Шаблоны**

Делать разметку непосредственно в коде Python плохо в 99,99 % случаев. Это сложно поддерживать, неудобно писать, и наверняка вы почувствовали дискомфорт, пока делали предыдущие примеры. А у нас были достаточно простые страницы и небольшое количество информации, которая менялась динамически.

Для того чтобы сделать жизнь программистов лучше, во Flask есть прекрасный механизм создания HTML-шаблонов, который мы сейчас и рассмотрим.

Практически всегда отделение логики приложения от макетов веб-страниц — отличная идея. Таким образом достигается нормальная организация внутри команды и становится возможным разделение работ. Каждый занимается своим делом: веб-дизайнер делает красиво, а разработчик — чтобы работало. Шаблоны как раз помогают достичь этого разделения. Во Flask шаблоны записываются как отдельные файлы, хранящиеся в папке templates, которая находится (по умолчанию) в корневой папке приложения. Давайте ее создадим. И добавим в эту папку файл с HTML-разметкой — index.html со следующим содержимым:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

</head>

<body>

<h1>Привет, {{ username }}!</h1>

</body>

</html>

Для нас такая разметка не представляет ничего сложного. Интерес вызывают разве что непонятные параметры в двух фигурных скобках. Давайте посмотрим, как шаблон будет работать. Для этого импортируем *render\_template* из flask и напишем для нашего приложения новый обработчик главной страницы:

@app.route('/')

@app.route('/index')

def index():

user = "Ученик Яндекс.Лицея"

return render\_template('index.html', title='Домашняя страница',

username=user)

И никакой разметки в нашем python-файле — прекрасно, не правда ли? Операция, которая преобразует шаблон в HTML-страницу, называется **рендерингом**. Чтобы отобразить шаблон, нам пришлось импортировать функцию render\_template(). Эта функция принимает имя файла шаблона и перечень аргументов шаблона и возвращает тот же шаблон, но при этом все блоки {{...}} в нем заменяются фактическими значениями переданных аргументов. Этот процесс работает схоже с прекрасно знакомым нам методом *format* для строк в Python. Механизм шаблонов, встроенный во Flask, называется Jinja2.

Запустите приложение и перейдите по ссылке http://127.0.0.1:8080/, из контекстного меню по правому клику в браузере выбирите «Показать исходный код» или «view page source». Посмотрите, какой HTML-код был сгенерирован на основе шаблона.

Важно не забывать, что кроме непосредственно переданных параметров внутри шаблона мы имеем доступ и к служебным объектам. Например, уже знакомый нам *request*, или *session*, о котором мы поговорим позже.

Параметры в шаблон не обязательно передавать по отдельности, так как их может быть много и получать мы их можем внутри нашей потенциально длиной функции в разных местах. Можно собирать их в словарь параметров, а потом распаковывать его в вызове функции *render\_template*:

@app.route('/')

@app.route('/index')

def index():

param = {}

param['username'] = "Ученик Яндекс.Лицея"

param['title'] = 'Домашняя страница'

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

Если мы не заполним какой-то из параметров шаблона, то он по умолчанию будет считаться равным пустой строке.

Кроме простой подстановки параметров Jinja2 умеет делать еще несколько полезных вещей. Речь о них пойдет далее.

**Условия в шаблонах**

Шаблонизатор Flask поддерживает условные операторы, заданные внутри блоков {% ...%}. Давайте создадим шаблон odd\_even.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Четное-нечетное</title>

</head>

<body>

{% if number % 2 == 0 %}

<div>{{ number }} - чётное</div>

{% else %}

<div>{{ number }} - нечётное</div>

{% endif %}

</body>

</html>

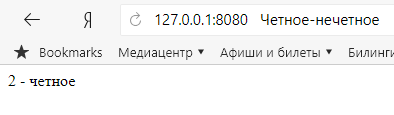
И обработчик:

@app.route('/odd\_even')

def odd\_even():

return render\_template('odd\_even.html', number=2)

Теперь в зависимости от того, что мы передали в шаблон, будет отрабатывать тот или иной блок.



Общий синтаксис условного оператора в шаблонах очень похож на тот, к которому мы привыкли в Python:

{% if условие\_1 %}

ветка 1

{% elif условие\_2 %} (не обязательно)

ветка 1

{% else %} (не обязательно)

ветка 2

{% endif %}

Поддерживаются вложенные условия.

**Циклы в шаблонах**

Jinja2 поддерживает еще и циклы *for*. Давайте создадим тестовый json-файл со списком новостей примерно следующего содержания:

{

"news": [

{

"title": "Сегодня хорошая погода",

"content": "Невероятно, сегодня хорошая погода"

},

{

"title": "Завтра хорошая погода",

"content": "С ума сойти, и завтра хорошая погода"

},

{

"title": "Послезавтра дождь",

"content": "Все вошло в норму"

}

]

}

Напишем обработчик:

@app.route('/news')

def news():

with open("news.json", "rt", encoding="utf8") as f:

news\_list = json.loads(f.read())

print(news\_list)

return render\_template('news.html', news=news\_list)

И шаблон:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Новости</title>

</head>

<body>

{% for item in news['news'] %}

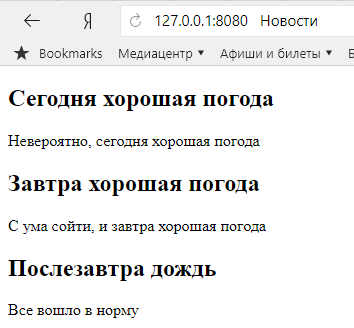
<h2>{{item["title"]}}</h2>

<div>{{item["content"]}}</div>

{% endfor %}

</body>

</html>



Как вы можете заметить, общий синтаксис цикла выглядит так:

{% for переменная цикла in набор значений %}

код

{% endfor %}

В качестве набора значений может выступать все то же, что и в обычном цикле на Python. Можно использовать и *range* (тогда можно смоделировать ситуации, когда значение в шаблон передавать не нужно, а цикл все равно сработает). Поддерживается вложенность.

Внутри цикла доступна переменная *loop* с рядом полезных атрибутов, например:

* index — индекс итерации с 1
* index0 — индекс итерации с нуля
* first — True, если первая итерация, иначе False
* last — True, если последняя итерация, иначе False

**Создание переменных в шаблоне**

Несмотря на то что практически все значения для шаблона мы передаем из нашего кода на Python, периодически возникают ситуации, когда возникает необходимость создать переменную непосредственно в шаблоне, например, для хранения промежуточного результата вычислений. Это можно сделать с помощью ключевого слова *set*.

{% set a = 10 %}

Давайте рассмотрим пример для иллюстрации создания переменных в шаблоне, а заодно посмотрим на переменную *loop* внутри цикла:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

</head>

<body>

<h1>Кто стоит в очереди?</h1>

{% set user\_list = ['Ваня', 'Петя', 'Саша', 'Кирилл'] %}

<ul>

{% for user in user\_list %}

<li>{{ loop.index }} - {{ user }} {% if loop.first %} первый в очереди {% elif loop.last %} последний. {% endif %}

</li>

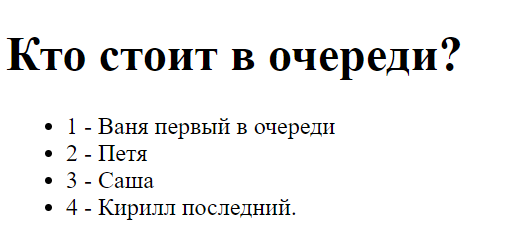
{% endfor %}

</ul>

</body>

</html>

Добавим обработчик, запустим:



**Наследование шаблонов**

В большинстве веб-приложений вверху или сбоку страницы есть главное меню, панели навигации с несколькими часто используемыми ссылками, внизу страницы зачастую располагается подвал (футер) с контактной информацией и т. д. Если веб-приложение содержит несколько страниц, не составит большого труда добавить такую информацию во все шаблоны. Но по мере увеличения масштаба это будет становиться все труднее и труднее. Может возникнуть ситуация, когда при изменении номера телефона или добавлении нового пункта меню придется изменить несколько сотен шаблонов. Кроме того, вы помните, что надо переиспользовать код, где это возможно, а писать одно и тоже несколько раз — плохая практика.

Jinja2 имеет функцию наследования шаблона, которая решает эту проблему. Мы можем разместить общие для всех шаблонов части макета страницы в базовом шаблоне, из которого выводятся все остальные шаблоны.

Давайте создадим базовый шаблон, который будет содержать небольшое верхнее меню, в файле base.html:

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<link rel="stylesheet"

href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"

integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"

crossorigin="anonymous">

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

</head>

<body>

<header>

<nav class="navbar navbar-light bg-light">

<a class="navbar-brand" href="#">Наше приложение</a>

</nav>

</header>

<!-- Begin page content -->

<main role="main" class="container">

{% block content %}{% endblock %}

</main>

</body>

</html>

В базовом шаблоне используется оператор управления блоком {% block %}{% endblock %}, чтобы определить место, куда будет вставляться содержимое дочерних шаблонов. Блокам присваивается уникальное имя (в нашем случае — content), на которое производные шаблоны могут ссылаться.

Давайте изменим и наш index.html, который теперь будет выглядеть следующим образом:

{% extends "base.html" %}

{% block content %}

<h1>Привет, {{ username }}!</h1>

{% endblock %}

В *extends* мы указываем, какой шаблон мы хотим расширить, а в *block* — какой именно блок (их может быть несколько).

Обычно базовые шаблоны делают таким образом, чтобы они отвечали за общую структуру страницы. Новые страницы веб-приложения создают как производные шаблоны из одного и того же базового шаблона для избежания дублирования кода. Дочерний шаблон может расширять несколько блоков родительского шаблона, и при этом сам быть родительским для другого шаблона. Про все тонкости можно почитать в [официальной документации](https://jinja.palletsprojects.com/en/2.10.x/templates/#template-inheritance).

**И еще.**

Jinja поддерживает еще много интересного:

* Макросы — аналог функций из Python — куски кода шаблона, которые можно переиспользовать, обращаясь по имени
* Фильтры — некоторые функции, которые можно применять к данным при выводе, в большинстве случаев они дублируют функциональность методов строк, и позволяют немного модифицировать данные при отображении в шаблоне

Про это и много другое можно почитать в [документации](https://jinja.palletsprojects.com/en/2.10.x/).

**Знакомство с Flask-WTF**

Микрофреймворк Flask силен, в том числе, и своей расширяемостью, которая позволяет значительно наращивать функциональность веб-приложения за счет дополнительных модулей с небольшими усилиями. Мы рассмотрим модуль flask-wtf для создания и обработки форм. У вас может возникнуть закономерный вопрос — зачем, ведь мы уже научились работать с формами? На самом деле, работа со сложными формами через разметку все равно достаточно непростая задача, а flask-wtf помогает не только скрыть эту сложность, но и использует при этом объектно-ориентированный подход.

Чтобы установить flask-wtf, достаточно выполнить команду:

pip install flask-wtf

Прежде чем приступить к дальнейшей работе, давайте сделаем небольшую настройку нашего приложения и добавим следующую строку после создания переменной *app*:

app.config['SECRET\_KEY'] = 'yandexlyceum\_secret\_key'

Эта настройка защитит наше приложение от [межсайтовой подделки запросов](https://ru.wikipedia.org/wiki/Межсайтовая_подделка_запроса).

Конечно, наш придуманный ключ довольно простой, но этот параметр необходим для корректной работы модуля. В принципе, защиту от CSRF-атаки можно отключить, но это не рекомендуется даже в учебных приложениях, как наше, чтобы при разработке своих больших проектов вы про это ненароком не забыли. Хорошая идея — хранить настройки приложения в отдельном файле конфигурации и считывать их при старте приложения.

Давайте создадим форму авторизации для входа в наше абстрактное приложение, которая будет содержать текстовое поле для ввода логина, поле для ввода пароля, чекбокс «Запомнить меня» и кнопку отправки формы на сервер. Для начала создадим класс нашей будущей формы. Создадим файл loginform.py, в котором напишем следующий код:

from flask\_wtf import FlaskForm

from wtforms import StringField, PasswordField, BooleanField, SubmitField

from wtforms.validators import DataRequired

class LoginForm(FlaskForm):

username = StringField('Логин', validators=[DataRequired()])

password = PasswordField('Пароль', validators=[DataRequired()])

remember\_me = BooleanField('Запомнить меня')

submit = SubmitField('Войти')

Как видно из примера, мы импортируем класс *FlaskForm* из модуля flask\_wtf — основной класс, от которого мы будем наследоваться при создании своей формы. Из модуля wtforms (flask\_wtf — обертка для этого модуля) мы импортируем типы полей, которые нам пригодятся для создания нашей формы: текстовое поле, поле ввода пароля, булевое поле (из него получается чекбокс), и кнопку отправки данных.

Кроме этого, из модуля wtforms.validators импортируем проверку, которая скажет нам о том, введены ли данные в поле или нет. Создаем необходимые поля, на поля ввода логина и пароля вешаем проверку наличия там введенной информации.

Прежде чем добраться до шаблона, давайте напишем обработчик, который будет оперировать пока не созданным шаблоном login.html (шаблон мы оставим на потом, чтобы посмотреть несколько вариантов):

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

def login():

form = LoginForm()

if form.validate\_on\_submit():

return redirect('/success')

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

Тут мы создаем нашу форму, и если все поля прошли валидацию, после нажатия на кнопку отправки данных отправляем нашего пользователя на страницу удачного логина (не забудьте ее сначала создать, а также импортировать *redirect* из модуля flask).

Теперь перейдем к шаблону. Давайте создадим новый шаблон login.html, который будет расширять уже существующий шаблон base.html. Есть несколько вариантов того, как мы можем отобразить поля нашей формы *LoginForm*. Если мы хотим просто создать поля и отобразить их с минимальной настройкой внешнего вида, тогда можно просто обойти их в цикле вот таким достаточно универсальным кодом:

{% extends "base.html" %}

{% block content %}

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

<div>

{{ form.csrf\_token }}

</div>

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

<div>

{{ field.label() }}

{{ field() }}

{% for error in field.errors %}

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

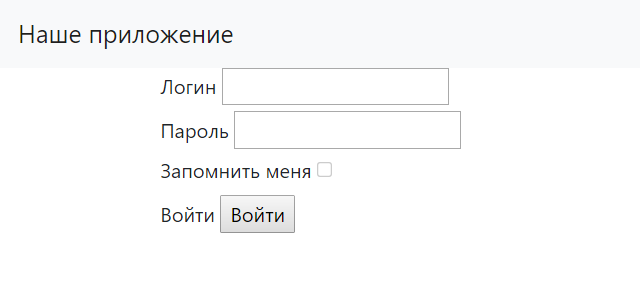
{% endfor %}

</div>

{% endfor %}

</form>

{% endblock %}



Если нам нужно больше контроля над отображением полей, можно обращаться в верстке к каждому из них по отдельности:

{% extends "base.html" %}

{% block content %}

<h1>Авторизация</h1>

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

{{ form.hidden\_tag() }}

<p>

{{ form.username.label }}<br>

{{ form.username(class="form-control") }}<br>

{% for error in form.username.errors %}

<div class="alert alert-danger" role="alert">

{{ error }}

</div>

{% endfor %}

</p>

<p>

{{ form.password.label }}<br>

{{ form.password(class="form-control", type="password") }}<br>

{% for error in form.password.errors %}

<div class="alert alert-danger" role="alert">

{{ error }}

</div>

{% endfor %}

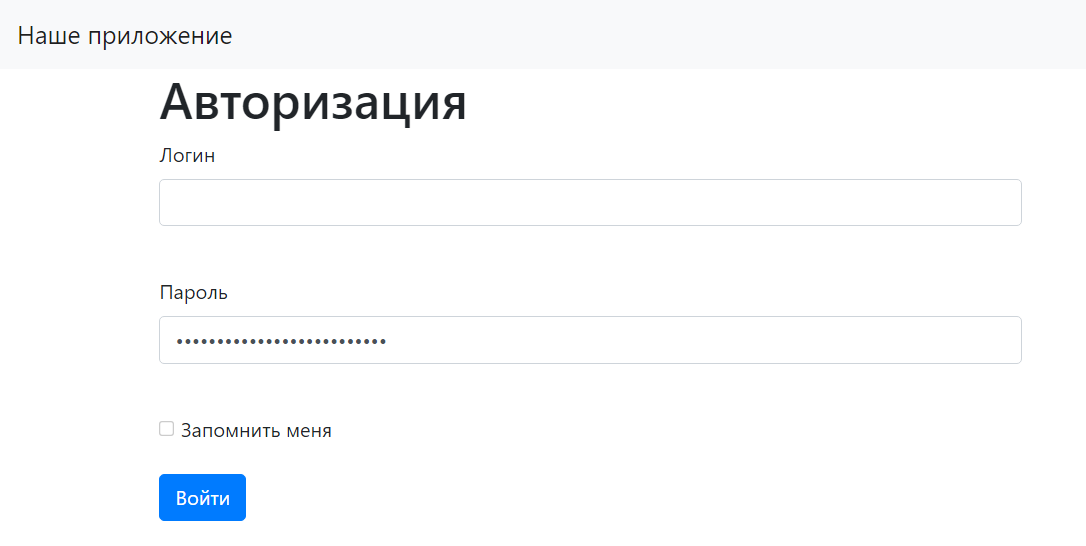
</p>

<p>{{ form.remember\_me() }} {{ form.remember\_me.label }}</p>

<p>{{ form.submit(type="submit", class="btn btn-primary") }}</p>

</form>

{% endblock %}



Как видите, здесь для создания формы мы оперируем уже не разметкой, а атрибутами объекта *form*. *form.hidden\_tag* — атрибут, который добавляет в форму токен для защиты от атаки, о которой мы говорили раньше. Из интересного в шаблоне есть еще циклы, которые добавляют вывод ошибок заполнения полей. В нашем случае валидатор только один, но в общем случае их может быть несколько, поэтому ошибки лучше выводить именно таким образом.

Если мы хотим добавить к компонентам какой-нибудь стиль, класс или указать какой-то другой атрибут, то их можно просто передать как параметры к вызову нужной части формы.

Проверьте, как все работает.

Обратите внимание: проверка правильности значений полей в нашем случае ведется на сервере. Поэтому важно поставить у нашей формы в шаблоне параметр *novalidate*, иначе Bootstrap будет проверять поля прямо в браузере и до сервера информация не дойдет. Вообще значения полей лучше проверять и там, и там:

* На клиенте для удобства пользователя (ему не надо ждать обновления страницы, чтобы получить ошибку)
* На сервере для безопасности, чтобы злоумышленники не смогли изменить информацию, которую вы проверили на клиенте. В конце концов, никто не запретит желающему сделать POST значений формы по нужному адресу не с HTML-страницы, а с помощью библиотеки *requests*

Может возникнуть ситуация, когда данные формы были проверены на клиенте, а затем они были изменены уже после отправки формы (например, добавлен вредоносный код в текст поля). Если на сервере не проверить данные еще раз, то ваше веб-приложение может утратить работоспособность и потерять все данные.

С загрузкой файлов в flask-wtf тоже есть свои небольшие особенности. Для загрузки файла достаточно создать поле типа *FileField*, а в обработке отправленной информации для получения содержимого файла добавить:

f = form.<название поля с файлом>.data

В библиотеке flask-wtf есть поля для всех самых распространенных типов полей ввода, которые мы рассматривали на прошлом уроке, различные валидаторы.

Уверены, что для вас не является секретом факт, что большинство веб-приложений используют в качестве источника информации базы данных, с которыми мы познакомились, когда создавали приложение c графическим пользовательским интерфейсом с использованием компонентов PyQt. На следующих нескольких уроках мы продолжим с ними работать. И рассмотрим очень мощный инструмент, который упростит разработку именно в части общения с базами данных.

**Знакомство с flask-sqlalchemy**

1. [Базы данных и ORM](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#1)
2. [Установка](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#2)
3. [Постановка задачи для первого примера](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#3)
4. [Начальный каркас приложения](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#4)
5. [Таблица с пользователями](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#5)
6. [Таблица с новостями](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#6)
7. [Взаимодействие с базой данных](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#7)
8. [Небольшая шпаргалка по filter](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#8)
9. [Отображение публичных записей](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#9)
10. [Регистрация пользователей](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2051/materials/4732#10)

**Аннотация**

На этом и следующем занятиях мы создадим полноценное веб-приложение с регистрацией и авторизацией пользователей, работающее с базой данных через ORM.

**Базы данных и ORM**

Мы уже достаточно тесно познакомились с реляционными базами данных: решали задачи с использованием БД, многие из вас уже использовали БД в своих проектах для хранения информации. Однако пока мы работали с данными исключительно с помощью запросов на языке SQL, которые составляли самостоятельно. Для небольших проектов это вполне оправдано и достаточно часто используется в реальной жизни. Flask «из коробки» не имеет собственных инструментов для работы с базами данных (в отличие, например, от Django) и предоставляет нам полную свободу выбора инструментов для работы.

Согласитесь: очень хочется перейти на более высокий уровень работы с данными. Представьте, что в вашем распоряжении есть объект, который связан с базой данных. Этот объект берет на себя всю работу по организации общения с данными. Вам лишь остается давать ему команды: получить данные, отфильтровать их по заданному условию, записать данные и т. д., а преобразование команд в SQL-запросы — это уже забота объекта. Работать с данными как с объектами чрезвычайно удобно — не зря же мы так много времени уделяем изучению объектного подхода.

К сожалению, написать универсальный модуль, который преобразовывает результат SQL-запроса в объекты и наоборот, достаточно нетривиальная задача. Хорошо, что она уже решена за нас.

В больших приложениях (не только для web) достаточно часто используется технология [ORM](https://ru.wikipedia.org/wiki/ORM) (Object-Relational Mapping — объектно-реляционное отображение) — прослойка, позволяющая работать с базой данных через объекты языка. Кроме того, большинство ORM позволяют генерировать скрипты миграции базы данных для поддержания версионности (отдаленно можно сравнить c git, но для баз данных), а также предоставляют разработчику еще немало другой полезной функциональности. Мы будем использовать библиотеку sqlalchemy. Ее можно использовать не только при создании веб-приложений, но и при разработке любых программ, которые взаимодействуют с базами данных.

**Установка**

Прежде чем мы начнем что-то делать, давайте установим sqlalchemy:

pip install sqlalchemy

**Постановка задачи для первого примера**

Давайте создадим веб-приложение, которое будет использовать базу данных и ORM для работы с ней. Для начала определимся с функциональностью.

У нас будет простое веб-приложение, в котором пользователи могут авторизовываться, просматривать, добавлять и удалять новости — такой мини-вариант личных блогов. Получается, нам надо хранить информацию о двух сущностях:

1. Пользователи
2. Новости

Причем каждая новость должна быть связана с автором новости, который ее написал.

**Начальный каркас приложения**

Начнем создавать наше приложение с создания основного файла, который назовем main.py. Для начала разместим в нем уже привычный код:

from flask import Flask

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

app.config['SECRET\_KEY'] = 'yandexlyceum\_secret\_key'

def main():

app.run()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Создадим папки:

* db — для хранения одного единственного файла базы данных
* data — для хранения классов, необходимых для взаимодействия с базой данных

Внутри папки data создадим два файла, один назовем \_\_all\_models.py (в нем будем хранить модели для работы с базой данных), а — другой db\_session.py — будет отвечать за подключение к базе данных и создание сессии для работы с ней.

Следующий шаг самый сложный. Не переживайте, этот код будет стандартным для всех ваших приложений и может переезжать от одного проекта к другому. Напишем в db\_session.py вот такой код:

import sqlalchemy as sa

import sqlalchemy.orm as orm

from sqlalchemy.orm import Session

import sqlalchemy.ext.declarative as dec

SqlAlchemyBase = dec.declarative\_base()

\_\_factory = None

Сначала импортируем необходимое — саму библиотеку sqlalchemy (в принципе, этого достаточно, остальные импорты нужны только для избавления от длинных путей), затем часть библиотеки, которая отвечает за функциональность ORM, потом объект *Session*, отвечающий за соединение с базой данных, и модуль declarative — он поможет нам объявить нашу базу данных.

Создадим две переменные: *SqlAlchemyBase* — некоторую абстрактную декларативную базу, в которую позднее будем наследовать все наши модели, и *\_\_factory*, которую будем использовать для получения сессий подключения к нашей базе данных.

Кроме того, в файле db\_session.py нам понадобится сделать еще две функции *global\_init* и *create\_session*.

def global\_init(db\_file):

global \_\_factory

if \_\_factory:

return

if not db\_file or not db\_file.strip():

raise Exception("Необходимо указать файл базы данных.")

conn\_str = f'sqlite:///{db\_file.strip()}?check\_same\_thread=False'

print(f"Подключение к базе данных по адресу {conn\_str}")

engine = sa.create\_engine(conn\_str, echo=False)

\_\_factory = orm.sessionmaker(bind=engine)

from . import \_\_all\_models

SqlAlchemyBase.metadata.create\_all(engine)

*global\_init* принимает на вход адрес базы данных, затем проверяет, не создали ли мы уже фабрику подключений (то есть не вызываем ли мы функцию не первый раз). Если уже создали, то завершаем работу, так как начальную инициализацию надо проводить только единожды.

Проверяем, что нам указали непустой адрес базы данных, а затем создаем строку подключения *conn\_str* (она состоит из типа базы данных, адреса до базы данных и параметров подключения), которую передаем Sqlalchemy для того, чтобы она выбрала правильный движок работы с базой данных (переменная *engine*). В нашем случае это будет движок для работы с SQLite базами данных.

Если в функцию create\_engine() передать параметр *echo* со значением True, в консоль будут выводиться все SQL-запросы, которые сделает SQLAlchemy, что очень удобно для отладки.

Наконец, создаем фабрику подключений к нашей базе данных, которая будет работать с нужным нам движком.

Импортируем все из файла \_\_all\_models.py — именно тут SQLalchemy узнает о всех наших моделях.

