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

**Заняття 2: REST API застосунок. Основи API**

**API** (Application programming interface) — опис способів (набір класів, процедур, функцій, структур або констант), якими одна комп'ютерна програма може взаємодіяти з іншою програмою.

API визначає функціональність, яку надає програма (модуль, бібліотека), при цьому API дозволяє абстрагуватись від того, як саме ця функціональність реалізована.

Практично всі операційні системи (UNIX, Windows, OS X, Linux тощо) мають API, за допомогою якого програмісти можуть створювати застосунки для цієї операційної системи. Головний API операційних систем - це безліч системних викликів.

В індустрії програмного забезпечення загальні стандартні API для стандартної функціональності відіграють важливу роль, оскільки вони гарантують, що всі програми, що використовують загальний API, працюватимуть однаково добре або, принаймні, типовим звичним чином. У випадку API графічних інтерфейсів це означає, що програми будуть мати схожий інтерфейс користувача, що полегшує процес освоєння нових програмних продуктів.

Наприклад: для того, щоб побачити в браузері рядок «Hello, world!», достатньо лише створити HTML-документ з мінімальним заголовком і найпростішим тілом, що містить цей рядок. Коли браузер відкриє цей документ, програма-браузер передасть ім'я файлу (або вже відкритий дескриптор файлу) бібліотеці, що обробляє HTML-документи, та, у свою чергу, за допомогою API операційної системи прочитає цей файл і розбереться у його будові, потім послідовно викличе через API бібліотеки стандартних графічних примітивів операції типу "очистити віконце", "написати "Hello, world!" вибраним шрифтом». Під час виконання цих операцій бібліотека графічних примітивів звернеться до бібліотеки віконного інтерфейсу з відповідними запитами, ця бібліотека вже звернеться до API операційної системи, щоб записати дані в буфер відеокарти.

При цьому практично на кожному з рівнів реально існує кілька можливих альтернативних API. Наприклад: ми могли б писати вихідний документ не на HTML, а на LaTeX, для відображення могли б використовувати будь-який браузер. До того ж, різні браузери використовують різні HTML-бібліотеки, і, крім того, все це може бути зібрано з використанням різних бібліотек примітивів та різних операційних систем.

Web API є практично синонімом для веб-служби.

API для Web застосунків нічим принципово не відрізняються від будь-яких інших видів API.

Найпоширенішим у Web зараз є стандарт REST API. Набирає популярності використання [**GraphQL**](https://graphql.org/) (https://graphql.org/) API та його похідні. Все ще багато де використовується **XML-RPC** і не здає позиції у своїй ніші [**SOAP**](https://en.wikipedia.org/wiki/SOAP) (https://en.wikipedia.org/wiki/SOAP).

[**XML-RPC**](https://ru.wikipedia.org/wiki/XML-RPC) (https://ru.wikipedia.org/wiki/XML-RPC)

**RPC (Remote Procedure Call)** — протокол виклику віддалених процедур.

Визначає набір стандартних типів даних та команд, які програміст може використовувати для доступу до функціональності іншої програми, яка знаходиться на іншому комп'ютері в мережі.

Очевидно з назви, що в якості формату зберігання та передачі даних в XML-RPC використовується XML. Це досить старий та надійний формат. Найчастіше для транспорту використовується HTTP поверх TCP/IP. Зараз цей стандарт дещо застарів, але все ще зустрічається у відносно старих системах, де він потрібний з міркувань зворотної сумісності.

Єдина перевага XML-RPC — простота.

[**JSON API**](https://jsonapi.org/) (https://jsonapi.org/)

[Python JSON-API implementations](https://jsonapi.org/implementations/#client-libraries-python)

Це дуже простий формат і він непогано реалізований у пакеті **json**, який входить до стандартного постачання Python.

JSON API створено поверх REST-API та додає низку додаткових вимог.

* body відповіді ЗАВЖДИ є. Там може бути порожній JSON-об'єкт, але щось обов'язково повинно бути.
* з попереднього пункту випливає, якщо у відповіді не повинно нічого міститись, то використовуйте статус коди відповідей, які не передбачають body (204 наприклад).
* документ відповіді обов'язково повинен містити хоча б одне із полів:
* data: дані, що запитуються клієнтом;
* errors: перелік помилок;
* meta: все, що не стандартизовано.

На жаль, на практиці рідко зустрічається суворе дотримання всіх вимог стандарту. Найчастіший випадок, коли зі стандарту беруть лише формат даних.

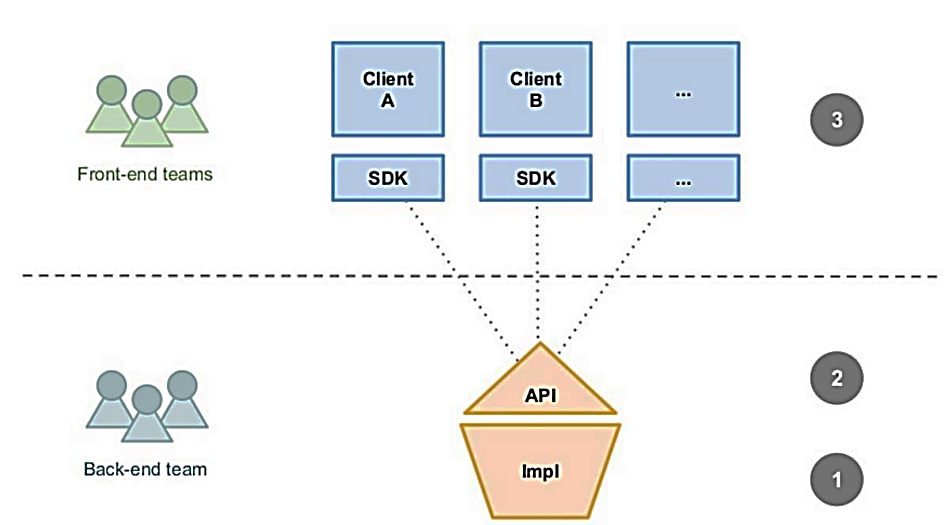
**Архітектурний підхід API First**

Сучасний Web-сервіс найчастіше складається з двох частин:

* Front-end
* Back-end

**Front-end** відповідає за відображення даних сервісу користувачу, а **Back-end**— за роботу з даними сервісу (зберігання, обробка, надання).

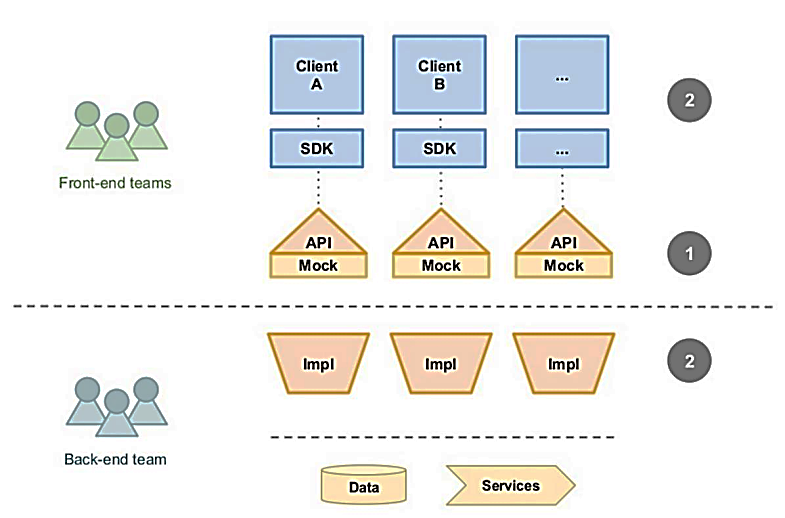
Зазвичай, ці частини розробляють кілька команд розробників. Очевидний підхід до розробки такого застосунку:



За такого підходу команди, які розробляють **Front-end**, не можуть розпочати роботу, доки**Back-end** не надасть працюючий API. Доки команди Front-end не розпочнуть розробку, використовуючи реальний API, недоліки та прорахунки, допущені при проектуванні API, не виявлять себе і команда **Back-end** не дізнається про них до релізу.

Такий підхід здорожчує розробку та розвиток застосунку.

Сучасніший і правильніший підхід полягає у тому, щоб насамперед стандартизувати API, за яким взаємодіятимуть **Front-end**та**Back-end.**



**Mock-об'єкт** (від англ. mock object, буквально: «об'єкт-пародія», «об'єкт-імітація», а також «підставка») — в об'єктно-орієнтованому програмуванні — тип об'єктів, що реалізують задані аспекти програмного оточення, що моделюється.

Mock-об'єкт - це конкретна фіктивна реалізація інтерфейсу, призначена виключно для тестування взаємодії, і щодо якого висловлюється твердження.

**SDK** (від англ. software development kit) — набір засобів розробки, який дозволяє фахівцям з програмного забезпечення створювати застосунки для певного пакету програм, програмного забезпечення базових засобів розробки, апаратної платформи, комп'ютерної системи, ігрових консолей, операційних систем та інших платформ.

За такого підходу є ймовірність, що розробники зможуть затвердити API, що відповідає вимогам обох команд, і команди можуть розпочати розробку зі створення Mock-реалізації API. Розробка Mock-реалізацій — це швидкий та легкий процес. Таким чином, обидві команди можуть практично відразу почати розробляти продукт, зосередившись на бізнес-вимогах і поклавшись на специфікацію.

