



# Лекция 5.

## Знакомство с FastAPI



# Оглавление

Введение в FastAPI и его возможности	3
Основные возможности FastAPI	3
Сравнение с другими фреймворками	4
Настройка среды разработки	4
Установка FastAPI	4
Настройка FastAPI	4
Запуск приложения	5
Создание базового приложения FastAPI	6
Создание модуля приложения	6
Настройка сервера и маршрутизации	6
Запуск приложения и проверка работоспособности	6
Обработка HTTP-запросов и ответов	7
Основы протокола HTTP	7
Обработка запросов GET	7
Обработка запросов POST	8
Обработка запросов PUT	9
Обработка запросов DELETE	9
Валидация данных запроса и ответа	10
Отправка запросов через curl	12
• POST запрос	12
• PUT запрос	12
• DELETE запрос	13
Создание конечных точек API	13
Определение конечных точек API	14
Работа с параметрами запроса и путями URL	14
Форматирование ответов API	16
• HTML текст	16
• JSON объект	16
Динамический HTML через шаблонизатор Jinja	17
Автоматическая документация по API	18
Интерактивная документация Swagger	18
Альтернативная документация ReDoc	21
Пример использования	22
Вывод	24

## На этой лекции мы

1. Узнаем про FastAPI и его возможности
2. Разберёмся в настройке среды разработки
3. Изучим создание базового приложения FastAPI
4. Узнаем об обработке HTTP-запросов и ответов
5. Разберёмся в создании конечных точек API
6. Изучим автоматическую документацию по API

## Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Вспомнили про синхронный подход к решению задач
2. Узнали о многопоточном подходе в программировании
3. Разобрались с многопроцессорным подходом в Python
4. Изучили асинхронный подход в решении задач

## Подробный текст лекции

### Введение в FastAPI и его возможности

**Framework FastAPI** — это современный фреймворк для создания веб-приложений на языке Python. Он был создан с учетом последних тенденций веб-разработки и имеет ряд преимуществ перед другими фреймворками. FastAPI совсем новый фреймворк, вышедший в 2018 году. С тех пор он активно развивается, набирает популярность.

### Основные возможности FastAPI

К основным возможностям FastAPI можно отнести следующие:

- Высокая скорость работы благодаря использованию асинхронных функций и типизации данных.
- Автоматическая генерация документации API на основе аннотаций функций и моделей данных.
- Встроенная валидация данных запросов и ответов.
- Поддержка OpenAPI и JSON Schema.
- Простота использования благодаря интуитивно понятному синтаксису и многочисленным примерам.

## Сравнение с другими фреймворками

FastAPI имеет ряд преимуществ перед другими популярными фреймворками, такими как Flask и Django. Он более быстрый благодаря использованию асинхронных функций, более безопасный благодаря встроенной валидации данных и поддержке OpenAPI, а также более простой в использовании благодаря интуитивно понятному синтаксису.

## Настройка среды разработки

Рассмотрим процесс настройки среды разработки для работы с FastAPI.

### Установка FastAPI

Первым шагом является установка FastAPI. Для этого необходимо использовать менеджер пакетов `pip`, который уже устанавливается вместе с Python. Откройте терминал и выполните следующую команду:

```
pip install fastapi
```

Эта команда установит FastAPI и все его зависимости.

Отдельно необходимо установить ASGI сервер для запуска приложения. Один из вариантов — установка `uvicorn`.

```
pip install "uvicorn[standard]"
```

## Настройка FastAPI

Для работы с FastAPI необходимо создать файл приложения и определить конечные точки API. Для этого можно использовать любой текстовый редактор или интегрированную среду разработки (IDE).

Пример кода:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

В этом примере мы создали объект FastAPI и определили конечную точку API с помощью декоратора `@app.get("/")`. Декоратор указывает, что это обработчик GET-запроса по пути `/`.

Внутри функции мы возвращаем словарь с сообщением "Hello World". Это сообщение будет отправлено в ответ на запрос.

## Запуск приложения

Для запуска приложения необходимо использовать сервер для запуска приложений `uvicorn`. Для этого открываем терминал ОС, переходим в каталог с проектом и выполняем следующую команду:

```
uvicorn main:app --reload
```

Эта команда запустит сервер на локальном хосте по адресу <http://127.0.0.1:8000/>.

Для остановки сервера нажмите сочетание клавиш `Ctrl + C` в терминале.