Наконец, заставляем нашу базу данных создать все объекты, которые она пока не создала. Обратите внимание: все таблицы, которые были уже созданы в базе данных, останутся без изменений.

def create\_session() -> Session:

global \_\_factory

return \_\_factory()

Функция *create\_session* нужна для получения сессии подключения к нашей базе данных. Часть -> Session нужна лишь для того, чтобы явно указать PyCharm, что наша функция возвращает объект типа sqlalchemy.orm.Session и среда могла показывать нам подсказки далее.

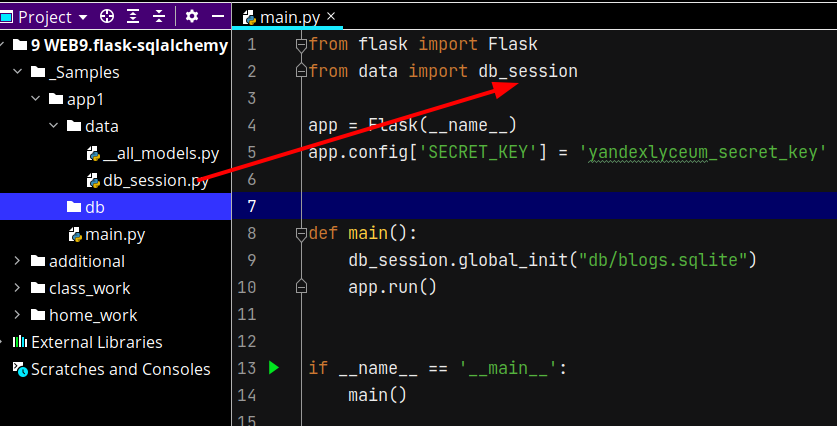
В качестве завершающего штриха давайте добавим в main.py импорт содержимого файла db\_session:

from data import db\_session

И перед запуском приложения app.run() добавим вызов глобальной инициализации всего, что связано с базой данных:

db\_session.global\_init("db/blogs.db")

В результате должна получиться примерно такая структура папок и файлов:



Запустим наше приложение. Мы увидим, что оно работает и после запуска создало пустую базу данных по адресу db/blogs.db.

Теперь давайте создадим таблицы.

**Таблица с пользователями**

Так как мы используем ORM, никакого SQL мы писать не будем и позволим sqlalchemy сделать всю работу за нас, а сами лишь опишем те классы, которые хотим получить для работы с базой в Python.

Давайте создадим в папке data файл users.py и опишем в нем класс *User* — модели для работы с таблицей, которая будет содержать информацию о пользователях нашего веб-приложения.

Подумаем, какая информация нам нужна о наших пользователях:

1. Уникальный идентификатор пользователя — некоторый его номер в нашей базе данных, желательно, чтобы он генерировался автоматически без нашего участия.
2. Имя пользователя — некоторое строковое значение.
3. Описание пользователя — некоторая текстовая информация, которой пользователь хочет поделиться о себе.
4. Адрес электронной почты — уникальная строка (чтобы мы могли точно знать, для какого пользователя восстанавливать пароль). Так как мы достаточно часто будем искать пользователя по адресу электронной почты (как минимум, при логине), было бы здорово, если бы база могла как-то ускорить такой поиск.
5. Зашифрованный пароль пользователя — строка. **Ни в коем случае нельзя хранить пароль пользователя в открытом виде!**
6. Дата создания пользователя — потому что нам хочется знать, когда пользователь зарегистрировался в нашем веб-приложении.

import datetime

import sqlalchemy

from .db\_session import SqlAlchemyBase

class User(SqlAlchemyBase):

\_\_tablename\_\_ = 'users'

id = sqlalchemy.Column(sqlalchemy.Integer,

primary\_key=True, autoincrement=True)

name = sqlalchemy.Column(sqlalchemy.String, nullable=True)

about = sqlalchemy.Column(sqlalchemy.String, nullable=True)

email = sqlalchemy.Column(sqlalchemy.String,

index=True, unique=True, nullable=True)

hashed\_password = sqlalchemy.Column(sqlalchemy.String, nullable=True)

created\_date = sqlalchemy.Column(sqlalchemy.DateTime,

default=datetime.datetime.now)

Чтобы обозначить, что *User* — не обычный класc, а именно класс модели, его необходимо унаследовать от объекта класса *SqlAlchemyBase*, который мы создали ранее.

В служебном атрибуте *\_\_tablename\_\_* указывается таблица, которая будет создана для хранения данных этой модели. Название таблицы можно не указывать, тогда sqlalchemy сделает таблицу сама, исходя из названия класса.

Для каждого из атрибутов типа *sqlalchemy.Column* будет создан одноименный столбец в базе данных согласно его описанию:

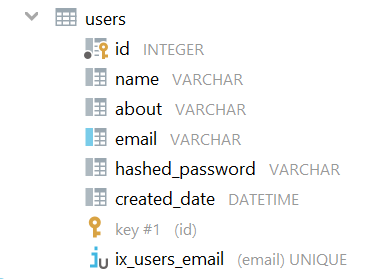
* sqlalchemy.Integer (sqlalchemy.String, sqlalchemy.DateTime и т. д.) — указания типа данных
* primary\_key=True — указание на то, что столбец является первичным ключом. Обычно первичный ключ — некоторый числовой идентификатор, который однозначно идентифицирует каждую запись в таблице
* autoincrement=True — признак автоинкрементного поля. Используется для увеличения значения первичного ключа на единицу при вставке каждой новой записи
* nullable=True/False — может ли поле не содержать никакой информации и быть пустым
* unique=True/False — содержит ли поле только уникальные значения или они могут повторяться
* default=datetime.datetime.now — значение по умолчанию. В данном случае мы говорим, что при вставке нового пользователя будет вставлена дата и время на момент его создания. Обратите внимание: мы не вызываем функцию, а передаем ее. Если бы мы вызвали функцию, то текущее время было бы вычислено только один раз при запуске сервера и было бы одинаковое для всех пользователей, которые были бы созданы после этого
* index=True — создать индекс по этому полю. [Индекс](https://ru.wikipedia.org/wiki/Индекс_(базы_данных)), если говорить упрощенно, позволяет значительно повысить скорость поиска по одному или нескольким полям базы данных. Цена этого — уменьшение скорости вставки данных, поэтому не стоит делать индексы на абсолютно все поля, а только на те (и ту их комбинацию), по которым будет часто осуществляться поиск

Запустим наше приложение. Но почему ничего не произошло? Почему не создалась таблица?

На самом деле все просто. Наша база пока не знает про нашу модель. Давайте добавим информацию о ней в файл \_\_all\_models.py.

from . import users

Запустим еще раз.



Ура, все заработало!

**Таблица с новостями**

По аналогии рядом с файлом users.py создадим файл news.py, а в нем — класс *News* для хранения следующей информации о новости:

* Уникальный идентификатор
* Заголовок новости
* Текст новости
* Дата создания новости
* Приватность новости (показывать ли ее всем или только автору)
* Идентификатор автора новости

import datetime

import sqlalchemy

from sqlalchemy import orm

from .db\_session import SqlAlchemyBase

class News(SqlAlchemyBase):

\_\_tablename\_\_ = 'news'

id = sqlalchemy.Column(sqlalchemy.Integer,

primary\_key=True, autoincrement=True)

title = sqlalchemy.Column(sqlalchemy.String, nullable=True)

content = sqlalchemy.Column(sqlalchemy.String, nullable=True)

created\_date = sqlalchemy.Column(sqlalchemy.DateTime,

default=datetime.datetime.now)

is\_private = sqlalchemy.Column(sqlalchemy.Boolean, default=True)

user\_id = sqlalchemy.Column(sqlalchemy.Integer,

sqlalchemy.ForeignKey("users.id"))

user = orm.relation('User')

Здесь стоит обратить внимание на поля *user\_id* и *user*.

*user\_id* — колонка, в которой указывается, что она ссылается на поле *id* таблицы *users*. А *user* — атрибут, который позволит нам получить для новости полноценный объект класса *User*.

Немного модифицируем наш класс с пользователями, добавим туда строчку:

news = orm.relation("News", back\_populates='user')

И в блок импортов строчку:

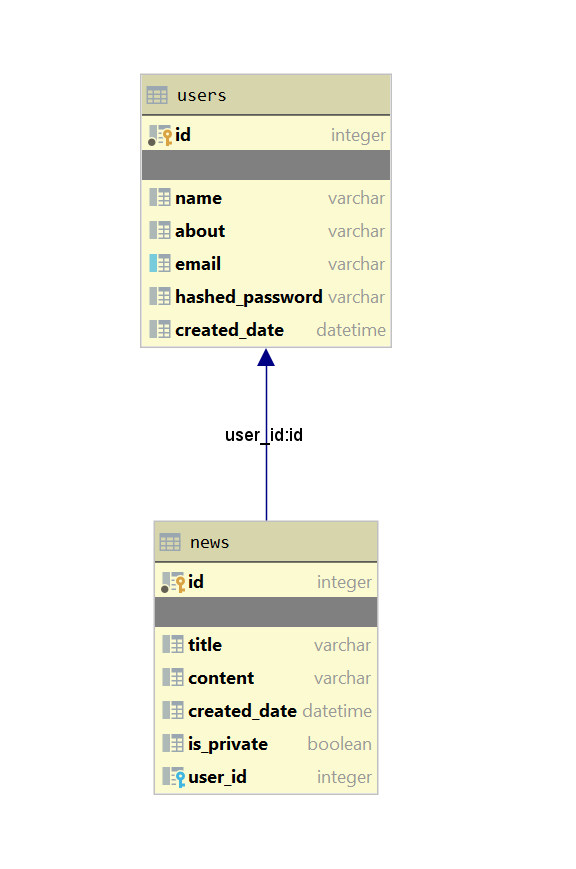
from sqlalchemy import orm

Это позволит нам легко получать для пользователя все его новости. Обратите внимание: значение параметра *back\_populates* должно указывать не на таблицу, а на атрибут класса *orm.relation*.

Чтобы подключить модель *News* к нашей базе данных, в файл \_\_all\_models.py надо добавить строку:

from . import news

Удалим базу данных и запустим наше приложение заново, чтобы все таблицы пересоздались. Вот какие две таблицы у нас получились в базе данных:



**Взаимодействие с базой данных**

Давайте ненадолго сместим фокус от создания веб-приложения на взаимодействие с базой данных с помощью sqlalchemy.

Можно пока временно закомментировать строку app.run() в файле main.py.

Давайте напишем код, который создает пользователя в нашей базе данных.

Добавьте в секцию импортов строку

from data.users import User

чтобы можно было использовать класс *User*.

За добавление объектов отвечает метод *add* у объекта *session*.

user = User()

user.name = "Пользователь 1"

user.about = "биография пользователя 1"

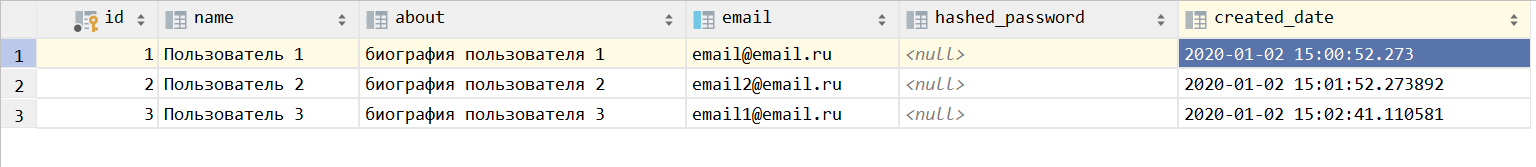
user.email = "email@email.ru"

db\_sess = db\_session.create\_session()

db\_sess.add(user)

db\_sess.commit()

Создадим еще нескольких пользователей, чтобы данные в таблице выглядели примерно вот так:



За получение данных отвечает метод *query* объекта *session*. В качестве параметра передаются классы объектов, которые мы хотим достать. Например, давайте достанем первого пользователя в выборке:

user = db\_sess.query(User).first()

print(user.name)

Пользователь 1

А теперь пройдемся вообще по всем пользователям в таблице. Для более красивого представления переопределите метод *\_\_repr\_\_* у класса *User*:

for user in db\_sess.query(User).all():

print(user)

<User> 1 Пользователь 1 email@email.ru

<User> 2 Пользователь 2 email2@email.ru

<User> 3 Пользователь 3 email1@email.ru

Метод filter() позволяет отфильтровать результаты с помощью оператора WHERE, примененного к запросу. Он принимает колонку, оператор или значение. Например, давайте отфильтруем пользователей и выберем только тех, у которых id > 1, а почта не содержит 1.

for user in db\_sess.query(User).filter(User.id > 1, User.email.notilike("%1%")):

print(user)

<User> 2 Пользователь 2 email2@email.ru

Условия, перечисленные в скобках *filter* через запятую, соединяются в запросе через AND. Давайте изменим запрос, чтобы условия фильтра соединялись через OR:

for user in db\_sess.query(User).filter((User.id > 1) | (User.email.notilike("%1%"))):

print(user)

Обратите внимание: в этом случае скобки вокруг частей фильтра **обязательны**.

Разумеется, sqlalchemy позволяет нам изменять и удалять записи в таблицах. Изменение записи сделать очень просто, надо выбрать нужную запись, поменять нужные атрибуты, а потом вызвать у сессии метод *commit*. Переименуем пользователя с id 1:

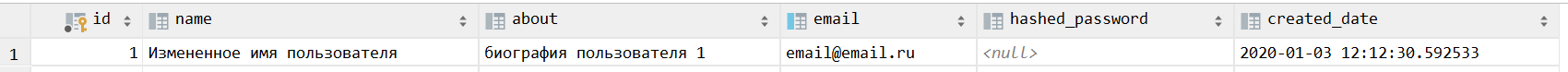
user = db\_sess.query(User).filter(User.id == 1).first()

print(user)

user.name = "Измененное имя пользователя"

user.created\_date = datetime.datetime.now()

db\_sess.commit()



За удаление записей отвечает метод delete().

Мы можем удалить как несколько записей по фильтру:

db\_sess.query(User).filter(User.id >= 3).delete()

db\_sess.commit()

Так и какую-то уже заранее выбранную запись:

user = db\_sess.query(User).filter(User.id == 2).first()

db\_sess.delete(user)

db\_sess.commit()

После всех изменений в базе данных не забывайте сделать *commit*.

После всех этих операций у нас должен остаться в базе всего один пользователь. Давайте добавим ему записи. Это можно сделать несколькими способами. Мы можем создать объект класса *News* и заполнить его поля, в том числе указать явно *id* записи автора:

news = News(title="Первая новость", content="Привет блог!",

user\_id=1, is\_private=False)

db\_sess.add(news)

db\_sess.commit()

Можем в качестве *user* указать объект класса *User*, выбранный (или созданный) заранее:

user = db\_sess.query(User).filter(User.id == 1).first()

news = News(title="Вторая новость", content="Уже вторая запись!",

user=user, is\_private=False)

db\_sess.add(news)

db\_sess.commit()

И самый удобный на наш взгляд вариант — использовать то, что через объект класса *User* мы можем взаимодействовать с его записями в таблице *News* почти как со списком:

user = db\_sess.query(User).filter(User.id == 1).first()

news = News(title="Личная запись", content="Эта запись личная",

is\_private=True)

user.news.append(news)

db\_sess.commit()

Так же легко мы можем обойти все новости конкретного пользователя:

for news in user.news:

print(news)

sqlalchemy — невероятно мощный и гибкий инструмент. Обязательно загляните в [документацию](https://docs.sqlalchemy.org/en/13/), потому что мы рассмотрели только самый минимум, необходимый для создания небольших веб-приложений и учебных проектов.

**Небольшая шпаргалка по filter**

| **Операция** | **Синтаксис ORM** |
| --- | --- |
| EQUALS | query(User).filter(User.name == 'Иван') |
| NOT EQUAL | query(User).filter(User.name != 'Иван') |
| LIKE | query(User).filter(User.name.like('%Иван%')) |
| NOT LIKE | query(User).filter(User.name.notlike('%Иван%')) |
| IN | query(User).filter(User.name.in\_(['Иван', 'Петр', 'Максим'])) |
| NOT IN | query(User).filter(User.name.notin\_(['Иван', 'Петр', 'Максим'])) или  query(User).filter(~User.name.in\_(['Иван', 'Петр', 'Максим'])) |
| NULL | query(User).filter(User.name == None) |
| AND | query(User).filter(User.name == 'Иван', User.id > 3) или query(User).filter(User.name == 'Иван').filter(User.id > 3) |
| OR | query(User).filter((User.name == 'Иван') | (User.id > 3)) или  query(User).filter(or\_(User.name == 'Иван', User.id > 3)) |

**Отображение публичных записей**

Воспользуемся наработками прошлых уроков и создадим новый шаблон index.html на основе базового шаблона base.html, который мы делали на прошлом уроке. Скопируйте папку templates со всем ее содержимым из материалов, созданных на прошлом уроке, и отредактируйте файл index.html, чтобы он выглядел так:

{% extends "base.html" %}

{% block content %}

<h1>Записи в блоге</h1>

{% for item in news%}

<div class="col-md6 border rounded">

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

<div>

{{item.content}}

</div>

<div>

Автор - {{item.user.name}}, Дата написания - {{item.created\_date}}

</div>

</div>

{% endfor %}

{% endblock %}

Обратите внимание: из каждой записи мы можем легко добраться до имени автора этой записи. А теперь добавим обработчик, чтобы при попадании на главную страницу нашего приложения все пользователи видели все публичные записи от всех авторов.

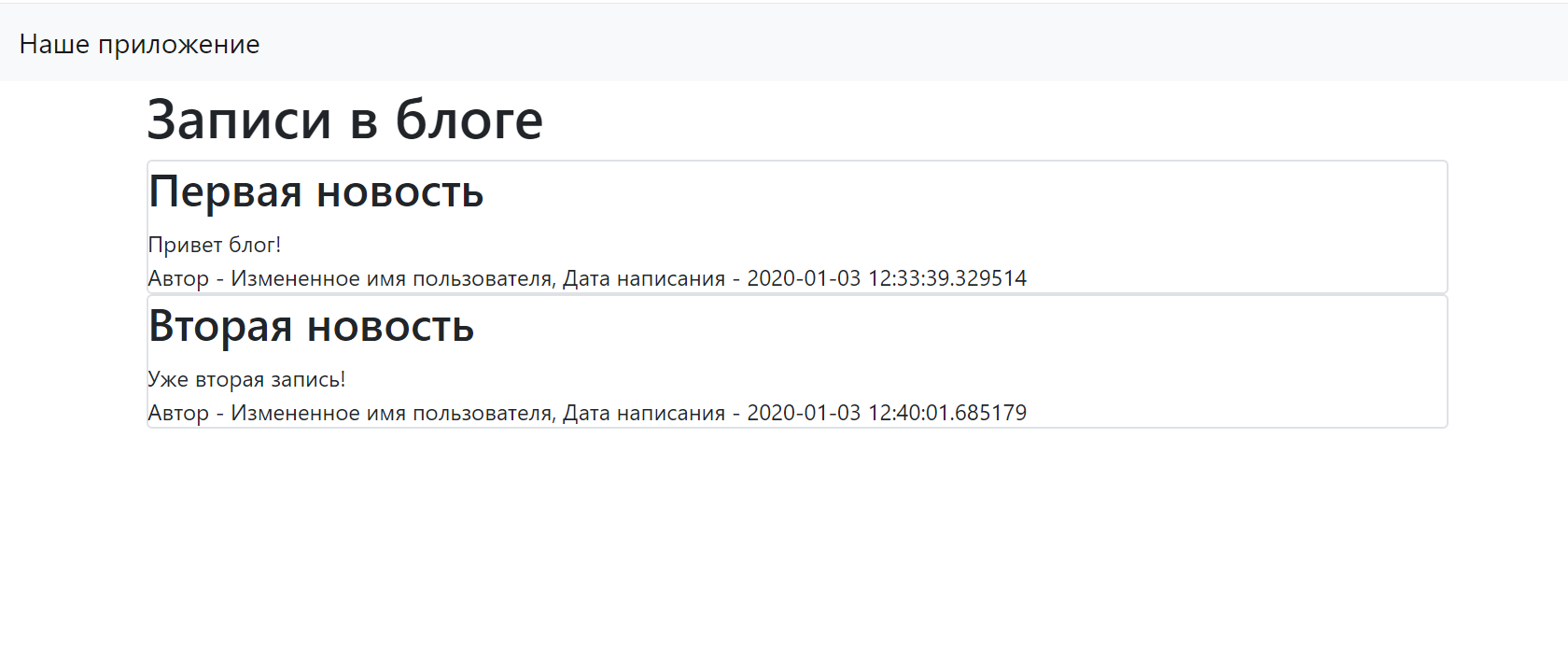
@app.route("/")

def index():

db\_sess = db\_session.create\_session()

news = db\_sess.query(News).filter(News.is\_private != True)

return render\_template("index.html", news=news)



**Регистрация пользователей**

Редко какое веб-приложение обходится без функциональности, доступной лишь зарегистрированным пользователям. Давайте добавим возможность пользователям регистрироваться в нашем приложении. Все классы для форм разместим в отдельном каталоге forms. Создадим в каталоге forms файл user.py, а в нем — класс, описывающий форму *RegisterForm*, в которую добавим все поля, которые будут в форме регистрации:

from flask\_wtf import FlaskForm

from wtforms import PasswordField, StringField, TextAreaField, SubmitField

from wtforms.fields.html5 import EmailField

from wtforms.validators import DataRequired

class RegisterForm(FlaskForm):

email = EmailField('Почта', validators=[DataRequired()])

password = PasswordField('Пароль', validators=[DataRequired()])

password\_again = PasswordField('Повторите пароль', validators=[DataRequired()])

name = StringField('Имя пользователя', validators=[DataRequired()])

about = TextAreaField("Немного о себе")

submit = SubmitField('Войти')

Спросим пароль у пользователя несколько раз, чтобы убедиться, что он нигде не опечатался.

Теперь в папке templates создадим шаблон register.html для отображения нашей формы, он может выглядеть так:

{% extends "base.html" %}

{% block content %}

<h1>Регистрация</h1>

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

{{ form.hidden\_tag() }}

<p>

{{ form.email.label }}<br>

{{ form.email(class="form-control", type="email") }}<br>

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

<p class="alert alert-danger" role="alert">

{{ error }}

</p>

{% endfor %}

</p>

<p>

{{ form.password.label }}<br>

{{ form.password(class="form-control", type="password") }}<br>

{% for error in form.password.errors %}

<p class="alert alert-danger" role="alert">

{{ error }}

</p>

{% endfor %}

</p>

<p>

{{ form.password\_again.label }}<br>

{{ form.password\_again(class="form-control", type="password") }}<br>

{% for error in form.password\_again.errors %}

<p class="alert alert-danger" role="alert">

{{ error }}

</p>

{% endfor %}

</p>

<p>

{{ form.name.label }}<br>

{{ form.name(class="form-control") }}<br>

{% for error in form.name.errors %}

<p class="alert alert-danger" role="alert">

{{ error }}

</p>

{% endfor %}

</p>

<p>

{{ form.about.label }}<br>

{{ form.about(class="form-control") }}<br>

{% for error in form.about.errors %}

<p class="alert alert-danger" role="alert">

{{ error }}

</p>

{% endfor %}

</p>

<p>{{ form.submit(type="submit", class="btn btn-primary") }}</p>

{{message}}

</form>

{% endblock %}

Прежде чем создавать обработчик URL, который будет отвечать за форму регистрации, давайте немного доработаем наш класс *User*. Как мы уже говорили, хранить пароль в открытом виде нельзя, поэтому во Flask есть инструменты, которые позволяют получить хешированное значение по строке и проверить, соответствует ли пароль хешу, который хранится в нашей базе данных. Функции *generate\_password\_hash* и *check\_password\_hash* хранятся в модуле werkzeug.security. Для большего удобства добавим в класс *User* две функции:

def set\_password(self, password):

self.hashed\_password = generate\_password\_hash(password)

def check\_password(self, password):

return check\_password\_hash(self.hashed\_password, password)

Первая устанавливает значение хэша пароля для переданной строки. А вторая проверяет, правильный ли пароль ввел пользователь. Одна понадобится нам для регистрации пользователя, а другая — позднее, когда мы будем делать авторизацию пользователей в нашем приложении.

А в раздел импортов добавим:

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

Добавим обработчик на адрес /register.

