

**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ
КОММУНИКАЦИЙ**

Ордена Трудового Красного Знамени

**Федеральное государственное бюджетное образовательное учреждение
высшего образования**

«Московский технический университет связи и информатики»

Кафедра «Математическая кибернетика и информационные технологии»

Отчет по практической работе №10

по дисциплине «Введение в информационные технологии»

на тему: «Работа с FastAPI»

Выполнила: студентка группы

БПИ2401

Алексеева Татьяна Игоревна

Проверил:

Мкртчян Грач Маратович

Москва

2025

Цель работы: освоить FastAPI, изучить типы параметров запросов, научиться валидировать данные, интегрировать внешний API и автоматизировать документацию.

Ход работы:

Для начала установка и подготовка:

```
Successfully installed annotated-types-0.7.0 anyio-4.9.0 fastapi-0.115.12 idna-3.10 pydantic-2.11.3 pydantic-core-2.33.1 sniffio-1.3.1 starlette-0.46.2 typing-extensions-4.13.2 typing-inspection-0.4.0

[notice] A new release of pip available: 22.3.1 -> 25.0.1
[notice] To update, run: python.exe -m pip install --upgrade pip
(venv) PS C:\Users\taale\PycharmProjects\LabWork_15>

Successfully installed click-8.1.8 colorama-0.4.6 h11-0.14.0 uvicorn-0.34.2

[notice] A new release of pip available: 22.3.1 -> 25.0.1
[notice] To update, run: python.exe -m pip install --upgrade pip
(venv) PS C:\Users\taale\PycharmProjects\LabWork_15>
```

Далее создаём файл main.py и импортируем fastapi и pyjokes в этот файл.

```
from fastapi import FastAPI
import pyjokes
```

Создадим объект fastapi, куда далее будут подключаться роуты. В дальнейшем будем называть его приложение fastapi.

```
from fastapi import FastAPI
import pyjokes
app = FastAPI()
```

Создаём простой роут. Для этого прописываем простую функцию, которую оборачиваем декоратором, он, в свою очередь, использует приложение, созданное ранее, http метод и путь по которому будет работать данный роут.

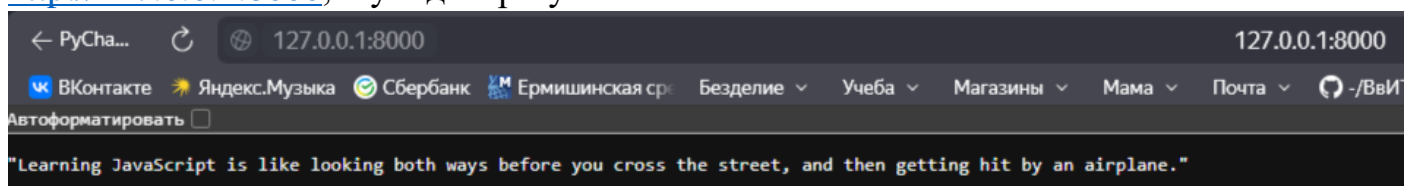
```
new *
@app.get("/")
def joke():
    return pyjokes.get_joke()
```

Для начала работы нужно запустить uvicorn – наш веб-сервер. Воспользуемся командой в консоли PyCharm.

Результат запуска команды uvicorn main_two:app --reload:

```
(venv) PS C:\Users\taale\PycharmProjects\LabWork_15> uvicorn main_two:app --reload
INFO: Will watch for changes in these directories: ['C:\\Users\\taale\\PycharmProjects\\LabWork_15']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [6116] using StatReload
INFO: Started server process [13076]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Перейдём по базовому адресу, который указывается при запуске uvicorn - <http://127.0.0.1:8000>, и увидим результат:



← PyCha... 127.0.0.1:8000 127.0.0.1:8000

ВКонтакте Яндекс.Музыка Сбербанк Ермишинская ср Безделие Учеба Магазины Мама Почта -/ВвИ

Автоформатировать

"Learning JavaScript is like looking both ways before you cross the street, and then getting hit by an airplane."

Будем использовать swagger, для удобной работы с нашим приложением. Его можно открыть по ссылке - <http://127.0.0.1:8000/docs>. Открывая данную ссылку мы видим:

FastAPI 0.1.0 OAS 3.1
/openapi.json

default

GET / Joke

На странице swagger будут отображаться все добавленные роуты. Развернём наш единственный роут и попробуем выполнить его. Для этого нажмём на кнопку “Try it out”, затем execute.

FastAPI 0.1.0 OAS 3.1
/openapi.json

default

GET / Joke

Parameters

Cancel

No parameters

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/' \
  -H 'accept: application/json'
```