Мы рассмотрели процесс настройки среды разработки для работы с FastAPI. Установили Fast API и сервер `unicorn`, создали файл приложения и определили

конечные точки API. Затем мы запустили сервер для запуска приложений uvicorn и проверили работу приложения в браузере.

## Создание базового приложения FastAPI

Рассмотрим процесс создания базового приложения FastAPI. Вы увидите много общего с Flask.

### Создание модуля приложения

Первым шагом является создание модуля приложения. Для этого создайте файл main.py и импортируйте FastAPI:

```
from fastapi import FastAPI

app = FastAPI()
```

В этом примере мы создали объект FastAPI и назвали его app.

### Настройка сервера и маршрутизации

Далее необходимо настроить сервер и определить маршрутизацию для нашего приложения. Для этого создайте функции-обработчики запросов и определите их маршруты.

```
@app.get("/")
async def read_root():
    return {"Hello": "World"}

@app.get("/items/{item_id}")
async def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}
```

В этом примере мы определили две функции-обработчика запросов. Первая функция обрабатывает GET-запрос по корневому пути "/" и возвращает словарь с сообщением "Hello World". Вторая функция обрабатывает GET-запрос по пути "/items/{item\_id}", где item\_id — это переменная пути, а q — это параметр запроса. Функция возвращает словарь с переданными параметрами.

## Запуск приложения и проверка работоспособности

Для запуска приложения необходимо использовать сервер для запуска приложений uvicorn. Для этого выполните следующую команду:

```
uvicorn main:app --reload
```

Эта команда запустит сервер на локальном хосте по адресу <http://127.0.0.1:8000/>.

Чтобы проверить работоспособность приложения, откройте браузер и перейдите по адресу <http://127.0.0.1:8000/>. Вы должны увидеть сообщение "Hello World".

Чтобы проверить работу второй функции, перейдите по адресу <http://127.0.0.1:8000/items/5?q=test>, где 5 — это значение переменной item\_id, а test — значение параметра q. Вы должны увидеть словарь с переданными параметрами.

## Обработка HTTP-запросов и ответов

**HTTP (Hypertext Transfer Protocol)** — это протокол передачи данных в интернете, используемый для обмена информацией между клиентом и сервером. В FastAPI обработка HTTP-запросов и ответов происходит автоматически.

### Основы протокола HTTP

Протокол HTTP работает по схеме "клиент-сервер". Клиент отправляет запрос на сервер, а сервер отвечает на этот запрос. Запрос состоит из трех частей: метод, адрес и версия протокола. Методы запроса могут быть GET, POST, PUT, DELETE и другие. Адрес - это URL-адрес ресурса, к которому обращается клиент. Версия протокола указывает на версию HTTP, которую использует клиент.

### Обработка запросов GET

Метод GET используется для получения ресурсов с сервера. В FastAPI обработка GET-запросов происходит с помощью декоратора `@app.get()`. Например:

```
import logging
from fastapi import FastAPI
```

```
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app = FastAPI()

@app.get("/")
async def read_root():
    logger.info('Отработал GET запрос.')
    return {"Hello": "World"}
```

Этот код создает приложение FastAPI и добавляет обработчик GET-запросов для корневого URL-адреса. Функция `read_root()` возвращает JSON-объект `{"Hello": "World"}`.

## Обработка запросов POST

Метод POST используется для отправки данных на сервер. В FastAPI обработка POST-запросов происходит с помощью декоратора `@app.post()`. Например:

```
import logging
from fastapi import FastAPI


logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app = FastAPI()

@app.post("/items/")
async def create_item(item: Item):
    logger.info('Отработал POST запрос.')
    return item
```

Этот код создает приложение FastAPI и добавляет обработчик POST-запросов для URL-адреса `/items/`. Функция `create_item()` принимает объект `Item` и возвращает его же.



 **Внимание!** Код выше не будет работать, так как мы не определили объект Item. Речь о модуле pydantic позволяющем создать класс Item будет позже в рамках курса.

## Обработка запросов PUT

Метод PUT используется для обновления данных на сервере. В FastAPI обработка PUT-запросов происходит с помощью декоратора `@app.put()`. Например:


```
import logging
from fastapi import FastAPI

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app = FastAPI()

@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item):
    logger.info(f'Отработал PUT запрос для item id = {item_id}.')
    return {"item_id": item_id, "item": item}
```

Этот код создает приложение FastAPI и добавляет обработчик PUT-запросов для URL-адреса `/items/{item_id}`. Функция `update_item()` принимает идентификатор элемента и объект Item и возвращает JSON-объект с этими данными.