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

def reqister():

form = RegisterForm()

if form.validate\_on\_submit():

if form.password.data != form.password\_again.data:

return render\_template('register.html', title='Регистрация',

form=form,

message="Пароли не совпадают")

db\_sess = db\_session.create\_session()

if db\_sess.query(User).filter(User.email == form.email.data).first():

return render\_template('register.html', title='Регистрация',

form=form,

message="Такой пользователь уже есть")

user = User(

name=form.name.data,

email=form.email.data,

about=form.about.data

)

user.set\_password(form.password.data)

db\_sess.add(user)

db\_sess.commit()

return redirect('/login')

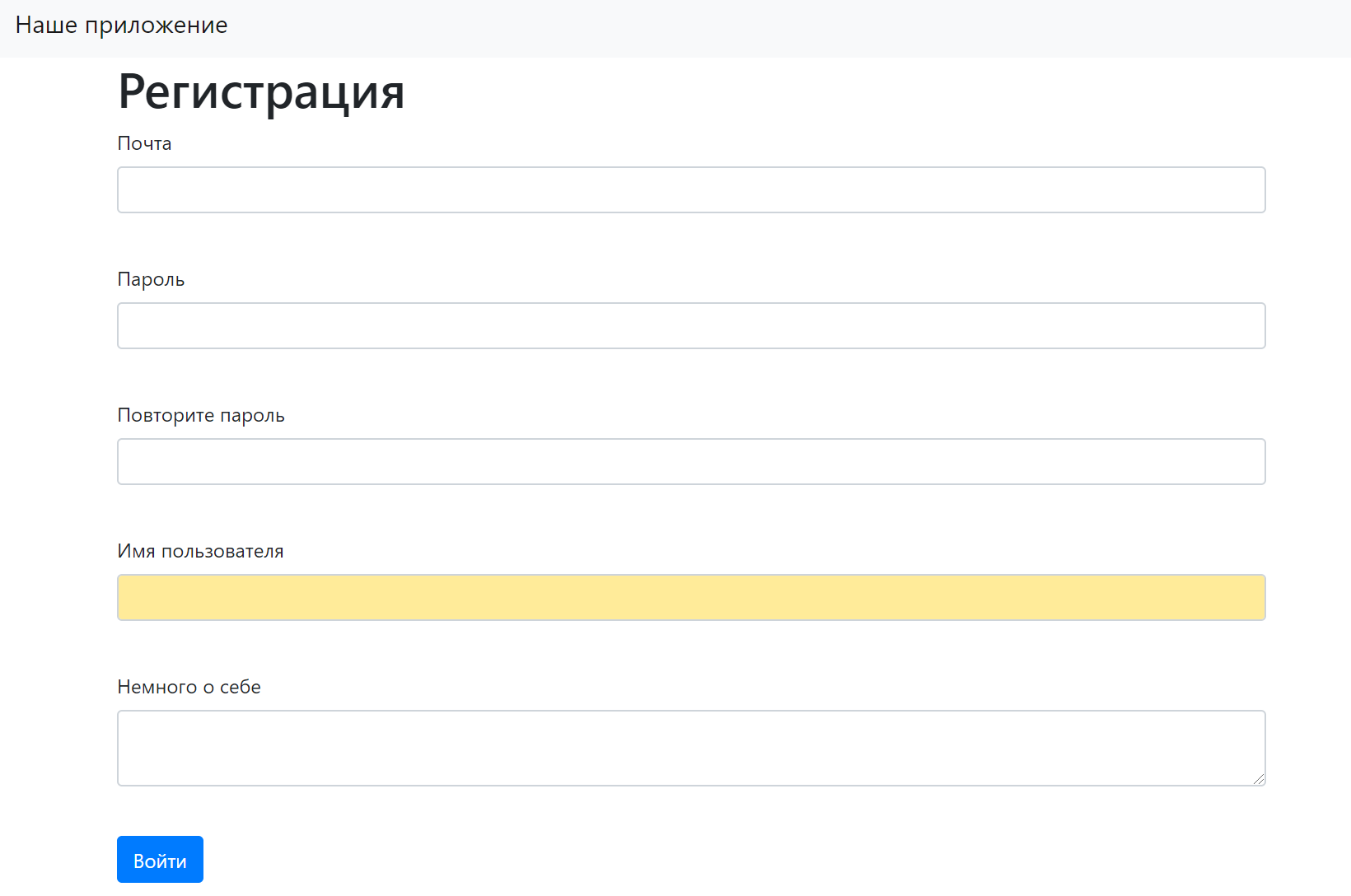
return render\_template('register.html', title='Регистрация', form=form)

Не забудьте импортировать форму:

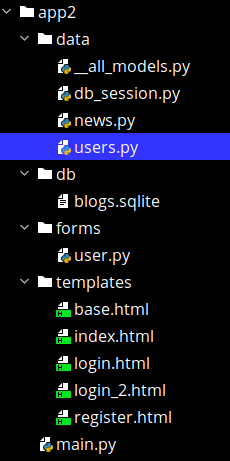
from forms.user import RegisterForm

После отправки формы на сервер проверяем, что пароли совпадают, а также что пользователя с таким адресом электронной почты пока нет в нашей базе данных.

Если все хорошо, добавляем пользователя в базу данных и отправляем его на страницу авторизации, которую сделаем на следующем уроке.



После добавления всех необходимых файлов в проект он должен выглядеть так:



**Заключение.**

На этом уроке мы научились работать с ORM sqlalchemy и написали каркас нашего приложения, страницу отображения новостей, а также страницу регистрации пользователей. На следующем уроке мы сделаем авторизацию и ее проверку и поговорим еще о нескольких важных вещах.

**Flask и sqlalchemy**

1. [Куки и сессии](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2052/materials/4759#1)
2. [Авторизация пользователя](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2052/materials/4759#2)
3. [Добавление, изменение и удаление данных](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2052/materials/4759#3)
4. [Отношение многие ко многим](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2052/materials/4759#4)
5. [Заключение](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2052/materials/4759#5)

**Аннотация**

Сегодня мы доделаем процесс авторизации пользователей в веб-приложении, а также рассмотрим еще несколько важных моментов, связанных с работой с библиотеками flask и sqlalchemy.

**Куки и сессии**

Продолжим работать над веб-приложением, которое мы начали делать на прошлом уроке. Прежде чем писать форму логина, давайте поговорим о том, как работает процесс аутентификации в вебе. Как вы наверняка заметили, пока наше веб-приложение не запоминает никакой информации о клиенте между его запросами, то есть мы никак не проверяем, пришел ли следующий запрос от того же клиента или от другого. Чтобы удостовериться в том, что пользователь был на нашем сайте и уже делал какие-то действия, есть несколько способов.

Первый из них — [куки](https://ru.wikipedia.org/wiki/Cookie). Куки — это небольшой фрагмент данных, который сервер устанавливает в браузере клиента. Это работает следующим образом:

1. Клиент отправляет запрос на получение страницы от сервера, то есть вызывает какой-то из наших flask-обработчиков url.
2. Сервер отвечает на запрос и вместе со страницей ответа отправляет одно или несколько куки.
3. При всех следующих запросах клиент отправляет информацию из полученных куки серверу (пока не истечет их «срок годности»).

Как и со всеми остальными параметрами, которые мы получаем от пользователя, работа с куки очень похожа на работу со словарями. Давайте сделаем вот такой небольшой пример:

@app.route("/cookie\_test")

def cookie\_test():

visits\_count = int(request.cookies.get("visits\_count", 0))

if visits\_count:

res = make\_response(

f"Вы пришли на эту страницу {visits\_count + 1} раз")

res.set\_cookie("visits\_count", str(visits\_count + 1),

max\_age=60 \* 60 \* 24 \* 365 \* 2)

else:

res = make\_response(

"Вы пришли на эту страницу в первый раз за последние 2 года")

res.set\_cookie("visits\_count", '1',

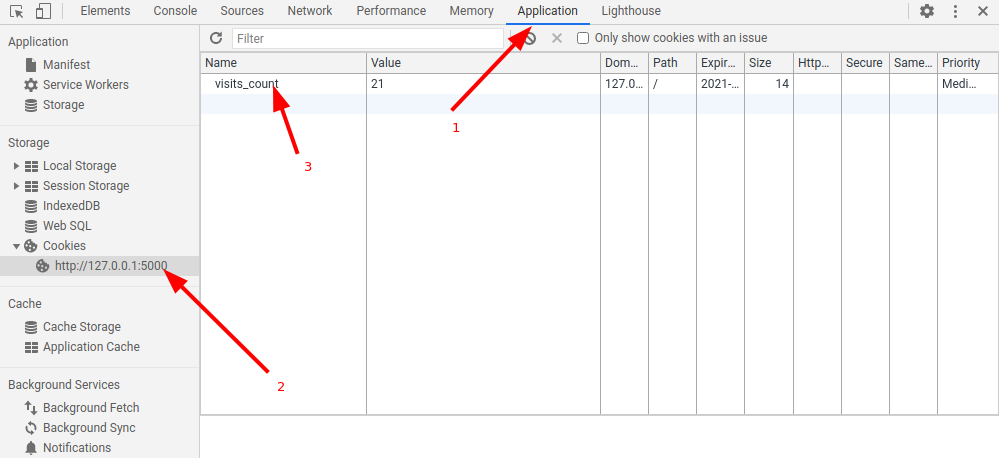
max\_age=60 \* 60 \* 24 \* 365 \* 2)

return res

Сначала мы пытаемся получить куки по ключу *visits\_count*, и если она не установлена у пользователя, получаем ноль. Ноль означает, что пользователь пришел на нашу страницу первый раз, о чем мы ему и сообщаем с помощью строки, которую передаем в функцию *flask.make\_response*. После чего у получившегося объекта вызываем метод *set\_cookie*, куда передаем имя куки, значение, а также максимальное время жизни, после которого браузер удалит куки.

Если куки уже установлено, мы увеличиваем счетчик на 1 и переустанавливаем куки, обновляя срок жизни. Обратите внимание: у куки ключи могут быть типа *str*, а значения — типа *str* или *bytes*.

Запустите программу, перейдите по адресу http://127.0.0.1:5000/cookie\_test и посмотрите, как будет меняться отображаемая информация в браузере, когда вы обновляете страницу (это можно делать при помощи клавиши F5). Также значение куки можно посмотреть непосредственно в браузере, перейдя в режим разработчика при помощи клавиши F12.



Если мы хотим сделать не простой текстовый ответ, необходимо передать в функцию *make\_response* результат функции *render\_template*.

Например, в обработчике для страницы новостей это могло выглядеть примерно вот так:

res = make\_response(render\_template("index.html", news=news))

res.set\_cookie("visits\_count", '1', max\_age=60 \* 60 \* 24 \* 365 \* 2)

Для удаления куки достаточно установить для нее нулевое время жизни:

res.set\_cookie("visits\_count", '1', max\_age=0)

Куки классные, но у них есть недостатки:

1. Вся информация, записанная в куки, хранится в открытом виде (не шифруется) и может быть доступна любому человеку для чтения и изменения. Поэтому в куки нельзя хранить пароли, данные банковских карт и другие чувствительные для потери данные.
2. Куки можно отключить в браузере, и если пользователь их отключит, мы об об этом не узнаем.
3. Куки не безразмерные. Каждая может хранить до 4 КБ данных, кроме того, у любого браузера есть свое ограничение на количество куки, которое может установить каждый сайт (это число всегда меньше 50).
4. Куки отправляются с каждым запросом к серверу, поэтому, если установить большое число больших куки, запросы к серверу будут тяжелыми и сайт будет работать медленно.

Сессии во Flask очень похожи на куки, но имеют большое преимущество: гарантируется, что содержимое сессии не может быть изменено пользователем (если у него нет нашего секретного ключа). Для работы с сессиями есть специальный объект *flask.session*, его надо импортировать. Давайте перепишем часть со счетчиком посещений на сессии:

@app.route("/session\_test")

def session\_test():

visits\_count = session.get('visits\_count', 0)

session['visits\_count'] = visits\_count + 1

return make\_response(

f"Вы пришли на эту страницу {visits\_count + 1} раз")

Сессии также хранятся в куках, но в зашифрованном виде. По умолчанию сессии существуют до тех пор, пока пользователь не закроет браузер. Чтобы продлить жизнь сессии, нужно присвоить атрибуту *session.permanent* значение True. В таком случае срок жизни сессии будет продлен до 31 дня. Если нужно еще больше (например, год) тогда после создания нашего приложения надо установить параметр PERMANENT\_SESSION\_LIFETIME.

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

app.config['PERMANENT\_SESSION\_LIFETIME'] = datetime.timedelta(

days=365

)

Удаление данных из сессии происходит так же, как и удаление пары «ключ-значение» из словаря. Например, это можно сделать так:

session.pop('visits\_count', None)

На использовании объекта сессии можно построить простую систему авторизации для нашего приложения. Для этого можно после правильного ввода пользователем логина и пароля записывать в его сессию некоторое сложноподбираемое значение, которое использовать как ключ доступа к личным разделам пользователя. Несмотря на то, что значение сессии нельзя изменить, злоумышленник может перехватить значение сессии и начать отправлять свои запросы, представляясь другим пользователем, так как при передаче данных от клиента до сервера и обратно по протоколу HTTP данные не шифруются. Для шифрованной передачи используется протокол HTTPS.

**Авторизация пользователя**

Для добавления функциональности авторизации пользователей можно воспользоваться библиотекой flask-login. Для начала установим библиотеку:

pip install flask-login

Выполним первоначальную настройку модуля. Сначала импортируем нужный класс:

from flask\_login import LoginManager

Затем сразу после создания приложения flask инициализируем *LoginManager*:

login\_manager = LoginManager()

login\_manager.init\_app(app)

Для верной работы flask-login у нас должна быть функция для получения пользователя, украшенная декоратором *login\_manager.user\_loader*. Добавим ее:

@login\_manager.user\_loader

def load\_user(user\_id):

db\_sess = db\_session.create\_session()

return db\_sess.query(User).get(user\_id)

Кроме того, наша модель для пользователей должна содержать ряд методов для корректной работы flask-login, но мы не будем создавать их руками, а воспользуемся множественным наследованием. И помимо *SqlAlchemyBase* унаследуем *User* от *UserMixin* из модуля flask-login, то есть заголовок класса модели пользователей будет выглядеть так:

class User(SqlAlchemyBase, UserMixin):

Сделаем форму авторизации пользователя, назовем ее *LoginForm*. Она будет практически совпадать с той, что мы делали на уроке знакомства с flask-wtf:

class LoginForm(FlaskForm):

email = EmailField('Почта', validators=[DataRequired()])

password = PasswordField('Пароль', validators=[DataRequired()])

remember\_me = BooleanField('Запомнить меня')

submit = SubmitField('Войти')

Сделаем к ней шаблон login.html:

{% extends "base.html" %}

{% block content %}

<h1>Авторизация</h1>

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

{{ form.hidden\_tag() }}

<p>

{{ form.email.label }}<br>

{{ form.email(class="form-control", type="email") }}<br>

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

<div class="alert alert-danger" role="alert">

{{ error }}

</div>

{% endfor %}

</p>

<p>

{{ form.password.label }}<br>

{{ form.password(class="form-control", type="password") }}<br>

{% for error in form.password.errors %}

<div class="alert alert-danger" role="alert">

{{ error }}

</div>

{% endfor %}

</p>

<p>{{ form.remember\_me() }} {{ form.remember\_me.label }}</p>

<p>{{ form.submit(type="submit", class="btn btn-primary") }}</p>

<div>{{ message }}</div>

</form>

{% endblock %}

И, наконец, сделаем обработчик адреса /login:

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

def login():

form = LoginForm()

if form.validate\_on\_submit():

db\_sess = db\_session.create\_session()

user = db\_sess.query(User).filter(User.email == form.email.data).first()

if user and user.check\_password(form.password.data):

login\_user(user, remember=form.remember\_me.data)

return redirect("/")

return render\_template('login.html',

message="Неправильный логин или пароль",

form=form)

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

(Не забудьте импортировать класс *LoginForm* и метод *login\_user* из модуля flask-login.)

Если форма логина прошла валидацию, мы находим пользователя с введенной почтой, проверяем, введен ли для него правильный пароль, если да, вызываем функцию *login\_user* модуля flask-login и передаем туда объект нашего пользователя, а также значение галочки «Запомнить меня». После чего перенаправляем пользователя на главную страницу нашего приложения.

Давайте запустим и попробуем. После ввода правильного логина и пароля нас действительно перенаправляет на главную страницу веб-приложения. Но как понять что что-то поменялось? Для этого существует атрибут *flask\_login.current\_user*, доступный в любом обработчике URL и в шаблонах. Если пользователь залогинен, то там содержится объект класса *User* текущего пользователя, а если никто не авторизовался — анонимного пользователя.

Давайте добавим следующий код в элемент nav базового шаблона:

{% if current\_user.is\_authenticated %}

<a class="navbar-brand" href="/logout">{{ current\_user.name }}</a>

{% else %}

<p>

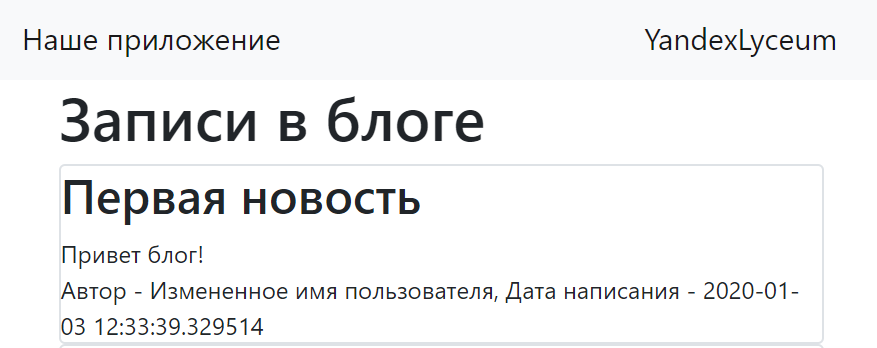
<a class="btn btn-primary " href="/register">Зарегистрироваться</a>

<a class="btn btn-success" href="/login">Войти</a>

</p>

{% endif %}

Теперь после входа и перенаправления на главную страницу мы увидим имя залогиненного пользователя:



Теперь добавим обработчик адреса /logout. Для него нам не понадобится отдельный шаблон, поскольку это не отдельная страница, а действие.

@app.route('/logout')

@login\_required

def logout():

logout\_user()

return redirect("/")

Тут все просто — мы «забываем» пользователя при помощи функции *logout\_user* и перенаправляем его на главную страницу нашего приложения. Из интересного здесь — декоратор *login\_required* (не забудьте это импортировать). Таким декоратором можно украшать обработчики страниц, на которые может попасть только авторизованный пользователь.

Давайте добавим небольшое изменение в главную страницу нашего приложения, чтобы для авторизованного пользователя отображались и его личные записи.

if current\_user.is\_authenticated:

news = db\_sess.query(News).filter(

(News.user == current\_user) | (News.is\_private != True))

else:

news = db\_sess.query(News).filter(News.is\_private != True)

**Добавление, изменение и удаление данных**

Чтобы далеко не ходить, давайте рассмотрим добавление, изменение и удаление данных на примере новостей. Начнем с добавления. Разумеется, добавлять новости у нас могут только авторизованные пользователи, поэтому давайте добавим в шаблон отображения списка новостей кнопку, доступную только им:

{% if current\_user.is\_authenticated %}

<a href="news" class="btn btn-secondary">Добавить новость</a>

{% endif %}

Создадим форму добавления новости *NewsForm*, в каталог forms добавим файл news.py:

from flask\_wtf import FlaskForm

from wtforms import StringField, TextAreaField

from wtforms import BooleanField, SubmitField

from wtforms.validators import DataRequired

class NewsForm(FlaskForm):

title = StringField('Заголовок', validators=[DataRequired()])

content = TextAreaField("Содержание")

is\_private = BooleanField("Личное")

submit = SubmitField('Применить')

Шаблон для редактирования новости news.html:

{% extends "base.html" %}

{% block content %}

<h1>Добавление новости</h1>

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

{{ form.hidden\_tag() }}

<p>

{{ form.title.label }}<br>

{{ form.title(class="form-control") }}<br>

{% for error in form.title.errors %}

<p class="alert alert-danger" role="alert">

{{ error }}

</p>

{% endfor %}

</p>

<p>

{{ form.content.label }}<br>

{{ form.content(class="form-control") }}<br>

{% for error in form.content.errors %}

<p content="alert alert-danger" role="alert">

{{ error }}

</p>

{% endfor %}

</p>

<p>{{ form.is\_private() }} {{ form.is\_private.label }}</p>

<p>{{ form.submit(type="submit", class="btn btn-primary") }}</p>

{{message}}

</form>

{% endblock %}

И обработчик:

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

@login\_required

def add\_news():

form = NewsForm()

if form.validate\_on\_submit():

db\_sess = db\_session.create\_session()

news = News()

news.title = form.title.data

news.content = form.content.data

news.is\_private = form.is\_private.data

current\_user.news.append(news)

db\_sess.merge(current\_user)

db\_sess.commit()

return redirect('/')

return render\_template('news.html', title='Добавление новости',

form=form)

Тут нет ничего такого, что мы уже не делали. Единственный интересный момент — сказать сессии, что мы изменили текущего пользователя с помощью метода *merge*.

Сделаем редактирование новости. Будем использовать уже созданную форму и шаблон, напишем только другой обработчик:

@app.route('/news/<int:id>', methods=['GET', 'POST'])

@login\_required

def edit\_news(id):

form = NewsForm()

if request.method == "GET":

db\_sess = db\_session.create\_session()

news = db\_sess.query(News).filter(News.id == id,

News.user == current\_user

).first()

if news:

form.title.data = news.title

form.content.data = news.content

form.is\_private.data = news.is\_private

else:

abort(404)

if form.validate\_on\_submit():

db\_sess = db\_session.create\_session()

news = db\_sess.query(News).filter(News.id == id,

News.user == current\_user

).first()

if news:

news.title = form.title.data

news.content = form.content.data

news.is\_private = form.is\_private.data

db\_sess.commit()

return redirect('/')

else:

abort(404)

return render\_template('news.html',

title='Редактирование новости',

form=form

)

Если мы запросили страницу записи, ищем ее в базе по *id*, причем автор новости должен совпадать с текущим пользователем. Если что-то нашли, предзаполняем форму, иначе показываем пользователю страницу 404. Такую же проверку на всякий случай делаем перед изменением новости.

Добавим кнопки «Изменить» и «Удалить» к каждой новости в списке новостей, но только для тех записей, автором которых является *current\_user*. Немного изменим шаблон index.html.

{% if current\_user.is\_authenticated and current\_user == item.user %}

<div>

<a href="/news/{{ item.id }}" class="btn btn-warning">

Изменить

</a>

<a href="/news\_delete/{{ item.id }}" class="btn btn-danger">

Удалить

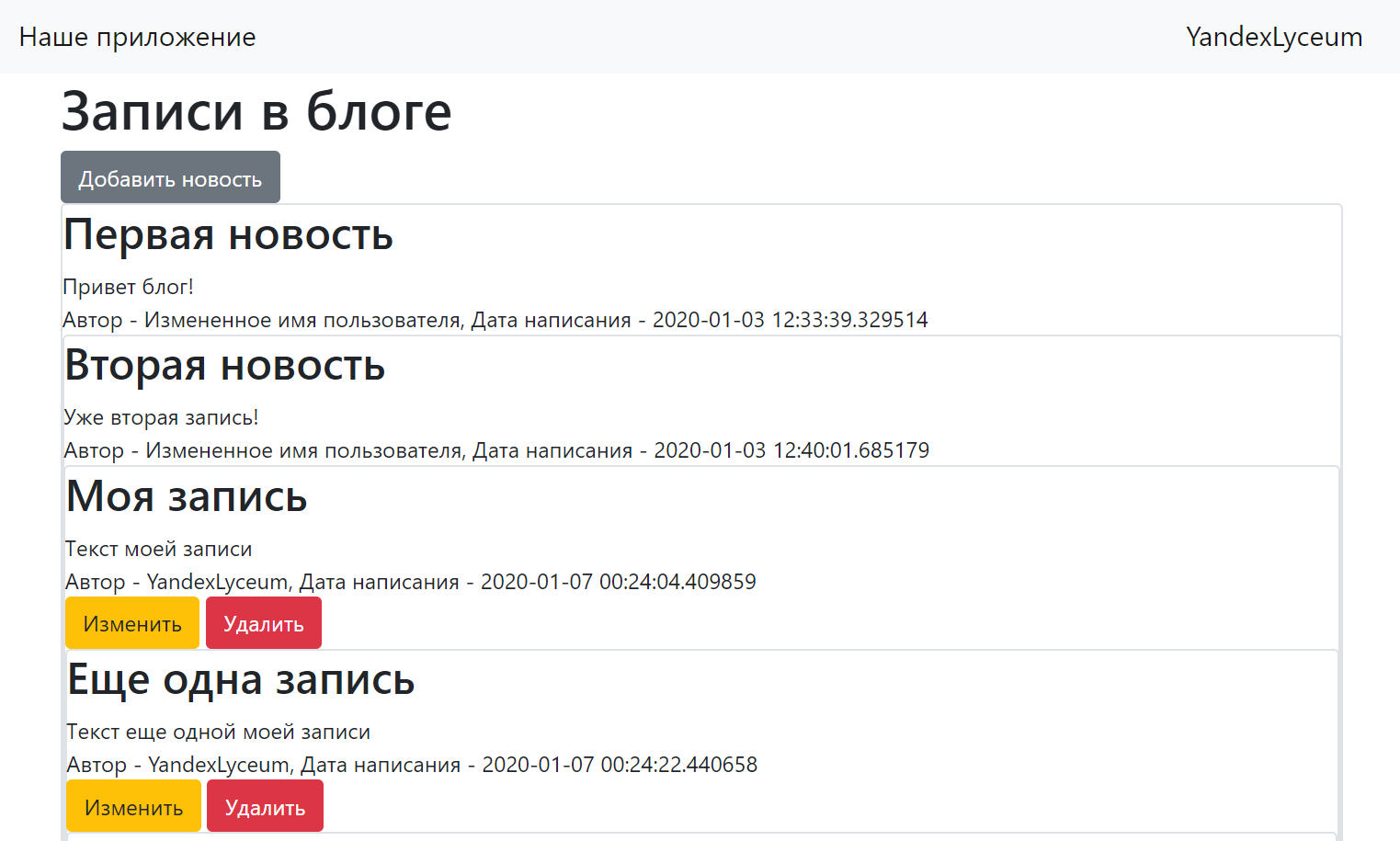
</a>

</div>

{% endif %}

В адреса каждой из ссылок допишем *id* новости.

У нас получится что-то вроде:



Добавим еще обработчик удаления записи:

@app.route('/news\_delete/<int:id>', methods=['GET', 'POST'])

@login\_required

def news\_delete(id):

db\_sess = db\_session.create\_session()

news = db\_sess.query(News).filter(News.id == id,

News.user == current\_user

).first()

if news:

db\_sess.delete(news)

db\_sess.commit()

else:

abort(404)

return redirect('/')

Итак, у нас получилось достаточно функциональное рабочее приложение. Давайте добавим еще пару штрихов.

**Отношение многие ко многим**

Когда вы создавали базы данных, наверняка обратили внимание, что когда сущности связаны **одна ко многим** (у одного пользователя есть несколько записей, в одном жанре есть несколько фильмов), то проблем с проектированием базы данных не возникает. Трудности начинаются, когда сущности связаны **многие ко многим** (в одном заказе может быть несколько наименований товаров, но эти же товары могут быть во многих заказах). Те, кто столкнулся с этой проблемой, вероятно, провели исследования и уже знают, что в SQL такая ситуация решается созданием промежуточной таблицы. Давайте посмотрим, как связь **многие ко многим** можно реализовать с помощью sqlalchemy.

Давайте представим, что наши записи могут принадлежать к одной или нескольким категориям, чтобы можно было проще их фильтровать по интересующей пользователя теме.