Request URL

http://127.0.0.1:8000/

Server response

Code Details

200

Response body

"If you put a million monkeys at a million keyboards, one of them will eventually write a Java program. The rest of them will write Perl."



Download

Добавляем ещё один роут, где будет параметр в пути, чтобы мы могли представить шутку от какого-то конкретного человека. Для этого в фигурных скобках добавим название желаемого параметра и добавим его де в параметрах функции. Итоговый вид роута:

```
new *
@app.get("/{friend}")
def friends_joke(friend: str):
    return friend + " tells his joke:" + pyjokes.get_joke()
```

Добавим к базовому пути - <http://127.0.0.1:8000/Mary> через слеш желаемое значение параметра name (friend), чтобы результат был, как на картинке ниже:

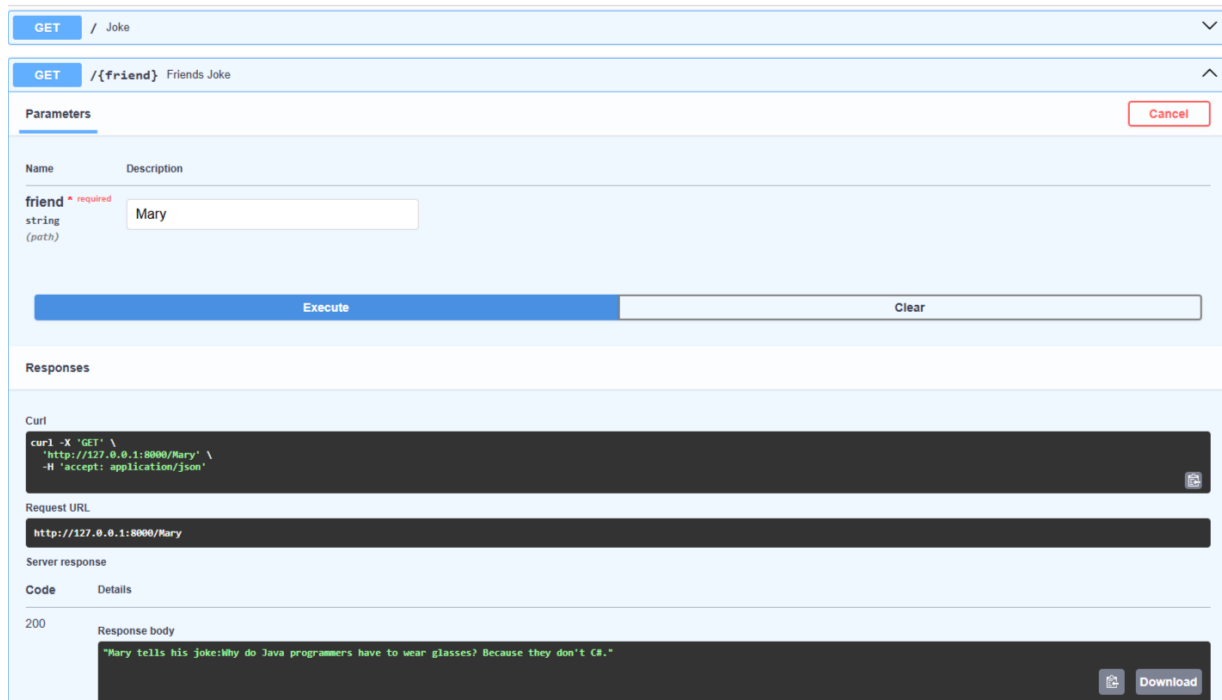
127.0.0.1:8000 127.0.0.1:8000/Mary

ВКонтакте Яндекс.Музыка Сбербанк Ермишинская ср Безделие Учеба Магазины Мама Почта -/ВВИТ Лаб

Автоформатировать

"Mary tells his joke:Speed Kills! Use Windows."

