**Модуль 11: Побудова REST API на FastAPI**

**Заняття 1: Основи FastAPI Фреймворк**[**FastAPI**](https://fastapi.tiangolo.com/)

[**FastAPI**](https://fastapi.tiangolo.com/) (https://fastapi.tiangolo.com/) — це вебфреймворк Python з відкритим вихідним кодом, який розроблений, щоб бути швидким та простим у використанні. Він побудований на основі Starlette та Pydantic, та натхненний попередніми вебфреймворками, такими як Flask і Falcon. Це сучасна, швидка (високопродуктивна) веб-інфраструктура для створення API-інтерфейсів за допомогою Python 3.7+ на основі стандартних підказок типів Python. Він побудований на основі Starlette і Pydantic та є одним із найшвидших доступний фреймворків Python на сьогоднішній день.

FastAPI має кілька переваг у порівнянні з іншими веб-фреймворками:

1. *Висока швидкість*: FastAPI використовує останні технології Python та має мінімальні накладні витрати на ресурси, тому він може обробляти велику кількість запитів за секунду.
2. *Легкість у використанні*: FastAPI має чистий та простий API, що дозволяє швидко та ефективно створювати складні вебзастосунки.
3. *Підтримка асинхронності*: FastAPI підтримує асинхронний код, що робить його оптимальним для обробки безлічі одночасних запитів.
4. *Валідація даних*: FastAPI має вбудовану валідацію даних, яка дозволяє легко перевірити параметри запиту і відповіді на відповідні специфікації.
5. *Автоматична документація*: FastAPI має вбудовану підтримку для створення документації у форматі OpenAPI і Swagger, що дозволяє легко відображати інформацію про доступні маршрути, параметри запиту та відповіді, а також тестувати API прямо з документації.
6. *Підтримка стандартних бібліотек*: FastAPI використовує стандартні бібліотеки Python, такі як Pydantic для моделювання даних і Starlette для обробки запитів та відповідей, що дозволяє легко інтегрувати його з іншими бібліотеками.
7. *Підтримка типізації*: FastAPI підтримує статичну типізацію на основі стандартної бібліотеки Python, що дозволяє виявляти помилки на етапі розробки та покращує читабельність коду.

INFO

FastAPI був третім за популярністю вебфреймворком в опитуванні розробників [Stack Overflow 2021](https://insights.stackoverflow.com/survey/2021/#section-most-loved-dreaded-and-wanted-web-frameworks)

(https://insights.stackoverflow.com/survey/2021/#section-most-loved-dreaded-and-wanted-web-frameworks).

Був визнаний найулюбленішим (*most loved*) python-вебфреймворком на підставі опитування серед розробників на порталі Stack Overflow в 2021 та в 2022 роках.

В цілому, FastAPI — чудовий вибір для створення сучасних високопродуктивних API за допомогою Python. Перевага автоматичної генерації документації з коду та вбудована перевірка типів роблять цей фреймворк чудовим вибором для мікросервісної інфраструктури. Крім цього, фреймворк асинхронний та дуже ефективно використовує ресурси.

Встановимо FastAPI, виконавши команди

poetry add fastapi

poetry add uvicorn[standard]

NOTE

Uvicorn — це реалізація веб-сервера ASGI для Python.

**Запуск сервера FastAPI**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-01/intro#%D0%B7%D0%B0%D0%BF%D1%83%D1%81%D0%BA-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%B0-fastapi)Тепер додайте наступний код до файлу main.py, щоб запустити сервер FastAPI з uvicorn.

**main.py**

from fastapi import FastAPI

app = FastAPI()

@app.get("/api/healthchecker")

def root():

return {"message": "Welcome to FastAPI!"}

В цьому прикладі ми імпортуємо клас FastAPI з модуля fastapi. Далі створюємо екземпляр FastAPI — наш застосунок app: app = FastAPI(). А також додаємо обробника маршруту /api/healthchecker — функцію root. Ця функція повертає словник {"message": "Welcome to FastAPI!"}, який буде автоматично перетворений у формат JSON та відправлений клієнту по HTTP протоколу.

Як бачимо, FastAPI — це клас Python, який надає всі функціональні можливості для нашого API. А змінна app — основна точка взаємодії для створення всіх наших API.

INFO

У загальному випадку, коли ми говоримо API (Application Programming Interface), мова йде про опис взаємодії однієї комп'ютерної програми з іншими

При побудові API, зазвичай, використовуються конкретні методи HTTP для виконання певної дії.

* POST: для створення даних.
* GET: для читання даних.
* PUT: для оновлення даних.
* PATCH: для часткового оновлення даних.
* DELETE: для видалення даних.

Стандарт OpenAPI та фреймворк FastAPI кожен із HTTP-методів називає *операцією* (operation). В подальшому будемо притримуватися використання цієї термінології.

Наш декоратор @app.get("/api/healthchecker") повідомляє FastAPI, що функція, яка знаходиться нижче, відповідає за обробку запитів, що ідуть за адресою: /api/healthchecker, використовуючи GET операцію.

Можна використовувати й інші операції:

* @app.post()
* @app.put()
* @app.delete()
* @app.patch()

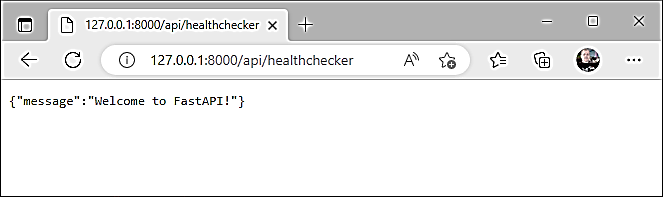
Виконайте наступну команду, щоб запустити сервер FastAPI з uvicorn:

uvicorn main:app --host localhost --port 8000 --reload

Тут параметри команди мають наступне значення:

* uvicorn — високопродуктивний веб-сервер ASGI;
* main — файл app/main.py;
* app — об'єкт, повернений після запиту app = FastAPI();
* --host — дозволяє прив'язати сокет до хоста. Значення за замовчуванням 127.0.0.1;
* --port — дозволяє прив'язати сокет до певного порту. За замовчуванням використовується значення 8000;
* --reload — забезпечує гаряче перезавантаження сервера під час розробки.

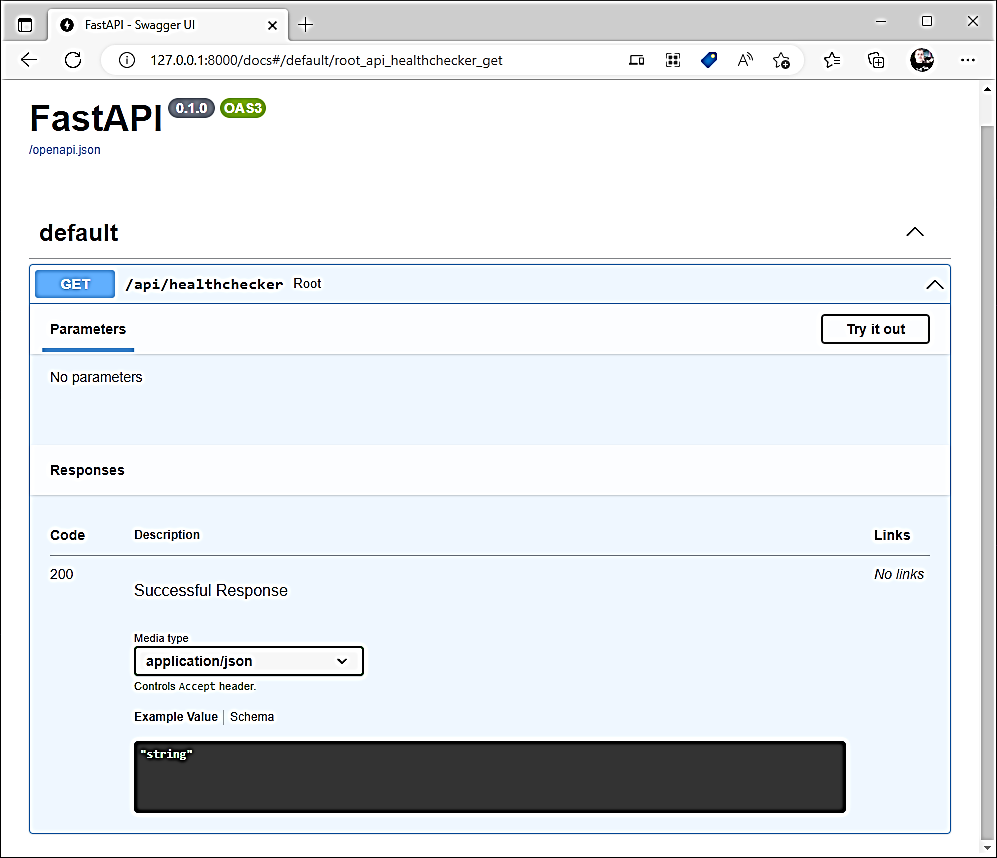
Після запуску сервера, за маршрутом <http://127.0.0.1:8000/api/healthchecker> буде відправлено повідомлення клієнту у форматі JSON