Щоб такий підхід взагалі був можливий, потрібні стандарти опису API та Web-API зокрема.

Найпоширенішим таким стандартом є SWAGGER (OpenAPI).

**Swagger** — це мова опису API, генератори серверного та клієнтського коду, документації, тестів. Існує набір [**інструментів**](https://swagger.io/tools/open-source/open-source-integrations/) (https://swagger.io/tools/open-source/open-source-integrations/) для роботи зі Swagger у Python.

Стандарт OpenAPI виріс із комерційного продукту SWAGGER, який став дуже популярним через свою безальтернативність та простоту.

Для опису API у SWAGGER використовується розмітка [**YAML**](https://uk.wikipedia.org/wiki/YAML). Онлайн [**редактор**](http://editor.swagger.io/#/) (https://editor.swagger.io/#/)для створення документації SWAGGER.

**YAML (Yet Another Markup Language)** "дружній" формат серіалізації даних, концептуально близький до мов розмітки, але орієнтований на зручність введення-виведення типових структур даних багатьох мов програмування.

Приклад опису одного веб-ресурсу за стандартом:

swagger: '2.0'

info:

description: This is the swagger file that goes with our server code

version: '1.0.0'

title: Swagger REST Article

consumes:

- 'application/json'

produces:

- 'application/json'

basePath: '/api'

# Paths supported by the server application

paths:

/people:

get:

operationId: 'people.read'

tags:

- 'People'

summary: 'The people data structure supported by the server application'

description: 'Read the list of people'

responses:

200:

description: 'Successful read people list operation'

schema:

type: 'array'

items:

properties:

fname:

type: 'string'

lname:

type: 'string'

timestamp:

type: 'string'

Отримувану специфікацію однаково легко читати людині і парсити комп'ютеру, схема не залежить від мови або технології, вона універсальна і досить виразна.

Найчастіше в Python доводиться описувати схему додатково, щоб потім клієнти могли нею скористатися. Але ми використовуємо фреймворк FastAPI для створення REST API, а цей фреймворк автоматизує написання SWAGGER документації, і це спрощує процес її створення.

**Що таке REST**

**REST** (REpresentational State Transfer) розшифровується як передача репрезентативного стану. Це набір принципів проектування для підвищення масштабованості і гнучкості мережевих комунікацій. Ці принципи відповідають на низку питань.

* Які компоненти системи?
* Як вони спілкуються між собою?
* Як можна гарантувати можливість змінити різні частини системи у будь-який час?
* Як можна масштабувати систему для обслуговування великої кількості користувачів?

Рой Т. Філдінг вперше ввів термін REST у 2000 році у своїй докторській дисертації «Архітектурні стилі та проектування мережевих архітектур програмного забезпечення». Її можна почитати за цією [адресою](https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)

На момент публікації дисертації Інтернет вже був дуже популярним. Філдинг по суті зробив крок назад і проаналізував риси, які зробили Інтернет успішнішим, ніж конкуруючі Інтернет-протоколи. Потім він представив схематичну структуру, яка зробить мережеве спілкування "веб-подібним". Отже, REST — це загальний набір принципів, які не специфічні для Інтернету. Його можна застосовувати до інших типів мереж, наприклад, до вбудованих систем. REST також не є протоколом, оскільки не потребує подробиць реалізації. У дисертації Філдінга викладається ряд архітектурних обмежень, які повинна задовольняти система, щоб вважатися RESTful.

Перш ніж ми перейдемо до того, які архітектурні обмеження накладає REST, згадаємо з розділу "Основи Web" термінологію URL та URI.

URI

Є ідентифікатором конкретного ресурсу. Як сторінка, книга або документ.

URL

Це особливий тип ідентифікатора, який також говорить вам, як отримати доступ до нього.

Існує наступна традиційна форма запису `URL:

<scheme>://[<login>[:<password>]@]<host>[:<port>]][/<path>][?<query>][#<fragment>]

Приклад:

https://user:password@host.com:80/resourse/path/?query=name&ttt=123#hash

У цьому записі:

* scheme — Це мережевий протокол, за яким відбувається звернення до ресурсу
* login — не обов'язковий параметр, ім'я користувача, яке використовується для доступу до ресурсу
* password — пароль вказаного користувача, якщо він необхідний
* host — повністю прописане доменне ім'я хоста в системі DNS (goit.ua) або IP-адреса хоста у формі чотирьох груп десяткових чисел, розділених крапками
* port — порт хоста для підключення. Згадайте, ми використовували http://localhost:3000, де 3000 — це і є порт. Виникає закономірне питання: чому порт не вказується, наприклад, для URL у браузері. Просто браузер знає, що для протоколу http порт дорівнює 80, а для https він дорівнює 443 і підставляє його за нас
* path — уточнююча інформація про місце знаходження ресурсу.
* query — рядок запиту з параметрами, що передаються на сервер (методом GET). Починається із символу ?, роздільник параметрів — знак &. Приклад: ?foo=123&baz=234&bar=value
* fragment — ідентифікатор із попереднім символом #. Часто його називають хеш — hash або якір. Якорем може бути вказаний заголовок всередині документа або атрибут id елемента. За таким посиланням браузер відкриє сторінку та перемістить вікно перегляду до зазначеного елемента. Наприклад, посилання на секцію контактів у лендінгу: https://example.com/#contact.

**Архітектурні обмеження RESTful**

**Клієнт-сервер**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#%D0%BA%D0%BB%D1%96%D1%94%D0%BD%D1%82-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80)Перше обмеження вказує, що мережа повинна складатися з клієнтів і серверів. Сервер — це комп'ютер, у якого є ресурси, що цікавлять, а клієнт — це комп'ютер, який хоче взаємодіяти з ресурсами, що зберігаються на сервері. Коли ви переглядаєте Інтернет, ваш комп'ютер діє як клієнт і відправляє HTTP-запити на сервер для доступу до інформації та управління нею.

Система RESTful повинна працювати за моделлю клієнт-сервер, навіть якщо компонент іноді діє як клієнт, а іноді — як сервер.

Альтернативою клієнт-серверної архітектури, яка відрізняється від RESTful, є архітектура інтеграції на основі подій. У цій моделі кожен компонент безперервно транслює події, при цьому, очікуючи відповідних подій від інших компонентів. Немає особистого спілкування, лише трансляція та підслуховування. REST вимагає індивідуальної взаємодії, тому архітектура інтеграції на основі подій не буде RESTful.

**Stateless**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#stateless)Відсутність стану. Відсутність стану не означає, що сервери та клієнти не мають стану, це просто означає, що їм не потрібно відслідковувати стан один одного. Коли клієнт не взаємодіє із сервером, сервер не знає про його існування. Сервер також не веде облік попередніх запитів. Кожен запит розглядається як окремий, тобто жодних сесій.

**Єдиний інтерфейс**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#%D1%94%D0%B4%D0%B8%D0%BD%D0%B8%D0%B9-%D1%96%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81)Це обмеження гарантує, що існує спільна мова між серверами та клієнтами, яка дозволяє замінювати або змінювати кожну частину без порушення роботи всієї системи. Це досягається за рахунок 4 додаткових обмежень:

* ідентифікація ресурсів
* маніпулювання ресурсами за допомогою представлень
* інформативні (самоописові) повідомлення
* гіпермедіа.

**Перше обмеження інтерфейсу: ідентифікація ресурсів**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#%D0%BF%D0%B5%D1%80%D1%88%D0%B5-%D0%BE%D0%B1%D0%BC%D0%B5%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D1%96%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%83-%D1%96%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F-%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D1%96%D0%B2)

Перше обмеження єдиного інтерфейсу впливає на спосіб ідентифікації ресурсів. У термінах REST ресурсом може бути будь-що — документ HTML, зображення, інформація про замовлення тощо. Кожен ресурс повинен бути однозначно ідентифікований стабільним ідентифікатором. "Стабільний" ідентифікатор означає, що він не змінюється при взаємодії, і він не змінюється навіть при зміні стану ресурсу. Якщо ресурс дійсно переміщається на інший ідентифікатор, сервер повинен дати клієнтові відповідь, що вказує, що запит був невірним, і дати посилання на нове місце розташування ресурсу.

Інтернет використовує URL для ідентифікації ресурсів та HTTP як стандарт зв'язку. Щоб отримати ресурс, який зберігається на сервері, клієнт відправляє HTTP-запит GET на URL, який ідентифікує цей ресурс. Щоразу, коли ви вводите адресу у свій браузер, ваш браузер робить запит GET на цей URL. Якщо він отримує статус 200 та HTML-документ, він відображає сторінку у вікні, щоб ви могли її переглянути.

**Друге обмеження інтерфейсу: маніпулювання ресурсами через представлення**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#%D0%B4%D1%80%D1%83%D0%B3%D0%B5-%D0%BE%D0%B1%D0%BC%D0%B5%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D1%96%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%83-%D0%BC%D0%B0%D0%BD%D1%96%D0%BF%D1%83%D0%BB%D1%8E%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D0%B0%D0%BC%D0%B8-%D1%87%D0%B5%D1%80%D0%B5%D0%B7-%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F)

Друге обмеження говорить про те, що клієнт управляє ресурсами, відправляючи представлення на сервер — зазвичай це об'єкт JSON або XML, що містить контент, який він хотів би додати, видалити або змінити. В REST сервер повністю контролює ресурси та відповідає за будь-які зміни. Ресурсами можуть слугувати записи в базі даних, файли тощо. Коли клієнт бажає внести зміни до ресурсів, він відправляє серверу представлення про те, як повинен виглядати отриманий ресурс. Сервер приймає запит як пропозицію, але має повний контроль і сам вчиняє дію.