В swagger новый роут будет выглядеть следующим образом:

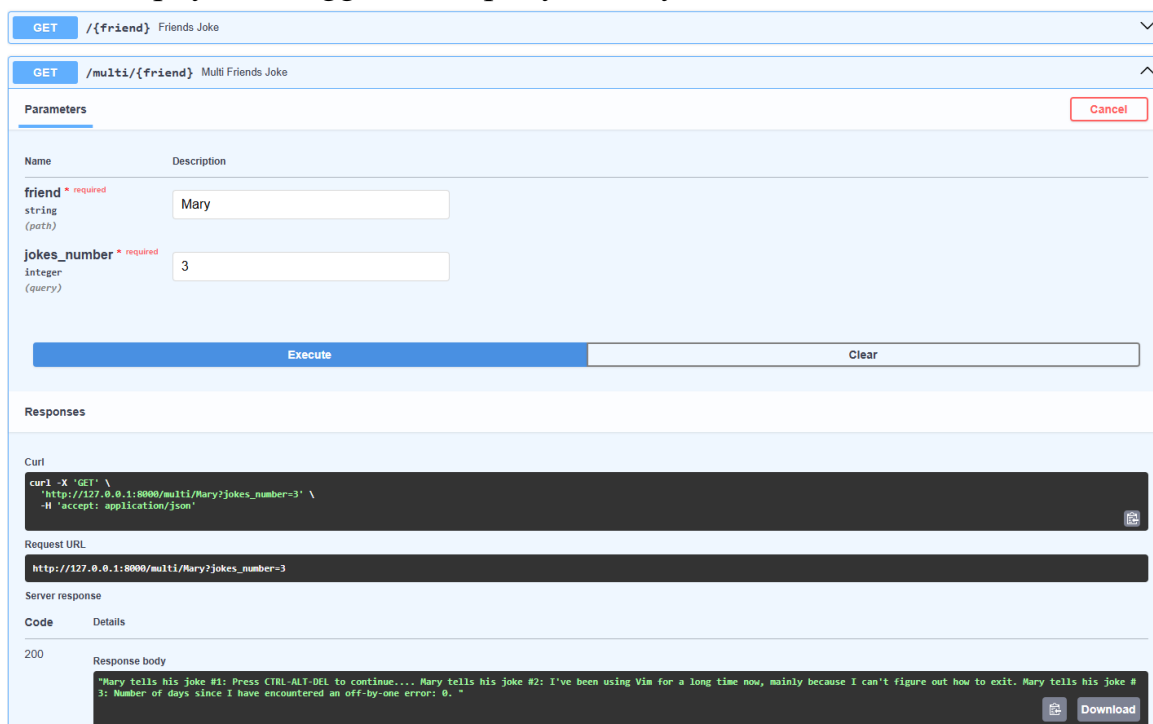


The image shows the Swagger UI for a REST API endpoint. At the top, it displays the method **GET** and the path `/Joke`. Below this, the endpoint is identified as `/{friend}` with the description "Friends Joke". The **Parameters** section shows a single required parameter named `friend` of type `string` (path), with the value `Mary` entered in the input field. Below the parameters are **Execute** and **Clear** buttons. The **Responses** section shows a **200** status code with a response body containing a joke: "Mary tells his joke: Why do Java programmers have to wear glasses? Because they don't C#." Below the response body are **Curl** and **Request URL** sections, and a **Server response** section with **Code** and **Details** tabs.