**FastAPI - http://127.0.0.1:8000/api/healthchecker**

**Документація OpenAPI**

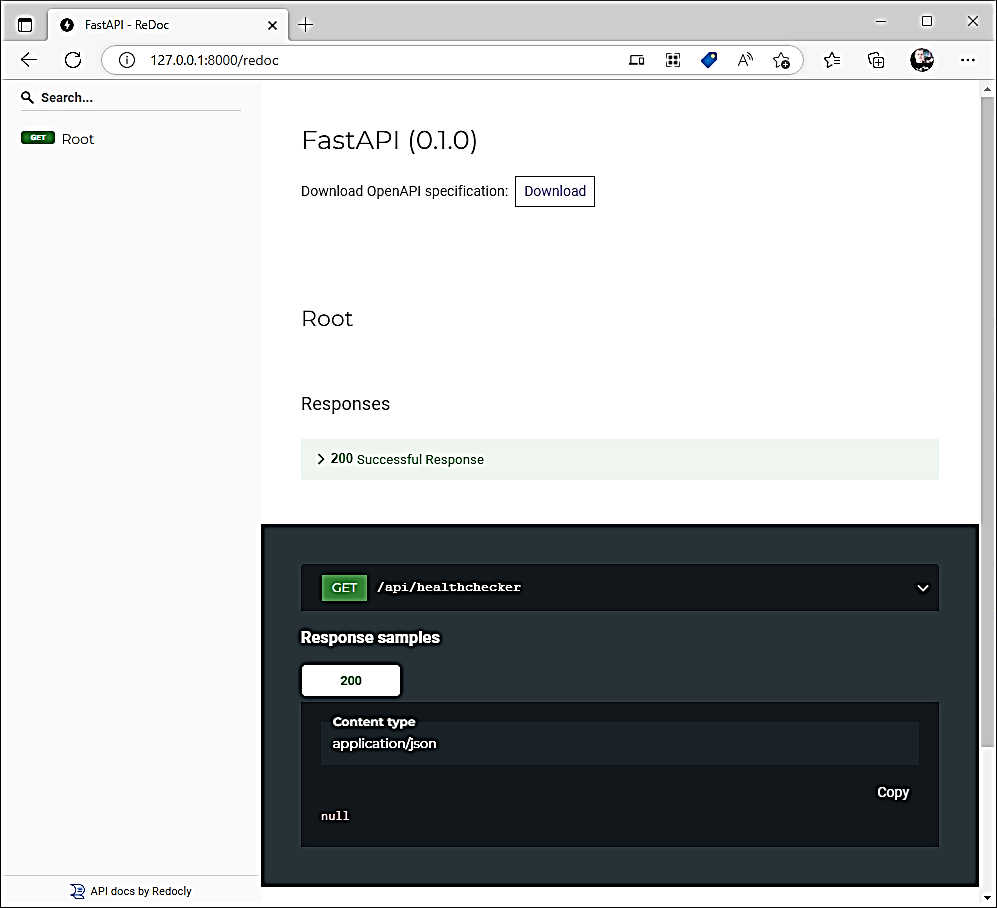
[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-01/intro#%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D1%96%D1%8F-openapi)Як вже зазначалося, FastAPI виконує автоматичну генерацію документації із коду. Це просто величезна перевага, що полегшує життя звичайному розробнику. Перейдемо за маршрутом <http://127.0.0.1:8000/docs>. Тут побачимо автоматично сформовану документацію [Swagger](https://swagger.io/) (https://swagger.io/). Swagger зазвичай використовується у зв'язці з архітектурою [RESTful API](https://en.wikipedia.org/wiki/Representational_state_transfer) (https://en.wikipedia.org/wiki/REST), до вивчення якої ми перейдемо після знайомства з FastAPI.



**Swagger документація проекту**

В подальшому, для взаємодії з нашим API, ми постійно будемо звертатися саме до цієї документації.

Також потрібно знати, що за маршрутом <http://127.0.0.1:8000/redoc> ми маємо документацію [ReDoc](https://redocly.github.io/redoc/) (https://redocly.github.io/redoc/)



**ReDoc документація проекту**

В обох випадках FastAPI створює «схему» зі всіма вашими API, використовуючи стандарт OpenAPI для визначення API. Якби ми створювали документацію вручну, нам би довелось створювати файл у форматі json або yaml. До речі, подивитись на нього можна за маршрутом <http://127.0.0.1:8000/openapi.json>

{

"openapi": "3.0.2",

"info": {

"title": "FastAPI",

"version": "0.1.0"

},

"paths": {

"/api/healthchecker": {

"get": {

"summary": "Root",

"operationId": "root\_api\_healthchecker\_get",

"responses": {

"200": {

"description": "Successful Response",

"content": {

"application/json": {

"schema": {}

}

}

}

}

}

}

}

}

І це прекрасно, що FastAPI мінімізує рутинну роботу у програміста та автоматизує її.

**Отримання та валідація даних**

Отримати дані від клієнтського застосунку на сервері можна кількома способами. Розглянемо їх детальніше та зупинимося на кожному, з урахуванням застосування його у фреймворку FastAPI.

**Параметри шляху (Path)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-01/request#%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8-%D1%88%D0%BB%D1%8F%D1%85%D1%83-path)Path параметр маршруту - це параметр, який визначає частину URL, що повинна збігатися з визначеним значенням під час виклику маршруту. Path параметри позначаються за допомогою фігурних дужок {} та можуть використовуватися для отримання динамічного контенту, наприклад, під час запиту конкретного користувача або запису в БД. Наприклад, маршрут /notes/{note\_id} використовує path параметр note\_id для ідентифікації конкретного запису в запиті.

@app.get("/notes/{note\_id}")

async def read\_note(note\_id: int):

return {"note": note\_id}

Тут ми, наприклад, запитуємо запис за його номером note\_id, наприклад, запит за маршрутом <http://127.0.0.1:8000/notes/10> призведе до того, що FastAPI помістить в параметр note\_id ціле значення 10 і ми отримаємо наступну відповідь:

{

"note": 10

}

Оскільки ми чітко вказали, що параметр note\_id має тип int, то окрім того, що ми отримаємо саме ціле число в note\_id, у нас з'являється відразу вбудована валідація даних. Наприклад, спробуйте передати в маршруті не число, а рядок foo — <http://127.0.0.1:8000/notes/foo>. Ми отримаємо повідомлення про помилку:

{

"detail": [

{

"loc": ["path", "note\_id"],

"msg": "value is not a valid integer",

"type": "type\_error.integer"

}

]

}

Вся перевірка даних виконується за допомогою Pydantic без втручання користувача.

Це достатньо простий та поширений спосіб взаємодії клієнта та сервера, але тут необхідно відзначити момент, коли можна зробити помилку — неправильно розмістити обробник для різних операцій.

Наприклад, ми розміщуємо обробник read\_new\_notes для отримання всіх нових записів за маршрутом /notes/new, відразу після обробника read\_note.

@app.get("/notes/{note\_id}")

async def read\_note(note\_id: int):

return {"note": note\_id}

@app.get("/notes/new")

async def read\_new\_notes():

return {"message": "Return new notes"}

Так ми робимо помилку, що ніколи не отримаємо відповідь від функції read\_new\_notes. Відправивши запит за маршрутом /notes/new, ми отримаємо попередню помилку валідації від обробника read\_note:

{

"detail": [

{

"loc": ["path", "note\_id"],

"msg": "value is not a valid integer",

"type": "type\_error.integer"

}

]

}

Як бачимо, спрацював обробник маршруту /notes/{note\_id}. Так відбувається тому, що FastAPI перебирає маршрути послідовно вниз по коду, а наш окремий випадок з маршрутом /notes/new цілком підходить для обробки функцією read\_post. Тому, щоб уникнути помилок маршрутизації, всі окремі випадки повинні розміщуватися перед загальними, навіть якщо вони додаються й після за часом.