 **Внимание!** Код выше не будет работать, так как мы не определили объект Item. Речь о модуле pydantic позволяющем создать класс Item будет позже в рамках курса.

## Обработка запросов DELETE

Метод DELETE используется для удаления данных на сервере. В FastAPI обработка DELETE-запросов происходит с помощью декоратора `@app.delete()`. Например:


```
import logging
from fastapi import FastAPI

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app = FastAPI()

@app.delete("/items/{item_id}")
async def delete_item(item_id: int):
    logger.info(f'Отработал DELETE запрос для item id = {item_id}.')
    return {"item_id": item_id}
```

Этот код создает приложение FastAPI и добавляет обработчик DELETE-запросов для URL-адреса `/items/{item_id}`. Функция `delete_item()` принимает идентификатор элемента и возвращает JSON-объект с этим идентификатором.

 **Важно!** Зачастую операция удаления не удаляет данные из базы данных, а изменяет специально созданное поле `is_deleted` на значение Истина. Таким образом вы сможете восстановить ранее удалённые данные пользователя, если он передумает спустя время.

## Валидация данных запроса и ответа

FastAPI позволяет автоматически валидировать данные запроса и ответа с помощью модуля `pydantic`. Например, можно создать класс `Item` для валидации данных:

```

...
from typing import Optional
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None
...

```

Этот класс содержит поля name, description, price и tax. Поля name и price обязательны, а поля description и tax необязательны. Затем можно использовать этот класс для валидации данных запроса и ответа:

```

...
from fastapi import FastAPI
from typing import Optional
from pydantic import BaseModel

...
app = FastAPI()

class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

...

@app.post("/items/")
async def create_item(item: Item):
    logger.info('Отработал POST запрос.')
    return item

@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item):
    logger.info(f'Отработал PUT запрос для item id = {item_id}.')
    return {"item_id": item_id, "item": item}
...

```

Этот код добавляет обработчики POST и PUT запросов, которые принимают объект Item и возвращают его же. Если данные не соответствуют описанию класса Item, то FastAPI вернет ошибку 422 с описанием ошибки.

## Отправка запросов через curl

Если с GET запросом проблем не было, то для тестирования POST, PUT и DELETE запросов воспользуемся curl.

**Curl (client URL)** — это инструмент командной строки на основе библиотеки libcurl для передачи данных с сервера и на сервер при помощи различных протоколов, в том числе HTTP, HTTPS, FTP, FTPS, IMAP, IMAPS, POP3, POP3S, SMTP и SMTPS. Он очень популярен в сфере автоматизации и скриптов благодаря широкому диапазону функций и поддерживаемых протоколов.

### ● POST запрос

Для отправки POST запроса нашему серверу введём в терминале следующую строку:

```
curl -X 'POST' 'http://127.0.0.1:8000/items/' -H 'accept: application/json' -H 'Content-Type: application/json' -d '{"name": "BestSale", "description": "The best of the best", "price": 9.99, "tax": 0.99}'
```

Эта строка отправляет POST запрос на URL-адрес «http://127.0.0.1:8000/items/» с данными JSON, содержащими поля «имя», «описание», «цена» и «налог» вместе с соответствующими значениями. Заголовки «accept» и «Content-Type» имеют значение «application/json», мы пересылаем запросом json объект на сервер и хотим получить json в качестве ответа.

### ● PUT запрос

Для отправки PUT запроса нашему серверу введём в терминале следующую строку:

```
curl -X 'PUT' 'http://127.0.0.1:8000/items/42' -H 'accept: application/json' -H 'Content-Type: application/json' -d '{"name": "NewName", "description": "New description of the object", "price": 77.7, "tax": 10.01}'
```

Эта строка отправляет HTTP-запрос PUT на локальный сервер по адресу `http://127.0.0.1:8000/`, обновляя элемент с идентификатором 42 новой информацией, предоставленной в формате JSON, такой как имя, описание, цена и налог.

Мы можем опускать необязательные поля объекта `Item` в запросе. Ответ от сервера будет 200. А вот отсутствие обязательных параметров приведёт к ответу 422 Unprocessable Entity.