Добавим ещё один роут, где будет возможность выбирать количество шуток. Для этого добавим ещё один параметр, который не будем указывать в пути роута. Это будет query параметр `jokes_number`. Он не будет указан в пути, но также необходим для корректной работы роута. Итоговый вид роута:

```
new *
@app.get("/multi/{friend}")
def multi_friends_joke(friend: str, jokes_number: int):
    result = ""
    for i in range(jokes_number):
        result += friend + f" tells his joke #{i + 1}: " + pyjokes.get_joke() + " "
    return result
```

Откроем новый роут в swagger и попробуем запустить его:



The image shows the Swagger UI for a REST API endpoint. At the top, it displays the method **GET** and the path `/multi/{friend}` with the description "Multi Friends Joke". The **Parameters** section shows two required parameters: `friend` (string, path) with the value `Mary`, and `jokes_number` (integer, query) with the value `3`. Below the parameters are **Execute** and **Clear** buttons. The **Responses** section shows a **200** status code with a response body containing three jokes: "Mary tells his joke #1: Press CTRL-ALT-DEL to continue...", "Mary tells his joke #2: I've been using Vim for a long time now, mainly because I can't figure out how to exit.", and "Mary tells his joke #3: Number of days since I have encountered an off-by-one error: 0." Below the response body are **Curl** and **Request URL** sections, and a **Server response** section with **Code** and **Details** tabs.

Рассмотрим другой способ передачи информации в роут. Например, некоторые http запросы поддерживают передачу данных в теле запроса (Body). Создадим новый роут, используя метод POST, и создадим схему тела запроса, которую будет принимать роут для корректной работы. Импортируем из библиотеки pydantic класс BaseModel:

```
from fastapi import FastAPI
import pyjokes
from pydantic import BaseModel
```

И создадим на его основе схему получения шутки, которую и передадим на вход, как показано ниже:

```
new *
@app.post("/")
def create_joke(joke_input: JokeInput):
    return joke_input.friend + " tells his joke:" + pyjokes.get_joke()
```

Затем откроем swagger и протестируем метод:

POST / Create Joke

Parameters

No parameters

Request body required

application/json

```
{
  "friend": "Mary"
}
```

Execute Clear

Результат запроса:

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "friend": "Mary"
  }'
```

Request URL

http://127.0.0.1:8000/

Server response

Code	Details
200	<p>Response body</p> <p>"Mary tells his joke: To understand recursion you must first understand recursion."</p> <p>Download</p>

Схемы позволяют задавать образец ожидаемого ответа. Обнови ранее созданный POST метод согласно примеру ниже, чтобы ответ выдавался в формате, ранее описанном в схеме Joke.

```
new *
@app.post("/")
def create_joke(joke_input: JokeInput):
    return Joke(friend=joke_input.friend, joke=pyjokes.get_joke())
```

Посмотрим на изменения в swagger и протестируем обновленный роут. Обратим внимание, что схема не показывается в ожидаемом ответе.

Example Value | Schema

```
"string"
```

Результат успешного выполнения:

Curl

```
curl -X 'POST' \
'http://127.0.0.1:8000/' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "friend": "Mary"
}'
```

Request URL