@app.get("/note/new")

async def read\_new\_notes():

return {"message": "Return new notes"}

@app.get("/notes/{note\_id}")

async def read\_note(note\_id: int):

return {"note": note\_id}

Відповідь:

{

"message": "Return new notes"

}

Також можна оголосити більше перевірок для параметра шляху за допомогою класу Path.

from fastapi import FastAPI, Path

app = FastAPI()

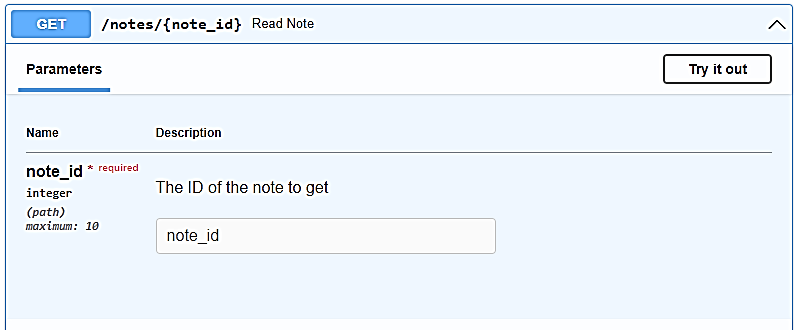
...

@app.get("/notes/{note\_id}")

async def read\_note(note\_id: int = Path(description="The ID of the note to get", gt=0, le=10)):

return {"note": note\_id}

Тут, для документації, ми надаємо пояснення для параметра note\_id за допомогою атрибута description та обмежуємо прийняте значення від 1 до 10.



**Валідація параметра шляху**

Ось деякі параметри для валідації, які може приймати клас Path:

* gt — значення повинно бути більшим,
* ge — значення повинно бути більшим або дорівнювати,
* lt — значення повинно бути меншим,
* le — значення повинно бути меншим або дорівнювати,
* min\_length — значення повинно мати мінімальну довжину (для рядків),
* max\_length — значення повинно мати максимальну довжину (для рядків),
* regex — значення повинно задовольняти регулярний вираз.

**Параметри запиту (Query)**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-01/request#%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8-%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D1%83-query)

Параметри запиту (Query) - це набір пар "ключ=значення", що йдуть після URL-адреси після знаку ? та розділені символом &. Вони використовуються для передачі додаткової інформації у запиті, наприклад, фільтри, пагінація та сортування.

Наприклад, в наступній URL-адресі ми передаємо два ключі skip та limit зі значеннями 0 і 10 відповідно:

http://127.0.0.1:8000/notes/?skip=0&limit=10

INFO

* skip - скільки записів потрібно пропустити з початку
* limit - кількість записів, що виводяться

Вони є частиною URL-адреси і є «звичайними» рядками. Але коли ви оголошуєте їх з типами Python, вони перетворюються на цей тип та перевіряють на його відповідність.

Оскільки параметри запиту не є фіксованою частиною шляху, вони можуть бути необов'язковими та мати значення за замовчуванням.

@app.get("/notes")

async def read\_notes(skip: int = 0, limit: int = 10):

return {"message": f"Return all notes: skip: {skip}, limit: {limit}"}

INFO

При оголошенні інших параметрів функції обробника, які не є частиною шляхів, вони автоматично інтерпретуються як параметри запиту. Якщо говорити простіше, то оскільки у маршруті /notes жодних параметрів немає, то параметри skip і limit у функції read\_notes будуть інтерпретовані як параметри запиту (Query).

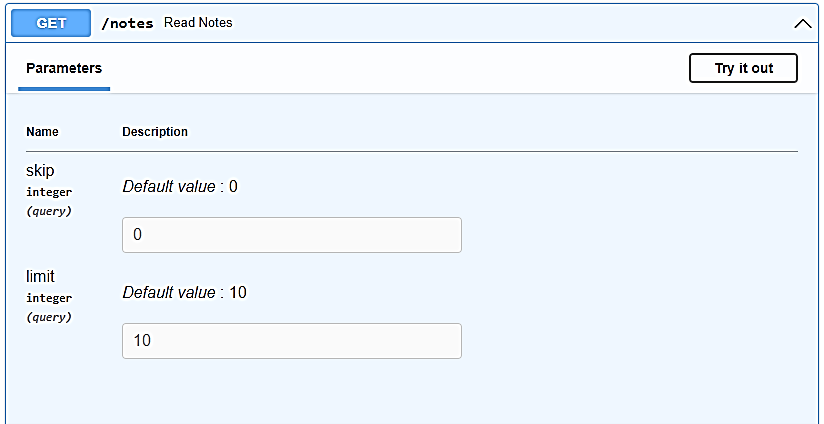
Відповідь:

{

"message": "Return all notes: skip: 0, limit: 10"

}

У документації ці параметри також відображаються:



**Параметри запиту (Query)**

INFO

Якщо ми хочемо зробити певний параметр запиту не обов'язковим, необхідно встановити для нього значення за замовчуванням: None. Наприклад, для наступного прикладу параметр q в маршруті передавати необов'язково:

@app.get("/notes")

async def read\_notes(skip: int = 0, limit: int = 10, q: str | None = None):

return {"message": f"Return all notes: skip: {skip}, limit: {limit}"}

Але, якщо ви хочете зробити параметр запиту обов'язковим, ви просто не оголошуєте жодного значення за замовчуванням.

FastAPI дозволяє оголошувати додаткову інформацію та перевірку параметрів. Для цього потрібно імпортувати Query з fastapi. Наприклад, встановимо, що значення limit може лежати в межах від 10 до 100.

from fastapi import FastAPI, Path, Query

app = FastAPI()

...

@app.get("/notes")

async def read\_notes(skip: int = 0, limit: int = Query(default=10, le=100, ge=10)):

return {"message": f"Return all notes: skip: {skip}, limit: {limit}"}

Параметри для валідації, які приймає клас Query, схожі з параметрами для Path. Щоб отримати більше інформації, прочитайте відповідний [розділ документації](https://fastapi.tiangolo.com/tutorial/query-params-str-validations/) (https://fastapi.tiangolo.com/tutorial/query-params-str-validations/).

**Тіло запиту (Body)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-01/request#%D1%82%D1%96%D0%BB%D0%BE-%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D1%83-body)Тіло запиту - це дані, які передаються разом з HTTP запитом на сервер. Вони можуть бути у форматі JSON, XML, або іншому форматі, та використовуються для передачі додаткової інформації від клієнта до сервера, наприклад, форми даних, файли або дані для створення або зміни об'єкта на сервері.

Тіло запиту відправляється разом із HTTP заголовком Content-Type, який вказує на формат даних у тілі запиту. Наприклад, якщо дані у форматі JSON, то заголовок Content-Type буде встановлений як application/json.

Тіло запиту може бути надіслано за допомогою різних операцій HTTP, таких як POST, PUT і PATCH, що використовуються для створення або зміни даних на сервері.

FastAPI надає можливість отримати тіло запиту за допомогою декоратора @app.post() або @app.put() для оголошення маршруту, та додаючи аргумент для відповідного типу даних, який очікується у тілі запиту. Ось приклад функції, що обробляє POST-запит із тілом запиту у форматі JSON:

from fastapi import FastAPI, Path, Query

from pydantic import BaseModel

app = FastAPI()

...

class Note(BaseModel):

name: str

description: str

done: bool

@app.post("/notes")

async def create\_note(note: Note):

return {"name": note.name, "description": note.description, "status": note.done}

В цьому прикладі ми створюємо клас Note, який визначає три поля: name (рядок str), description ( рядок str) та done (булеве значення bool). Потім визначаємо функцію create\_note, яка приймає аргумент note типу Note. Функція повертає словник {"name": note.name, "description": note.description, "status": note.done}.

Декоратор @app.post("/notes") реєструє функцію create\_note в якості обробника POST-запитів на маршрут /notes.

Відправлення тіла запиту можна здійснити за допомогою різних бібліотек та Frontend фреймворків, таких як requests в Python, axios в JavaScript, або HttpClient в .NET. У цих бібліотеках є методи для відправлення тіла запиту разом з іншою інформацією: заголовками та параметрами.

Наприклад, в Python за допомогою використання бібліотеки requests можна відправити тіло запиту у форматі JSON наступним чином:

import json

import requests

data = {"name": "Test note", "description": "Testing", "done": False}

headers = {'Content-type': 'application/json'}

response = requests.post('http://127.0.0.1:8000/notes', data=json.dumps(data), headers=headers)

print(response.json())

На цьому прикладі ми відправляємо POST-запит на URL http://127.0.0.1:8000/notes разом з тілом запиту у форматі JSON та заголовком Content-type: application/json. FastAPI автоматично валідує тіло запиту та перетворює його в об'єкт типу Note. Потім функція create\_note буде викликана разом з аргументом note, що містить в собі об'єкт Note з полями name="Test note", "description": "Testing" та "done": False.

Клієнт повинен отримати наступне виведення:

{'name': 'Test note', 'description': 'Testing', 'status': False}

INFO

Frontend фреймворк - це набір бібліотек та інструментів, які використовуються для розробки веб-інтерфейсів. Зазвичай, вони містять функції для роботи з HTML, CSS та JavaScript, а також надають різні компоненти, такі як кнопки, форми, таблиці тощо, які можна використовувати в своєму застосунку.