Хороший короткий PUT запрос:

```
curl -X 'PUT' 'http://127.0.0.1:8000/items/42' -H 'accept: application/json' -H 'Content-Type: application/json' -d '{"name": "NewName", "price": 77.7}'
```

Плохой PUT запрос:

```
curl -X 'PUT' 'http://127.0.0.1:8000/items/42' -H 'accept: application/json' -H 'Content-Type: application/json' -d '{"name": "NewName", "tax": 77.7}'
```

В данном запросе отсутствует обязательное поле `price`. Его мы сделали обязательным в классе `Item` строкой `price: float`.

Код состояния ответа HTTP **422 Unprocessable Entity** указывает, что сервер понимает тип содержимого в теле запроса и синтаксис запроса является правильным, но серверу не удалось обработать инструкции содержимого.

## ● DELETE запрос

Чтобы удалить объект нужен лишь его идентификатор, без передачи самого объекта. `curl` будет выглядеть следующим образом:

```
curl -X 'DELETE' 'http://127.0.0.1:8000/items/13' -H 'accept: application/json'
```

Запрос DELETE сообщает серверу о желании удалить объект с id 13.

# Создание конечных точек API

FastAPI позволяет легко создавать конечные точки (endpoints) API для взаимодействия с клиентами. Рассмотрим, как определять конечные точки, работать с параметрами запроса и путями URL, а также форматировать ответы API.

## Определение конечных точек API

Конечная точка API — это URL-адрес, по которому клиент может отправлять запросы к серверу. В FastAPI определение конечных точек происходит с помощью декораторов.

Например так:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def read_root():
    return {"Hello": "World"}

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

Этот код создает две конечные точки: одну для корневого URL-адреса, другую для URL-адреса `/items/{item_id}`. Функции `read_root()` и `read_item()` обрабатывают GET-запросы и возвращают JSON-объекты.

## Работа с параметрами запроса и путями URL

Часто клиенты отправляют запросы с параметрами, которые нужно обработать на сервере. В FastAPI параметры запроса и пути URL определяются в декораторах конечных точек.

Например:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id: int, q: str = None):
    if q:
        return {"item_id": item_id, "q": q}
    return {"item_id": item_id}
```

Этот код создает конечную точку для URL-адреса `/items/{item_id}`, которая принимает параметр `item_id` типа `int` и параметр `q` типа `str` со значением по умолчанию `None`. Если параметр `q` задан, функция возвращает JSON-объект с обоими параметрами, иначе — только с `item_id`.

Мы также можем определить несколько параметров URL-адреса в пути, например `/users/{user_id}/orders/{order_id}`, а затем определить соответствующие параметры в функции для доступа к ним.

```
...
@app.get("/users/{user_id}/orders/{order_id}")
async def read_item(user_id: int, order_id: int):
    # обработка данных
    return {"user_id": user_id, "order_id": order_id}
```

Использование параметров запроса с FastAPI может быть любым удобным для решения поставленной задачи.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/")
async def read_item(skip: int = 0, limit: int = 10):
    return {"skip": skip, "limit": limit}
```

В этом примере мы определяем новый маршрут `/items/`, который принимает два параметра запроса `skip` и `limit`. Значения по умолчанию для этих параметров равны

0 и 10 соответственно. Когда мы вызываем этот маршрут без каких-либо параметров запроса, он возвращает значения по умолчанию.

Например перейдя по адресу <http://127.0.0.1:8000/items/> получим json с {"skip": 0, "limit": 10}.

Мы также можем передать параметры запроса в URL-адресе, например <http://127.0.0.1:8000/items/?skip=20&limit=30>. В таком случае ответ будет следующим json объектом {"skip": 20, "limit": 30}.

## Форматирование ответов API

FastAPI позволяет форматировать ответы API в различных форматах, например, в JSON или HTML. Для этого нужно использовать соответствующие функции модуля `fastapi.responses`.

### • HTML текст

Например:

```
from fastapi import FastAPI
from fastapi.responses import HTMLResponse

app = FastAPI()

@app.get("/", response_class=HTMLResponse)
async def read_root():
    return "<h1>Hello World</h1>"
```

Этот код создает конечную точку для корневого URL-адреса, которая возвращает HTML-страницу с текстом "Hello World". Функция `read_root()` использует класс `HTMLResponse` для форматирования ответа в HTML.