Добавим модель *Category*, сделаем для этого новый файл category.py в папке data:

import sqlalchemy

from .db\_session import SqlAlchemyBase

class Category(SqlAlchemyBase):

\_\_tablename\_\_ = 'category'

id = sqlalchemy.Column(sqlalchemy.Integer, primary\_key=True,

autoincrement=True)

name = sqlalchemy.Column(sqlalchemy.String, nullable=True)

Кроме того, добавим перед моделью с категориями информацию о промежуточной таблице:

association\_table = sqlalchemy.Table(

'association',

SqlAlchemyBase.metadata,

sqlalchemy.Column('news', sqlalchemy.Integer,

sqlalchemy.ForeignKey('news.id')),

sqlalchemy.Column('category', sqlalchemy.Integer,

sqlalchemy.ForeignKey('category.id'))

)

Тут мы говорим sqlalchemy, что нам нужна вспомогательная таблица *association* (не обязательно такое имя, часто их называют имя\_сущности1\_to\_имя\_сущности2, то есть в нашем случае news\_to\_category), которая будет содержать только несколько внешних ключей на каждую из таблиц.

Немного обновим модель *News*, чтобы можно было получать доступ к категориям новости как к списку:

categories = orm.relation("Category",

secondary="association",

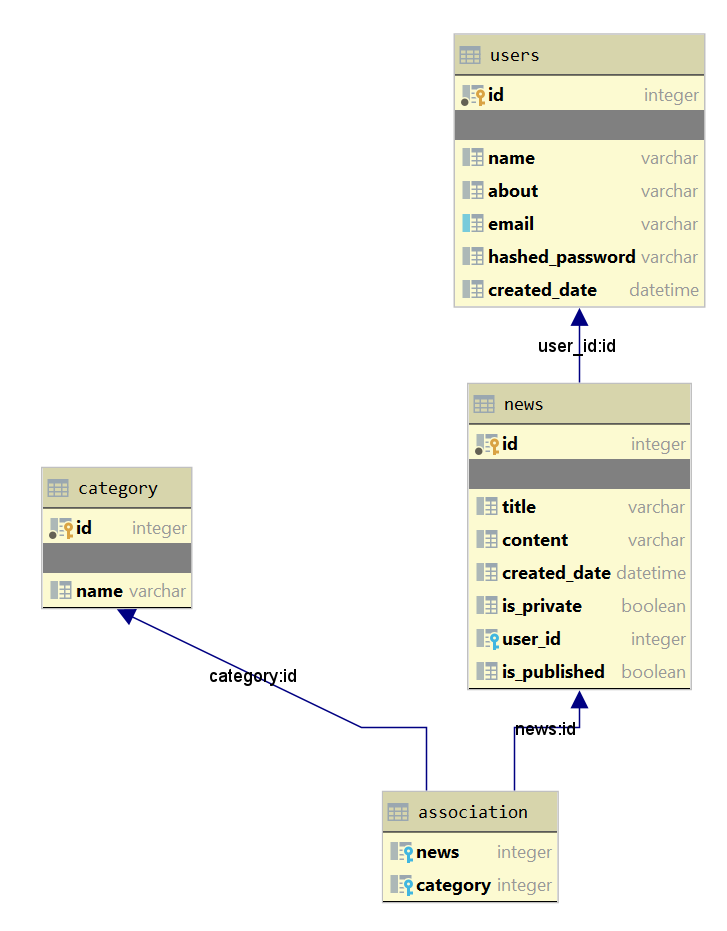
backref="news")

К сожалению, чтобы sqlalchemy применил изменения, надо удалить уже существующие таблицы в базе данных, которые затрагивают эти изменения. В данном случае необходимо удалить таблицу *news*. После перезапуска приложения таблицы будут созданы уже с правильной структурой. Такое поведение не очень удобно, скоро мы узнаем, как с этим можно справиться.

Не забудьте добавить в файл \_\_all\_models.py импорт новой модели:

from . import category

Если мы все сделали правильно, наша база данных станет выглядеть примерно вот так:



Теперь, когда мы будем вызывать метод новость.category.append и передавать туда объект типа *Category*, у нас будет создаваться запись именно в промежуточной таблице.

Чтобы удалить категорию у новости, достаточно сделать:

news.categories.remove(category)

**Заключение**

На этом мы завершаем рассмотрение функциональности SQLalchemy, хотя в следующих темах будем использовать полученные знания (и еще вернемся к ней, когда будем рассматривать механизм миграций), а теперь нас ждет создание своего собственного API.

**REST-API. Понятие. Делаем простое REST-API**

1. [REST API на flask. Микросервисы](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2053/materials/4721#1)
2. [REST](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2053/materials/4721#2)
3. [Blueprint](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2053/materials/4721#3)
4. [Сериализация в json](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2053/materials/4721#4)
5. [Создание и тестирование сервисов](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2053/materials/4721#5)

**Аннотация**

На этом занятии мы научимся создавать собственное API для наших веб-приложений.

**REST API на flask. Микросервисы**

На прошлых уроках вы изучили базовые возможности фреймворка Flask и уже можете создавать полноценные веб-приложения, которые могут взаимодействовать с пользователем и работать с базой данных. На этом уроке мы предлагаем подняться на уровень выше и рассмотреть архитектуру веб-приложения: по каким правилам будут взаимодействовать компоненты вашего приложения и как организовать код, чтобы в дальнейшем его было легче поддерживать и проще добавлять новые функции.

Можно выделить два основных подхода к организации архитектуры веб-приложений:

* Монолитная архитектура (Monolithic Style)
* Сервис-ориентированная архитектура (SOA, Service-oriented Architecture)

При монолитном подходе приложение строится как единое целое. Как правило, монолитное веб-приложение включает три части: веб-сервер, который содержит всю логику приложения, базу данных и пользовательский интерфейс в виде HTML-страниц. Любые изменения в логике сервера, даже самые небольшие, приводят к выпуску новой версии всего приложения. А с учетом того, что все больше приложений развертываются на облачных серверах, после внесенных изменений нужно заново развертывать в облаке монолитное приложение целиком. Монолитный подход является самым старым, именно с него началась разработка всего программного обеспечения. Тем не менее, монолитные приложения живы и по сей день, и в некоторых проектах такой подход является оправданным.

Недостатки монолитной архитектуры привели к появлению сначала **модульной**, а потом и сервис-ориентированной архитектуры. **Сервис-ориентированная архитектура** (SOA) основана на использовании отдельных полностью самодостаточных модулей, называемых **сервисами**. Разработка сервисов происходит отдельно друг от друга, а потому изменения в коде одного сервиса будут влиять только на него.

Около 10 лет назад (сам термин устоялся только в 2012 году) появился еще один подход, «современный взгляд» на проектирование распределенных приложений — **архитектура микросервисов** (MSA, Micro Service Architecture). Можно сказать, что MSA является разновидностью, частным случаем SOA. Такая архитектура подразумевает, что ваше приложение представляет собой много небольших сервисов, которые взаимодействуют между собой путем обмена сообщениями, как правило, по протоколу HTTP (или разные сервисы взаимодействуют по разным протоколам). MSA предполагает, что каждый микросервис — это полностью независимое приложение, содержащее всего несколько сотен строк кода. Согласно MSA, микросервисы могут располагаться даже на различных серверах. Интересно, что при таком подходе разные микросервисы в рамках одного приложения могут быть написаны на разных языках программирования и с применением разных технологий.

Рекомендуем прочитать подробную статью про микросервисную архитектуру на [Хабре](https://habr.com/company/dataart/blog/280083/).

Можно сказать, что микросервисы — это теория, а на практике она может выражаться по-разному. Мы рассмотрим архитектурный стиль, основанный на идеологии микросервисов, который на настоящее время становится стандартом при разработке веб-приложений — **архитектурный стиль REST** (REpresentational State Transfer).

**REST**

REST — архитектурный стиль программного обеспечения, который определяет правила управления информационными потоками и правила взаимодействия компонентов распределенного приложения. Другими словами, REST — это набор ограничений и принципов взаимодействия сервера и клиента в сети, который использует существующие стандарты: протокол HTTP, стандарт построения URL, форматы данных JSON и XML.

Ключевым понятием в архитектуре REST является URL. Каждая единица информации имеет уникальный идентификатор — URL, который строится по строго заданному формату. Например, вторая книга с книжной полки будет иметь URL /book/2, а 50-я страница в этой книге — /book/2/page/50.

То, каким образом происходит управление этой информацией, определяет протокол передачи данных. REST использует протокол HTTP, соответственно, управление информацией происходит с помощью HTTP-запросов: GET (получить), PUT (заменить), POST (добавить), DELETE (удалить). Для каждой единицы информации определяется пять операций:

| **№** | **Операция** | **Запрос** | **Пример URL** | **Примечание** |
| --- | --- | --- | --- | --- |
| 1 | Получить список всех объектов | GET | /book/ | Получить список всех книг на полке |
| 2 | Получить информацию об объекте | GET | /book/2 | Получить информацию о книге № 2 |
| 3 | Создать новый объект | POST | /book/ | Добавить новую книгу на основе информации, переданной с запросом |
| 4 | Изменить существующий объект | PUT | /book/1 | Изменить книгу № 1 на основе информации, переданной с запросом |
| 5 | Удалить существующий объект | DELETE | /book/2 | Удалить книгу № 2 |

Данные для изменения и добавления (PUT- и POST-запросы) передаются в теле запроса в формате JSON или XML. Глядя на запрос, можно сразу определить, для чего он служит (к сожалению, в некоторых других архитектурных стилях — SOAP, XML-RPC — это не так).

Приложения, поддерживающие архитектуру REST, называются **RESTful-приложениями**. Давайте создадим наш первый RESTful веб-сервис.

Перед написанием кода нужно продумать, с какой информацией мы будем работать и на основе каких правил будут строиться URL. Давайте вернемся к нашему приложению по работе с новостями из прошлого урока и построим его по принципам REST. Для получения списка всех новостей можно использовать такой URL:

http://127.0.0.1:5000/api/news

**Blueprint**

Чтобы не смешивать код обработчиков API для новостей с кодом основного приложения, мы воспользуемся механизмом разделения приложения Flask на независимые модули или Blueprint. Как правило, blueprint — логически выделяемый набор обработчиков адресов, который можно вынести в отдельный файл или модуль. Blueprint работает аналогично объекту приложения Flask, но в действительности он не является приложением. Обычно это лишь эскиз для сборки или расширения приложения.

Давайте создадим файл news\_api.py и добавим в него следующий код:

import flask

from . import db\_session

from .news import News

blueprint = flask.Blueprint(

'news\_api',

\_\_name\_\_,

template\_folder='templates'

)

@blueprint.route('/api/news')

def get\_news():

return "Обработчик в news\_api"

Мы создали схему, в которую добавили обработчик. Обратите внимание: мы воспользовались не декоратором app.route, а аналогичным для схемы — blueprint.route. Теперь нужно, чтобы о нашей схеме узнало основное приложение, для чего в main.py перед запуском добавим регистрацию схемы:

from data import db\_session, news\_api

def main():

db\_session.global\_init("db/blogs.db")

app.register\_blueprint(news\_api.blueprint)

app.run()

Перейдем по адресу http://127.0.0.1:5000/api/news и проверим, что у нас получилось.



**Сериализация в json**

Обычно REST-сервисы обмениваются информацией в формате json. Поэтому надо научить объекты наших моделей превращаться в словарь, для чего надо добавить каждой из моделей метод *to\_dict*. Это достаточно просто сделать руками, но мы воспользуемся возможностями модуля SQLAlchemy-serializer. Установим его:

pip install SQLAlchemy-serializer

Теперь мы снова сможем воспользоваться множественным наследованием и добавить к каждой из наших моделей еще один класс-примесь *SerializerMixin*, добавляющий метод *to\_dict*.

Заголовок класса *Users* теперь у нас выглядит так:

class User(SqlAlchemyBase, UserMixin, SerializerMixin):

Не забудьте добавить *SerializerMixin* и в остальные модели.

**Создание и тестирование сервисов**

Теперь допишем в функцию get\_news() код для получения всех новостей. Давайте пока опустим проверку авторизации пользователя.

@blueprint.route('/api/news')

def get\_news():

db\_sess = db\_session.create\_session()

news = db\_sess.query(News).all()

return jsonify(

{

'news':

[item.to\_dict(only=('title', 'content', 'user.name'))

for item in news]

}

)

Как видите, мы можем явно указать, какие поля хотим оставить в получившемся словаре.

Согласно архитектуре REST, обмен данными между клиентом и сервером осуществляется в формате JSON (реже — XML). Поэтому мы изменили формат ответа сервера flask с помощью метода *jsonify*, который преобразует наши данные в JSON.

Посмотреть, что вернул сервер, можно и не открывая браузер. По сути, браузер — это средство для отправки запросов на сервер и отображения ответа. Отправить запрос на сервер можно и другими способами (например, консольная утилита curl или GUI-программа Postman), но как программисты на Python, мы воспользуемся его средствами, а именно — модулем requests. Мы уже неплохо знакомы с этим модулем и помним, что он позволяет отправлять на сервер разные типы запросов, передавать данные вместе с POST- и PUT-запросами. Для этого используются специальные функции, их имена совпадают с именем HTTP-запроса. Для тестирования нашего RESTful-сервиса создадим файл test.py со следующим содержимым:

from requests import get

print(get('http://localhost:5000/api/news').json())

Запустим файл (не забудьте запустить сервер) и увидим, что на наш GET-запрос пришел ответ от сервера со всеми новостями из базы данных.

Согласно REST, далее нужно реализовать получение информации об одной новости. Фактически, мы уже получили из списка всю информацию о каждой новости. При проектировании приложений по архитектуре REST обычно поступают таким образом: когда возвращается список объектов, он содержит только краткую информацию (например, только id и заголовок), а полную информацию (текст и автора) можно посмотреть с помощью запроса, который мы обработаем далее.

Напишем функцию *get\_one\_news* для получения информации о новости по ее идентификатору. Как нам говорит REST, идентификатор объекта содержится в URL, по которому мы обращаемся. Как же передать этот идентификатор, если он различный у всех новостей, а в декораторе функции-обработчика мы указываем статический адрес? Flask позволяет решить эту проблему с помощью параметров адреса: мы можем передать параметр в URL, в декораторе заключив его в угловые скобки и указав тип и имя (например, <int:news\_id>), и он будет передан функции-обработчику как аргумент. Обратите внимание: теперь наша функция принимает аргумент *news\_id*:

@blueprint.route('/api/news/<int:news\_id>', methods=['GET'])

def get\_one\_news(news\_id):

db\_sess = db\_session.create\_session()

news = db\_sess.query(News).get(news\_id)

if not news:

return jsonify({'error': 'Not found'})

return jsonify(

{

'news': news.to\_dict(only=(

'title', 'content', 'user\_id', 'is\_private'))

}

)

Посмотрим ответ сервера с корректным *id* и с ошибочным, добавив еще два запроса в файл test.py:

print(get('http://localhost:5000/api/news/1').json())

print(get('http://localhost:5000/api/news/999').json())

# новости с id = 999 нет в базе

Как видим, в первом случае нам вернулась одна новость, во втором — сообщение об ошибке. В типе параметра URL мы указали *int*. Что же будет, если мы передадим не число? Давайте проверим.

print(get('http://localhost:5000/api/news/q').json())

Файл test.py завершится с ошибкой. Так как параметр не типа *int*, в функцию *get\_one\_news* мы даже не попадем — нет полного совпадения декоратора. На такой запрос сервер вернет ответ со статусом 404 (страница не найдена), но не в формате JSON. Поэтому при попытке преобразовать ответ с помощью метода json() программа завершается с ошибкой, так как клиентское приложение, в нашем случае, ожидает от сервера ответ в формате JSON. Уберите приведение ответа к JSON и посмотрите, что вернул сервер.

Чтобы такой ошибки не возникало, воспользуемся модулем make\_response. Импортируем модуль и добавим функцию с декоратором 404 ошибки:

from flask import make\_response

@app.errorhandler(404)

def not\_found(error):

return make\_response(jsonify({'error': 'Not found'}), 404)

После этого даже при передаче неправильного параметра ответ от сервера будет приходить в формате JSON, и клиентское приложение не будет падать. Верните метод json() и проверьте результат.

Следующая операция, которую нам нужно реализовать — добавление новой новости с помощью POST-запроса. Как уже было сказано, данные для создаваемой новости будут переданы в теле запроса. Чтобы добраться до тела запроса, нам понадобиться модуль request из flask (не путайте с модулем requests).

@blueprint.route('/api/news', methods=['POST'])

def create\_news():

if not request.json:

return jsonify({'error': 'Empty request'})

elif not all(key in request.json for key in

['title', 'content', 'user\_id', 'is\_private']):

return jsonify({'error': 'Bad request'})

db\_sess = db\_session.create\_session()

news = News(

title=request.json['title'],

content=request.json['content'],

user\_id=request.json['user\_id'],

is\_private=request.json['is\_private']

)

db\_sess.add(news)

db\_sess.commit()

return jsonify({'success': 'OK'})

Проверив, что запрос содержит все требуемые поля, мы заносим новую запись в базу данных. *request.json* содержит тело запроса, с ним можно работать, как со словарем. В нашем случае все ключи заранее известны и все поля обязательны. В противном случае для работы со словарем *request.json* нужно использовать метод словарей *get*. Вспомните, в чем его особенность.

При тестировании проверим пустой запрос, передачу только заголовка новости и корректный запрос. Все данные передаются в параметре json:

print(post('http://localhost:5000/api/news').json())

print(post('http://localhost:5000/api/news',

json={'title': 'Заголовок'}).json())

print(post('http://localhost:5000/api/news',

json={'title': 'Заголовок',

'content': 'Текст новости',

'user\_id': 1,

'is\_private': False}).json())

Проверьте, что в каждом случае от сервера приходит нужное сообщение. Также давайте отправим запрос на получение всех новостей и убедимся, что новость добавлена.

Добавим функцию удаления новости:

@blueprint.route('/api/news/<int:news\_id>', methods=['DELETE'])

def delete\_news(news\_id):

db\_sess = db\_session.create\_session()

news = db\_sess.query(News).get(news\_id)

if not news:

return jsonify({'error': 'Not found'})

db\_sess.delete(news)

db\_sess.commit()

return jsonify({'success': 'OK'})

И протестируем ее:

print(delete('http://localhost:5000/api/news/999').json())

# новости с id = 999 нет в базе

print(delete('http://localhost:5000/api/news/1').json())

Такой «ручной» способ создания RESTful-сервисов достаточно объемный и трудоемкий. Нужно держать в голове все URL и методы, создавать много функций-обработчиков с декораторами для каждого типа запросов (эти функции как раз и являются микросервисами). Программисты на Python не были бы программистами на Python, если бы не стремились упростить объемный код и сделать его более лаконичным. Так и появился легкий способ создавать RESTful-сервисы на flask — модуль Flask-RESTful, который мы рассмотрим уже на следующем уроке.

Еще один важный вопрос не был рассмотрен в этом уроке — вопрос авторизации. Сейчас удалять, добавлять и просматривать новости с помощью нашего API может любой пользователь, а точнее любая программа, которая будет использовать этот API. В настоящих API для выполнения таких действий пользователь должен представиться. Эта тема раскрыта в видео к уроку.

Основы промышленного программирования | Д20Кольцово - Технополис - 1 - Д2

[Задать вопрос](https://connect.yandex.ru/forms/question-form.staff-id/)

[[](https://passport.yandex.ru/?)](https://passport.yandex.ru?" \t "_parent)

[Маракасова Юлия](https://passport.yandex.ru?" \t "_parent)

* [Обзор курса](https://lyceum.yandex.ru/courses/352/groups/2321)
* [Решения](https://lyceum.yandex.ru/courses/352/groups/2321/solutions)
* [Расписание](https://lyceum.yandex.ru/courses/352/groups/2321/schedule)
* [Успеваемость](https://lyceum.yandex.ru/courses/352/groups/2321/progress)

[Урок Flask-restful](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2054)

**REST-API. Flask-RESTful**

1. [Flask-RESTful](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2054/materials/4746#1)
2. [Миграции](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2054/materials/4746#2)

**Аннотация**

Сегодня мы посмотрим, как создавать RESTful-сервисы с помощью библиотеки flask-restful.

**Flask-RESTful**

Все то, что мы сделали ранее при создании RESTful-сервиса, можно обернуть в более лаконичную, а главное — объектно-ориентированную оболочку, забыв о множестве разбросанных функций-обработчиков. Для этого установим дополнительный модуль flask-restful с помощью pip.

pip install flask-restful

Из него нам понадобятся следующие объекты и функции:

from flask\_restful import reqparse, abort, Api, Resource

Создадим вторую версию нашего REST-сервиса. После создания flask-приложения создадим объект RESTful-API:

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

api = Api(app)

Для каждого ресурса (единица информации в REST называется **ресурсом**: новости, пользователи и т. д.) создается два класса: для одного объекта и для списка объектов. В нашем случае это классы *NewsResource* и *NewsListResource* соответственно. Создадим их в отдельном модуле, который назовем news\_resources.py. Оба этих класса необходимо унаследовать от класса *Resource* из модуля flask-restful.

В классе одного объекта (*NewsResource*) определяются операции, которые мы можем сделать с **одним** объектом: получить информацию об объекте, изменить информацию, удалить объект. flask-restful подразумевает, что эти методы будут иметь имена, аналогичные соответствующим HTTP-запросам: *get*, *put*, *post*, *delete*. И все они, конечно, должны принимать в качестве аргумента идентификатор объекта.

Когда мы проектировали REST-сервис вручную, в каждой функции-обработчике мы проверяли, существует ли новость с таким идентификатором, и если нет, то отправляли ошибку. А также мы определили функцию *not\_found* с декоратором @app.errorhandler(404), чтобы изменить формат ответа сервера в случае ошибки. Давайте вынесем эту проверку в отдельную функцию в файле с ресурсами для записей:

def abort\_if\_news\_not\_found(news\_id):

session = db\_session.create\_session()

news = session.query(News).get(news\_id)

if not news:

abort(404, message=f"News {news\_id} not found")

Функция *abort* генерирует HTTP-ошибку с нужным кодом и возвращает ответ в формате JSON, поэтому функция *not\_found* нам больше не нужна. В предыдущем нашем варианте мы никак не использовали коды ответа сервера, это неверно. Наше сообщение об ошибке мог прочитать только человек, а для программы-клиента оно ничего не значит, ответ пришел со статусом ОК (200). Давно знакомый вам код 404 означает, что запрашиваемый ресурс не найден. Подробнее про статусы состояния протокола HTTP можно почитать [здесь](https://ru.wikipedia.org/wiki/Список_кодов_состояния_HTTP).

Класс *NewsResource* с методами получения информации и удаления будет иметь вид:

class NewsResource(Resource):

def get(self, news\_id):

abort\_if\_news\_not\_found(news\_id)

session = db\_session.create\_session()

news = session.query(News).get(news\_id)

return jsonify({'news': news.to\_dict(

only=('title', 'content', 'user\_id', 'is\_private'))})

def delete(self, news\_id):

abort\_if\_news\_not\_found(news\_id)

session = db\_session.create\_session()

news = session.query(News).get(news\_id)

session.delete(news)

session.commit()

return jsonify({'success': 'OK'})

В классе списка объектов (*NewsListResource*) определяются операции, которые мы можем сделать с **набором** объектов: показать список объектов и добавить объект в список. В этом классе нам потребуется реализовать два метода: *get* и *post* без аргументов. Доступ к данным, переданным в теле POST-запроса, осуществляется с помощью парсера аргументов из модуля reqparse, который предварительно нужно создать и добавить в него аргументы:

parser = reqparse.RequestParser()

parser.add\_argument('title', required=True)

parser.add\_argument('content', required=True)

parser.add\_argument('is\_private', required=True, type=bool)

parser.add\_argument('is\_published', required=True, type=bool)

parser.add\_argument('user\_id', required=True, type=int)

По-прежнему считаем, что все поля новости являются обязательными. Также укажем тип идентификатора пользователя — целое число. Теперь всю проверку аргументов запроса за нас будет делать модуль reqparse.

Таким образом, класс *NewsListResource* будет иметь вид:

class NewsListResource(Resource):

def get(self):

session = db\_session.create\_session()

news = session.query(News).all()

return jsonify({'news': [item.to\_dict(

only=('title', 'content', 'user.name')) for item in news]})

def post(self):

args = parser.parse\_args()

session = db\_session.create\_session()

news = News(

title=args['title'],

content=args['content'],

user\_id=args['user\_id'],

is\_published=args['is\_published'],

is\_private=args['is\_private']

)

session.add(news)

session.commit()

return jsonify({'success': 'OK'})

После того как мы создали классы ресурсов, нам надо внести их в настройки нашего RESTful-API, указав имя класса и URL. Параметр URL также указывается в угловых скобках. Для этого в main.py перед запуском нашего приложения необходимо добавить вот такой код:

# для списка объектов

api.add\_resource(news\_resources.NewsListResource, '/api/v2/news')

# для одного объекта

api.add\_resource(news\_resources.NewsResource, '/api/v2/news/<int:news\_id>')

Давайте проверим такую реализацию RESTful-сервиса с помощью запросов в нашем тестовом файле test.py. Убедитесь, что все работает, как нужно.

Документация по модулю flask-RESTful есть на [официальном сайте](http://flask-restful.readthedocs.io/en/latest/index.html).

Если ваше приложение построено по правилам REST, вы получите логичную организацию работы с ресурсами, даже если их в приложении большое количество. Клиенты используют простые и понятные URL, а новым клиентам не составит труда разобраться с интерфейсом вашего приложения. Также в мире веб-разработки стандартом становится предоставлять [RESTful API](https://restfulapi.net/rest-api-design-tutorial-with-example/) для сторонних приложений.

Есть ли подводные камни при использовании REST? Да, и этим камнем является HTML. Дело в том, что спецификация HTML позволяет создавать формы, отправляющие только GET- или POST-запросы. Поэтому для нормальной работы с другими методами (PUT, DELETE) приходится имитировать их искусственно. Часто в этих случаях используют JavaScript-библиотеки, например, JQuery.

Как и предыдущем уроке, здесь мы не рассматривали вопрос авторизации. В видеоматериалах к уроку эта тема раскрыта.