Існує велика кількість різноманітних frontend фреймворків, кожний з яких має свої особливості та можливості. Ось деякі із найпопулярніших frontend фреймворків:

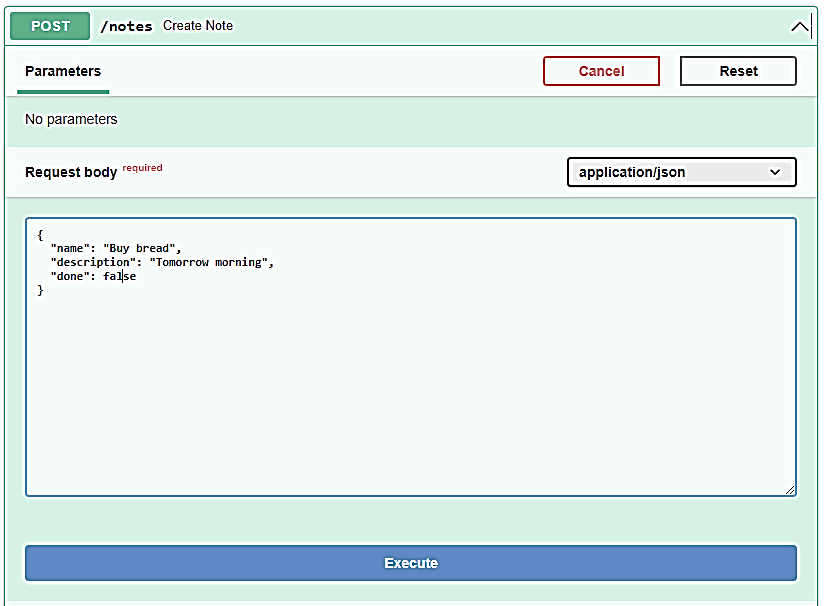
* React
* Angular
* Vue.js
* Ember.js
* Svelte

Валідація даних у цьому прикладі виконується за допомогою модуля pydantic. Клас Note, визначений в цьому прикладі, є підкласом BaseModel з pydantic. Він використовує анотації типів, щоб визначити поля name, description і done.

При обробці запиту FastAPI викликає метод валідації в об'єкта Note. Цей метод перевіряє, що передані дані відповідають зазначеним типам та встановлює атрибути об'єкта відповідним чином. Якщо ж дані невалідні (наприклад, поле "done" містить не булевий вираз), то буде викинутий виняток ValidationError з відповідним повідомленням.

Цей метод валідації дуже зручний, оскільки він автоматично валідує вхідні дані та перетворює їх на об'єкт вказаного типу. Ви можете вказати різні обмеження для полів, такі як мінімальне або максимальне значення, необхідність наявності поля тощо.

Перейдіть за маршрутом http://127.0.0.1:8000/docs та спробуйте відправити POST запит на сервер:



**Відправляємо тіло запиту**

Наприклад, відправимо тіло запиту наступного виду:

{

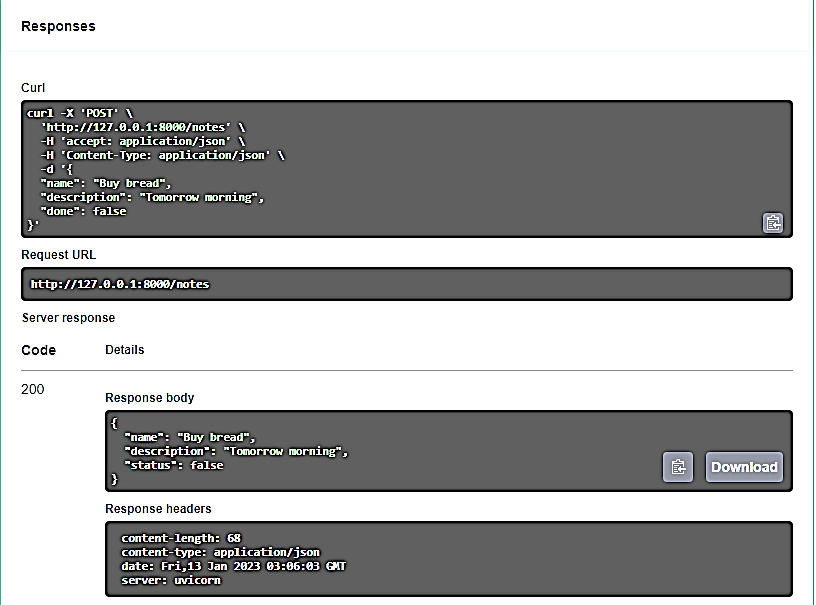
"name": "Buy bread",

"description": "Tomorrow morning",

"done": false

}

Як бачимо, документація Swagger теж є клієнтом для нашого сервера. Отримаємо очікувану відповідь.



**Отримуємо відповідь**

**Валідація даних Pydantic**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-01/request#%D0%B2%D0%B0%D0%BB%D1%96%D0%B4%D0%B0%D1%86%D1%96%D1%8F-%D0%B4%D0%B0%D0%BD%D0%B8%D1%85-pydantic)Як бачимо, завдяки pydantic FastAPI знімає більшість питань при валідації вхідних даних, а значить зменшує ймовірність людського фактору. Ось приклад складнішої валідації даних із використанням pydantic:

from typing import Optional

from pydantic import BaseModel, Field, EmailStr, HttpUrl

class User(BaseModel):

name: str

email: EmailStr

website: HttpUrl

age: Optional[int] = Field(None, ge=13, le=90)

friends: Optional[int] = 0

user = User(name="John", email="john@example.com", website="https://john.com", age=25, friends=10)

print(user)

# Output: User(name='John', email='john@example.com', website='https://john.com', age=25, friends=10)

# Validation error (age is below the minimum)

user = User(name="Jane", email="jane@example.com", website="https://jane.com", age=12)

print(user)

# Output: pydantic.error\_wrappers.ValidationError: 1 validation error for User

# age

# ensure this value is greater than or equal to 13 (type=value\_error.number.not\_ge; limit\_value=13)

# Validation error (website is not a valid URL)

user = User(name="Bob", email="bob@example.com", website="invalid url", age=20)

print(user)

# Output: pydantic.error\_wrappers.ValidationError: 1 validation error for User

# website

# invalid or missing URL scheme (type=value\_error.url.scheme)

В цьому прикладі ми створюємо клас User з полями name (рядок), email (рядок з валідною адресою електронної пошти), website (рядок з валідним URL), age (ціле число, необов'язкове, оскільки ми присвоюємо йому значення None) і friends (ціле число, теж необов'язкове, зі значенням за замовчуванням 0).

Ми також вказуємо деякі обмеження у класі Field: ge=13 і le=90. Це означає, що значення поля age повинно бути в діапазоні від 13 до 90.

При створенні екземпляра класу User з коректними значеннями всі поля валідуються та екземпляр створюється успішно. Однак, якщо якесь поле невалідне (наприклад, age дорівнює 12, що є меншим за мінімально допустиме значення 13), то отримаємо наступний виняток ValidationError з відповідним повідомленням.

В pydantic також є безліч інших типів даних, які можна використовувати для валідації, такі як UUID, дата і час, паролі тощо.

Подивіться[**документацію** pydantic](https://docs.pydantic.dev/) (https://docs.pydantic.dev/latest/), щоб дізнатися детальніше про всі доступні типи даних та їх можливості.

Зверніть увагу, що валідація даних із використанням pydantic лише визначає, що дані відповідають зазначеним типам та обмеженням. Вона не перевіряє, чи дані коректні у ширшому розумінні. Наприклад, якщо ви хочете перевірити, чи email є унікальним в системі, то для цього вам потрібно буде написати спеціальну функцію.

**Підключення до бази даних**

З FastAPI ви можете використовувати будь-яку реляційну базу даних, яка вам знадобиться. Ми ж будемо використовувати SQLAlchemy та PostgreSQL.

Підключення SQLAlchemy до FastAPI може відбуватися в декілька кроків.