### • JSON объект

В этом примере возвращается ответ JSON с настраиваемым сообщением и кодом состояния.

```
from fastapi import FastAPI
from fastapi.responses import JSONResponse

app = FastAPI()

@app.get("/message")
async def read_message():
```



```
message = {"message": "Hello World"}
return JSONResponse(content=message, status_code=200)
```

В этом примере мы импортируем класс `JSONResponse` из модуля `FastAPI.responses`. Внутри функции `read_message` мы определяем словарь, содержащий ключ сообщения со значением «Hello World». Затем мы возвращаем объект `JSONResponse` со словарем сообщений в качестве содержимого и кодом состояния 200.

## Динамический HTML через шаблонизатор Jinja

В следующем примере используется шаблонизация `Jinja2` для создания ответа HTML с динамическим содержимым.

```
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
from fastapi.templating import Jinja2Templates

app = FastAPI()
templates = Jinja2Templates(directory="templates")

@app.get("/{name}", response_class=HTMLResponse)
async def read_item(request: Request, name: str):
    return templates.TemplateResponse("item.html", {"request":
request, "name": name})
```

В этом примере мы импортируем класс `Jinja2Templates` из модуля `FastAPI.templating`. Мы создаем экземпляр этого класса и передаем каталог, в котором расположены наши шаблоны. В функции `read_item` мы получаем параметр имени из пути URL и генерируем динамический HTML-ответ, используя шаблон `Jinja2` (`item.html`). Шаблон получает объект запроса и параметр имени в качестве переменных контекста для отображения в ответе HTML.

Простейший шаблон `item.html`

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>Item - {{ name }}</title>
</head>
```