Найпростіший приклад — це блог у мережі. Коли користувач створює нове повідомлення у блозі, комп'ютер-клієнт повідомляє серверу, що необхідно додати нове повідомлення у блог. Для цього він відправляє HTTP-запит POST, можливо PUT, з контентом для нового повідомлення у блозі. Сервер надсилає відповідь клієнту, який вказує, чи було створено повідомлення або виникла проблема створення запису.

**Третє обмеження інтерфейсу: інформативні (self-descriptive) повідомлення**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#%D1%82%D1%80%D0%B5%D1%82%D1%94-%D0%BE%D0%B1%D0%BC%D0%B5%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D1%96%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%83-%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%96-self-descriptive-%D0%BF%D0%BE%D0%B2%D1%96%D0%B4%D0%BE%D0%BC%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F)

Інформативні повідомлення — ще одне обмеження, що забезпечує одноманітний інтерфейс між клієнтами та серверами. Інформативне повідомлення — це повідомлення, що містить всю інформацію, яка необхідна одержувачу для його розуміння. Додаткової інформації в окремій документації або іншому повідомленні бути не повинно.

Щоб зрозуміти, як це стосується Інтернету, давайте проаналізуємо набір HTTP-запитів і відповідей.

Коли користувач вводить http://www.example.com в адресному рядку свого веб-браузера, браузер надсилає наступний HTTP-запит:

GET / HTTP/1.1

Host: www.example.com

Це повідомлення є інформативним, оскільки воно повідомляє серверу, який метод HTTP був використаний і який протокол використовувався (HTTP 1.1).

Сервер може надіслати відповідь, схожу на цю:

HTTP/1.1 200 OK

Content-Type: text/html

<!DOCTYPE html>

<html lang="en-US">

<head>

<meta charset="UTF-8" />

<title>Your Site Title Here</title>

</head>

<body>

Hello world!

<a href="https://goit.ua">GoIT world!</a>

<img src="https://goit.ua/wp-content/themes/1/images/Layer.png">

</body>

</html>

Це повідомлення є інформативним (самоописовим), тому що воно повідомляє клієнту, як йому потрібно інтерпретувати тіло повідомлення, вказуючи, що це Content-type було text/html. У клієнта є все необхідне у цьому єдиному повідомленні, щоб обробити його належним чином.

**Четверте обмеження інтерфейсу: гіпермедіа**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#%D1%87%D0%B5%D1%82%D0%B2%D0%B5%D1%80%D1%82%D0%B5-%D0%BE%D0%B1%D0%BC%D0%B5%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D1%96%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%83-%D0%B3%D1%96%D0%BF%D0%B5%D1%80%D0%BC%D0%B5%D0%B4%D1%96%D0%B0)Останнє обмеження інтерфейсу — це обмеження гіпермедіа. Гіпермедіа — це модне слово для даних, що надсилаються з сервера клієнту, які містять інформацію про те, що клієнт може зробити далі, іншими словами, які подальші запити він може робити. В REST сервери повинні надсилати клієнтам лише гіпермедіа. HTML — це різновид гіпермедіа. Щоб краще зрозуміти це, давайте ще раз подивимося на відповідь сервера вище.

<a href="https://goit.ua">GoIT world!</a>

повідомляє клієнту, що він повинен зробити запит GET на https://goit.ua, якщо користувач клацне на посилання.

<img src="https://goit.ua/wp-content/themes/1/images/Layer.png" />

повідомляє клієнту, що потрібно негайно надіслати запит GET на https://goit.ua/wp-content/themes/1/images/Layer.png, щоб він міг відобразити зображення користувачеві.

Коли система має ідентифікатори для кожного ресурсу, маніпулює ними шляхом надсилання представлень від клієнта на сервер та має повідомлення, які є інформативними та складаються з гіпермедіа, вважається, що вона має єдиний інтерфейс.

**Кешування**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#%D0%BA%D0%B5%D1%88%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)Кешування належить до обмеження, за якого відповіді сервера повинні бути позначені як кешовані або некешовані. Кешування відбувається, коли клієнт зберігає попередні відповіді, отримані від сервера, тому, коли ці дані знову знадобляться, він може не робити запит мережею, а використовувати кешовані дані. Кешування частково або повністю усуває деякі клієнт-серверні взаємодії, сприяючи масштабованості та продуктивності застосунку.

**Багатошарова (багаторівнева) система**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#%D0%B1%D0%B0%D0%B3%D0%B0%D1%82%D0%BE%D1%88%D0%B0%D1%80%D0%BE%D0%B2%D0%B0-%D0%B1%D0%B0%D0%B3%D0%B0%D1%82%D0%BE%D1%80%D1%96%D0%B2%D0%BD%D0%B5%D0%B2%D0%B0-%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0)Багатошарова (багаторівнева) система полягає в тому, що компонентів може бути більше, ніж просто серверів та клієнтів. Це означає, що у системі може бути більше одного рівня. Однак, кожен компонент обмежений, щоб бачити та взаємодіяти лише з наступним шаром. Проксі — це додатковий компонент, який передає HTTP-запити на сервери або інші проксі. Проксі-сервери корисні для балансування навантаження та перевірки безпеки. Проксі-сервер діє як сервер для початкового клієнта, який надсилає запит, а потім діє вже як клієнт, коли передає цей запит далі. Шлюз може бути ще одним додатковим шаром, який перетворює HTTP-запит в інший протокол і поширює цей запит, а потім перетворює отриману відповідь назад в HTTP. Клієнт, при цьому, взаємодіє зі шлюзом як із звичайним сервером.

**Код за запитом**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/arch_04#%D0%BA%D0%BE%D0%B4-%D0%B7%D0%B0-%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D0%BE%D0%BC)Необов'язкове обмеження, яке належить до можливості сервера відправляти виконуваний код клієнту. Коли документ HTML завантажений, браузер автоматично завантажує код JavaScript із сервера та виконує його локально.

Таким чином, система RESTful — це будь-яка мережа, що задовольняє розглянуті обмеження. Система RESTful повинна бути гнучкою для різних випадків використання, масштабованою для підтримки великої кількості користувачів та компонентів, та такою, що адаптується з часом. Але пам'ятайте, що REST — це теоретичний проект.

**Основні методи HTTP**

**Ідемпотентність**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/http_05#%D1%96%D0%B4%D0%B5%D0%BC%D0%BF%D0%BE%D1%82%D0%B5%D0%BD%D1%82%D0%BD%D1%96%D1%81%D1%82%D1%8C)Властивість об'єкта або операції при повторному застосуванні операції до об'єкта давати той самий результат, як і за першого. Приклади:

* додавання з нулем: a=a+0=(a+0)+0=((a+0)+0)+0=...
* множення на одиницю: x = x\*1 = (x\*1)\*1 = ((x\*1)\*1)\*1 = ...;

З точки зору служби RESTful, щоб операція була ідемпотентною, клієнти можуть здійснювати один і той самий виклик кілька разів, генеруючи той самий результат, працюючи як "сеттер" метод мовою програмування. Іншими словами, створення кількох однакових запитів має той самий ефект, що і надсилання одного запиту. Хоча ідемпотентні операції генерують один і той самий результат на сервері, сама відповідь може відрізнятися, тобто стан ресурсу може змінитися між запитами.

Основні або HTTP-методи, що найчастіше використовуються, GET/POST/PUT/DELETE

* GET надає запит представлення ресурсу. Запити з використанням цього методу можуть лише витягувати дані.
* POST використовується для надсилання сутностей до певного ресурсу. Викликає зміну стану та побічні ефекти на сервері, оскільки найчастіше створює новий ресурс за надісланим представленням
* PUT замінює ресурс поточним представленням запиту
* DELETE видаляє вказаний ресурс.
* PATCH використовується для часткової зміни ресурсу

**HTTP метод GET**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/http_05#http-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-get)Використовується для отримання (або читання) представлення ресурсу. У разі успішної адреси, метод GET повертає представлення ресурсу у форматі XML або JSON в поєднанні з кодом стану HTTP 200 (OK). У разі невдачі повертається код 404 (NOT FOUND) або 400 (BAD REQUEST). Є безпечним (ідемпотентним) методом. Це означає, що він призначений лише для отримання інформації та не повинен змінювати стан сервера або мати побічні ефекти.

**HTTP метод POST**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/http_05#http-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-post)Запит найчастіше використовується для створення нових ресурсів. На практиці він також використовується для створення вкладених ресурсів. При створенні нового ресурсу POST запит надсилає представлення, а сервіс бере на себе відповідальність за призначення новому ресурсу ID тощо. У разі успішного створення ресурсу повертається HTTP код 201 Create, а також може бути встановлений заголовок Location з адресою створеного ресурсу. Метод POST не є безпечним або ідемпотентним запитом, оскільки відбувається побічний ефект — створення ресурсу.