Запустіть докер-контейнер з базою даних PostgreSQL:

docker run --name db-postgres -p 5432:5432 -e POSTGRES\_PASSWORD=567234 -d postgres

Необхідно встановити залежності: SQLAlchemy та драйвер бази даних psycopg2 для PostgreSQL, використовуючи poetry.

poetry add sqlalchemy psycopg2

Далі створимо файл для підключення до бази даних db.py з наступним вмістом:

**db.py**

from sqlalchemy import create\_engine, Column, Integer, String, Boolean

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = "postgresql+psycopg2://postgres:567234@localhost:5432/fast\_db"

engine = create\_engine(SQLALCHEMY\_DATABASE\_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

class Note(Base):

\_\_tablename\_\_ = "notes"

id = Column(Integer, primary\_key=True, index=True)

name = Column(String(50))

description = Column(String(250))

done = Column(Boolean, default=False)

Base.metadata.create\_all(bind=engine)

# Dependency

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

В коді вище ми створили двигун SQLAlchemy engine та підключили його до бази даних за допомогою URL-адреси бази даних. Далі створюємо сесію за допомогою sessionmaker та визначаємо базовий клас для наших моделей. Також створюємо модель Note, яка містить інформацію про нотатку, та використаємо declarative\_base для створення таблиці в базі даних із відповідними полями.

Потім ми створюємо функцію get\_db, яка повертає нову сесію та використовує контекстний менеджер yield для автоматичного закриття сесії після використання. У функціях роутера можна використовувати її як: db: Session = Depends(get\_db) для отримання сесії та здійснення запитів до бази даних.

Вираз db: Session = Depends(get\_db) - це аргумент функції, який оголошує змінну db типу Session та присвоює їй значення, яке повертає функція get\_db().

Depends - це декоратор, який використовується для надання вказівки FastAPI, що аргумент залежить від зовнішнього джерела і повинен бути отриманий з нього.

В цьому випадку функція get\_db повертає сесію, яка використовується для взаємодії з базою даних.

Коли ви викликаєте функцію цим аргументом, то FastAPI викличе get\_db() для отримання сесії та передасть її у функцію як аргумент db. Це дозволяє вам використовувати сесію всередині функції для виконання запитів до бази даних та обробки результатів.

Давайте розглянемо приклад, як ми можемо виконати запит від обробника маршруту:

from fastapi import FastAPI, Path, Query, Depends, HTTPException

from pydantic import BaseModel

from sqlalchemy.orm import Session

from db import get\_db

app = FastAPI()

@app.get("/api/healthchecker")

def healthchecker(db: Session = Depends(get\_db)):

try:

# Make request

result = db.execute("SELECT 1").fetchone()

if result is None:

raise HTTPException(status\_code=500, detail="Database is not configured correctly")

return {"message": "Welcome to FastAPI!"}

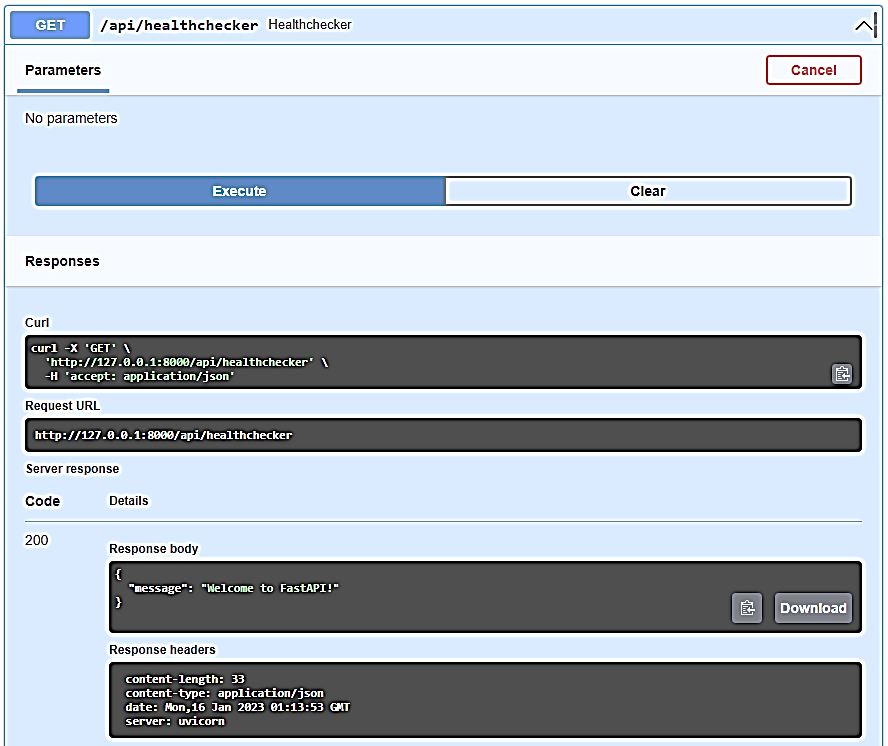
except Exception as e:

print(e)

raise HTTPException(status\_code=500, detail="Error connecting to the database")

...

Цей код створює маршрут /api/healthchecker для перевірки стану бази даних. Функція healthchecker отримує сесію з базою даних за допомогою встановленої залежності Depends(get\_db), виконує запит SELECT 1 та обробляє результат. Якщо результат не знайдено, тоді генерується виняток з кодом 500 та повідомленням "Database is not configured correctly". Якщо під час запиту сталася помилка, тоді генерується виняток з кодом 500 та повідомленням "Error connecting to the database"



**Перевірка стану бази даних**

Давайте тепер перепишемо операцію створення нотатки для збереження в базі даних:

**main.py**

from fastapi import FastAPI, Path, Query

from pydantic import BaseModel

from db import get\_db, Note

app = FastAPI()

...

class NoteModel(BaseModel):

name: str

description: str

done: bool

@app.post("/notes")

async def create\_note(note: NoteModel, db: Session = Depends(get\_db)):

new\_note = Note(name=note.name, description=note.description, done=note.done)

db.add(new\_note)

db.commit()

db.refresh(new\_note)

return new\_note

В коді вище ми використовуємо створену раніше модель NoteModel для опису даних, які повинні бути передані у функцію create\_note. Функція приймає note як аргумент, який відповідає моделі NoteModel і використовує його для створення нового запису в базі даних. Потім функція використовує db сесію для створення нового запису, завершення транзакції та оновлення нового запису db.refresh(new\_note). Нарешті, вона повертає новий запис як відповідь.

Перевіримо працездатність коду, створивши запис виду:

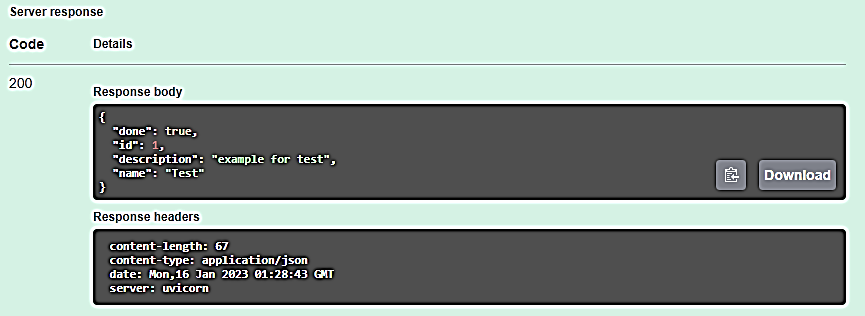
{

"name": "Test",

"description": "example for test",

"done": true

}



**Результат POST операції**

Як бачимо, все вдалося.

**Модель відповіді**

Перепишемо код функцій обробників операцій: отримання всіх записів read\_notes та записи за її ідентифікатором read\_note з урахуванням підключення бази даних:

@app.get("/notes")

async def read\_notes(skip: int = 0, limit: int = Query(default=10, le=100, ge=10), db: Session = Depends(get\_db)):

notes = db.query(Note).offset(skip).limit(limit).all()

return notes

@app.get("/notes/{note\_id}")

async def read\_note(note\_id: int = Path(description="The ID of the note to get", gt=0, le=10),

db: Session = Depends(get\_db)):

note = db.query(Note).filter(Note.id == note\_id).first()

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail='Not found')

return note

В першій функції read\_notes створений обробник маршруту для кінцевої точки /notes, вона використовується для витягування списку нотаток з бази даних. Вона приймає два необов'язкових параметри запиту skip та limit, що використовуються для визначення зсуву та обмеження кількості нотаток, що повертаються. Параметр skip використовується для визначення кількості записів, які потрібно пропустити під час вибірки, а параметр limit використовується для визначення максимальної кількості нотаток, що повертаються. Значення за замовчуванням skip=0 та limit=10.