**Миграции**

Мы уже говорили о том, что sqlalchemy при старте не вносит изменения в таблицы, если они уже созданы. Но абсолютно всегда бывает ситуация, когда после запуска первой версии нам захочется что-то поменять такого, что затронет и наши модели. Наверняка нам захочется хранить в базе дополнительную информацию. Что делать в таком случае? Не удалять же базу данных с информацией о куче пользователей, они такого не простят. Конечно, можно написать SQL-скрипты для приведения базы данных в актуальное состояние, но в общем случае это занятие не из приятных, потому что столбцы могут изменяться, удаляться, могут меняться связи между моделями и т. д. Тут на помощь приходят инструменты для **миграции**.

Уже упоминалось, что это такая штука, вроде системы контроля версий, только для базы данных. Инструменты миграции созданы для того, чтобы поддерживать вашу базу данных в актуальном состоянии и автоматически генерировать скрипты для перехода состояния базы из состояния n к состоянию n + 1 и обратно (примерно как переходы по истории коммитов).

Какие-то ORM содержат модули миграции прямо из коробки (например, в Django), для sqlalchemy нам придется устанавливать и настраивать такую библиотеку отдельно. Она называется [Alembic](https://alembic.sqlalchemy.org/en/latest/).

Есть еще flask-migrate — обертка для alembic, упрощающая работу именно для flask-приложений, но мы рассмотрим более общий пример.

Установим библиотеку:

pip install alembic

После в командной строке нужно перейти в директорию с нашим проектом и выполнить команду:

alembic init alembic

Эта команда создаст директорию с именем alembic, в которой модуль будет хранить все необходимые файлы. А также alembic.ini в директории вашего проекта.

Откройте alembic.ini и измените строку с параметром *sqlalchemy.url* на путь к вашей базе данных:

sqlalchemy.url = sqlite:///db/blogs.sqlite?check\_same\_thread=False

После этого перейдем в папку alembic. Нас интересует файл env.py. Откроем его и найдем строчку с переменной *target\_metadata*. Эта переменная должна «узнать» о всех файлах нашей модели. Напишем там следующий код:

import sys

sys.path.insert(0, 'Путь к папке вашего проекта')

from data.db\_session import SqlAlchemyBase

import data.\_\_all\_models

target\_metadata = SqlAlchemyBase.metadata

sys.path.insert — вставляет путь к каталогу в PATH. Это нужно для того, чтобы было проще писать импорты данных остальных файлов.

Затем импортируем нашу базу данных, напоминаем ей о всех моделях, после чего получаем данные о всех моделях.

Давайте попробуем. Например, мы хотим добавить в модель с новостью поле, которое показывает, опубликована запись или нет, чтобы потом показывать в списке только опубликованные записи. Идем в модель *News* и добавляем в нее новое поле:

is\_published = sqlalchemy.Column(sqlalchemy.Boolean, default=True)

Теперь запускаем консоль в папке нашего проекта и выполняем команду:

alembic revision --autogenerate -m "добавили признак публикации"

Alembic посмотрит на наши модели и на базу и создаст в папке alembic/versions новую миграцию вот с таким кодом:

from alembic import op

import sqlalchemy as sa

# revision identifiers, used by Alembic.

revision = '3acec80b2659'

down\_revision = None

branch\_labels = None

depends\_on = None

def upgrade():

# ### commands auto generated by Alembic - please adjust! ###

op.add\_column('news', sa.Column('is\_published', sa.Boolean(),

nullable=True))

# ### end Alembic commands ###

def downgrade():

# ### commands auto generated by Alembic - please adjust! ###

op.drop\_column('news', 'is\_published')

# ### end Alembic commands ###

Тут всего две функции. *upgrade* выполняет код для изменения состояния базы, *downgrade* — код для возврата к предыдущему состоянию. Давайте обновим нашу базу, для чего выполним в консоли команду:

alembic upgrade head

*head* означает, что мы хотим применить все миграции друг за другом для приведения базы в самое актуальное состояние. Вместо *head* можно указать номер ревизии или написать, например, +2, чтобы обновиться только на 2 следующие версии.

Посмотрим, что случилось с нашей базой данных. В таблице news появилось поле *is\_published*, как и ожидалось, но кроме этого появилась новая таблица *alembic\_version*. В этой таблице хранится номер ревизии, которой в настоящий момент соответствует база данных.

К сожалению, alembic не всемогущ. Вот что он умеет определять.

Операции, которые Alembic умеет выявлять:

* Добавление и удаление таблиц
* Добавление и удаление колонок
* Изменения во внешних ключах
* Изменения в типах колонок
* Изменения в индексах и использованных уникальных ограничениях

А вот какие изменения не умеет:

* Изменение имени таблицы
* Изменение имени колонки

Поэтому бывают ситуации, когда в файлы миграции надо вмешиваться и дописывать в них необходимые изменения. Не бойтесь этого.

Основы промышленного программирования | Д20Кольцово - Технополис - 1 - Д2

[Задать вопрос](https://connect.yandex.ru/forms/question-form.staff-id/)

[[](https://passport.yandex.ru/?)](https://passport.yandex.ru?" \t "_parent)

[Маракасова Юлия](https://passport.yandex.ru?" \t "_parent)

* [Обзор курса](https://lyceum.yandex.ru/courses/352/groups/2321)
* [Решения](https://lyceum.yandex.ru/courses/352/groups/2321/solutions)
* [Расписание](https://lyceum.yandex.ru/courses/352/groups/2321/schedule)
* [Успеваемость](https://lyceum.yandex.ru/courses/352/groups/2321/progress)

[Урок Облако](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2055)

**Разворачиваем проект в облаке. Дорешка**

1. [Введение](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2055/materials/4728#1)
2. [Туннель в Интернет](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2055/materials/4728#2)
3. [Heroku](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2055/materials/4728#3)
4. [Яндекс.Облако](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2055/materials/4728#4)

**Аннотация**

Сегодня мы посмотрим, как разместить свой проект в Интернете, чтобы к нему могли получить доступ все желающие, а также изучим несколько сервисов, которые смогут нам в этом помочь.

Этот урок не содержит задач.

**Введение**

Разрабатывать веб-приложения, которые «живут» только на нашем локальном компьютере, согласитесь, не очень весело. Само слово «веб» подразумевает, что к результатам нашей работы должны иметь доступ все пользователи сети Интернет, ну или те, которым мы даем на это право. Процесс размещения ресурсов в Интернете называется deploy.

Пока наше flask-приложение размещено на локальном компьютере, и мы помним, что каждый компьютер в мире думает про себя, что его IP-адрес 127.0.0.1. Как же тогда пользовательский запрос из Интернета может добраться до нашего компьютера и получить нужную веб-страницу или json-ответ от сервиса? Ведь таких машин с адресом 127.0.0.1 миллионы!

Для решения этой задачи есть несколько способов и множество разных сервисов. С некоторыми из них мы сегодня познакомимся.

Давайте сделаем простейшее веб-приложение, на примере которого мы будем проверять различные способы организации доступа к веб-приложению через Интернет.

import os

from flask import Flask

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

@app.route("/")

def index():

return "Привет от приложения Flask"

if \_\_name\_\_ == '\_\_main\_\_':

port = int(os.environ.get("PORT", 5000))

app.run(host='0.0.0.0', port=port)

Из нового: *os.environ* пытается получить значение переменной окружения с именем *PORT*, а если не получается, то выставляет порт 5000. Это нам пригодится позднее.

host = 0.0.0.0 означает, что мы разрешаем подключения из любой сети по любому интерфейсу.

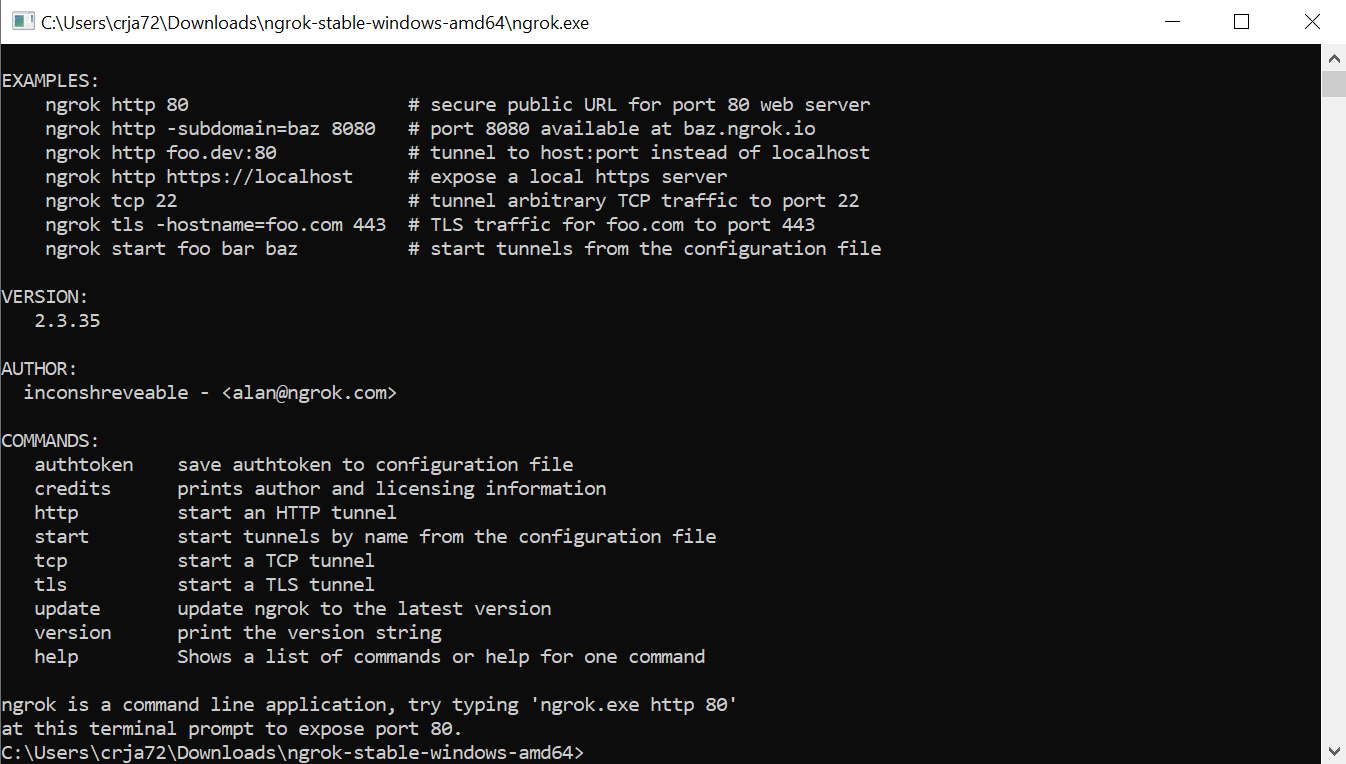
**Туннель в Интернет**

Очень часто при отладке веб-приложений хочется посмотреть, что происходит, когда в приложение приходят запросы из Интернета, или продемонстрировать результаты работы другу/заказчику. Не всегда удобно ради тестирования или ради одноразовой демонстрации арендовать и настраивать сервер в Интернете.

Согласитесь, было бы удобно получить какой-то временный адрес в Интернете «в один клик» для таких целей. К счастью, для этого есть сервисы, которые организуют виртуальный туннель из Интернета на ваш локальный компьютер. Один из них — [ngrok](https://ngrok.com/).

Важная его особенность: у ngrok есть бесплатный план, который не требует даже регистрации на сервисе. Но даже этой ограниченной версии хватит для наших целей.

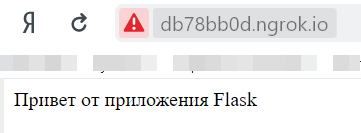
Ngrok не требует практически никакой настройки. Необходимо скачать исполняемый файл приложения для нужной операционной системы со [страницы загрузки](https://ngrok.com/download) и запустить его. Нас встретит такое окно:



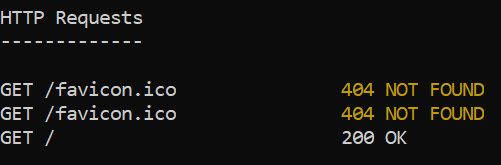
Теперь запустим наше тестовое приложение в PyCharm или через командную строку. Выполним в окне ngrok команду:

ngrok http 5000

После этого ngrok создаст временный адрес в Интернете вида набор\_символов.ngrok.io и виртуальный туннель с этого адреса на пятитысячный порт нашего локального компьютера. Теперь все запросы по протоколу http и https, которые придут на созданный временный адрес, будут перенаправляться на наш компьютер. Давайте проверим. Перейдем по этому адресу (скопируем и вставим его в строку запроса браузера) и посмотрим на результат.



Все работает! А в консоли ngrok появилось сообщение, что GET-запрос успешно обработан:



Таким образом можно организовывать тестирование приложения в Интернете. Особенно такой подход удобен, когда для проверки функциональности приложения надо получить запрос извне. В дальнейшем мы будем использовать эту методику при отладке приложений для голосового помощника Алисы.

Кроме ручного запуска ngrok есть возможность автоматизировать этот процесс с помощью модуля flask-ngrok. Установим модуль:

pip install flask-ngrok

Чтобы все заработало как надо, необходимо внести небольшие изменения в код нашего примера:

from flask import Flask

from flask\_ngrok import run\_with\_ngrok

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

run\_with\_ngrok(app)

@app.route("/")

def index():

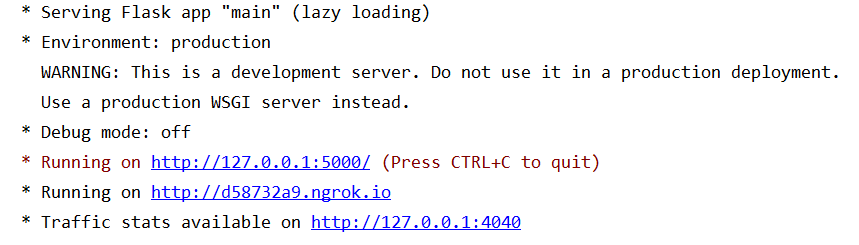
return "Привет от приложения Flask"

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

После создания объекта класса *Flask* необходимо передать его в функцию *run\_with\_ngrok* библиотеки flask-ngrok. Все предельно просто. Однако обратите внимание: при использовании этой библиотеки при вызове метода *run* объекта класса *Flask* пока нельзя передавать дополнительные параметры (хост, порт, и т. д.).

После запуска программы сначала запустится прослушивание локального порта, а затем и тунель через ngrok.



**Heroku**

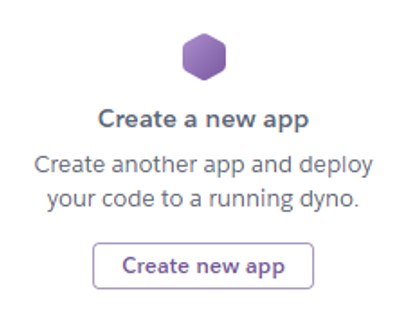
Более продвинутый способ, который (при использовании платных планов) подойдет для «боевого» разворачивания приложения в Интернете (деплоя) — аренда контейнеров для приложений на одном из множества сервисов, которые предоставляют такие услуги.

Сервисов действительно очень много, и все они разнятся как в стоимости (некоторые предлагают бесплатный тариф, который подойдет для приложения с небольшим количеством пользователей), так и в сложности настроек для запуска.

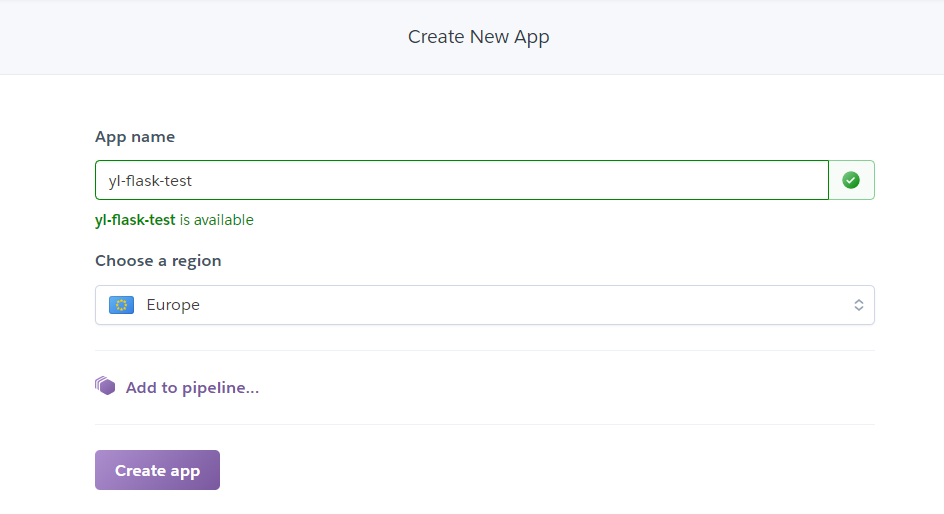
В качестве одного из самых легких стоит привести в пример [pythonanywhere.com](https://www.pythonanywhere.com/), который содержит ряд преднастроенных шаблонов для различных приложений, разработанных на Python. К недостатку этого сервиса можно отнести то, что приложения, размещенные на нем, могут обращаться только к тем ресурсам, которые были внесены в «белый список» доступных адресов.

Немного более сложным в обращении является сервис [Heroku](https://dashboard.heroku.com/apps). Он поддерживает приложения не только на Python, но и на других языках программирования, различные базы данных и разные преднастроенные сторонние приложения.

Зарегистрируйтесь на Heroku. После чего создайте новое приложение по адресу <https://dashboard.heroku.com/apps>



Придумайте приложению имя и выберите расположение.



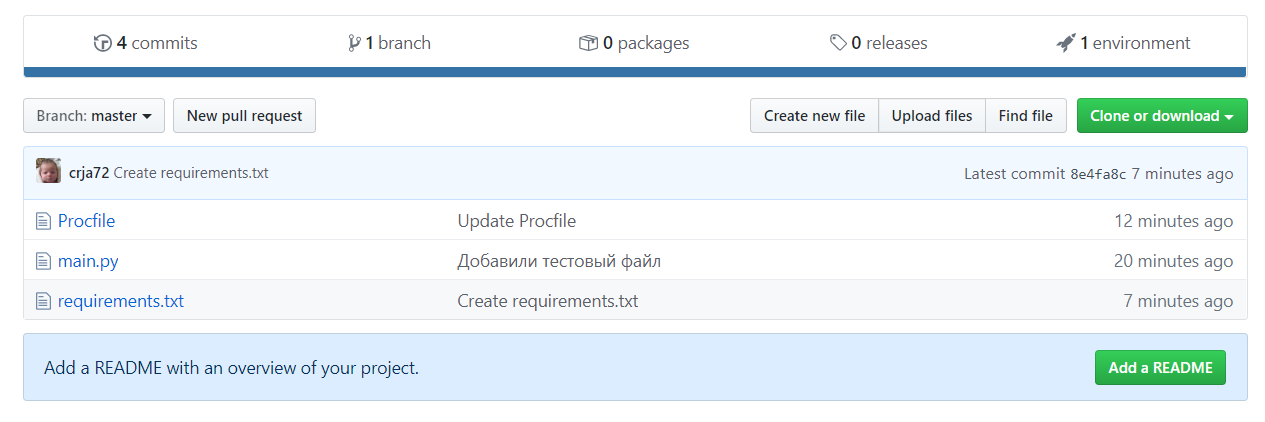
Есть несколько способов разместить свое приложение на Heroku:

* Через консольный клиент Heroku CLI
* Подключив свой репозиторий на GitHub

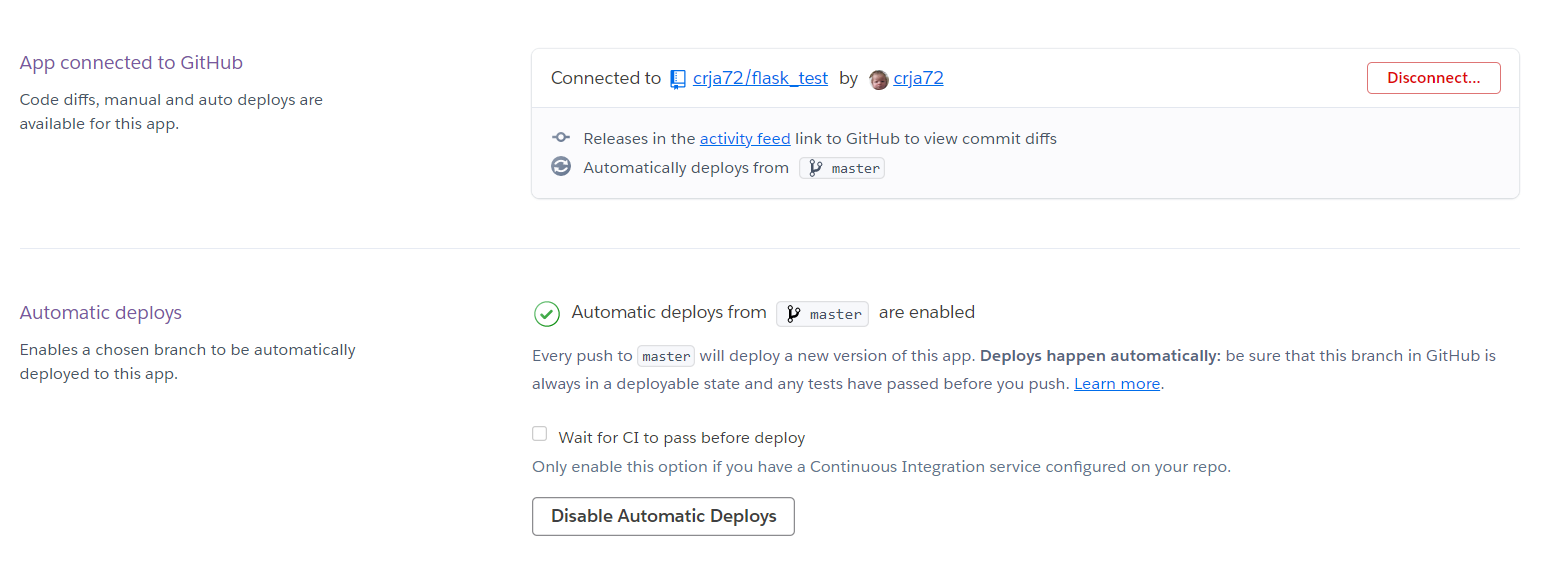
Давайте пойдем по второму пути и создадим репозиторий (не имеет значение публичный или приватный), в который **запушим** файл с нашим простейшим веб-приложением, а также еще несколько файлов. Первый из них — служебный текстовый файл Procfile (без расширения), в котором для Heroku будет указано, какой файл запускать, с текстом:

web: python main.py

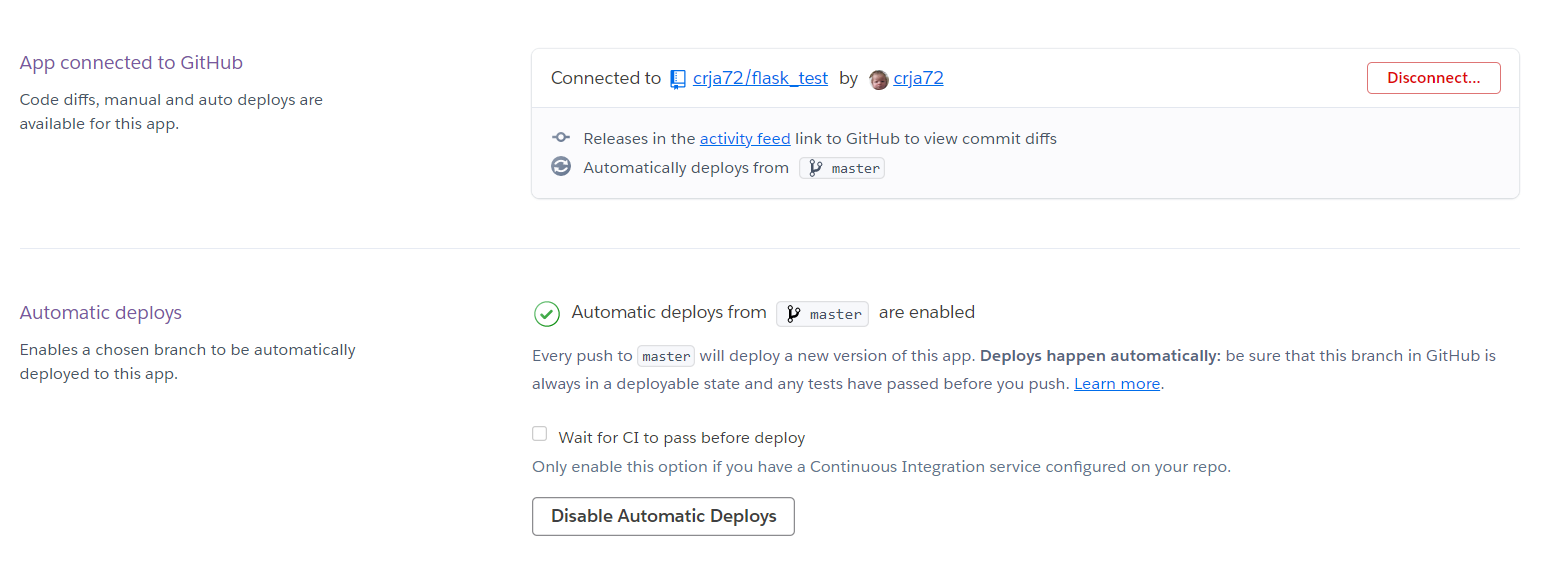
Второй — уже знакомый нам из раздела PyQT requirements.txt, в котором необходимо указать все используемые сторонние модули (flask).



Подключим GitHub к Heroku и выберем нужный репозиторий, а затем и ветку, с которой будем забирать для размещения код. Это можно сделать автоматически или в ручном режиме.



Если не включено автоматическое размещение кода, надо нажать на кнопку Delpoy branch.



Все. Можно переходить по адресу имя\_приложения.herokuapp.com и наслаждаться результатом.