**HTTP метод PUT (PATCH)**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/http_05#http-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-put-patch)Зазвичай використовується для оновлення ресурсу. Тіло запиту при відправленні PUT-запиту до існуючого ресурсу URL повинна містити оновлені дані оригінального ресурсу (повністю, або тільки частину, що оновлюється, — PATCH). У разі успішного оновлення повертається код 200 (або 204, якщо не було передано будь-який контент у тілі відповіді). Метод PUT вважається небезпечною операцією, оскільки в процесі виконання відбувається модифікація (або створення) екземплярів ресурсу на стороні сервера, але цей метод ідемпотентний. Іншими словами, створення або оновлення ресурсу через відправлення PUT запиту — ресурс не зникне, буде розташовуватися там, де і був.

**HTTP метод DELETE**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/http_05#http-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-delete)Використовується для видалення ресурсу, ідентифікованого конкретним URL (ID). У разі успішного видалення ресурсу повертається код 200 (OK) HTTP спільно з тілом відповіді, що містить дані видаленого ресурсу. Також можливе використання HTTP коду 204 (NO CONTENT) без тіла відповіді. Відповідно до специфікації HTTP метод DELETE ідемпотентний. Якщо ви виконуєте DELETE запит до ресурсу, він видаляється. Повторний DELETE запит до ресурсу завершиться так само: ресурс видалений.

**Коди відповіді HTTP**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/http_05#%D0%BA%D0%BE%D0%B4%D0%B8-%D0%B2%D1%96%D0%B4%D0%BF%D0%BE%D0%B2%D1%96%D0%B4%D1%96-http)Як видно, будь-яка відповідь від сервера повинна вказувати код стану HTTP. Він показує, чи був успішно виконаний певний запит HTTP. Коди поділяються на п'ять груп:

1. Інформаційні 100 - 199

100: Continue

2.Успішні 200 - 299

200: OK

201: Created

202: Accepted

204: No Content

3.Перенаправлення 300 - 399

301: Moved Permanently

307: Temporary Redirect

4.Клієнтські помилки 400 - 499

400: Bad Request

401: Unauthorized

403: Forbidden

404: Not Found

5.Серверні помилки 500 - 599

500: Internal Server Error

501: Not Implemented

502: Bad Gateway

503: Service Unavailable

504: Gateway Timeout

Детальніший опис почитайте у стандарті [RFC 2616](https://tools.ietf.org/html/rfc2616#section-10) (https://datatracker.ietf.org/doc/html/rfc2616#section-10) або простіший у документації [MDN](<https://developer.mozilla.org/ru/> docs/Web/HTTP/Status)

**Формування URL для REST API**

У цьому розділі ми поговоримо про правильне найменування ресурсів для API. Коли ресурси названі правильно, API інтуїтивно зрозумілий та простий у використанні. Часто кінцеві URL за запитом ресурсу називають endpoints API.

Що є хорошою практикою для правильного іменування?

Ну, по-перше, необхідно використовувати для опису базових URL іменники у множині — users, contacts. Також потрібно використовувати конкретніші та чіткіші імена news, videos, а не абстрактні items або elements. Складну логіку для URL необхідно описувати за рахунок додаткових параметрів, тобто ховати за знаком ?. Приклад використання пагінації /users?limit=25&offset=50, фільтрація відповіді /friends?fields=id,name,picture

Виходячи зі сказаного, наступні назви будуть поганою практикою

/api/users/13/remove

/api/getusers

/api/v1/users-get

Тут ми використовуємо дієслова, вказуємо видалення для /api/users/13/remove, а необхідно просто використовувати HTTP метод DELETE

Давайте розглянемо хороші практики іменування URL для різноманітних ситуацій.

Додавання нового клієнта в систему:

HTTP метод: POST

URL: https://www.example.com/customers

Отримати дані клієнта з ідентифікатором клієнта ID 112233:

HTTP метод: GET

URL: https://www.example.com/customers/112233

Той самий URL ми використовуємо для HTTP методів PUT та DELETE для оновлення та видалення, відповідно.

Створення нового продукту:

HTTP метод: POST

URL: https://www.example.com/products

Для читання, оновлення, видалення продукту з ID 432111, відповідно:

HTTP метод: GET, PUT, DELETE

URL: https://www.example.com/products/432111

Створення нового замовлення для клієнта поза контекстом клієнта

HTTP метод: POST

URL: https://www.example.com/orders

Створення того самого замовлення, але в контексті конкретного клієнта з ID 332244

HTTP метод: POST

URL: https://www.example.com/customers/332244/orders

Список замовлень, що належать клієнту ID 332244:

HTTP метод: GET

URL: https://www.example.com/customers/332244/orders

Припустимо потрібний URL для додавання нової позиції в замовленні з ID 1234, для клієнта з ID 332244:

HTTP метод: POST

URL: https://www.example.com/customers/332244/orders/1234/lineorders

Отримання списку замовлення за ID замовлення без знання ID конкретного клієнта

HTTP метод: GET

URL: https://www.example.com/orders/8769

Пагінація здійснюється через query рядок за допомогою параметра offset — це початковий номер позиції і параметра limit — максимальна кількість елементів, що повертаються. Вони можуть називатися і по-іншому, наприклад skip, limit

HTTP метод: GET

URL: https://api.example.com/resources?offset=0&limit=25

Складна фільтрація за значеннями. Можна використовувати роздільник — подвійна двокрапка ::, яка відокремлює ім'я властивості від значення порівняння.

HTTP метод: GET

URL: https://www.example.com/users?filter="name::sam|city::denver"

Сортування. Один із способів, коли для кожної властивості, що передається, здійснюється сортування в порядку зростання, а для кожної властивості з префіксним тире ("-") - сортування здійснюється в порядку спадання.

Сепаратор для кожного імені властивості — вертикальна смуга ("|").

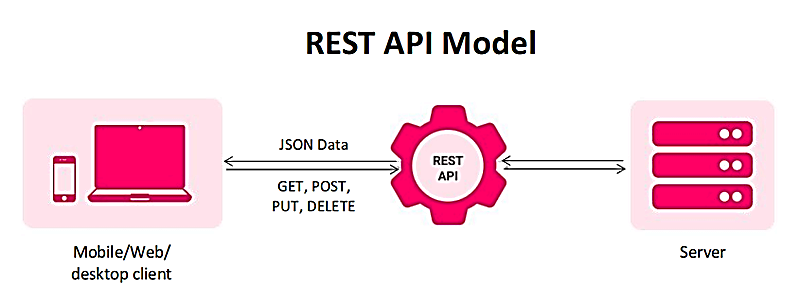
HTTP метод: GET

URL: https://www.example.com/users?sort=lastName|firstName|-birthdate

Сподіваюся, з цих прикладів вам стало зрозуміло, як потрібно називати ресурси для вашого API.

**Базовий приклад REST API**

Для розуміння як це виглядає в коді – почнемо з базового прикладу. Нам необхідно реалізувати код, який відповідає зображенню нижче.



Нам необхідна реалізація чотирьох операцій:

* GET — отримання ресурсу
* POST — створення ресурсу
* PUT — оновлення ресурсу
* DELETE — видалення ресурсу

Вважатимемо, що працюємо з деякою абстрактною сутністю item. Значить маршрут повинен бути іменником у множині /items. Наприклад, для item з унікальним ідентифікатором, що дорівнює 1, це будуть маршрути:

* GET — /items
* POST — /items
* PUT — /items/1
* DELETE — /items/1

Наведемо мінімально необхідний для цього код і розберемо його:

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str = None

price: float

tax: float = None

@app.post("/items")

async def create\_item(item: Item):

return item

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

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

return {"item\_id": item\_id}

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

async def update\_item(item\_id: int, item: Item):

return {"item\_id": item\_id, "item": item}

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

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

return {"item\_id": item\_id}

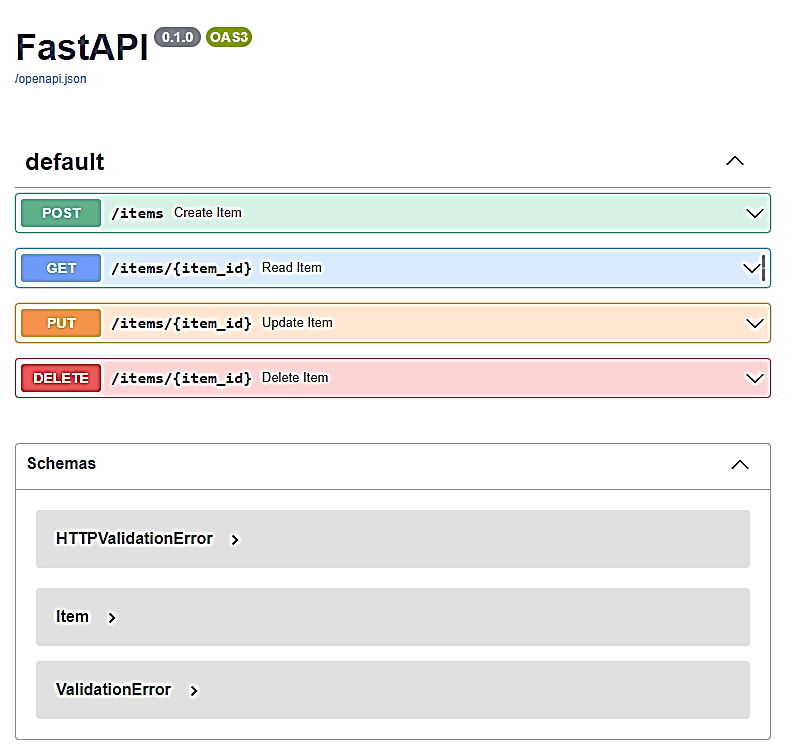
У цьому прикладі ми створюємо API з чотирма основними методами: POST, GET, PUT і DELETE.