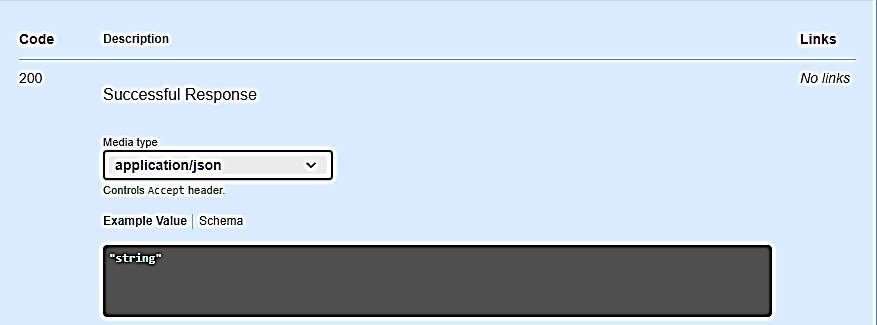
Друга функція read\_note — це обробник маршруту для кінцевої точки /notes/{note\_id}. Вона використовується для витягування однієї нотатки з бази даних за ідентифікатором нотатки. Він приймає параметр маршруту note\_id, який використовується для позначення ідентифікатора отриманої нотатки. Він використовує цей ідентифікатор для фільтрації таблиці нотаток та витягування першої нотатки, яка відповідає вказаному ідентифікатору: метод first. Якщо нотатку не знайдено, виникає виняток HTTPException з кодом стану 404.

Тут потрібно пояснити про об'єкт status, що є частиною коду fastapi і містить визначені коди стану HTTP в якості констант. Ці константи можна використовувати для позначення коду стану HTTP у відповіді HTTP. Наприклад, у функції read\_note ми використовуємо status.HTTP\_404\_NOT\_FOUND, щоб вказати, що ресурс, який ми шукаємо, не знайдено. Це скорочений спосіб позначення коду стану 404, який робить код читабельнішим та зручнішим в обслуговування. HTTPException — це вбудований виняток fastapi для повернення відповіді про помилку із зазначеним кодом стану HTTP та подробицями повідомлення.

INFO

Ви можете використовувати й інші коди стану з fastapi.status, такі як HTTP\_200\_OK, HTTP\_201\_CREATED, HTTP\_204\_NO\_CONTENT, HTTP\_400\_BAD\_REQUEST тощо.

Якщо зараз ми подивимось документацію до цих функцій, то побачимо, що там немає опису об'єкта, який вони повертають.



**Перевірка стану бази даних**

Зараз там є просто опис "string", тоді як відповідь, наприклад, для функції read\_note - це об'єкт із створеною нотаткою із попереднього прикладу:

{

"name": "Test",

"id": 1,

"description": "example for test",

"done": true

}

Для створення моделей відповіді ми можемо використовувати анотації типів так само, як і для вхідних даних у параметрах функцій, ви можете використовувати pydantic моделі, списки, словники, скалярні значення, такі як цілі числа, логічні значення тощо.

Визначимо модель відповіді функції read\_notes:

class ResponseNoteModel(BaseModel):

id: int = Field(default=1, ge=1)

name: str

description: str

done: bool

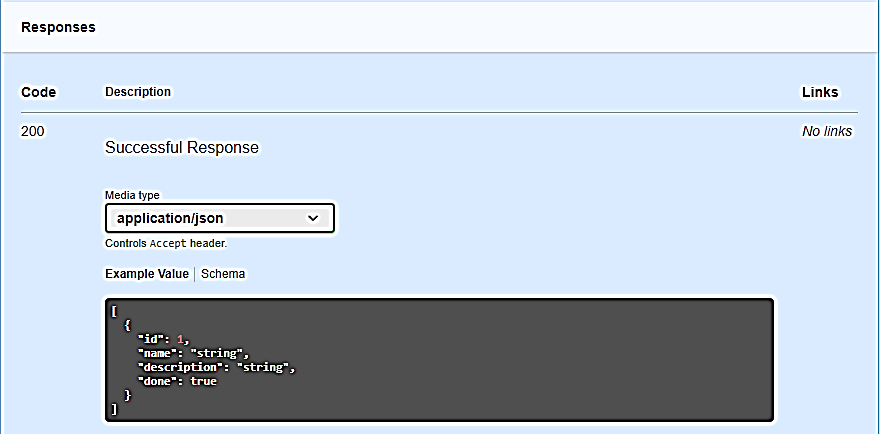
@app.get("/notes")

async def read\_notes(skip: int = 0, limit: int = Query(default=10, le=100, ge=10), db: Session = Depends(get\_db)) -> list[ResponseNoteModel]:

notes = db.query(Note).offset(skip).limit(limit).all()

return notes

Тепер у документації ми побачимо модель відповіді:



**Модель відповіді функції read\_notes**

Також в FastAPI ви можете використовувати декоратори роутерів, щоб встановити властивості відповіді автоматично. Наприклад, ви можете встановити модель відповіді за допомогою аргументів декоратору:

@app.get("/notes/{note\_id}", response\_model=ResponseNoteModel)

async def read\_note(note\_id: int = Path(description="The ID of the note to get", gt=0, le=10),

db: Session = Depends(get\_db)):

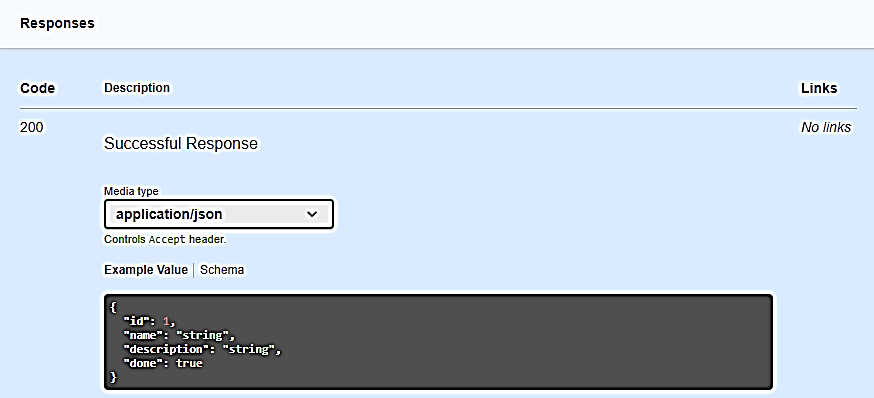
note = db.query(Note).filter(Note.id == note\_id).first()

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail='Not found')

return note

Перевіряємо в документації, що для функції read\_note модель відповіді також правильна:



**Модель відповіді функції read\_note**

Підіб'ємо підсумки. В цьому коді ви додали декоратор @app.get("/notes/{note\_id}", response\_model=ResponseNoteModel) до функції read\_note. Параметр response\_model використовується для позначення моделі Pydantic, яку варто використовувати для перевірки даних відповіді. Це означає, що дані відповіді, які повертаються функцією read\_note, будуть перевірені на відповідність моделі ResponseNoteModel Pydantic. Якщо дані відповіді недійсні відповідно до моделі, *виникне помилка*!

За допомогою цього декоратора ми також вказуємо, що кінцева точка повинна використовувати ResponseNoteModel для перевірки даних відповіді, що є досить хорошою практикою, оскільки дозволяє гарантувати, що дані відповіді відповідають потрібному формату та обмеженням.

TIP

Використання моделей відповіді допомагає нам виявити помилки на ранній стадії, перш ніж вони потраплять до клієнта. Це також дозволяє нам задокументувати очікуваний формат відповіді.

Далі в конспекті ми дотримуватимемося додавання моделі відповіді за допомогою декоратора.

Повний код прикладу

**main.py**

from fastapi import FastAPI, Path, Query, Depends, HTTPException, status

from pydantic import BaseModel, Field

from sqlalchemy.orm import Session

from db import get\_db, Note

app = FastAPI()

class ResponseNoteModel(BaseModel):

id: int = Field(default=1, ge=1)

name: str

description: str

done: bool

@app.get("/api/healthchecker")

def healthchecker(db: Session = Depends(get\_db)):

try:

# Здійснюємо запит

result = db.execute("SELECT 1").fetchone()

if result is None:

raise HTTPException(status\_code=500, detail="Database is not configured correctly")

return {"message": "Welcome to FastAPI!"}

except Exception as e:

print(e)

raise HTTPException(status\_code=500, detail="Error connecting to the database")

@app.get("/notes")

async def read\_notes(skip: int = 0, limit: int = Query(default=10, le=100, ge=10), db: Session = Depends(get\_db)) -> \

list[ResponseNoteModel]:

notes = db.query(Note).offset(skip).limit(limit).all()

return notes

@app.get("/notes/{note\_id}", response\_model=ResponseNoteModel)

async def read\_note(note\_id: int = Path(description="The ID of the note to get", gt=0, le=10),

db: Session = Depends(get\_db)):

note = db.query(Note).filter(Note.id == note\_id).first()

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail='Not found')

return note

class NoteModel(BaseModel):

name: str

description: str

done: bool

@app.post("/notes")

async def create\_note(note: NoteModel, db: Session = Depends(get\_db)):

new\_note = Note(name=note.name, description=note.description, done=note.done)

db.add(new\_note)

db.commit()

db.refresh(new\_note)

return new\_note

**Обробка помилок**

Обробка помилок - це важлива складова будь-якого API, а FastAPI забезпечує різні способи обробки помилок.