```
<body>
  <h1>Hello, {{ name|title }}!</h1>
</body>
</html>
```

## Автоматическая документация по API

FastAPI обладает встроенным инструментом для автоматической документации API, который позволяет быстро и удобно ознакомиться с функциональностью приложения. Рассмотрим два варианта документации API: интерактивную документацию Swagger и альтернативную документацию ReDoc.

### Интерактивная документация Swagger

**Swagger** — это инструмент для создания и документирования API. FastAPI использует Swagger UI для генерации интерактивной документации, которая отображает все маршруты, параметры и модели данных, которые были определены в приложении.

Для просмотра интерактивной документации Swagger нужно запустить приложение и перейти по адресу <http://localhost:8000/docs>. На странице будут представлены все маршруты и параметры, доступные в приложении.

Fast API 0.1.0 OAS3  
/openapi.json

default

GET /items/{item\_id} Read Item Get

Try it out

Parameters

Name	Description
item_id <span style="color: red;">* required</span>	
integer	
(path)	
q	
string	
(query)	

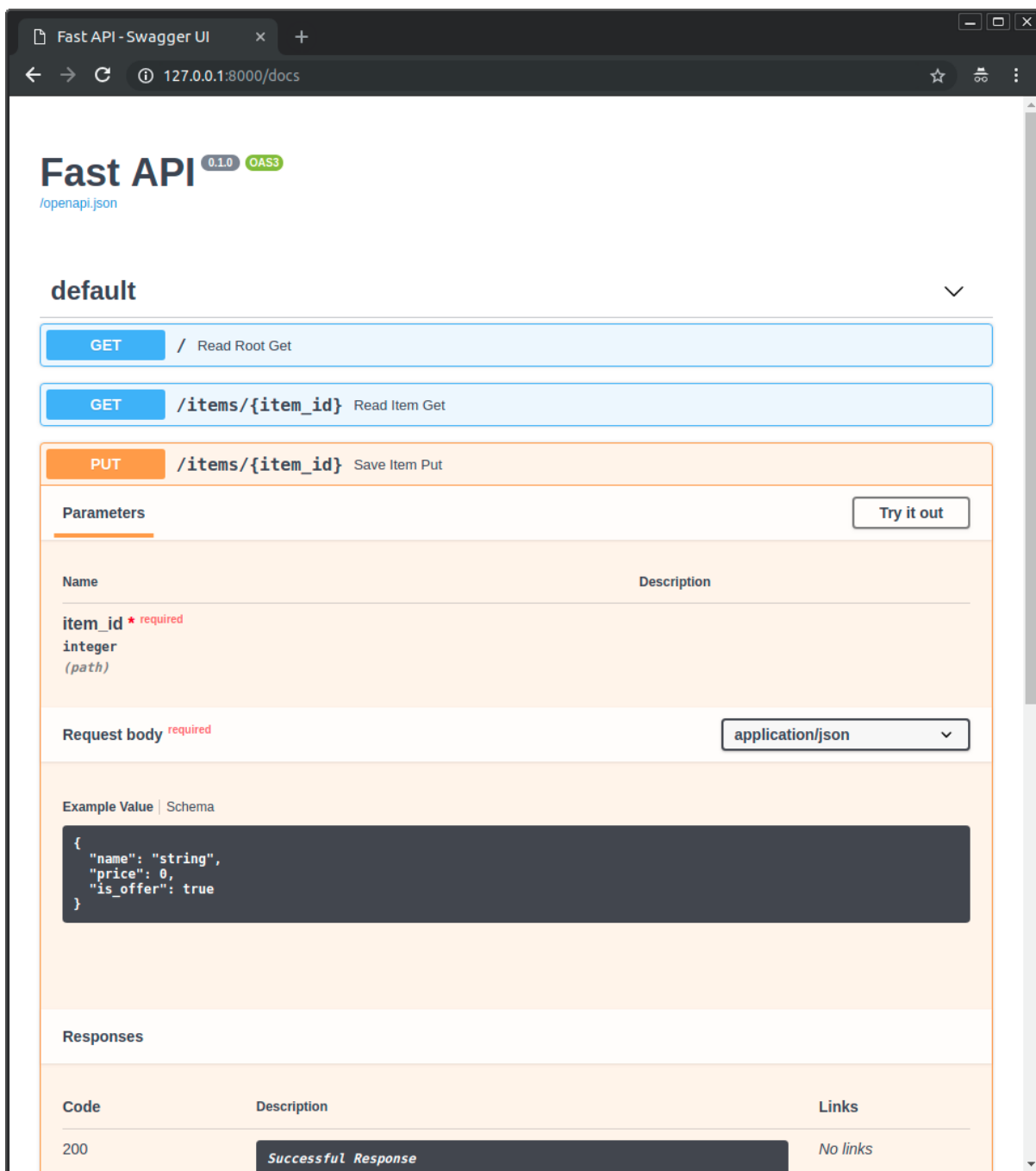
Responses

Code	Description	Links
200	<b>Successful Response</b>	No links
	application/json	
	Controls Accept header.	
422	<b>Validation Error</b>	No links
	application/json	

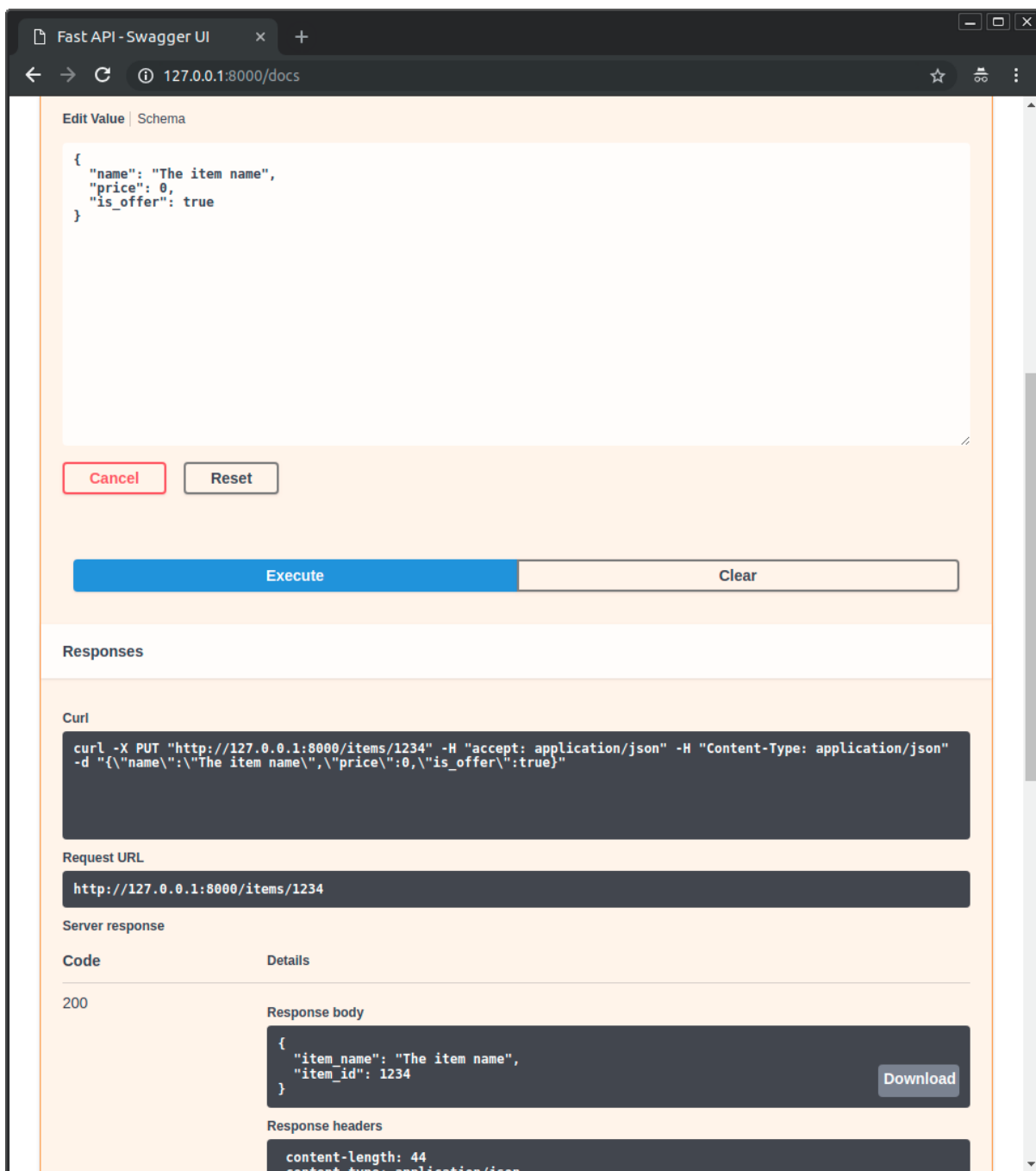
Example Value | Schema