Операція POST використовується для створення нового елемента. Ми використовуємо декоратор @app.post та визначаємо відповідний URL для цього маршруту. Ми також вказуємо, що функція create\_item очікує як аргумент об'єкт типу Item, який визначений у нашому класі Item. Цей клас Item є прикладом Pydantic-моделі, яка використовується для валідації та серіалізації вхідних даних.

Операція GET використовується для отримання існуючого елемента за ідентифікатором. Ми використовуємо декоратор @app.get і вказуємо URL зі змінною item\_id, яку ми можемо використовувати у функції read\_item.

Операція PUT використовується для оновлення існуючого елемента. Ми використовуємо декоратор @app.put і вказуємо URL зі змінною item\_id, як і у випадку з GET методом. Ми також вказуємо, що функція update\_item очікує як другий аргумент об'єкт типу Item.

Операція DELETE використовується для видалення існуючого елемента. Ми використовуємо декоратор @app.delete і вказуємо URL зі змінною item\_id, як і у випадку з GET та PUT методами. Функція delete\_item не очікує жодних інших аргументів, тому що ми видаляємо елемент за його ідентифікатором.



**Отримана Swagger документація**

**Готовий застосунок**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/example#%D0%B3%D0%BE%D1%82%D0%BE%D0%B2%D0%B8%D0%B9-%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BE%D0%BA)Готовий результат повинен бути як у цьому живому прикладі, запустіть та подивіться код:

Щоб побачити Swagger документацію нашого застосунку, після запуску прикладу, натисніть [посилання](https://fastapi-base-rest-api.krabaton.repl.co/docs) (https://replit.com/@Krabaton/FastAPI-Base-Rest-API)

﻿Ми розглянули приклад базового **RESTful API** з використанням FastAPI. Ви можете розширити його, додаючи додаткові маршрути та функції, а також реалізовуючи логіку для роботи з базою даних та іншими сервісами. І в наступному розділі ми і створимо саме такий застосунок.

**Застосунок REST API**

У цьому практичному завданні ми реалізуємо систему управління нотатками як сервіс**REST API** з використанням фреймворку [**FastAPI**](https://fastapi.tiangolo.com/) (https://fastapi.tiangolo.com/)**.**

**Встановлення необхідних пакетів**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#%D0%B2%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BD%D0%B5%D0%BE%D0%B1%D1%85%D1%96%D0%B4%D0%BD%D0%B8%D1%85-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%96%D0%B2)

Для створення віртуального оточення будемо використовувати «Poetry» (https://python-poetry.org/).

Для активації віртуального оточення необхідно виконати команду:

poetry shell

Встановимо безпосередньо фреймворк FastAPI.

poetry add fastapi

Потрібно встановити пакет uvicorn, який буде працювати як сервер:

poetry add uvicorn[standard]

Також нам необхідно ще встановити наступний набір пакетів для реалізації нашого REST API застосунку:

* sqlalchemy — ORM SQLAlchemy
* psycopg2 — драйвер для роботи з PostgreSQL
* alembic — пакет для виконання міграцій

Попереднє налаштування застосунку завершено, переходимо до реалізації застосунку нотаток.

**Архітектура проекту**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#%D0%B0%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D1%83)

Структура нашого застосунку буде наступною:

├── src

|  ├─database

│  |   ├── db.py

│  |   └── models.py

│  ├── repository

│  |   ├── notes.py

│  |   └── tags.py

│  ├── routes

│  |   ├── notes.py

│  |   └── tags.py

│  └── schemas.py

├── pyproject.toml

└── main.py

Точка входу в наш застосунок - main.py, а всередині папки src ми помістимо файли для роботи з нашим застосунком.

Кожен файл буде мати свою зону відповідальності:

* repository/notes.py, repository/tags.py - додатковий шар абстракції, який містить методи для роботи з базою даних. Кожен файл відповідає за роботу із конкретною таблицею.
* database/db.py - буде відповідати за підключення до бази даних PostgreSQL
* database/models.py - буде містити наші моделі бази даних
* routes/notes.py, routes/tags.py - тут будуть знаходитись маршрути нашого REST API застосунку, для роботи з нашими сутностями – нотатками (notes) та тегами (tags). Звідси ми будемо викликати методи репозиторію, а не звертатись до бази даних напряму
* schemas/notes.py - схеми для валідації наших вхідних та вихідних даних

**Підключення до бази даних**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#%D0%BF%D1%96%D0%B4%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B4%D0%BE-%D0%B1%D0%B0%D0%B7%D0%B8-%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)Для роботи з базою даних нам потрібно спочатку виконати підключення до бази даних.

**src/database/db.py**

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

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

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

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

# Dependency

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

Цей код створює з'єднання з базою даних PostgreSQL за допомогою SQLAlchemy. Рядок з'єднання визначений у змінній SQLALCHEMY\_DATABASE\_URL і передається у функцію create\_engine для створення двигуна engine.

Константа SQLALCHEMY\_DATABASE\_URL - це рядок з'єднання з базою даних.

* postgresql+psycopg2:// — це драйвер для підключення до PostgreSQL бази даних.
* postgres — ім'я користувача, яке використовується для підключення до бази даних.
* 567234 — пароль для підключення до бази даних.
* localhost — хост бази даних.
* 5432 — порт для підключення до бази даних.
* rest\_app — назва бази даних.

ПРИМІТКА

Не забудьте підняти докер-контейнер з PostgreSQL і створити в ньому базу даних rest\_app

В цілому, цей рядок визначає з'єднання до бази даних PostgreSQL, яка знаходиться на локальному хості і доступна на порту 5432, з ім'ям користувача postgres та паролем 567234.

Функція sessionmaker використовується для створення фабрики сесій, яка використовується для створення сесій для взаємодії з базою даних.

Фабрика SessionLocal налаштована так, щоб не виконувати автокомміт та автоскидання сесії, і прив'язана до двигуна, створеного раніше.

Параметр autocommit - це режим, коли кожна операція з базою даних автоматично підтверджується (комітиться). Тобто будь-які зміни, які ви вносите до бази даних, відразу ж стають активними і не можуть бути скасовані. За замовчуванням для роботи з SQLAlchemy увімкнено режим автокомміту.

Параметр autoflush - це режим, коли будь-які зміни, які ви вносите в об'єкти сесії, автоматично відправляються в базу даних. Іншими словами, будь-які зміни, які ви вносите в об'єкти, відразу ж стають активними і можуть бути помітні в базі даних. За замовчуванням для роботи з SQLAlchemy увімкнено режим автоскидання.

В цьому разі сесія налаштована на роботу з відключеним автокоммітом та автоскиданням і означає, що сесія не буде авто комітити і не буде автоматично скидати зміни в базу даних. Це дає можливість працювати з транзакціями, де можна виконувати кілька операцій і або підтвердити їх всі відразу commit, або відкотити rollback у разі помилки.

Функція get\_db – це залежність, яка повертає сесію з використанням фабрики SessionLocal. Сесія закривається при виході з функції з використанням блоку finally.

**Створення моделей**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B5%D0%B9)У файлі models.py створимо моделі. Імпортуємо наш клас Base та виконаємо наслідування від нього.

from sqlalchemy import Column, Integer, String, Boolean, func, Table

from sqlalchemy.orm import relationship

from sqlalchemy.sql.schema import ForeignKey

from sqlalchemy.sql.sqltypes import DateTime

from sqlalchemy.ext.declarative import declarative\_base

Base = declarative\_base()

note\_m2m\_tag = Table(

"note\_m2m\_tag",

Base.metadata,

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

Column("note\_id", Integer, ForeignKey("notes.id", ondelete="CASCADE")),

Column("tag\_id", Integer, ForeignKey("tags.id", ondelete="CASCADE")),

)

class Note(Base):

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

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

title = Column(String(50), nullable=False)

created\_at = Column('created\_at', DateTime, default=func.now())

description = Column(String(150), nullable=False)

done = Column(Boolean, default=False)

tags = relationship("Tag", secondary=note\_m2m\_tag, backref="notes")

class Tag(Base):

\_\_tablename\_\_ = "tags"

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

name = Column(String(25), nullable=False, unique=True)

Цей код визначає два класи, Note та Tag, з використанням декларативної системи SQLAlchemy. Клас Base - це декларативний базовий клас, що надається SQLAlchemy, який використовується для визначення структури класів та наслідування від нього.

Змінна note\_m2m\_tag - це таблиця, яка визначена для зберігання зв'язку багато-до-багатьох між нотатками і тегами. Таблиця має три стовпці: id, note\_id та tag\_id. Стовпці note\_id та tag\_id є зовнішніми ключами з параметром ondelete, встановленим на CASCADE, що означає, що при видаленні нотатки або тегу, всі відповідні рядки у таблиці зв'язків також будуть видалені.