Одним із таких способів є використання винятків. У FastAPI ви можете "викинути" виняток HTTPException для повернення відповіді про помилку.

Наприклад, ви можете викинути виняток зі статусом 404 Not Found, якщо ресурс не знайдено:

@app.get("/items/{item\_id}")

async def read\_item(item\_id: int):

item = {"item\_id": item\_id, "name": "Foo"}

if item is None:

raise HTTPException(status\_code=404, detail="Item not found")

return item

В цьому прикладі ми створюємо функцію read\_item, яка повертає об'єкт із заданим item\_id, якщо він існує, або викидує виняток HTTPException зі статусом 404 Not Found, якщо ресурс не знайдено.

Також можна використовувати виняток для обробки інших типів помилок як, наприклад, невалідні дані, помилки аутентифікації, тощо. Наприклад, ви можете викинути виняток зі статусом 400 Bad Request, якщо валідація даних не пройдена:

@app.post("/items")

async def create\_item(item: Item):

if item.name is None:

raise HTTPException(status\_code=400, detail="Name is required")

return item

В цьому прикладі ми створюємо функцію create\_item, яка приймає об'єкт Item в якості аргументу та валідує його. Якщо поле name дорівнює None, ми викидуємо виняток зі статусом 400 Bad Request.

Ще одним способом обробки помилок є використання

декоратора @app.exception\_handler(). За допомогою цього декоратора ви можете вказати функцію, яка буде викликатися при виникненні винятків у всіх фукціях роутерів:

@app.exception\_handler(HTTPException)

def handle\_http\_exception(request: Request, exc: HTTPException):

return {"message": str(exc.detail)}

В цьому прикладі ми створюємо фукцію handle\_http\_exception, яка приймає об'єкти Request та HTTPException в якості аргументів та повертає об'єкт разом із повідомленням про помилку.

@app.exception\_handler() - це декоратор в FastAPI, який дозволяє вказувати функцію, що буде викликатися у випадку виникнення винятків у всіх функціях роутерів. Він корисний для обробки помилок, що можуть виникнути в різних частинах вашого коду, наприклад, валідація даних, помилки аутентифікації тощо.

Щоб використовувати @app.exception\_handler(), вам потрібно створити функцію, що буде обробляти винятки, та декорувати її за допомогою @app.exception\_handler().

У функції-обробника винятків ви можете використовувати аргументи request та exc для отримання доступу до об'єкта запиту та об'єкта винятку відповідно. Наприклад, ви можете повертати різноманітні повідомлення про помилки, залежно від типу винятку:

@app.exception\_handler(ValidationError)

def validation\_error\_handler(request: Request, exc: ValidationError):

return JSONResponse(

status\_code=status.HTTP\_400\_BAD\_REQUEST,

content={"message": "Invalid input data"}

)

@app.exception\_handler(HTTPException)

def http\_exception\_handler(request: Request, exc: HTTPException):

return JSONResponse(

status\_code=exc.status\_code,

content={"message": exc.detail},

)

@app.exception\_handler(Exception)

def unexpected\_exception\_handler(request: Request, exc: Exception):

return JSONResponse(

status\_code=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR,

content={"message": "An unexpected error occurred"},

)

В цьому прикладі ми створюємо три функції-обробника винятків: одну - для обробки помилок валідації даних, одну - для обробки HTTP-винятків та одну - для обробки неочікуваних винятків. Кожна функція декорована @app.exception\_handler(), вказуючи тип винятку, який вона буде опрацьовувати.

Після того, як ви створите функцію-обробника винятків, ви можете використовувати їх в своєму коді, викидуючи винятки в тих місцях, де хочете опрацювати помилку. Наприклад:

@app.post("/items/")

async def create\_item(item: Item):

if item.price < 0:

raise HTTPException(

status\_code=status.HTTP\_400\_BAD\_REQUEST,

detail="Price should be a positive number",

)

return item

В цьому прикладі ми викидуємо виняток HTTPException, якщо ціна товару від'ємна. Після цього FastAPI буде викликати відповідну функцію-обробник винятків через @app.exception\_handler(HTTPException), для обробки цієї помилки.

Ви також можете створювати власні класи винятків для обробки специфічних типів помилок. Наприклад:

class ItemNotFoundError(Exception):

pass

@app.exception\_handler(ItemNotFoundError)

def item\_not\_found\_error\_handler(request: Request, exc: ItemNotFoundError):

return JSONResponse(

status\_code=status.HTTP\_404\_NOT\_FOUND,

content={"message": "Item not found"},

)

@app.get("/items/{item\_id}")

async def read\_item(item\_id: int):

item = get\_item\_by\_id(item\_id)

if item is None:

raise ItemNotFoundError

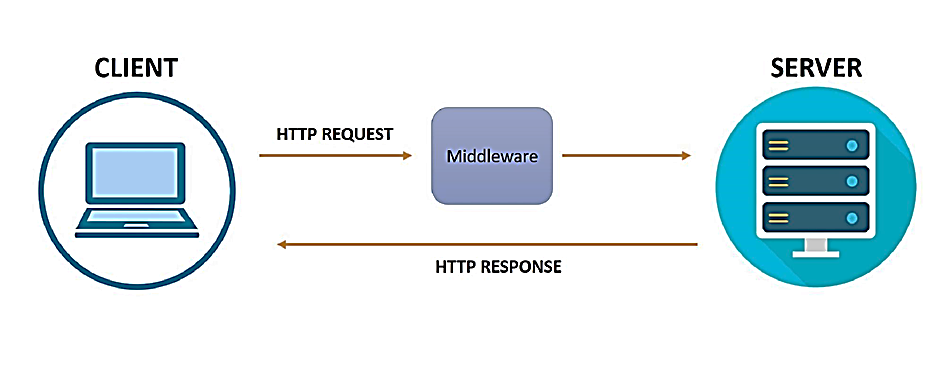
return item

В цьому прикладі ми створюємо клас ItemNotFoundError, який наслідується від класу Exception, та функцію-обробника винятків item\_not\_found\_error\_handler, яка обробляє цей тип винятків. Потім ми викидуємо цей виняток у функції read\_item, якщо товар не знайдений. FastAPI викликатиме item\_not\_found\_error\_handler для обробки цієї помилки.

Обробка помилок за допомогою @app.exception\_handler() може бути дуже корисною для управління помилками та покращення досвіду користувача у вашому застосунку FastAPI. Вона дозволяє легко управляти повідомленнями про помилки та статусами HTTP для різноманітних типів помилок, що, в результаті, може значно покращити документування та спростити налагодження вашого коду.

**Проміжне програмне забезпечення**

Проміжне програмне забезпечення (middleware) - це функція, яка є програмним компонентом. Цей компонент знаходиться між двома кінцевими точками: клієнтом та сервером, виконуючи певні дії з даними, що проходять через нього.



**Проміжне програмне забезпечення (middleware)**

Проміжне ПЗ часто використовується для збільшення функціональності вебзастосунку без зміни його основного коду. Завдяки цьому, застосунок стає модульнішим та простішим в обслуговуванні, а також спрощується процес додавання нових функцій та можливостей.

FastAPI також підтримує проміжне ПЗ. Воно дозволяє вам визначити функції проміжного ПЗ, які будуть виконуватися на певних маршрутах або на всьому шляху до та після обробки запиту функцією кінцевої точки.

У FastAPI можете використати проміжне програмне забезпечення за допомогою декоратора @app.middleware(). Наприклад:

from fastapi import FastAPI

app = FastAPI()

@app.middleware("http")

async def add\_process\_time\_header(request: Request, call\_next):

start\_time = time.time()

response = await call\_next(request)

process\_time = time.time() - start\_time

response.headers["X-Process-Time"] = str(process\_time)

return response

Цей код визначає проміжну функцію add\_process\_time\_header, яка буде виконуватися для всіх HTTP запитів, що обробляються застосунком.

Функція приймає два аргументи:

* request: Request, який є екземпляром fastapi.Request та містить вхідні дані запиту.
* call\_next виступає в ролі функції, що викликається, та яку необхідно викликати для запиту наступного проміжного програмного забезпечення або функції обробника маршруту.

Функція починає з вимірювання поточного часу, а потім викликає функцію call\_next, передаючи запит в якості аргументу. Це викличе наступну функцію проміжного програмного забезпечення в ланцюжку або, якщо функцій проміжного програмного забезпечення далі немає, то викличе функцію обробника маршруту.

Потім функція знову вимірює поточний час та вираховує час початку для розрахунку загального часу обробки. Після цього до об'єкта відповіді додається налаштовуваний заголовок response.headers["X-Process-Time"], який містить загальний час обробки, і відповідь повертається.