```
http://127.0.0.1:8000/
```

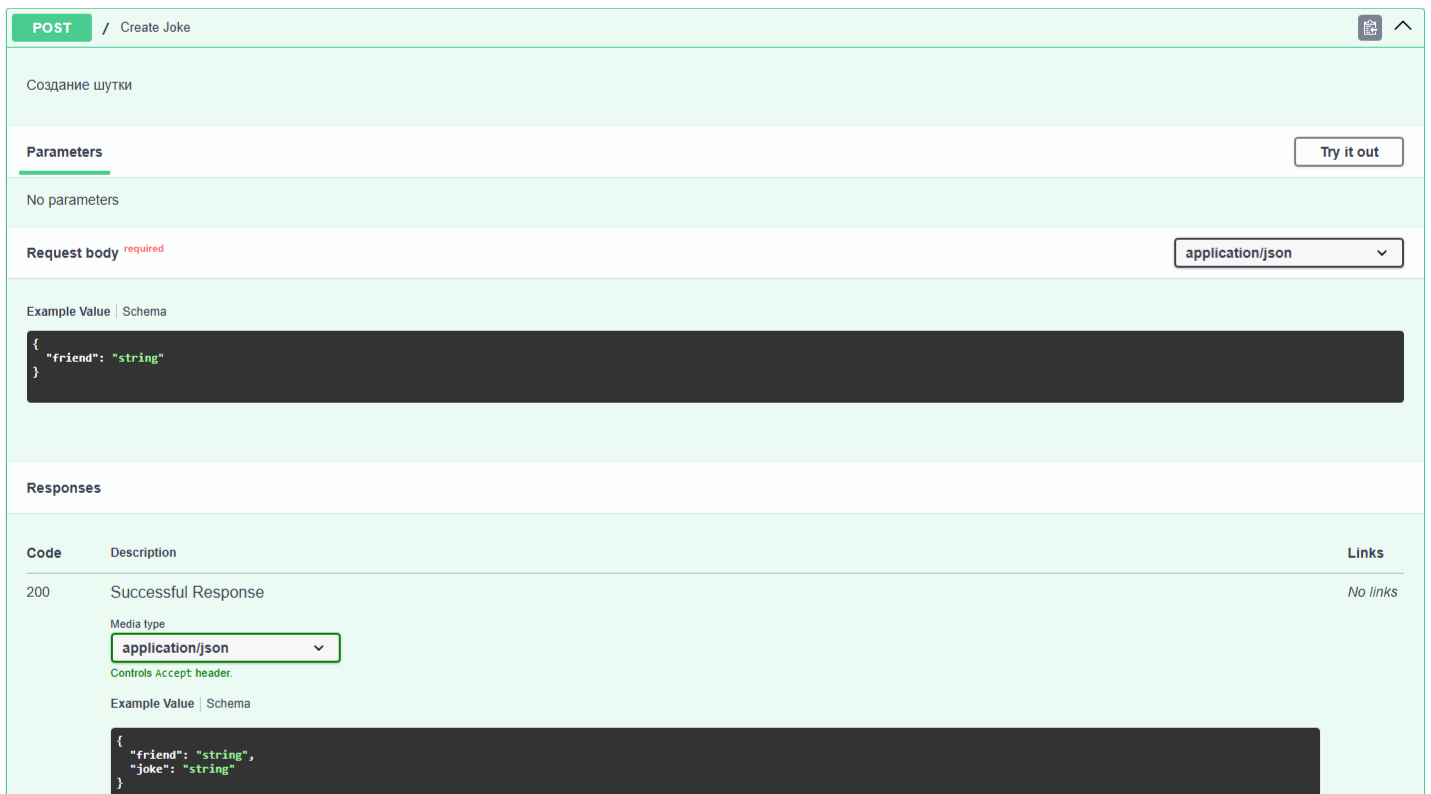
Server response

Code	Details
200	<p>Response body</p> <pre>{ "friend": "Mary", "joke": "Speed dating is useless. 5 minutes is not enough to properly explain the benefits of the Unix philosophy." }</pre> <p>Download</p>

Добавим комментарий с описанием роута и поставим в параметрах `response_model`, чтобы добавить валидацию ответа функции. Обновленный эндпоинт будет выглядеть следующим образом:

```
new *
@app.post(path="/", response_model=Joke)
def create_joke(joke_input: JokeInput):
    """Создание шутки"""
    return Joke(friend=joke_input.friend, joke=pyjokes.get_joke())
```

Обновим страницу со swagger и увидим добавленное описание эндпоинта и схему ожидаемого ответа:



Самостоятельное задание:

Чтобы мы могли извлечь данные из Википедии, мы должны сначала установить библиотеку Python Wikipedia, которая является облачной для официального API Википедии. Вводим в терминал следующую команду – `pip install wikipedia`. Результат успешной установки:

```
Successfully built wikipedia
Installing collected packages: urllib3, soupsieve, charset-normalizer, certifi, requests, beautifulsoup4, wikipedia
Successfully installed beautifulsoup4-4.13.4 certifi-2025.1.31 charset-normalizer-3.4.1 requests-2.32.3 soupsieve-2.7 urllib3-2.4.0 wikipedia-1.4.0
```

Импортируем все необходимые нам библиотеки:

```
from fastapi import FastAPI
from pydantic import BaseModel
import wikipedia
```

Установим язык для Wikipedia, как показано ниже (по умолчанию он английский, для удобства возьмём русский):

```
wikipedia.set_lang("ru")
```

Далее создадим объект `fastapi`, куда будут подключаться роуты.