Клас Note має наступні стовпці:

* id — цілочисельний первинний ключ
* title — рядок довжиною 50 символів, який не може бути порожнім. Заголовок нотатки
* created\_at — стовпчик типу datetime зі значенням за замовчуванням поточної дати та часу. Час створення нотатки.
* description — рядок довжиною 150 символів, який не може бути порожнім. Опис нотатки.
* done — стовпчик типу boolean зі значенням за замовчуванням False. Визначає статус нотатки.
* tags — зв'язок із класом Tag, за допомогою додаткової таблиці note\_m2m\_tag

Стовпчик created\_at — це стовпчик типу DateTime і визначений зі значенням за замовчуванням як поточна дата і час з використанням SQLAlchemy функції func.now(). Це означає, що коли до бази даних додається нова нотатка, якщо для стовпчика created\_at не надано значення, воно буде автоматично встановлено на поточну дату та час. Це корисно для відстеження того, коли була створена нотатка в базі даних.

Атрибут tags у класі Note є SQLAlchemy зв'язком, який визначає зв'язок багато-до-багатьох між класами Note і Tag.

* Перший параметр, `Tag', - це клас, з яким визначається зв'язок.
* Параметр secondary використовується для вказівки таблиці асоціації і, яка зберігає зв'язок багато-до-багатьох. У нашому випадку це таблиця note\_m2m\_tag, яку ми визначили раніше.
* Параметр backref використовується для визначення нової властивості на зв'язаному класі, яка забезпечуватиме доступ до зв'язаних об'єктів в іншому напрямку. У цьому випадку backref="notes" додасть властивість notes до класу Tag. Вона дозволить отримати доступ до всіх нотаток, пов'язаних із певним тегом.

Таким чином рядок коду tags = relationship("Tag", secondary=note\_m2m\_tag, backref="notes") визначає зв'язок багато-до-багатьох між класами Note і Tag, з таблицею асоціації note\_m2m\_tag, і властивістю зворотного посилання на клас Tag, що називається notes. Коли ви отримуєте доступ до note.tags, SQLAlchemy фактично вибере елементи з бази даних у таблиці тегів та заповнить їх.

Клас Tag досить простий і має наступні стовпці:

* id — цілочисельний первинний ключ
* name — рядок довжиною 25, який не може бути порожнім і повинен бути унікальним. Назва тегу.

**Виконуємо міграції моделей**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D1%83%D1%94%D0%BC%D0%BE-%D0%BC%D1%96%D0%B3%D1%80%D0%B0%D1%86%D1%96%D1%97-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B5%D0%B9)Створивши файл з моделями, потрібно виконати процес міграції за допомогою alembic. Насамперед необхідно ініціалізувати оточення alembic за допомогою команди:

alembic init migrations

Оскільки ми хочемо використовувати автогенерацію SQL скриптів у міграціях alembic, нам необхідно повідомити про це оточення alembic у файлі env.py, який розташований у папці migrations. Відкриємо його і насамперед імпортуємо нашу декларативну базу Base з файлу models.py та рядок підключення SQLALCHEMY\_DATABASE\_URL до нашої бази даних.

**migrations/env.py**

from logging.config import fileConfig

from sqlalchemy import engine\_from\_config

from sqlalchemy import pool

from alembic import context

from src.database.models import Base

from src.database.db import SQLALCHEMY\_DATABASE\_URL

Далі нам необхідно знайти наступний рядок нижче в коді файлу env.py:

target\_metadata = None

і замість None, вказати наші метадані:

target\_metadata = Base.metadata

Тут виконаємо заміну рядка підключення до бази даних на актуальну:

config.set\_main\_option("sqlalchemy.url", SQLALCHEMY\_DATABASE\_URL)

Створюємо міграцію наступною консольною командою в корені проекту.

alembic revision --autogenerate -m 'Init'

Якщо файл з міграцією успішно створився в папці migrations/versions, то застосуємо створену міграцію:

alembic upgrade head

**Створюємо схеми валідації**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D1%8E%D1%94%D0%BC%D0%BE-%D1%81%D1%85%D0%B5%D0%BC%D0%B8-%D0%B2%D0%B0%D0%BB%D1%96%D0%B4%D0%B0%D1%86%D1%96%D1%97)Створимо моделі Pydantic і помістимо їх у файл schemas.py. Моделі Pydantic визначають «схему» або іншими словами дійсну форму даних.

**src/schemas.py**

from datetime import datetime

from typing import List, Optional

from pydantic import BaseModel, Field

class TagModel(BaseModel):

name: str = Field(max\_length=25)

class TagResponse(TagModel):

id: int

class Config:

orm\_mode = True

class NoteBase(BaseModel):

title: str = Field(max\_length=50)

description: str = Field(max\_length=150)

class NoteModel(NoteBase):

tags: List[int]

class NoteUpdate(NoteModel):

done: bool

class NoteStatusUpdate(BaseModel):

done: bool

class NoteResponse(NoteBase):

id: int

created\_at: datetime

tags: List[TagResponse]

class Config:

orm\_mode = True

Давайте тепер детально розберемо код. Клас TagModel є моделлю Pydantic, яка визначає поля класу Tag. Він використовує поле name типу str та максимальну довжину за замовчуванням 25 символів.

Клас TagResponse - це інша модель Pydantic, яка наслідується від TagModel і додає додаткове поле id типу int. Цей клас призначений для використання як модель відповіді при поверненні даних Tag клієнту.

Атрибут orm\_mode у класі Config використовується для увімкнення режиму ORM для цієї моделі. Таким чином, атрибут orm\_mode у класі Config дозволяє Pydantic автоматично генерувати модель бази даних з певних полів, що дозволяє використовувати модель Pydantic як об'єкт передачі даних, а також як модель бази даних без необхідності окремого використання моделі бази даних.

Таким чином, ці два класи визначають модель даних для Tag з класом TagModel для валідації та створення поля name та іншим класом TagResponse для повернення даних клієнту з додатковим полем id.

Далі в коді йдуть кілька моделей Pydantic, пов'язаних із нотатками.

* NoteBase — визначає основні поля нотатки, такі як title і description з максимальною довжиною 50 та 150 відповідно.
* NoteModel — наслідується від NoteBase і додає поле tags типу List[int]. Це буде список з id необхідних тегів.
* NoteUpdate — наслідується від NoteModel і додає обов'язкове поле done типу boolean.
* NoteStatusUpdate — визначає лише одне поле done типу boolean, використовується для оновлення статусу завершеності нотатки.
* NoteResponse — наслідується від NoteBase і додає поля id, created\_at та tags. Поле id має тип int, created - тип datetime та поле tags є списком об'єктів TagResponse, використовується для отримання даних Нотатки з додатковою інформацією.

Загалом, ці моделі визначають поля та правила валідації для створення, оновлення та отримання даних для нотатки Note і тегів Tag.

**Реалізуємо REST API**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#%D1%80%D0%B5%D0%B0%D0%BB%D1%96%D0%B7%D1%83%D1%94%D0%BC%D0%BE-rest-api)

**CRUD операції для тегів**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#crud-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D1%97-%D0%B4%D0%BB%D1%8F-%D1%82%D0%B5%D0%B3%D1%96%D0%B2)

Щоб реалізувати повний цикл **CRUD** операцій, нам необхідно реалізувати набір маршрутів нашого REST API застосунку

* Отримати список усіх тегів: маршрут - /api/tags, HTTP метод GET
* Створити тег: маршрут - /api/tags, HTTP метод POST
* Отримати тег за id: маршрут - /api/tags/<int:id>, HTTP метод GET
* Оновити тег: маршрут - /api/tags/<int:id>, HTTP метод PUT
* Видалити тег: маршрут - /api/tags/<int:id>, HTTP метод DELETE

Додаємо реалізацію маршрутів у файл main.py за допомогою механізму роутингу:

from fastapi import FastAPI

from src.routes import notes, tags

app = FastAPI()

app.include\_router(tags.router, prefix='/api')

app.include\_router(notes.router, prefix='/api')

@app.get("/")

def read\_root():

return {"message": "Hello World"}

Цей код створює екземпляр веб-фреймворку FastAPI і визначає два маршрути: один для модуля notes та один для модуля tags. Функція include\_router використовується для включення маршрутизації, визначеної в кожному модулі, а параметр prefix використовується для зазначення загального префікса URL для всіх маршрутів цього модуля.

Також визначена функція read\_root, яка прив'язана до маршруту '/' і повертає об'єкт JSON з повідомленням `Hello World', яке є маршрутом за замовчуванням для застосунку.

Роутери для модулів notes та tags містять точки доступу для операцій CRUD для нотаток та тегів відповідно.

Розберемо код для операцій над тегами src/routes/tags.py.