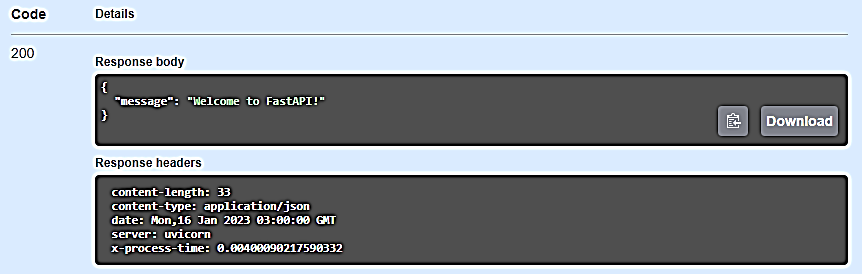
Ця проміжна функція додає до відповіді налаштовуваний заголовок з часом, який знадобився для обробки запиту. Наприклад, це може бути корисним для відлагодження або моніторингу продуктивності застосунку.

Проміжне програмне забезпечення може бути корисним для реалізації різних функцій у вашому застосунку FastAPI, наприклад, аутентифікації, авторизації, моніторингу тощо. Зверніть увагу, що проміжне програмне забезпечення виконується до виконання маршруту, тому його можна використовувати для маніпуляції запитом або обробки помилок перед його обробкою на маршруті.

Ви також можете використовувати проміжне програмне забезпечення для додавання додаткової функціональності до відповіді сервера, наприклад, додавання HTTP-заголовків, модифікації тіла відповіді тощо.

Ви можете створювати будь-яку кількість проміжного програмного забезпечення та застосовувати його до необхідних маршрутів або типів запитів. Це може бути корисно для розподілу різноманітних функцій вашого застосунку та спрощення його обслуговування або відлагодження.

Додамо код проміжного програмного забезпечення до попереднього прикладу та виконаємо запит на маршрут /api/healthchecker за допомогою Swagger документації.



**Проміжне програмне забезпечення (middleware)**

Як бачимо, все працює, а у відповіді присутній наш доданий заголовок x-process-time з часом виконання запиту.

**Робота з формами та файлами**

**Завантаження файлів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-01/formdata#%D0%B7%D0%B0%D0%B2%D0%B0%D0%BD%D1%82%D0%B0%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D1%84%D0%B0%D0%B9%D0%BB%D1%96%D0%B2)Щоб дозволити користувачам завантажувати файли у ваш застосунок за допомогою FastAPI, нам також знадобиться пакет python-multipart:

poetry add python-multipart

FastAPI надає вбудовану підтримку для обробки завантаження файлів. Ви можете використовувати класи File та Form з модуля fastapi для обробки файлів.

Для завантаження файлів можна використовувати тип File в параметрах функції-обробника маршруту, наприклад:

from fastapi import FastAPI, File, UploadFile

import pathlib

app = FastAPI()

@app.post("/uploadfile/")

async def create\_upload\_file(file: UploadFile = File()):

pathlib.Path("uploads").mkdir(exist\_ok=True)

file\_path = f"uploads/{file.filename}"

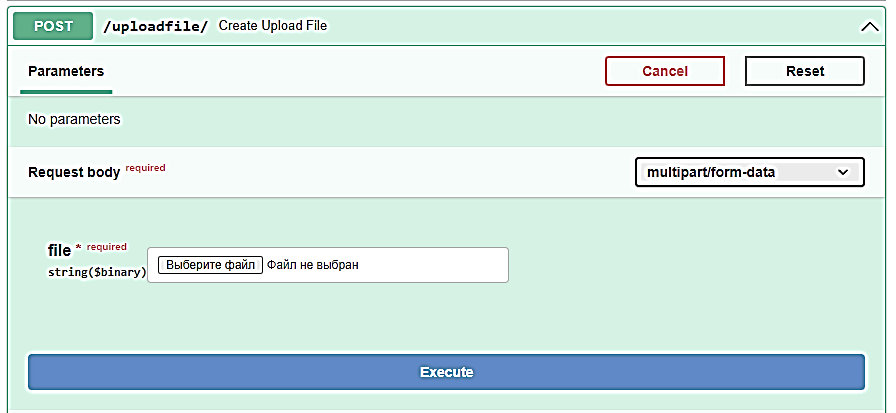
with open(file\_path, "wb") as f:

f.write(await file.read())

return {"file\_path": file\_path}

В цьому прикладі ми використовуємо UploadFile для отримання файлів із запиту. Потім ми перевіряємо наявність директорії uploads, якщо вона відсутня, то створюємо її. Потім зберігаємо файл в цю директорію з іменем, отриманим з файлу. А в кінці ми повертаємо маршрут до файлу як результат функції.

Перейдемо до документації та виконаємо завантаження файлу:



**Проміжне програмне забезпечення (middleware)**

Після виконання операції в корені проекту повинна з'явитися папка uploads, а всередині - завантажений файл.

Варто відзначити, що в цьому прикладі ми зберігаємо файли на диску сервера, і, якщо ваш застосунок планує обслуговувати велику кількість завантажень або великі файли, то слід розглянути можливість використання спеціальних сервісів зберігання файлів, наприклад AWS S3, Google Cloud Storage або Cloudinary. Це дозволить уникнути проблем з обмеженням місця на диску сервера та покращить продуктивність вашого застосунку.

Також варто зауважити, що під час роботи з файлами потрібно слідкувати за безпекою та перевіряти права доступу до файлів, не варто дозволяти завантажувати або завантажувати файли з кореневої директорії вашого проекту, щоб уникнути випадкового розкриття чутливої інформації.

**Статичні ресурси**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-01/formdata#%D1%81%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D1%96-%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D0%B8)Статичні ресурси - це файли, що не змінюються під час виконання застосунку та не залежать від контексту або вхідних даних. Наприклад, статичними ресурсами можуть бути файли зображень, стильові таблиці, скрипти тощо. Ці файли зазвичай не змінюються та є однаковими для кожного запиту, саме тому сервер може швидко та ефективно віддавати їх клієнту.

FastAPI надає можливість легко створювати маршрути для статичних ресурсів за допомогою класу StaticFiles. Цей клас дозволяє створювати маршрути для статичних даних, що знаходяться в певній директорії.

from fastapi import FastAPI

from fastapi.staticfiles import StaticFiles

app = FastAPI()

app.mount("/static", StaticFiles(directory="static"), name="static")

В цьому прикладі ми використовуємо метод mount для створення маршруту /static, який буде відповідати директорії static на диску. Таким чином, якщо в директорії static знаходиться файл image.jpg, то він буде доступний за адресою http://localhost:8000/static/image.jpg

**Завантаження файлів із форми HTML**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-01/formdata#%D0%B7%D0%B0%D0%B2%D0%B0%D0%BD%D1%82%D0%B0%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D1%84%D0%B0%D0%B9%D0%BB%D1%96%D0%B2-%D1%96%D0%B7-%D1%84%D0%BE%D1%80%D0%BC%D0%B8-html)Для реалізації завантаження файлів за допомогою форми HTML ви можете використати той самий підхід, що і в попередньому прикладі, але у вашій HTML формі вкажіть enctype як multipart/form-data.

Наприклад, ось як можна створити HTML форму для завантаження одного файлу та помістити її у файл static/index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

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

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

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

</form>

</body>

</html>

Тепер код нашого застосунку повинен виглядати наступним чином:

import pathlib

from fastapi import FastAPI, File, UploadFile

from fastapi.staticfiles import StaticFiles

app = FastAPI()

app.mount("/static", StaticFiles(directory="static"), name="static")

@app.post("/uploadfile/")

async def create\_upload\_file(file: UploadFile = File()):

pathlib.Path("uploads").mkdir(exist\_ok=True)

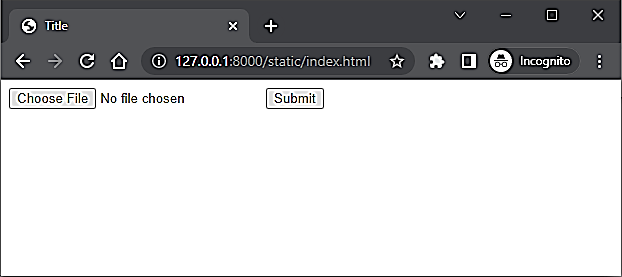
file\_path = f"uploads/{file.filename}"

with open(file\_path, "wb") as f:

f.write(await file.read())

return {"file\_path": file\_path}

А при переході за маршрутом http://127.0.0.1:8000/static/index.html в браузері ми отримаємо примітивну форму



**Форма завантаження файлу**

Щойно форму буде відправлено, FastAPI отримуватиме та оброблятиме файли так само, як це було розглянуто у попередньому прикладі.