```
{
  "detail": [
    {
      "loc": [
        "string"
      ]
    }
  ]
}
```

При выборе конкретного маршрута откроется его описание, включая возможные параметры запроса и ответа. Для каждого параметра указаны его тип, описание и необходимость.



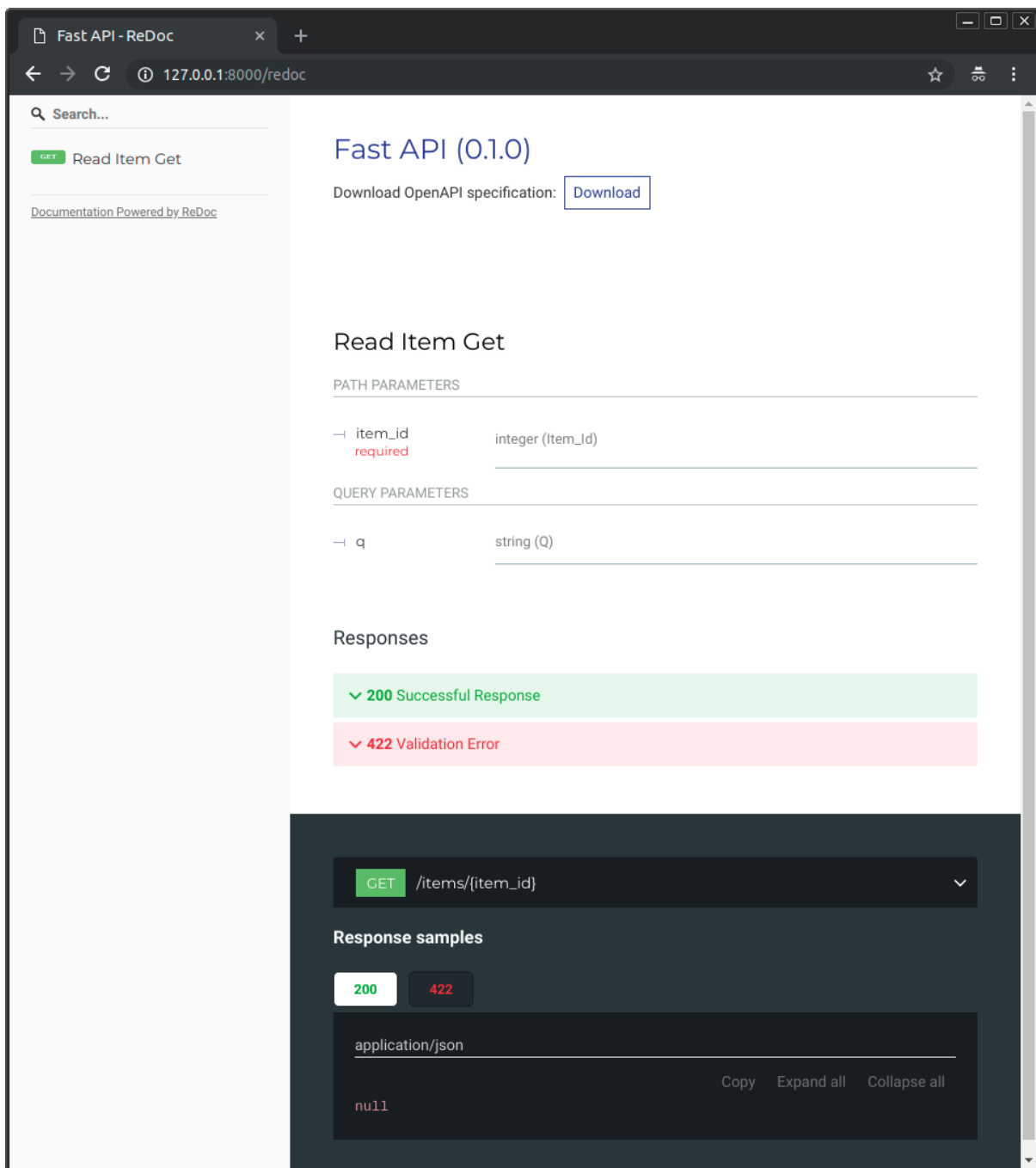
Также Swagger UI позволяет отправлять запросы к API прямо из интерфейса. Для этого нужно выбрать метод запроса и заполнить параметры запроса, если таковые имеются. После отправки запроса будет отображен его результат.



## Альтернативная документация ReDoc

**ReDoc** — это альтернативный инструмент для документирования API, который также поддерживается FastAPI. Он предоставляет более простой и лаконичный интерфейс для просмотра документации.

Для просмотра документации ReDoc нужно запустить приложение и перейти по адресу <http://localhost:8000/redoc>. На странице будет отображена документация API в формате OpenAPI.



Как и в Swagger, каждый маршрут содержит описание его параметров и возможных ответов. Однако ReDoc не позволяет отправлять запросы к API из интерфейса.

## Пример использования

Для того чтобы включить генерацию документации API в FastAPI, нужно использовать модуль `fastapi.openapi`. Например, вот как выглядит простой пример приложения с одним маршрутом:

```

from fastapi import FastAPI

app = FastAPI()

@app.get("/hello/{name}")
async def read_item(name: str, age: int):
    return {"Hello": name, "Age": age}

```

Для генерации документации нужно создать экземпляр класса FastAPI с параметром openapi\_url:

```

from fastapi import FastAPI
from fastapi.openapi.utils import get_openapi

app = FastAPI(openapi_url="/api/v1/openapi.json")

@app.get("/hello/{name}")
async def read_item(name: str, age: int):
    return {"Hello": name, "Age": age}

def custom_openapi():
    if app.openapi_schema:
        return app.openapi_schema
    openapi_schema = get_openapi(
        title="Custom title",
        version="1.0.0",
        description="This is a very custom OpenAPI schema",
        routes=app.routes,
    )
    app.openapi_schema = openapi_schema
    return app.openapi_schema

app.openapi = custom_openapi

```

В этом примере мы переопределили метод custom\_openapi, который генерирует схему OpenAPI вручную. Мы также установили значение параметра openapi\_url, чтобы FastAPI знал, где разместить схему OpenAPI.

После запуска приложения можно перейти по адресу <http://localhost:8000/api/v1/openapi.json> и убедиться, что схема OpenAPI была успешно сгенерирована.

Затем можно запустить приложение и перейти по адресу <http://localhost:8000/docs> или <http://localhost:8000/redoc>, чтобы просмотреть сгенерированную документацию.

## Вывод

На этой лекции мы:

1. Узнали про FastAPI и его возможности
2. Разобрались в настройке среды разработки
3. Изучили создание базового приложения FastAPI
4. Узнали об обработке HTTP-запросов и ответов
5. Разобрались в создании конечных точек API
6. Изучили автоматическую документацию по API

## Домашнее задание

Для закрепления материалов лекции попробуйте самостоятельно набрать и запустить демонстрируемые примеры.