При размещении приложений на Heroku очень важно указывать порт, на котором будет работать ваше приложение именно так, как мы сделали в первом примере этого урока.

if \_\_name\_\_ == '\_\_main\_\_':

port = int(os.environ.get("PORT", 5000))

app.run(host='0.0.0.0', port=port)

Когда приложение разворачивается в Heroku, для него будет выделен порт и установлен в переменную окружения. Важно, чтобы ваше приложение слушало именно этот порт, иначе оно не заработает.

**Яндекс.Облако**

Самым сложным, но одновременно самым гибким и продвинутым вариантом является размещение приложения на облачной виртуальной машине с использованием сервиса Яндекс.Облако или аналогов.

Создание и настройка таких виртуальных машин требует определенных усилий и несколько сложнее, чем сервисы, которые мы рассматривали ранее, но зато вы сможете полностью контролировать результат и реализовывать разные продвинутые сценарии разворачивания вашего приложения.

Для начала необходимо зайти на [Яндекс.Облако](http://cloud.yandex.ru/) и нажать кнопку «Подключиться» в правом верхнем углу экрана. Далее алгоритм работы можно описать так:

1. Необходимо создать виртуальную машину
2. Выбрать и установить образ операционной системы, которую вы планируете использовать. Можно использовать как «чистый» образ (только операционная система), так и подготовленный командой Яндекс.Облака или сторонними разработчиками
3. Установить все необходимое ПО: интерпретатор Python, все библиотеки, системные утилиты, которые обеспечат работу при большой нагрузке и т. д.
4. Загрузить разработанный вами сервер и ...
5. Поддерживать работу вашего ресурса

Поверьте, это далеко не просто.

Поэтому мы немного вам помогли. Мы подготовили для вас видео, в котором рассказываем, как сделать базовую настройку сервера и установить на него ваше приложение.

Основы промышленного программирования | Д20Кольцово - Технополис - 1 - Д2

[Задать вопрос](https://connect.yandex.ru/forms/question-form.staff-id/)

[[](https://passport.yandex.ru/?)](https://passport.yandex.ru?" \t "_parent)

[Маракасова Юлия](https://passport.yandex.ru?" \t "_parent)

* [Обзор курса](https://lyceum.yandex.ru/courses/352/groups/2321)
* [Решения](https://lyceum.yandex.ru/courses/352/groups/2321/solutions)
* [Расписание](https://lyceum.yandex.ru/courses/352/groups/2321/schedule)
* [Успеваемость](https://lyceum.yandex.ru/courses/352/groups/2321/progress)

[Урок Алиса](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056)

**Алиса — урок 1**

1. [Алиса](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#1)
2. [Интеграция с Алисой](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#2)
3. [Логирование (журналирование)](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#3)
4. [Уровни логирования](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#4)
5. [Пишем код](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#5)
6. [Как работает данный код](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#6)
7. [Отладка с Postman](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#7)
8. [Регистрация и тестирование навыка](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#8)
9. [Именованные сущности](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#9)
10. [Ресурсы](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2056/materials/4725#10)

**Аннотация**

Мы начинаем блок по работе с голосовым помощником от Яндекса — Алисой.

**Алиса**

Алиса — это голосовой помощник от Яндекса. Фактически, это робот, который умеет общаться с нами посредством компьютера или любого другого вычислительного устройства: мобильного телефона, часов, «умной колонки» и даже системы управления автомобилем. В процессе общения Алиса может отвечать на вопросы, искать информацию в Интернете, играть с нами в различные игры, торговать на бирже, информировать о погоде, распознавать образы, управлять компонентами «умного дома» и делать множество других вещей.

**Навыками** принято называть отдельные задачи, которые Алиса умеет решать. Например, программист может научить Алису заказывать пиццу. Это умение и будет навыком. Сейчас различными разработчиками создано [большое количество навыков](https://dialogs.yandex.ru/store/), мы же с вами сегодня сделаем простой навык под названием «Купи слона», в котором Алиса будет с нами играть в простую игру:

* Купи слона
* Нет
* Все говорят «Нет». А ты купи слона
* Хорошо
* Слона можно найти на Яндекс.Маркете!

И на этом все. Эта игра — пример, который приведен на официальном сайте платформы Яндекс.Диалоги — места, где любой человек может публиковать свои навыки для Алисы.

Но для начала работы узнаем несколько вещей о разработке навыков.

**Интеграция с Алисой**

Первое, в чем нужно разобраться — как интегрироваться с Алисой. Алиса не имеет API в классическом понимании — как, например, у Яндекс.Карт. Интеграция происходит по технологии WebHook (технология — это громкое слово, скорее, идея).

Почему Алиса просто не могла предоставить нам API для работы? Как вы думаете, в чем сложность?

Алиса может обратиться к нашему навыку в любое время, поскольку никто не знает, когда один из миллионов пользователей Алисы решит им воспользоваться. Если бы Алиса предоставляла свое API, мы должны были бы регулярно ее спрашивать: «Эй, Алиса, никто не запрашивал наш навык?», затем при положительном ответе: «Запрашивали? Вот мой ответ. А что там ответил пользователь?» и так далее.

Как же быть?

Оказывается, лучшим решением является такое, в котором Алиса сама сообщит нам обо всех событиях, которые нас касаются. Идея WebHook’ов состоит в том, что мы не обращаемся к API, а реализуем свое, но по правилам, описанным в документации. Даем доступ к созданному API Алисе, и уже она начинает общаться с ним самостоятельно. Получается как бы «интеграция наоборот». В документации Алисы есть строгие требования к нашему API. После того, как API реализован, мы «говорим» Алисе, куда обращаться: сообщаем наш адрес. Если в интерфейсе Алисы кто-то вызвал наш навык, то Алиса сама нам об этом сообщит. Очень удобно! Не надо постоянно «пинать» Алису.

**Логирование (журналирование)**

Представьте ситуацию: вы не можете видеть на мониторе работу вашей программы. У вас просто может не быть монитора. Или программа размещена не на вашем компьютере, а где-то в облаке. И вы понимаете, что программа работает некорректно. Как же узнать, как она работает, и где вкралась ошибка? В таком случае помогает логирование.

**Логирование**

Логирование — это запись и вывод информации о выполняемых операциях, ошибках и других событиях в коде. Это могут быть значения переменных или любой текст.

Логирование может происходить в консоль или файл. Логи выводятся в консоль, например, в процессе отладки приложения. Если вы уже написали программу, которая выполняется на сервере, то записывать логи желательно в файл, чтобы потом их можно было изучить в случае остановки, падения или перезагрузки сервера.

Для логирования в Python используется библиотека logging.

Пример:

import logging

def log():

i = 0

while i < 10:

logging.warning(i)

i += 1

if \_\_name\_\_ == '\_\_main\_\_':

log()

Выводит в консоль:

WARNING:root:0

WARNING:root:1

WARNING:root:2

WARNING:root:3

WARNING:root:4

WARNING:root:5

WARNING:root:6

WARNING:root:7

WARNING:root:8

WARNING:root:9

В этом примере мы используем библиотеку logging с параметрами по умолчанию, поэтому вся информация выводится в консоль. Мы использовали функцию warning(), которая предназначена для привлечения внимания к проблеме, которая еще не считается ошибкой. *root* — это имя журнала, его можно менять.

Давайте научимся сохранять данные журнала в файл. Еще раз напомним, что это — правильная практика, когда программа используется в боевом режиме (production), так как файл сохранится в случае остановки, падения или перезагрузки сервера.

Для логирования в файл нужно добавить в код следующую строку:

logging.basicConfig(filename='example.log')

Теперь код целиком выглядит так:

import logging

logging.basicConfig(filename='example.log')

def log\_to\_file():

i = 0

while i < 10:

logging.warning(i)

i += 1

if \_\_name\_\_ == '\_\_main\_\_':

log\_to\_file()

После того, как мы запустим программу, в папке программы создастся файл example.log, и в нем окажется та же информация, что ранее выводилась в консоль:

WARNING:root:0

WARNING:root:1

WARNING:root:2

WARNING:root:3

WARNING:root:4

WARNING:root:5

WARNING:root:6

WARNING:root:7

WARNING:root:8

WARNING:root:9

Если мы запустим программу повторно, то увидим, что старые данные из файла не удаляются. Таким образом, информация будет записана дважды.

Чего же не хватает в файле? Что еще нам важно знать? Правильно — время события, ведь без него никакого расследования не получится.

Для того чтобы вывести временную метку, надо в функцию logging.basicConfig() передать еще один параметр — *format*. Как несложно догадаться, это формат сообщения (как выглядит и что содержит), которое записывается в лог. Сообщение в логе может содержать много полезной информации (подробно можно изучить [тут](https://docs.python.org/3/library/logging.html#logrecord-attributes)). Мы рассмотрим только четыре атрибута, которые можно вывести, и подробно расскажем о них.

С добавлением параметра *format* наш код будет выглядеть так:

import logging

logging.basicConfig(

filename='example.log',

format='%(asctime)s %(levelname)s %(name)s %(message)s'

)

def log\_to\_file():

i = 0

while i < 10:

logging.warning(i)

i += 1

if \_\_name\_\_ == '\_\_main\_\_':

log\_to\_file()

Рассмотрим параметр format='%(asctime)s %(levelname)s %(name)s %(message)s'.

Здесь:

|  |  |
| --- | --- |
| **asctime** | Время события |
| **levelname** | Уровень логирования (подробно об этом расскажем ниже) |
| **name** | Имя логера (журнала). По умолчанию — root, но вы можете задавать разные имена, чтобы сделать свои логи более информативными. Например, считается хорошей практикой назначать имя файла с кодом |
| **message** | Сообщение, которое вы отправили в logging.warning() |

После запуска мы увидим в файле:

2018-12-23 21:08:58,805 WARNING root 0

2018-12-23 21:08:58,805 WARNING root 1

2018-12-23 21:08:58,805 WARNING root 2

2018-12-23 21:08:58,805 WARNING root 3

2018-12-23 21:08:58,805 WARNING root 4

2018-12-23 21:08:58,806 WARNING root 5

2018-12-23 21:08:58,806 WARNING root 6

2018-12-23 21:08:58,806 WARNING root 7

2018-12-23 21:08:58,806 WARNING root 8

2018-12-23 21:08:58,806 WARNING root 9

**Уровни логирования**

Запись в логи принято разбирать на типы — уровни. Любое сообщение несет в себе информацию определенной важности, и время реакции на сообщения отличается. Приведем примеры:

|  |  |
| --- | --- |
| **Уровень** | **Пример события** |
| **Debug** | Отправлен запрос в базу данных на сохранение |
| **Debug** | Завершен запрос в базу данных на сохранение |
| **Debug** | Запрос в базу занял 0.02 секунды, извлечено 1000 записей |
| **Info** | Зарегистрирован новый пользователь (user\_id = 123123) |
| **Warning** | Отклонена транзакция с суммой платежа 0 |
| **Error** | Ошибка при сохранении данных пользователя (user\_id = 123124, operation\_id = 12312313) |
| **Critical (Fatal)** | Ошибка ответа API Яндекс Карт, код: 404. Маршруты не рассчитываются |

Начинающему программисту бывает сложно определить, к какому уровню следует отнести то или иное событие, возникающее при работе программы. Мы приведем базовые принципы, однако надо иметь в виду, что правила всегда определяет руководитель команды, которая работает над проектом.

|  |  |
| --- | --- |
| **Уровень** | **Относящиеся события** |
| **Debug** | Сообщения отладки. В боевом режиме (production) сообщения этого уровня обычно отключены, чтобы не засорять файлы журналов. Но они могут включаться для поиска багов, которые не удалось воспроизвести |
| **Info** | Обычные сообщения, информирующие о действиях программы. Реагировать на такие сообщения вообще не надо, но они могут помочь, например, при поиске багов, расследовании интересных ситуаций и т. д. |
| **Warning** | Записывая такое сообщение, программа обычно пытается привлечь внимание. Произошло что-то странное. Возможно, это новый тип ситуации, еще не известный на текущий момент. Следует разобраться в том, что произошло, что это означает, и отнести ситуацию либо к инфо-сообщению, либо к ошибке. Соответственно, придется доработать код обработки таких ситуаций |
| **Error** | Ошибка в работе программы, требующая вмешательства. Что-то не сохранилось, что-то отвалилось. Необходимо принимать меры довольно быстро! Ошибки этого уровня и выше требуют немедленной записи в лог, чтобы ускорить реакцию на них |
| **Critical (Fatal)** | Это — особый класс ошибок, приводящих к неработоспособности программы в целом или неработоспособности одного из ее модулей. Чаще всего случаются фатальные ошибки из-за неверной конфигурации или отказов оборудования. Требуют срочной, немедленной реакции |

Имейте в виду, что это — рекомендации. Вы как архитектор программы вправе определять критичность события или ошибки в своей программе, а если работа происходит в команде, то решение о критичности той или иной ситуации принимается руководителем команды или коллегиально.

Разработчик может настраивать уровень логирования. Логируются сообщения установленного уровня и уровней более высокой критичности. Например, если установлен уровень Info, то в консоль или в файл будут выводиться сообщения уровней: Info, Warning, Error и Fatal.

Уровень логирования по умолчанию — WARNING.

Уровень логирования в Python настраивается все в той же функции logging.basicConfig() следующим образом:

import logging

logging.basicConfig(level=logging.DEBUG)

def log():

logging.debug('Debug')

logging.info('Info')

logging.warning('Warning')

logging.error('Error')

logging.critical('Critical or Fatal')

if \_\_name\_\_ == '\_\_main\_\_':

log()

Видим в консоли:

DEBUG:root:Debug

INFO:root:Info

WARNING:root:Warning

ERROR:root:Error

CRITICAL:root:Critical or Fatal

Если вызвать logging.basicConfig(level=logging.ERROR), в консоли мы увидим:

ERROR:root:Error

CRITICAL:root:Critical or Fatal

Используйте логирование в процессе разработки. Это поможет вам правильно и своевременно реагировать на ошибки, искать баги и понимать, что происходит в вашей программе в любой момент времени.

**Пишем код**

Теперь переходим к самому интересному. Начнем! Мы разработаем навык, который описан в разделе [«Быстрый старт»](https://tech.yandex.ru/dialogs/alice/doc/quickstart-python-docpage/) документации Алисы.

Вы уже знакомы с библиотекой Flask. С ее помощью мы напишем небольшой веб-сервер, который будет обрабатывать запросы от Алисы.

Алиса будет передавать нам JSON, содержащий данные о пользователе, и информацию, которую пользователь ввел. Мы же должны будем вернуть в ответ свой JSON (соответственно документации). Алиса его обработает и покажет пользователю.

Рассмотрим JSON запроса, который отправит нам Алиса. На самом деле JSON выглядит сложнее и имеет больше полей, но мы рассмотрим только те из них, которые будем использовать (внимательно изучить содержание JSON’ов можно в [документации](https://yandex.ru/dev/dialogs/alice/doc/protocol-docpage/)):

{

"request": {

"command": "закажи пиццу на улицу льва толстого, 16 на завтра",

"original\_utterance": "закажи пиццу на улицу льва толстого, 16 на завтра"

},

"session": {

"new": true,

"message\_id": 4,

"session\_id": "2eac4854-fce721f3-b845abba-20d60",

"skill\_id": "3ad36498-f5rd-4079-a14b-788652932056",

"user\_id": "AC9WC3DF6FCE052E45A4566A48E6B7193774B84814CE49A922E163B8B29881DC"

},

"version": "1.0"

}

* *request* — данные, полученные от пользователя
  + *command* — запрос, который был передан вместе с командой активации навыка. Например, если пользователь активирует навык словами «спроси у Сбербанка, где ближайшее отделение», в этом поле будет передана строка «где ближайшее отделение»
  + *original\_utterance* — полный текст пользовательского запроса, максимум 1024 символа
* *session* — данные о сессии (разговоре с Алисой)
  + *new* — признак новой сессии. Возможные значения: true — пользователь начинает новый разговор с навыком, false — запрос отправлен в рамках уже начатого разговора
  + *message\_id* — идентификатор сообщения в рамках сессии, максимум 8 символов. Инкрементируется (увеличивается на единицу) с каждым следующим запросом
  + *session\_id* — уникальный идентификатор сессии, максимум 64 символа
  + *skill\_id* — идентификатор вызываемого навыка, присвоенный при создании
  + *user\_id* — идентификатор экземпляра приложения, в котором пользователь общается с Алисой, максимум 64 символа. Даже если пользователь авторизован с одним и тем же аккаунтом в приложении Яндекс для Android и iOS, Яндекс.Диалоги (так называется сервис Яндекса, управляющий навыками для Алисы) присвоят отдельный *user\_id* каждому из этих приложений
* *version* — версия протокола. Текущая версия — 1.0

JSON ответа Алисы (то есть тот JSON, что сформирует наша программа) будет выглядеть так (конечно же, полное описание можно посмотреть в [документации](https://tech.yandex.ru/dialogs/alice/doc/protocol-docpage/)):

{

"response": {

"text": "Здравствуйте! Это мы, хороводоведы.",

"tts": "Здравствуйте! Это мы, хоров+одо в+еды.",

"buttons": [

{

"title": "Надпись на кнопке",

"payload": {},

"url": "https://example.com/",

"hide": true

}

],

"end\_session": false

},

"session": {

"session\_id": "2eac4854-fce721f3-b845abba-20d60",

"message\_id": 4,

"user\_id": "AC9WC3DF6FCE052E45A4566A48E6B7193774B84814CE49A922E163B8B29881DC"

},

"version": "1.0"

}

* *response* — данные для ответа пользователю
  + *text* — текст, который следует показать. Максимум 1024 символа. Не должен быть пустым
  + *tts* — текст, который следует сказать пользователю. Максимум 1024 символа. Не должен быть пустым
  + *buttons* — массив кнопок, которые следует показать пользователю. Все указанные кнопки выводятся после основного ответа Алисы, описанного в свойстве *text*. Кнопки можно использовать как релевантные ответу ссылки или подсказки для продолжения разговора
  + *title* — текст кнопки, обязателен для каждой кнопки. Максимум 64 символа
  + *url* — URL, который должна открывать кнопка, максимум 1024 байта
  + *payload* — значения, которые передадутся в программу после нажатия этой кнопки
  + *hide* — признак того, что кнопку нужно убрать после следующей реплики пользователя. Допустимые значения: false — кнопка должна оставаться активной (значение по умолчанию), true — кнопку нужно скрывать после нажатия
  + *end\_session* — признак конца разговора. Допустимые значения: false — сессию следует продолжить, true — сессию следует завершить
* *session* — данные о сессии (аналогичные запросу)
* *version* — версия протокола. Текущая версия — 1.0

Вы можете скачать [файл с шаблоном кода](https://yastatic.net/s3/lyceum/content/resources/flask_app.py) или скопировать текст программы ниже.

# импортируем библиотеки

from flask import Flask, request

import logging

# библиотека, которая нам понадобится для работы с JSON

import json

# создаём приложение

# мы передаём \_\_name\_\_, в нем содержится информация,

# в каком модуле мы находимся.

# В данном случае там содержится '\_\_main\_\_',

# так как мы обращаемся к переменной из запущенного модуля.

# если бы такое обращение, например,

# произошло внутри модуля logging, то мы бы получили 'logging'

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

# Устанавливаем уровень логирования

logging.basicConfig(level=logging.INFO)

# Создадим словарь, чтобы для каждой сессии общения

# с навыком хранились подсказки, которые видел пользователь.

# Это поможет нам немного разнообразить подсказки ответов

# (buttons в JSON ответа).

# Когда новый пользователь напишет нашему навыку,

# то мы сохраним в этот словарь запись формата

# sessionStorage[user\_id] = {'suggests': ["Не хочу.", "Не буду.", "Отстань!" ]}

# Такая запись говорит, что мы показали пользователю эти три подсказки.

# Когда он откажется купить слона,

# то мы уберем одну подсказку. Как будто что-то меняется :)

sessionStorage = {}

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

# Функция получает тело запроса и возвращает ответ.

# Внутри функции доступен request.json - это JSON,

# который отправила нам Алиса в запросе POST

def main():

logging.info(f'Request: {request.json!r}')

# Начинаем формировать ответ, согласно документации

# мы собираем словарь, который потом при помощи

# библиотеки json преобразуем в JSON и отдадим Алисе

response = {

'session': request.json['session'],

'version': request.json['version'],

'response': {

'end\_session': False

}

}

# Отправляем request.json и response в функцию handle\_dialog.

# Она сформирует оставшиеся поля JSON, которые отвечают

# непосредственно за ведение диалога

handle\_dialog(request.json, response)

logging.info(f'Response: {response!r}')

# Преобразовываем в JSON и возвращаем

return json.dumps(response)

def handle\_dialog(req, res):

user\_id = req['session']['user\_id']

if req['session']['new']:

# Это новый пользователь.

# Инициализируем сессию и поприветствуем его.

# Запишем подсказки, которые мы ему покажем в первый раз

sessionStorage[user\_id] = {

'suggests': [

"Не хочу.",

"Не буду.",

"Отстань!",

]

}

# Заполняем текст ответа

res['response']['text'] = 'Привет! Купи слона!'

# Получим подсказки

res['response']['buttons'] = get\_suggests(user\_id)

return

# Сюда дойдем только, если пользователь не новый,

# и разговор с Алисой уже был начат

# Обрабатываем ответ пользователя.

# В req['request']['original\_utterance'] лежит весь текст,

# что нам прислал пользователь

# Если он написал 'ладно', 'куплю', 'покупаю', 'хорошо',

# то мы считаем, что пользователь согласился.

# Подумайте, всё ли в этом фрагменте написано "красиво"?

if req['request']['original\_utterance'].lower() in [

'ладно',

'куплю',

'покупаю',

'хорошо'

]:

# Пользователь согласился, прощаемся.

res['response']['text'] = 'Слона можно найти на Яндекс.Маркете!'

res['response']['end\_session'] = True

return

# Если нет, то убеждаем его купить слона!

res['response']['text'] = \

f"Все говорят '{req['request']['original\_utterance']}', а ты купи слона!"

res['response']['buttons'] = get\_suggests(user\_id)

# Функция возвращает две подсказки для ответа.

def get\_suggests(user\_id):

session = sessionStorage[user\_id]

# Выбираем две первые подсказки из массива.

suggests = [

{'title': suggest, 'hide': True}

for suggest in session['suggests'][:2]

]

# Убираем первую подсказку, чтобы подсказки менялись каждый раз.

session['suggests'] = session['suggests'][1:]

sessionStorage[user\_id] = session

# Если осталась только одна подсказка, предлагаем подсказку

# со ссылкой на Яндекс.Маркет.

if len(suggests) < 2:

suggests.append({

"title": "Ладно",

"url": "https://market.yandex.ru/search?text=слон",

"hide": True

})

return suggests

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

Сохраните ваш файл под именем flask\_app.py.

**Как работает данный код**

Разберем тезисно, как работает приведенная программа.

1. Когда мы запускаем программу, точкой входа в нее со стороны Алисы является функция main(), которая «обернута» декоратором *app.route*.
2. Внутри main() мы сначала логируем полученный запрос, затем начинаем формировать ответ, потом вызываем функцию handle\_dialog() для обработки диалога с пользователем, в результате же логируем и возвращаем ответ.
3. Функция handle\_dialog() формирует ответ, исходя из данных запроса и состояния сессии (новый разговор или продолжение старого).
4. Вспомогательная функция get\_suggests() формирует подсказки в ответе.
5. В глобальном словаре *sessionStorage* мы будем хранить информацию о подсказках для каждого пользователя.

**Отладка с Postman**

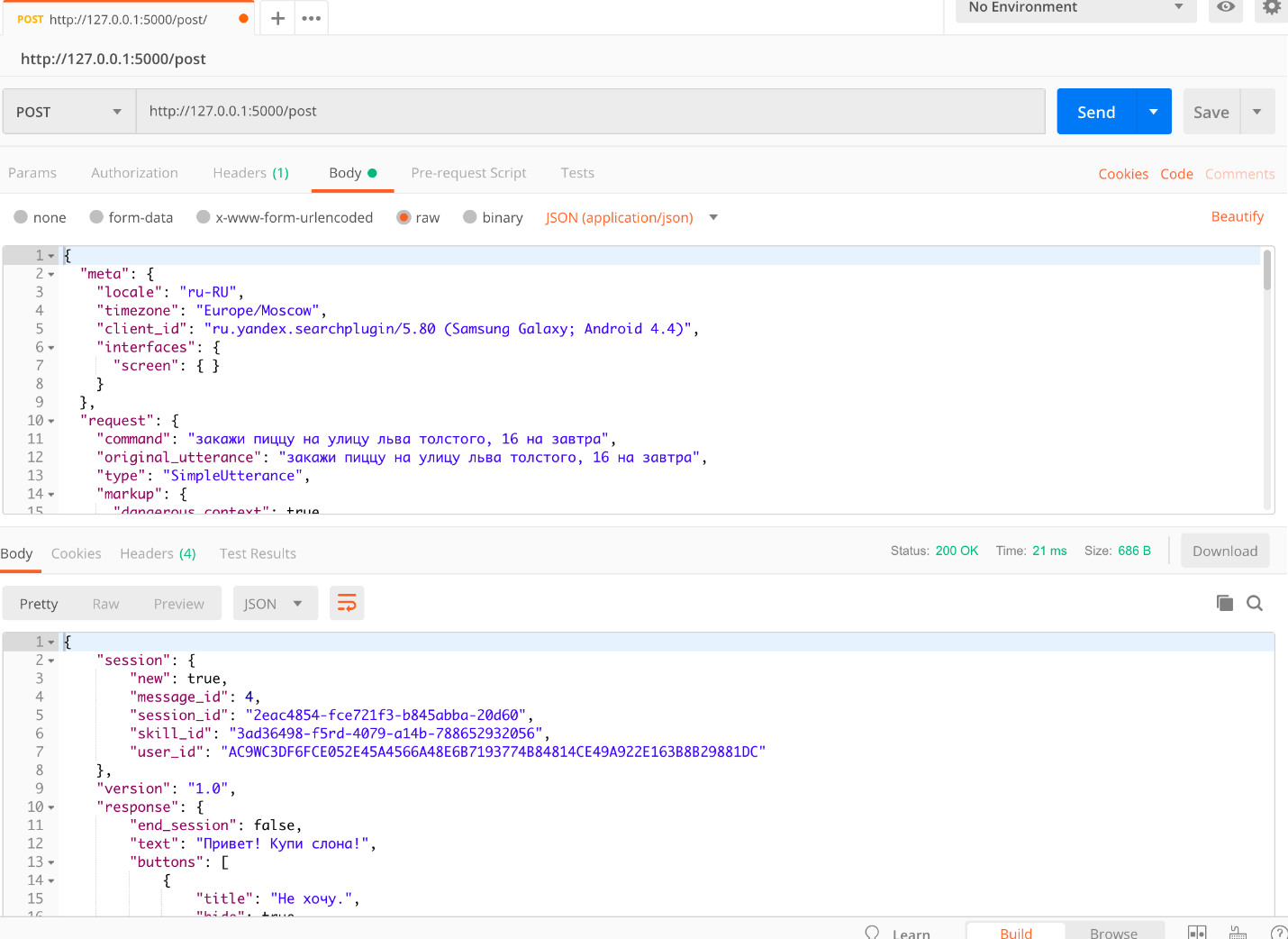
Перед тем, как продолжить, нам надо убедиться, что созданный нами веб-сервер возвращает правильный ответ (или похожий на правильный). Воспользуемся для этого программой [Postman](https://www.getpostman.com/).