**src/routes/tags.py**

from typing import List

from fastapi import APIRouter, HTTPException, Depends, status

from sqlalchemy.orm import Session

from src.database.db import get\_db

from src.schemas import TagModel, TagResponse

from src.repository import tags as repository\_tags

router = APIRouter(prefix='/tags', tags=["tags"])

@router.get("/", response\_model=List[TagResponse])

async def read\_tags(skip: int = 0, limit: int = 100, db: Session = Depends(get\_db)):

tags = await repository\_tags.get\_tags(skip, limit, db)

return tags

@router.get("/{tag\_id}", response\_model=TagResponse)

async def read\_tag(tag\_id: int, db: Session = Depends(get\_db)):

tag = await repository\_tags.get\_tag(tag\_id, db)

if tag is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Tag not found")

return tag

@router.post("/", response\_model=TagResponse)

async def create\_tag(body: TagModel, db: Session = Depends(get\_db)):

return await repository\_tags.create\_tag(body, db)

@router.put("/{tag\_id}", response\_model=TagResponse)

async def update\_tag(body: TagModel, tag\_id: int, db: Session = Depends(get\_db)):

tag = await repository\_tags.update\_tag(tag\_id, body, db)

if tag is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Tag not found")

return tag

@router.delete("/{tag\_id}", response\_model=TagResponse)

async def remove\_tag(tag\_id: int, db: Session = Depends(get\_db)):

tag = await repository\_tags.remove\_tag(tag\_id, db)

if tag is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Tag not found")

return tag

Цей код визначає набір маршрутів для модуля тегів з використанням класу APIRouter з бібліотеки fastapi. Маршрути прив'язані до певних операцій HTTP (GET, POST, PUT, DELETE) та призначені для обробки операцій CRUD тегів.

Маршрут router.get("/", response\_model=List[TagResponse]) прив'язаний до операції GET і призначений для отримання списку тегів. Він приймає два необов'язкові параметри запиту, skip та limit, які використовуються для розбиття результатів на сторінки або іншими словами для пагінації.

INFO

Пагінація результатів – це техніка розбиття великої кількості результатів на кілька сторінок для зручності перегляду та навігації. Це особливо корисно для великих наборів даних, таких як список постів у блозі, результати пошуку або таблиці бази даних.

Маршрут router.get("/{tag\_id}", response\_model=TagResponse) прив'язаний до операції GET і призначений для отримання певного тегу за його ідентифікатором. Якщо тегу із наданим ідентифікатором не існує, він викликає виняток HTTPException з кодом стану 404.

Маршрут router.post("/", response\_model=TagResponse) прив'язаний до операції POST і призначений для створення нового тегу. Він приймає об'єкт TagModel у тілі запиту та повертає створений тег.

Маршрут router.put("/{tag\_id}", response\_model=TagResponse) прив'язаний до операції PUT і призначений для оновлення певного тегу за його ідентифікатором. Якщо тегу із наданим ідентифікатором не існує, він викликає виняток HTTPException з кодом стану 404.

Маршрут router.delete("/{tag\_id}", response\_model=TagResponse) прив'язаний до операції DELETE і призначений для видалення певного тегу за його ідентифікатором. Якщо тегу із наданим ідентифікатором не існує, він викликає виняток HTTPException з кодом стану 404.

Всі маршрути в цьому файлі залежать від функції get\_db, яка забезпечує сеанс бази даних.

Маршрути, визначені в цьому файлі, обробляють операцію CRUD для тегів і використовують функції, визначені в модулі репозиторію, для взаємодії з базою даних.

Усі методи роботи бази даних із тегами ми поміщаємо у файл src/repository/tags.py. Вони виконують безпосередні операції, додавання, зміни та видалення тегів всередині бази даних.

**src/repository/tags.py**

from typing import List

from sqlalchemy.orm import Session

from src.database.models import Tag

from src.schemas import TagModel

async def get\_tags(skip: int, limit: int, db: Session) -> List[Tag]:

return db.query(Tag).offset(skip).limit(limit).all()

async def get\_tag(tag\_id: int, db: Session) -> Tag:

return db.query(Tag).filter(Tag.id == tag\_id).first()

async def create\_tag(body: TagModel, db: Session) -> Tag:

tag = Tag(name=body.name)

db.add(tag)

db.commit()

db.refresh(tag)

return tag

async def update\_tag(tag\_id: int, body: TagModel, db: Session) -> Tag | None:

tag = db.query(Tag).filter(Tag.id == tag\_id).first()

if tag:

tag .name = body.name

db.commit()

return tag

async def remove\_tag(tag\_id: int, db: Session) -> Tag | None:

tag = db.query(Tag).filter(Tag.id == tag\_id).first()

if tag:

db.delete(tag)

db.commit()

return tag

Ми визначили набір функцій, які використовуються для виконання операцій CRUD над моделлю тегів у базі даних.

Функція get\_tags використовується для отримання списку тегів з бази даних з можливістю пропустити певну кількість тегів та обмежити кількість тегів, що повертаються. Функція get\_tag використовується для отримання певного тегу за його ідентифікатором. Функція create\_tag використовується для створення нового тегу в базі даних. Він бере об'єкт TagModel і використовує інформацію з нього для створення нового об'єкта Tag, потім додає його до сеансу і фіксує зміни в базі даних. Функція update\_tag використовується для оновлення певного тегу за його ідентифікатором. Він бере об'єкт TagModel та оновлює за ім'ям тегу інформацію з нього. Якщо тегу не існує, повертається None.

Функція remove\_tag використовується для видалення певного тегу за його ідентифікатором. Якщо тегу не існує, повертається None. Всі ці функції використовують SQLAlchemy ORM для взаємодії з базою даних.

**CRUD операції для нотаток**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#crud-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D1%97-%D0%B4%D0%BB%D1%8F-%D0%BD%D0%BE%D1%82%D0%B0%D1%82%D0%BE%D0%BA)Робота з нотатками будується аналогічно до роботи з тегами.

* Отримати список нотаток: маршрут - /api/notes, HTTP метод GET
* Створити нотатку: маршрут - /api/notes, HTTP метод POST
* Отримати нотатку за id: маршрут - /api/notes/<int:id>, HTTP метод GET
* Оновити статус нотатки: маршрут - /api/notes/<int:id>, HTTP метод patch
* Видалити нотатку: маршрут - /api/notes/<int:id>, HTTP метод DELETE

**src/routes/notes.py**

from typing import List

from fastapi import APIRouter, HTTPException, Depends, status

from sqlalchemy.orm import Session

from src.database.db import get\_db

from src.schemas import NoteModel, NoteUpdate, NoteStatusUpdate, NoteResponse

from src.repository import notes as repository\_notes

router = APIRouter(prefix='/notes', tags=["notes"])

@router.get("/", response\_model=List[NoteResponse])

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

notes = await repository\_notes.get\_notes(skip, limit, db)

return notes

@router.get("/{note\_id}", response\_model=NoteResponse)

async def read\_note(note\_id: int, db: Session = Depends(get\_db)):

note = await repository\_notes.get\_note(note\_id, db)

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Note not found")

return note

@router.post("/", response\_model=NoteResponse)

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

return await repository\_notes.create\_note(body, db)

@router.put("/{note\_id}", response\_model=NoteResponse)

async def update\_note(body: NoteUpdate, note\_id: int, db: Session = Depends(get\_db)):

note = await repository\_notes.update\_note(note\_id, body, db)

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Note not found")

return note

@router.patch("/{note\_id}", response\_model=NoteResponse)

async def update\_status\_note(body: NoteStatusUpdate, note\_id: int, db: Session = Depends(get\_db)):

note = await repository\_notes.update\_status\_note(note\_id, body, db)

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Note not found")

return note

@router.delete("/{note\_id}", response\_model=NoteResponse)

async def remove\_note(note\_id: int, db: Session = Depends(get\_db)):

note = await repository\_notes.remove\_note(note\_id, db)

if note is None:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND, detail="Note not found")

return note

Цей код визначає маршрути API застосунку для створення нотаток. API має наступні кінцеві точки:

* GET /notes/ — повертає список нотаток з додатковими параметрами запиту skip та limit для управління пагінацією сторінок.
* GET /notes/{note\_id} — повертає конкретну нотатку за ідентифікатором. Якщо нотатка не знайдена, виникає помилка 404.
* POST /notes/ — створює нову нотатку з даними нотатки, переданими в запиті.
* PUT /notes/{note\_id} — оновлює існуючу нотатку за ідентифікатором з використанням оновлених даних, переданих у запиті. Якщо нотатка не знайдена, виникає помилка 404.
* PATCH /notes/{note\_id} — оновлює статус існуючої нотатки за ідентифікатором, при цьому оновлений статус передається в запиті. Якщо нотатка не знайдена, виникає помилка 404.
* DELETE /notes/{note\_id} — видаляє існуючу нотатку за ідентифікатором. Якщо нотатка не знайдена, виникає помилка 404.

Ми знову використовуємо впровадження залежностей для отримання сеансу бази даних з функції get\_db.

ПРИМІТКА

Можливо ви вже звернули увагу, але для оновлення нотатки ми використовували HTTP операцію PUT @router.put("/{note\_id}",

response\_model=NoteResponse), а для оновлення статусу нотатки замінили її на HTTP операцію PATCH @router.patch("/{note\_id}", response\_model=NoteResponse). Виникає закономірне питання, чому? Річ у тому, що операція PUT передбачає повну зміну об'єкта, а операція PATCH - часткову, що і відбувається в нашому випадку. У нотатки змінюється лише одна властивість done - виконана вона чи ні.

Модуль src.repository.notes містить функції, що взаємодіють із базою даних.