```
app = FastAPI()
```

Прописываем первый роут с параметром `path`. Здесь мы будем получать статью из википедии по названию (в виде словаря) и отображать заголовок, `url` и резюме:

```
@app.get("/article/{title}")
def get_article(title: str):
    """Получаем статью по названию"""
    page = wikipedia.page(title)
    return {
        "title": page.title,
        "url": page.url,
        "summary": page.summary
    }
```

Для начала работы запускаем `uvicorn` – нам веб-сервер. Прописываем в консоли `uvicorn independent_task:app --reload`

Получаем:

```
(venv) PS C:\Users\taale\PycharmProjects\LabWork_15> uvicorn independent_task:app --reload
INFO: Will watch for changes in these directories: ['C:\\Users\\taale\\PycharmProjects\\LabWork_15']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [10916] using StatReload
INFO: Started server process [23516]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Посмотрим результат в swagger:

FastAPI 0.1.0 OAS 3.1
/openapi.json

default

GET /article/{title} Get Article

Получаем статью по названию

Parameters

Name Description

title * required
string
(path)

Python

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/article/Python' \
  -H 'accept: application/json'
```

Request URL

http://127.0.0.1:8000/article/Python

Server response

Code Details

200

Response body

```
{
  "title": "Python",
  "url": "https://ru.wikipedia.org/wiki/Python",
  "summary": "Python (МФА: [ˈpaɪθ(ə)n]; в русском языке встречается названия питон или пайтон) – мультипарадигмальный высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью, ориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение переносимости написанных на нём программ. Язык является полностью объектно-ориентированным в том плане, что всё является объектами. Необычной особенностью языка является выделение блоков кода отступами. Синтаксис ядра языка минималистичен, за счёт чего на практике редко возникает необходимость обращаться к документации. Python – интерпретируемый язык, использующийся в том числе для написания скриптов. Недостатками языка являются зачастую более низкая скорость работы и более высокое потребление памяти написанными на нём программами по сравнению с аналогичным кодом, написанным на компилируемых языках, таких как C или C++. Python является мультипарадигмным языком программирования, поддерживающим императивное, процедурное, структурное, функциональное, асинхронное, объектно-ориентированное программирование и метапрограммирование. Задачи обобщённого программирования решаются за счёт динамической типизации. Аспектно-ориентированное программирование частично поддерживается через декораторы, более полноценная поддержка обеспечивается дополнительными фреймворками. Такие методики как контрактное и логическое программирование можно реализовать с помощью библиотек или расширений. Основные архитектурные черты – динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений с глобальной блокировкой интерпретатора (GIL), высокоуровневая структура данных. Поддерживается разбиение программ на модули, которые, в свою очередь, могут объединяться в пакеты. В эталонной реализации Python является интерпретатор CPython, который поддерживает большинство активно используемых платформ, являющийся стандартом де-факто языка. Он распространяется под свободной лицензией Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях, включая проприетарные. CPython компилирует исходные тексты в высокоуровневый байт-код, который исполняется в стековой виртуальной машине. К другим трём основным реализациям языка относятся Jython (для JVM), IronPython (для CLR/.NET) и PyPy. PyPy написан на подмножестве языка Python (RPython) и разрабатывался как альтернатива CPython с целью повышения скорости исполнения программ, в том числе за счёт использования JIT-компиляции. Поддержка версии Python 2 закончилась в 2020 году. На текущий момент активно развивается версия языка Python 3. Разработка языка ведётся через предложения по расширению языка PEP (англ. Python Enhancement Proposal), в которых описываются нововведения, делаются корректировки согласно обратной связи от сообщества и документируются итоговые решения. Стандартная библиотека включает большой набор полезных переносимых функций, начиная с возможностей для работы с текстом и заканчивая средствами для написания сетевых приложений. Дополнительные возможности, такие как математическое моделирование, работа с оборудованием, написание веб-приложений или разработка игр, могут реализовываться посредством обширного количества сторонних библиотек, а также интеграцией библиотек, написанных на C или C++, при этом и сам интерпретатор Python может интегрироваться в проекты, написанные на этих языках. Существует и специализированный репозиторий программного обеспечения, написанного на Python, – PyPI. Данный репозиторий предоставляет средства для простой установки пакетов в операционную систему и стал стандартом де-факто для Python. По состоянию на 2019 год в нём содержится более 175 тысяч пакетов. Python стал одним из самых популярных языков, он используется в анализе данных, машинном обучении, DevOps и веб-разработке, а также в других сферах, включая разработку игр. За счёт читабельности, простого синтаксиса и отсутствия необходимости в компиляции язык хорошо подходит для обучения программированию, позволяя концентрироваться на изучении алгоритмов, концептов и парадигм. Отладка же и экспериментирование в значительной степени облегчаются тем фактом, что язык является интерпретируемым. Применяется язык многими крупными компаниями, такими как Google или Facebook."
}
```

Download

Следующий шаг – пропишем роут с параметром query. Будем искать статьи с указанием количества предложений (ответ выведем в виде словаря).