Postman — это приложение, которое позволяет взаимодействовать с API без написания кода. Это бывает удобно, чтобы находить ошибки и «пробовать» новое API перед разработкой. Postman не единственное подобное приложение, можете попробовать еще, например, [Insomnia](https://insomnia.rest/).

Сейчас мы проверим ответ от нашего приложения.

Запустите созданное ранее приложение в среде разработки и скопируйте адрес вида http://127.0.0.1:5000/post в строку запроса программы Postman.

JSON запроса можно взять из [документации](https://tech.yandex.ru/dialogs/alice/doc/protocol-docpage/) Алисы или, на крайний случай, из рассмотренного нами примера.



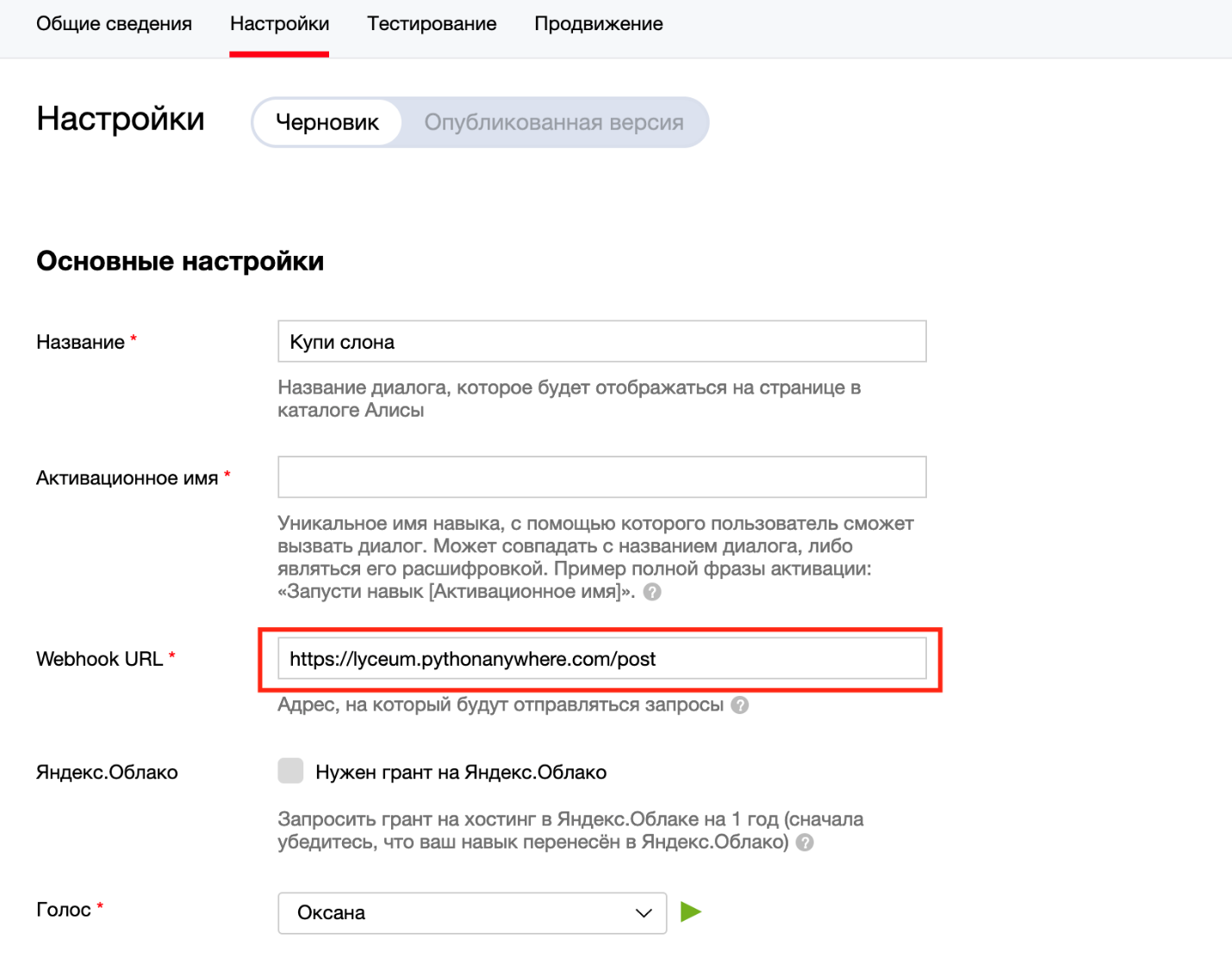
Проверьте, что ваш ответ похож на тот, что требует документация Алисы. Если это так, то поздравляем, мы добились результата :)

Для следующего шага нам понадобится, чтобы наш навык был доступен из Интернета, поэтому давайте либо задеплоим (разместим) его на Heroku, либо просто сделаем туннель с помощью ngrok.

**Регистрация и тестирование навыка**

Перейдем к завершающему и самому простому пункту. Это регистрация навыка в Алисе.

1. Зарегистрируйтесь в Яндексе или залогиньтесь, если вы уже зарегистрированы.
2. Перейдите по ссылке <https://dialogs.yandex.ru/developer/>.
3. Нажмите **создать диалог**.
4. Выберите вариант **Навык в Алисе**.
5. Заполните «активационное имя». По этой фразе Алиса будет вызывать ваш навык.
6. Заполните поля, указав ссылку в поле Webhook URL.



Перейдите на вкладку **Тестирование** и проверьте работу своего навыка.

**Именованные сущности**

Часто при создании навыка из ответа пользователя необходимо извлечь некоторую информацию, и не всегда это делается просто. Но разработчики Алисы уже позаботились о нас и сами выделяют некоторые такие сущности, которые в терминах Алисы называются **именованными**.

Под именованными сущностями подразумеваются:

* Имена
* Фамилии
* Названия городов и т. д.

Когда пользователь в своем сообщении использует имя, фамилию или город, эти данные попадают в специальный раздел JSON’а, который отправляет нам Алиса.

Рассмотрим пример:

{

"meta": {

"locale": "ru-RU",

"timezone": "Europe/Moscow",

"client\_id": "ru.yandex.searchplugin/5.80 (Samsung Galaxy; Android 4.4)",

"interfaces": {

"screen": { }

}

},

"request": {

"command": "закажи пиццу на улицу льва толстого, 16 на завтра",

"original\_utterance": "закажи пиццу на улицу льва толстого, 16 на завтра",

"type": "SimpleUtterance",

"markup": {

"dangerous\_context": true

},

"payload": {},

"nlu": {

"tokens": [

"закажи",

"пиццу",

"на",

"льва",

"толстого",

"16",

"на",

"завтра"

],

"entities": [

{

"tokens": {

"start": 2,

"end": 6

},

"type": "YANDEX.GEO",

"value": {

"house\_number": "16",

"street": "льва толстого"

}

},

{

"tokens": {

"start": 3,

"end": 5

},

"type": "YANDEX.FIO",

"value": {

"first\_name": "лев",

"last\_name": "толстой"

}

},

{

"tokens": {

"start": 5,

"end": 6

},

"type": "YANDEX.NUMBER",

"value": 16

},

{

"tokens": {

"start": 6,

"end": 8

},

"type": "YANDEX.DATETIME",

"value": {

"day": 1,

"day\_is\_relative": true

}

}

]

}

},

"session": {

"new": true,

"message\_id": 4,

"session\_id": "2eac4854-fce721f3-b845abba-20d60",

"skill\_id": "3ad36498-f5rd-4079-a14b-788652932056",

"user\_id": "AC9WC3DF6FCE052E45A4566A48E6B7193774B84814CE49A922E163B8B29881DC"

},

"version": "1.0"

}

Обратите внимание на раздел request → nlu → tokens:

"tokens": [

"закажи",

"пиццу",

"на",

"льва",

"толстого",

"16",

"на",

"завтра"

]

Можно заметить, что это полученный от пользователя текст, разобранный на отдельные слова. Каждое слово при этом приводится к нижнему регистру. Это очень удобно, если надо искать вхождения каких-то определенных слов. Например, «да» или «нет».

Второй раздел, на который стоит обратить внимание — это entities (то есть **сущности**):

"entities": [

{

"tokens": {

"start": 2,

"end": 6

},

"type": "YANDEX.GEO",

"value": {

"house\_number": "16",

"street": "льва толстого"

}

},

{

"tokens": {

"start": 3,

"end": 5

},

"type": "YANDEX.FIO",

"value": {

"first\_name": "лев",

"last\_name": "толстой"

}

},

{

"tokens": {

"start": 5,

"end": 6

},

"type": "YANDEX.NUMBER",

"value": 16

},

{

"tokens": {

"start": 6,

"end": 8

},

"type": "YANDEX.DATETIME",

"value": {

"day": 1,

"day\_is\_relative": true

}

}

]

Мы видим, что текст от пользователя подробно разобран и из него выделены все сущности от имени до даты.

entities — это массив таких сущностей. Рассмотрим одну сущность и подробно распишем, из чего же она состоит:

{

"tokens": {

"start": 3,

"end": 5

},

"type": "YANDEX.FIO",

"value": {

"first\_name": "лев",

"last\_name": "толстой"

}

}

* *tokens* — обозначение начала и конца именованной сущности в массиве слов (*tokens*, что мы рассматривали выше). Нумерация слов в массиве начинается с 0
  + *start* — первое слово именованной сущности
  + *end* — последнее слово после именованной сущности
* *type* — тип именованной сущности. Возможные значения:
  + YANDEX.DATETIME — дата и время
  + YANDEX.FIO — фамилия, имя и отчество
  + YANDEX.GEO — местоположение (адрес или аэропорт)
  + YANDEX.NUMBER — число, целое или с плавающей точкой
* *value* — формальное описание именованной сущности (мы приведем ниже пример для типа YANDEX.FIO. Описание остальных типов ищите в [документации](https://yandex.ru/dev/dialogs/alice/doc/naming-entities-docpage/))
  + first\_name — имя
  + last\_name — фамилия

Пример кода, получающий имя человека из JSON:

def get\_first\_name(req):

# перебираем сущности

for entity in req['request']['nlu']['entities']:

# находим сущность с типом 'YANDEX.FIO'

if entity['type'] == 'YANDEX.FIO':

# Если есть сущность с ключом 'first\_name',

# возвращаем ее значение.

# Во всех остальных случаях возвращаем None.

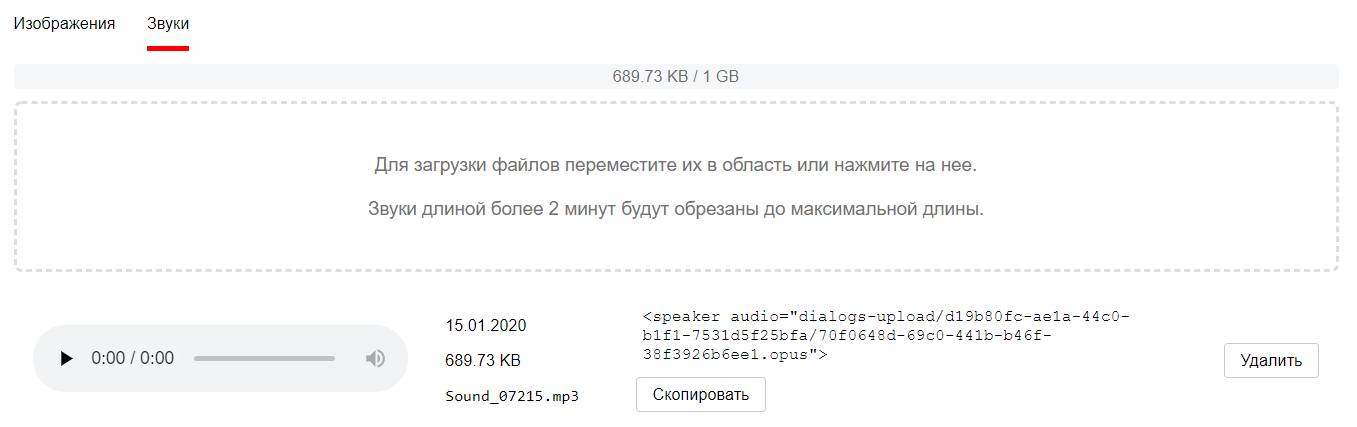
return entity['value'].get('first\_name', None)

**Ресурсы**

Для каждого навыка Алиса позволяет размещать до 100 MB дополнительных медиафайлов (изображения и звуки), но с некоторыми ограничениями:

* Изображение должно быть размером от 1 KB до 1 MB
* Звуковой файл должен быть не длиннее 2-х минут

Ресурсы можно добавить на вкладке «Ресурсы» в разделе управления навыком. Как работать с изображениями мы рассмотрим на следующем уроке, а пока давайте добавим какой-нибудь звук.

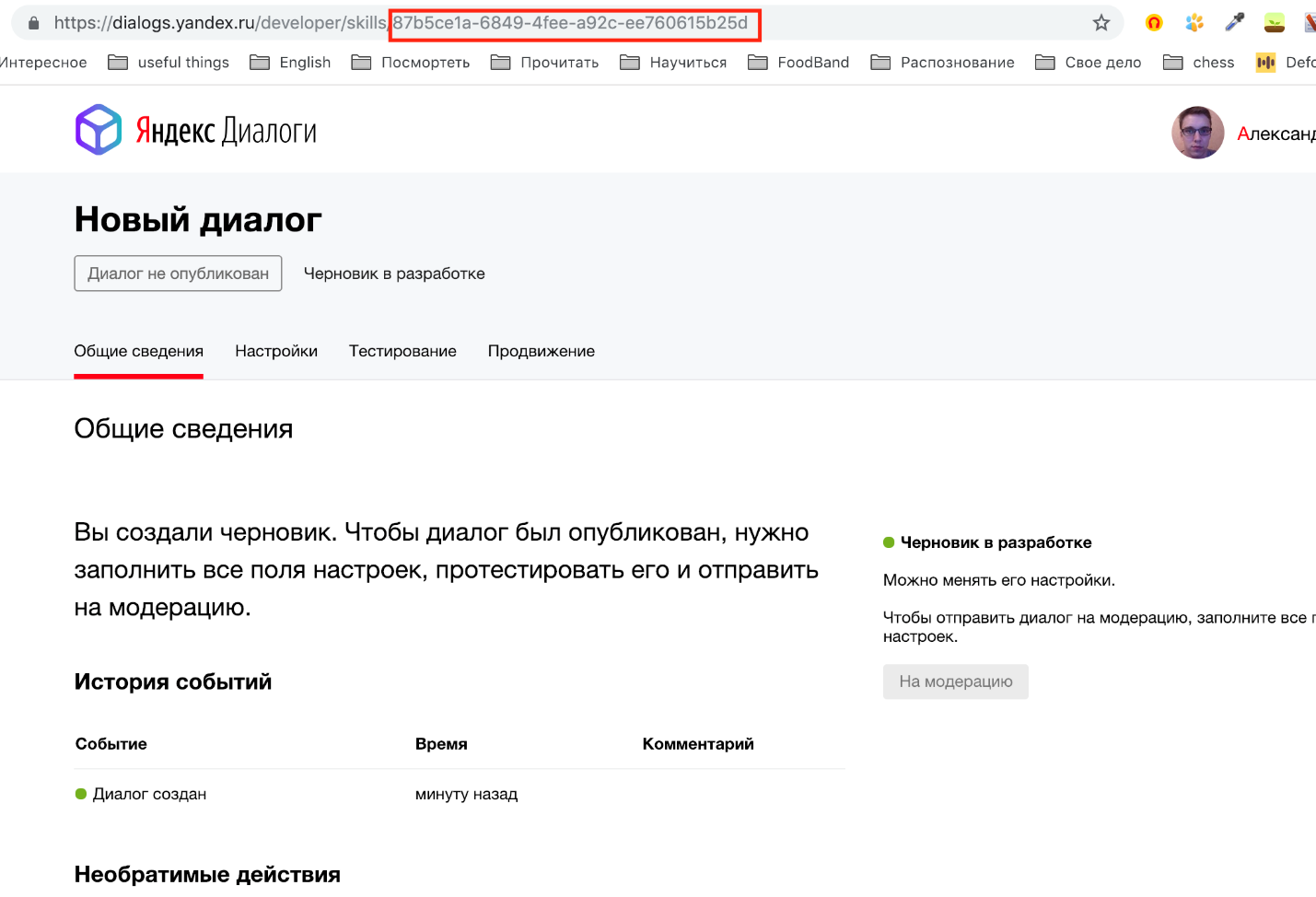


После загрузки звукового файла добавится код его вставки в ответ. Чтобы добавить проигрывание звукового файла, надо добавить этот код в поле *tts* ответа.

Загружать ресурсы к навыку можно не только через веб-интерфейс, но и с помощью [HTTP API Алисы](https://tech.yandex.ru/dialogs/alice/doc/resource-upload-docpage/). Можно написать код с использованием библиотеки requests или воспользоваться Postman.

Идем по пунктам:

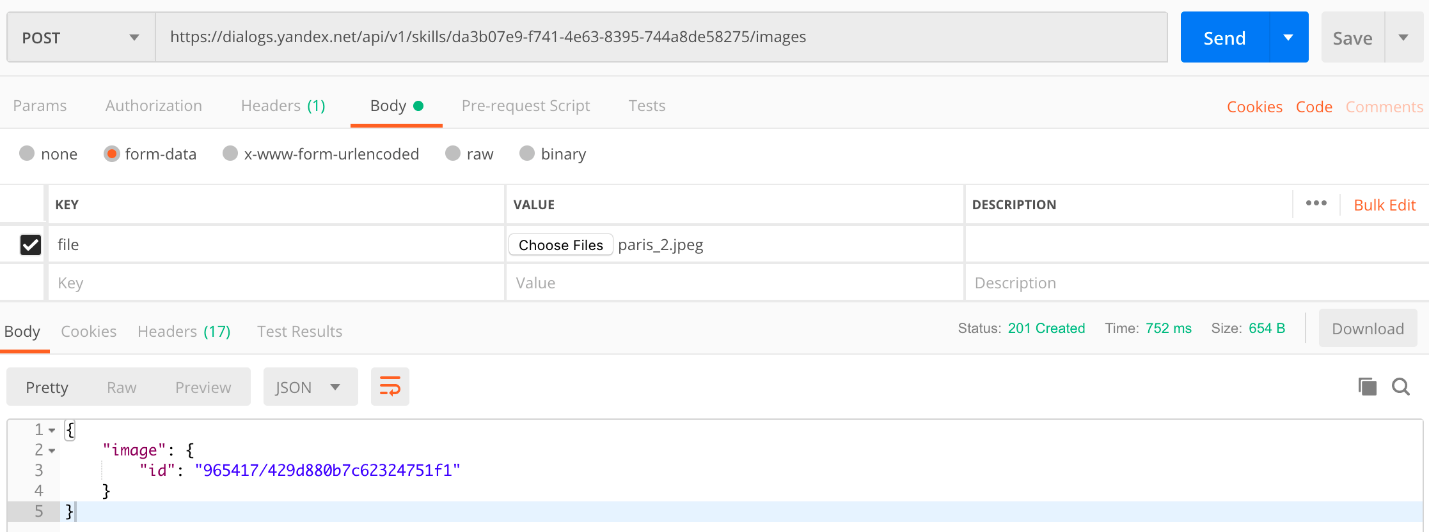
1. Создадим новый навык. Зайдем на [платформу диалогов](https://dialogs.yandex.ru/developer/) и нажмем **создать новый навык**
2. Скопируем из URL и сохраним его идентификатор (ID):



1. Получим token по [ссылке](https://oauth.yandex.ru/authorize?response_type=token&client_id=c473ca268cd749d3a8371351a8f2bcbd). Сохраните его тоже. Он вам пригодится
2. Теперь запускаем Postman
   * Заполняем поля на вкладке Headers
   * Добавьте параметр с названием Authorization. Для этого в колонке KEY пишем Authorization, а в колонке VALUE — «OAuth Token» (OAuth пробел token, который вы получили)
   * В поле ссылки добавляем ссылку https://dialogs.yandex.net/api/v1/skills/id\_навыка/images (заменить id\_навыка на id из пункта 2) для изображения или https://dialogs.yandex.net/api/v1/skills/id\_навыка/sounds для звукового файла



1. Теперь на вкладке Body выбираем радио-кнопку form-data и добавляем файл. В колонке key напишите «file», в колонке value нажмите «choose file» и выберите файл с вашего компьютера. Далее нажимаем Send.



Если все сделано правильно, в ответе вы получите идентификатор картинки или звукового файла, который вы сможете использовать в дальнейшем.

Основы промышленного программирования | Д20Кольцово - Технополис - 1 - Д2

[Задать вопрос](https://connect.yandex.ru/forms/question-form.staff-id/)

[[](https://passport.yandex.ru/?)](https://passport.yandex.ru?" \t "_parent)

[Маракасова Юлия](https://passport.yandex.ru?" \t "_parent)

* [Обзор курса](https://lyceum.yandex.ru/courses/352/groups/2321)
* [Решения](https://lyceum.yandex.ru/courses/352/groups/2321/solutions)
* [Расписание](https://lyceum.yandex.ru/courses/352/groups/2321/schedule)
* [Успеваемость](https://lyceum.yandex.ru/courses/352/groups/2321/progress)

[Урок Алиса 2](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057)

**Алиса — урок 2**

1. [Введение](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#1)
2. [Картинка в ответе](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#2)
3. [Навык, который знакомится](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#3)
4. [Игра](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#4)
5. [Пользуемся API Яндекс.Карт](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#5)
6. [Получаем координаты города](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#6)
7. [Получаем страну города](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#7)
8. [Рассчитываем расстояние от города до города](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#8)
9. [Программируем навык](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#9)
10. [Функции Яндекс.Облака](https://lyceum.yandex.ru/courses/352/groups/2321/lessons/2057/materials/4832#10)

**Аннотация**

Сегодня мы продолжим разрабатывать навыки для Алисы: научим ее знакомиться с пользователем и взаимодействовать с внешними API. В качестве бонуса расскажем, как разместить свой навык в Яндекс.Облаке.

**Введение**

На этом уроке мы напишем игру «Угадай город». Алиса показывает пользователю фотографию города, а он должен его отгадать.

Важный момент: перед игрой пользователь должен представиться, а Алиса — его поприветствовать. Это добавит нашему навыку немного человечности.

Находить именованные сущности (город и имя) мы уже умеем, осталось научиться отправлять пользователю картинку.

Итак, приступим.

**Картинка в ответе**

Научимся показывать картинку в ответе. Например, для нашей игры нужно будет демонстрировать картинку города.

Для начала рассмотрим JSON ответа Алисы, в котором прикреплена картинка:

{

"response": {

"text": "Здравствуйте! Это мы, хороводоведы.",

"tts": "Здравствуйте! Это мы, хоров+одо в+еды.",

"card": {

"type": "BigImage",

"image\_id": "1027858/46r960da47f60207e924",

"title": "Заголовок для изображения",

"description": "Описание изображения.",

"button": {

"text": "Надпись на кнопке",

"url": "http://example.com/",

"payload": {}

}

},

"buttons": [

{

"title": "Надпись на кнопке",

"payload": {},

"url": "https://example.com/",

"hide": true

}

],

"end\_session": false

},

"session": {

"session\_id": "2eac4854-fce721f3-b845abba-20d60",

"message\_id": 4,

"user\_id": "AC9WC3DF6FCE052E45A4566A48E6B7193774B84814CE49A922E163B8B29881DC"

},

"version": "1.0"

}

В отличие от примера из предыдущего урока, в ответе появился раздел *card*. Он и содержит в себе картинку.

Рассмотрим подробно его содержание:

"card": {

"type": "BigImage",

"image\_id": "1027858/46r960da47f60207e924",

"title": "Заголовок для изображения",

"description": "Описание изображения.",

"button": {

"text": "Надпись на кнопке",

"url": "http://example.com/",

"payload": {}

}

Мы опишем только те поля, которые будем использовать. Про оставшиеся можно посмотреть в [документации Алисы](https://tech.yandex.ru/dialogs/alice/doc/protocol-docpage/#response).