```
@app.get("/search")
def search_articles(query: str, sentences: int):
    """Поиск статей с указанием количества предложений"""
    summary = wikipedia.summary(query, sentences=sentences)
```



```

return {
    "query": query,
    "sentences": sentences,
    "summary": summary
}

```

Проверяем результат в swagger:

GET /search Search Articles

Parameters

Cancel

Name	Description
query * required string (query)	Artificial Intelligence
sentences * required integer (query)	2

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/search?query=Artificial%20Intelligence&sentences=2' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/search?query=Artificial%20Intelligence&sentences=2
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "query": "Artificial Intelligence", "sentences": 2, "summary": "Искусственный интеллект (англ. artificial intelligence; AI) в самом широком смысле — это интеллект, демонстрируемый машинами, в частности компьютерными системами." }</pre> <p>Download</p>

И наконец реализуем POST-запрос. Пропишем необходимые модели pydantic и сам роут, где мы будем получать случайную статью в строгом соответствии с моделью.

```

class Random(BaseModel):
    title: str
    url: str
    summary: str

@app.post("/random-article/", response_model=Random)
def get_random_article(search_input: SearchInput):
    """Получить случайную статью по теме"""
    random_title = wikipedia.random(pages=1)
    page = wikipedia.page(random_title)
    summary = wikipedia.summary(random_title,
    sentences=search_input.sentences)

    return Random(
        title = page.title,
        url = page.url,

```

```
)  
    summary = summary
```

Результат в swagger:

POST

/random-article/ Get Random Article

Parameters

No parameters

Request body required

application/json

```
{  
  "search_term": "Science",  
  "sentences": 3  
}
```

Execute

Clear

Responses

Curl

```
curl -X 'POST' \  
  'http://127.0.0.1:8000/random-article/' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "search_term": "Science",  
    "sentences": 3  
  }'
```

Request URL

http://127.0.0.1:8000/random-article/

Code

Details

200

Response body

```
{  
  "title": "Тарелка",  
  "url": "https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D1%80%D0%B5%D0%BB%D0%BA%D0%80",  
  "summary": "Тарелка, блюдо – плоское или вогнутое изделие из керамики, стекла, металла и иных материалов. Тарелки бывают различной формы с приподнятыми краями и предназначены для приёма пищи и хранения – в частности, столовой посуды. Обычно круглой формы, служит для подачи пищи на стол, а так же для чаш" }  
Download
```

Пропишем модель pydantic для валидации входных и выходных данных роутов path и query:

```

class ArticleSummary(BaseModel):
    title:str
    url: str
    summary: str

@app.get("/article/{title}", response_model=ArticleSummary)
def get_article(title: str):
    """Получаем статью по названию"""
    page = wikipedia.page(title)
    return ArticleSummary(title=page.title, url=page.url,
summary=page.summary)

class ArticleQuery(BaseModel):
    query:str
    summary: str

@app.get("/search/", response_model=ArticleQuery)
def search_articles(query: str, sentences: int = 3):
    """Поиск статей по ключевому слову"""
    summary = wikipedia.summary(query, sentences=sentences)
    return ArticleQuery(query=query, summary=summary)

```

Вывод: в ходе лабораторной работы я изучила основы FastAPI, научилась создавать REST-сервисы и применять Pydantic для валидации данных.