*card* — описание карточки-сообщения с поддержкой изображений. Если приложению удается отобразить карточку для пользователя, свойство *response.text* не используется

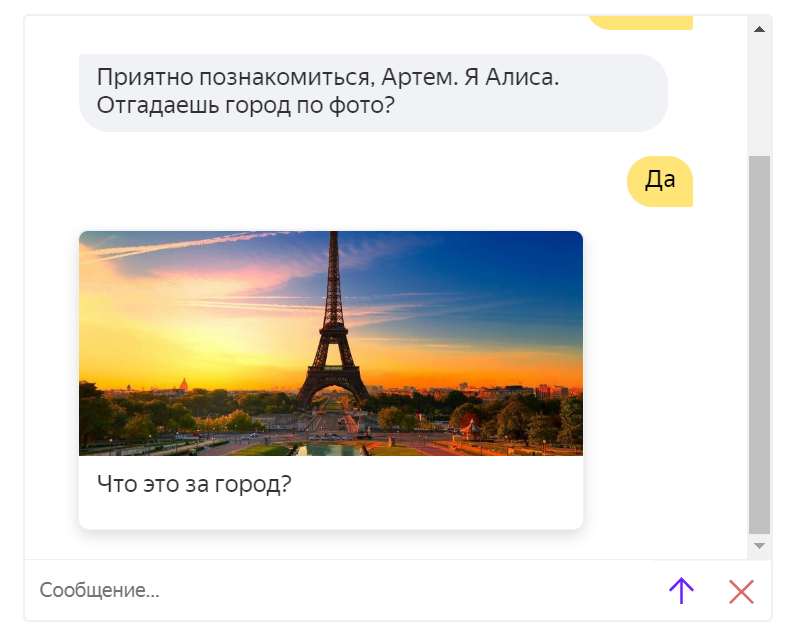
* *type* — тип карточки. Поддерживаемые значения:
  + *BigImage* — одно изображение
  + *ItemsList* — галерея изображений (от 1 до 5)

Требуемый формат ответа зависит от типа карточки. Мы будем использовать BigImage

* *image\_id* — идентификатор изображения, который возвращается в ответ на запрос загрузки
* *title* — заголовок для изображения. Это на самом деле увидит пользователь

В этом [архиве](https://yastatic.net/s3/lyceum/content/resources/cities.zip) мы собрали для вас картинки, которые можно использовать для решения задач урока (но можете использовать свои изображения). Подробно про то, как загружать ресурсы к своему навыку, мы говорили на прошлом уроке.

Вот как может выглядеть ответ Алисы с картинкой:



**Навык, который знакомится**

Напишем простой навык, который ведет следующий диалог:

* Привет! Назови свое имя!
* Саша.
* Приятно познакомиться, Саша. Я — Алиса. Какой город хочешь увидеть?
* Москва (Нью-йорк, Париж).
* Этот город я знаю. *И показывает фото города*

Если вдруг пользователь не назвал имя, Алиса должна сказать:

* Не расслышала имя. Повтори, пожалуйста!

Если пользователь не назвал город или назвал город, для которого нет картинки, Алиса должна сказать:

* Первый раз слышу об этом городе. Попробуй еще разок!

Код приведен ниже:

from flask import Flask, request

import logging

import json

import random

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

logging.basicConfig(level=logging.INFO)

# создаем словарь, в котором ключ — название города,

# а значение — массив, где перечислены id картинок,

# которые мы записали в прошлом пункте.

cities = {

'москва': ['1540737/daa6e420d33102bf6947',

'213044/7df73ae4cc715175059e'],

'нью-йорк': ['1652229/728d5c86707054d4745f',

'1030494/aca7ed7acefde2606bdc'],

'париж': ["1652229/f77136c2364eb90a3ea8",

'3450494/aca7ed7acefde22341bdc']

}

# создаем словарь, где для каждого пользователя

# мы будем хранить его имя

sessionStorage = {}

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

def main():

logging.info(f'Request: {request.json!r}')

response = {

'session': request.json['session'],

'version': request.json['version'],

'response': {

'end\_session': False

}

}

handle\_dialog(response, request.json)

logging.info(f'Response: {response!r}')

return json.dumps(response)

def handle\_dialog(res, req):

user\_id = req['session']['user\_id']

# если пользователь новый, то просим его представиться.

if req['session']['new']:

res['response']['text'] = 'Привет! Назови свое имя!'

# создаем словарь в который в будущем положим имя пользователя

sessionStorage[user\_id] = {

'first\_name': None

}

return

# если пользователь не новый, то попадаем сюда.

# если поле имени пустое, то это говорит о том,

# что пользователь еще не представился.

if sessionStorage[user\_id]['first\_name'] is None:

# в последнем его сообщение ищем имя.

first\_name = get\_first\_name(req)

# если не нашли, то сообщаем пользователю что не расслышали.

if first\_name is None:

res['response']['text'] = \

'Не расслышала имя. Повтори, пожалуйста!'

# если нашли, то приветствуем пользователя.

# И спрашиваем какой город он хочет увидеть.

else:

sessionStorage[user\_id]['first\_name'] = first\_name

res['response'][

'text'] = 'Приятно познакомиться, ' \

+ first\_name.title() \

+ '. Я - Алиса. Какой город хочешь увидеть?'

# получаем варианты buttons из ключей нашего словаря cities

res['response']['buttons'] = [

{

'title': city.title(),

'hide': True

} for city in cities

]

# если мы знакомы с пользователем и он нам что-то написал,

# то это говорит о том, что он уже говорит о городе,

# что хочет увидеть.

else:

# ищем город в сообщение от пользователя

city = get\_city(req)

# если этот город среди известных нам,

# то показываем его (выбираем одну из двух картинок случайно)

if city in cities:

res['response']['card'] = {}

res['response']['card']['type'] = 'BigImage'

res['response']['card']['title'] = 'Этот город я знаю.'

res['response']['card']['image\_id'] = random.choice(cities[city])

res['response']['text'] = 'Я угадал!'

# если не нашел, то отвечает пользователю

# 'Первый раз слышу об этом городе.'

else:

res['response']['text'] = \

'Первый раз слышу об этом городе. Попробуй еще разок!'

def get\_city(req):

# перебираем именованные сущности

for entity in req['request']['nlu']['entities']:

# если тип YANDEX.GEO то пытаемся получить город(city),

# если нет, то возвращаем None

if entity['type'] == 'YANDEX.GEO':

# возвращаем None, если не нашли сущности с типом YANDEX.GEO

return entity['value'].get('city', None)

def get\_first\_name(req):

# перебираем сущности

for entity in req['request']['nlu']['entities']:

# находим сущность с типом 'YANDEX.FIO'

if entity['type'] == 'YANDEX.FIO':

# Если есть сущность с ключом 'first\_name',

# то возвращаем ее значение.

# Во всех остальных случаях возвращаем None.

return entity['value'].get('first\_name', None)

if \_\_name\_\_ == '\_\_main\_\_':

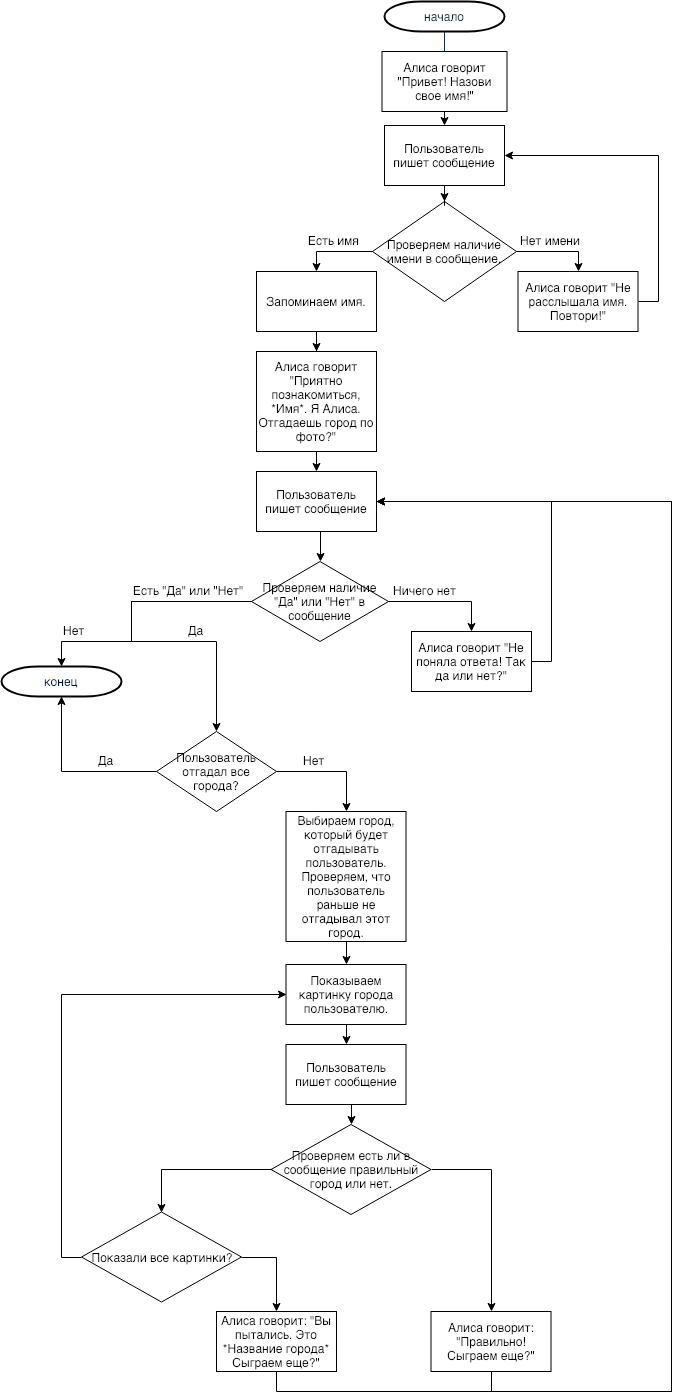
app.run()

Загружаем этот код в Heroku или делаем туннель в ngrok, заполняем данные в Алисе и отправляемся тестировать!

**Игра**

Доработаем предыдущую программу до конца.

Рассмотрим блок-схему, чтобы понять логику общения нашего навыка с пользователем.



Код игры приведен в [файле](https://yastatic.net/s3/lyceum/content/resources/alice-2-game.py).

**Пользуемся API Яндекс.Карт**

Сегодня мы попробуем совместить наш опыт по работе со сторонними API (Яндекс.Карты) и знания Алисы, чтобы создать новый навык, который будет сообщать пользователю, в какой стране находится загаданный город, а также вычислять расстояние от одного города до другого.

Алгоритм работы навыка следующий:

* Если мы пишем название одного города, Алиса сообщит нам, в какой стране находится этот город
* Если мы пишем названия двух городов, Алиса посчитает расстояние между ними
* Если мы вдруг напишем названия трех и более городов, Алиса возмутится и сообщит: «Слишком много городов. Я запуталась»

Для получения информации о географических объектах мы воспользуемся [геокодером Яндекс.Карт](https://tech.yandex.ru/maps/geocoder/).

**Получаем координаты города**

Мы уже знаем, что Алиса умеет вычленять из текста разные сущности, в том числе и названия городов.

Напишем функцию get\_coordinates(city\_name), которая получает географические координаты города по его имени.

Напомним: надо отправить **HTTP-запрос**, например, такой https://geocode-maps.yandex.ru/1.x/?geocode=Москва&format=json Яндекс.Картам, а потом разобрать ответ.

Возвращает эта функция кортеж с координатами города. В случае, если в процессе работы произошла **любая ошибка**, мы вернем исключение.

import requests

def get\_coordinates(city\_name):

try:

# url, по которому доступно API Яндекс.Карт

url = "https://geocode-maps.yandex.ru/1.x/"

# параметры запроса

params = {

"apikey": "40d1649f-0493-4b70-98ba-98533de7710b",

# город, координаты которого мы ищем

'geocode': city\_name,

# формат ответа от сервера, в данном случае JSON

'format': 'json'

}

# отправляем запрос

response = requests.get(url, params)

# получаем JSON ответа

json = response.json()

# получаем координаты города

# (там написаны долгота(longitude), широта(latitude) через пробел)

# посмотреть подробное описание JSON-ответа можно

# в документации по адресу https://tech.yandex.ru/maps/geocoder/

coordinates\_str = json['response']['GeoObjectCollection'][

'featureMember'][0]['GeoObject']['Point']['pos']

# Превращаем string в список, так как

# точка - это пара двух чисел - координат

long, lat = map(float, coordinates\_str.split())

# Вернем ответ

return long, lat

except Exception as e:

return e

**Получаем страну города**

Функция get\_country(city\_name) вернет нам страну, в которой находится указанный город. Отличие от предыдущей функции заключается лишь в получении других данных из ответа геокодера.

def get\_country(city\_name):

try:

url = "https://geocode-maps.yandex.ru/1.x/"

params = {

"apikey": "40d1649f-0493-4b70-98ba-98533de7710b",

'geocode': city\_name,

'format': 'json'

}

data = requests.get(url, params).json()

# все отличие тут, мы получаем имя страны

return data['response']['GeoObjectCollection'][

'featureMember'][0]['GeoObject']['metaDataProperty'][

'GeocoderMetaData']['AddressDetails']['Country']['CountryName']

except Exception as e:

return e

**Рассчитываем расстояние от города до города**

А вот для вычисления расстояний между двумя точками необходимы знания **тригонометрических** функций. Ведь Земля — круглая!

Подробно о расчете коротких расстояний на Земле можно прочитать [тут](https://en.wikipedia.org/wiki/Haversine_formula). Пока же можно просто воспользоваться приведенной функцией.

import math

def get\_distance(p1, p2):

# p1 и p2 - это кортежи из двух элементов - координаты точек

radius = 6373.0

lon1 = math.radians(p1[0])

lat1 = math.radians(p1[1])

lon2 = math.radians(p2[0])

lat2 = math.radians(p2[1])

d\_lon = lon2 - lon1

d\_lat = lat2 - lat1

a = math.sin(d\_lat / 2) \*\* 2 + math.cos(lat1) \* math.cos(lat2) \* math.sin(d\_lon / 2) \*\* 2

c = 2 \* math.atan2(a \*\* 0.5, (1 - a) \*\* 0.5)

distance = radius \* c

return distance

**Программируем навык**

Наша программа будет состоять из двух файлов.

В первом файле мы разместим код, который будет отвечать за общение с Алисой, а во втором — функции, которые связаны с общением с API Яндекс.Карт.

Всю функциональность общения с картами мы уже реализовали, поэтому осталось объединить все функции в один файл — [geo.py](https://yastatic.net/s3/lyceum/content/resources/geo.py).

А во втором файле мы расположим код, который отвечает за общение с Алисой — [app.py](https://yastatic.net/s3/lyceum/content/resources/app.py).

Многое из того, что мы делаем тут, уже было сделано в других уроках. Так что объясним только новые моменты.

from flask import Flask, request

import logging

import json

# импортируем функции из нашего второго файла geo

from geo import get\_country, get\_distance, get\_coordinates

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

# Добавляем логирование в файл.

# Чтобы найти файл, перейдите на pythonwhere в раздел files,

# он лежит в корневой папке

logging.basicConfig(level=logging.INFO, filename='app.log',

format='%(asctime)s %(levelname)s %(name)s %(message)s')

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

def main():

logging.info('Request: %r', request.json)

response = {

'session': request.json['session'],

'version': request.json['version'],

'response': {

'end\_session': False

}

}

handle\_dialog(response, request.json)

logging.info('Request: %r', response)

return json.dumps(response)

def handle\_dialog(res, req):

user\_id = req['session']['user\_id']

if req['session']['new']:

res['response']['text'] = \

'Привет! Я могу показать город или сказать расстояние между городами!'

return

# Получаем города из нашего

cities = get\_cities(req)

if not cities:

res['response']['text'] = 'Ты не написал название не одного города!'

elif len(cities) == 1:

res['response']['text'] = 'Этот город в стране - ' + \

get\_country(cities[0])

elif len(cities) == 2:

distance = get\_distance(get\_coordinates(

cities[0]), get\_coordinates(cities[1]))

res['response']['text'] = 'Расстояние между этими городами: ' + \

str(round(distance)) + ' км.'

else:

res['response']['text'] = 'Слишком много городов!'

def get\_cities(req):

cities = []

for entity in req['request']['nlu']['entities']:

if entity['type'] == 'YANDEX.GEO':

if 'city' in entity['value']:

cities.append(entity['value']['city'])

return cities

if \_\_name\_\_ == '\_\_main\_\_':

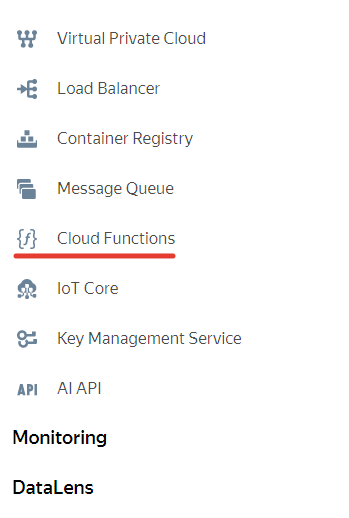
app.run()

Осталось протестировать и задеплоить наш навык.

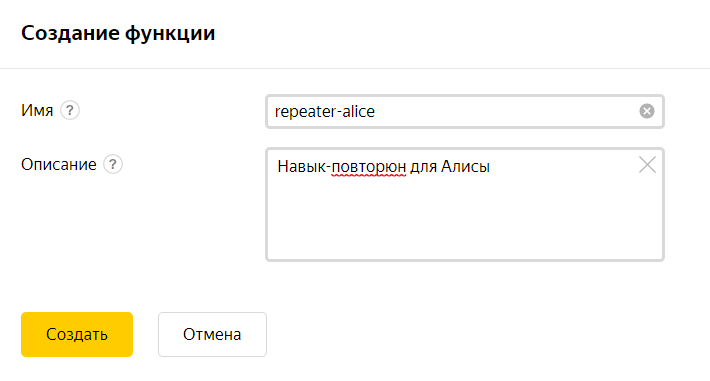
**Функции Яндекс.Облака**

Яндекс.Облако добавило возможность создавать свои функции в облаке без необходимости настраивать виртуальную машину и различные права доступа. У функций ограниченная функциональность, но вполне достаточная для, например, небольших навыков для голосового помощника Алиса. Большим плюсом использования функций с этой целью является то, что в этом случае они **не тарифицируются**, то есть пользователь может размещать сколько угодно таких функций бесплатно.

Давайте посмотрим, как сделать свою функцию. Зайдем в Яндекс.Облако и выберем пункт меню Cloud Functions.



Мы рассмотрим пример создания функции для самого простого навыка, который просто повторяет пользовательский ввод. Поэтому введем название и описание навыка исходя из его функциональности:



После этого создадим функцию. Файл можно создать как в редакторе, так и сделать его на своем компьютере, а потом загрузить в облако zip-архив.

Сама функция будет немного отличаться от того кода, который мы привыкли делать с использованием flask. Вот пример кода файла main.py:

def handler(event, context):

"""

Точка входа для облачной функции.

:param event: содержимое request.json().

:param context: информация о текущем контексте выполнения.

:return: ответ будет представлен в виде json автоматически.

"""

text = 'Привет, я повторю все, что вы скажете'

if 'request' in event and \

'original\_utterance' in event['request'] \

and len(event['request']['original\_utterance']) > 0:

text = event['request']['original\_utterance']

return {

'version': event['version'],

'session': event['session'],

'response': {

'text': text,

'end\_session': 'false'

},

}

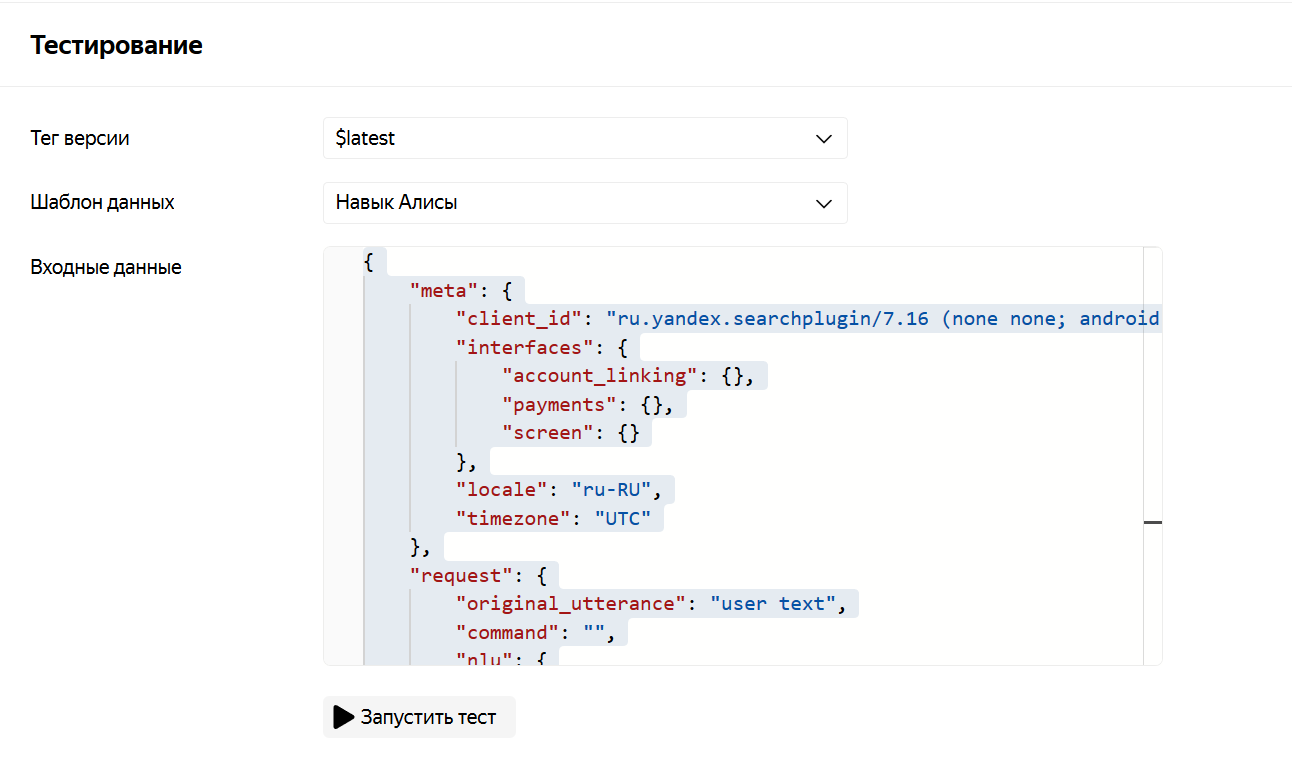
Обратите внимание: flask тут нам уже не нужен. Мы просто делаем функцию с определенной сигнатурой. Запрос к ней обработает Яндекс.Облако и положит все, что было, в request.json() в параметр *event*. Кроме того, нет необходимости самостоятельно формировать json в ответ, это произойдет автоматически.

Для того чтобы наша функция работала корректно, надо указать еще следующие значения в настройках:

* Среда выполнения — python37
* Точка входа — main.handler (имя файла без py + имя функции в этом файле, которую надо запускать)

После этого можно нажать кнопку «Создать версию».

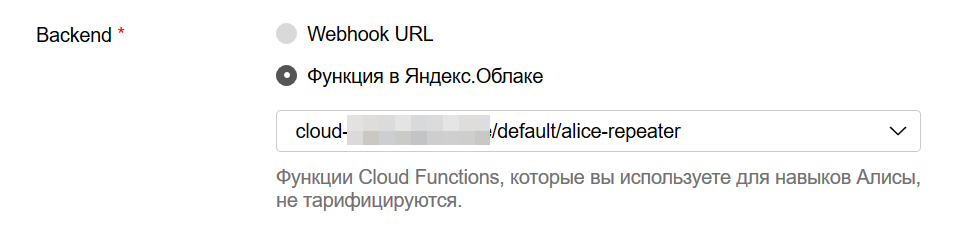
На вкладке «Тестирование» можно отправить тестовый запрос в нашу функцию почти как в Postman, причем есть возможность указать шаблон данных в формате запроса Алисы.



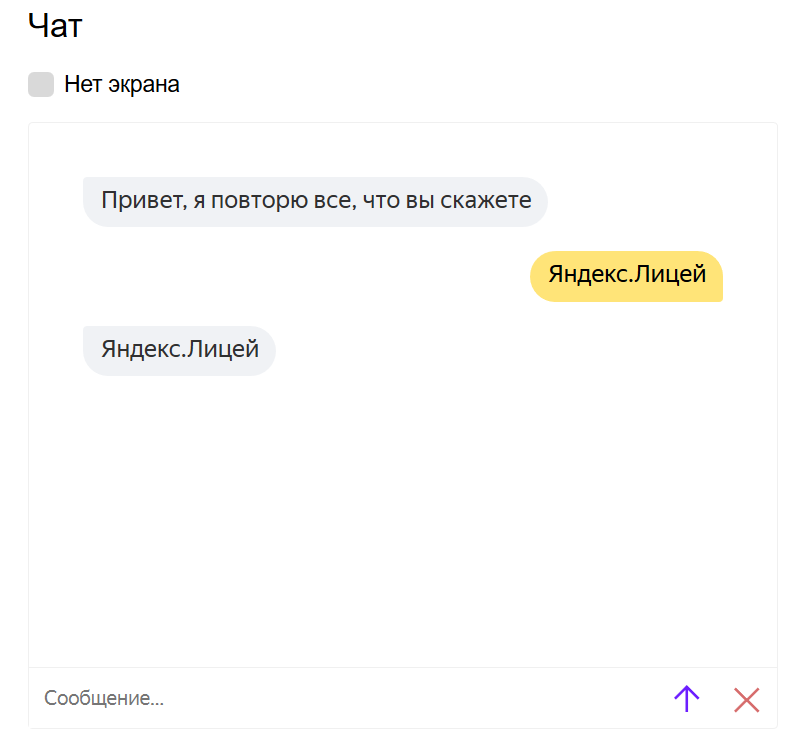
Протестируйте функцию.

Для того чтобы Алиса смогла отправить запрос к функции, ее надо сделать **публичной**. Этот переключатель находится на вкладке **Обзор**. Ссылку для Алисы копировать необязательно.

Теперь перейдем в настройки навыка Алисы и вместо Webhook выберем нужную публичную функцию в Яндекс.Облаке из списка:



Убедимся, что все работает:



То, что функция не использует flask, немного неудобно для локальной отладки. Но достаточно просто вынести ее в отдельный файл так, чтобы обработчик URL из flask вызывал ее, подстраивая данные до нужной сигнатуры и затем превращая ответ в json. Таким образом будет удобно и локально тестировать функцию, и затем заливать ее в Яндекс.Облако.