**src/repository/notes.py**

from typing import List

from sqlalchemy.orm import Session

from src.database.models import Note, Tag

from src.schemas import NoteModel, NoteUpdate, NoteStatusUpdate

async def get\_notes(skip: int, limit: int, db: Session) -> List[Note]:

return db.query(Note).offset(skip).limit(limit).all()

async def get\_note(note\_id: int, db: Session) -> Note:

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

async def create\_note(body: NoteModel, db: Session) -> Note:

tags = db.query(Tag).filter(Tag.id.in\_(body.tags)).all()

note = Note(title=body.title, description=body.description, tags=tags)

db.add(note)

db.commit()

db.refresh(note)

return note

async def remove\_note(note\_id: int, db: Session) -> Note | None:

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

if note:

db.delete(note)

db.commit()

return note

async def update\_note(note\_id: int, body: NoteUpdate, db: Session) -> Note | None:

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

if note:

tags = db.query(Tag).filter(Tag.id.in\_(body.tags)).all()

note.title = body.title

note.description = body.description

note.done = body.done

note.tags = tags

db.commit()

return note

async def update\_status\_note(note\_id: int, body: NoteStatusUpdate, db: Session) -> Note | None:

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

if note:

note.done = body.done

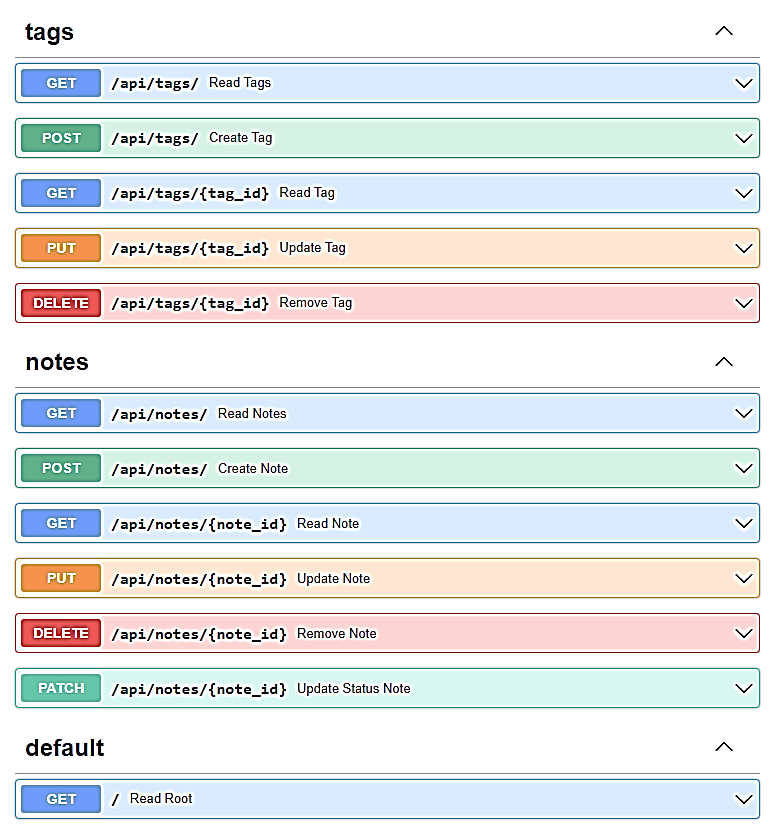
db.commit()

return note

Цей код містить функції репозиторію для обробки нотаток notes у базі даних. Ці функції використовуються для отримання, створення, оновлення та видалення нотаток з бази даних.

* get\_notes — витягує записи notes з бази даних із використанням заданих значень пропуску та обмеження.
* get\_note — витягує одну нотатку з бази даних за її id.
* create\_note — створює нову нотатку у базі даних, використовуючи дані з body.
* remove\_note — видаляємо нотатку за її унікальним id.
* update\_note — повне оновлення нотатки.
* update\_status\_note — оновлення статусу нотатки.

Ми отримали остаточний код для RESTful API, побудованого з використанням платформи FastAPI і SQLAlchemy. Він включає маршрути для обробки операцій CRUD на ресурсах notes і tags, а також функції в модулі repository, які взаємодіють з базою даних за допомогою SQLAlchemy для виконання цих операцій. Крім того, він також включає класи моделей і схем, які використовуються для визначення структури даних, що передаються в API та з нього.



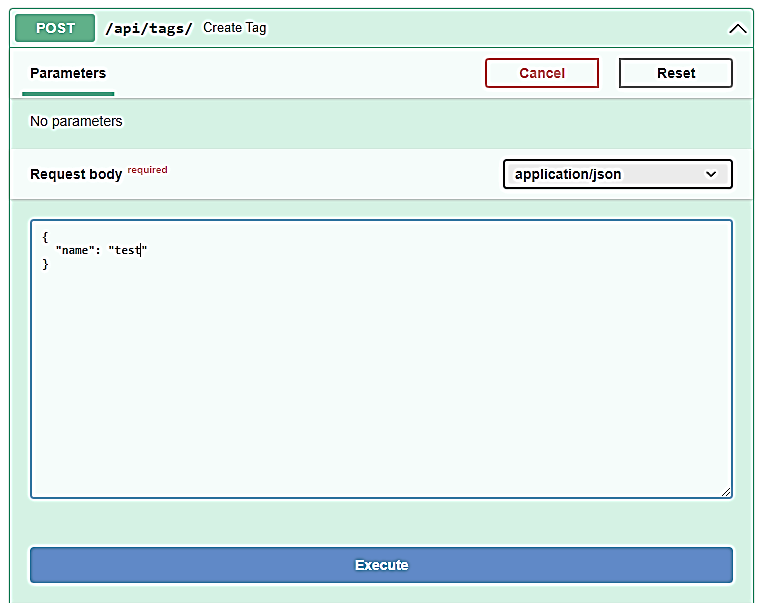
**Swagger документація отриманого RESTful API**

**Робота з API**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-api)Для роботи з нашим API потрібний клієнтський застосунок. Він може бути розроблений на якомусь фреймворку - React, Vue, Angular тощо. Також можна використовувати інструменти розробника, такі як [Postman](https://www.postman.com/) (https://www.postman.com/).

Ми, для взаємодії з побудованим REST API, будемо використовувати нашу Swagger документацію <http://127.0.0.1:8000/docs>.

Виконаємо операцію POST для маршруту http://127.0.0.1:8000/tags/. І створимо тег з назвою name=test.



**Створюємо перший тег**

У нас з'явився перший тег test.

{

"name": "test",

"id": 1

}

Створимо ще один тег, наприклад buy, і отримаємо від нашого API наступну відповідь.

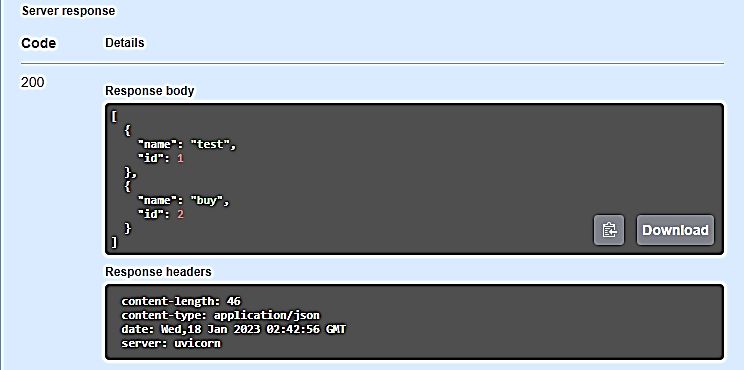
{

"name": "buy",

"id": 2

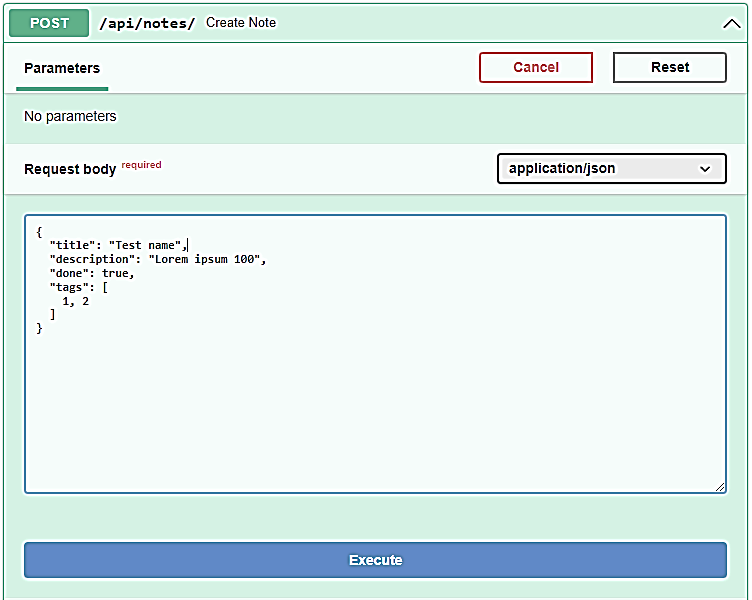
}

Тепер GET запит на адресу http://127.0.0.1:8000/tags/, який повинен повернути список створених тегів:



**Список створених тегів**

Тепер створімо нотатку з тегами, що існують.



**Створення нотатки**

Формат JSON, що відправляється на сервер, буде наступним:

{

"name": "Test name",

"description": "Lorem Ipsum 100",

"tags": [

1,

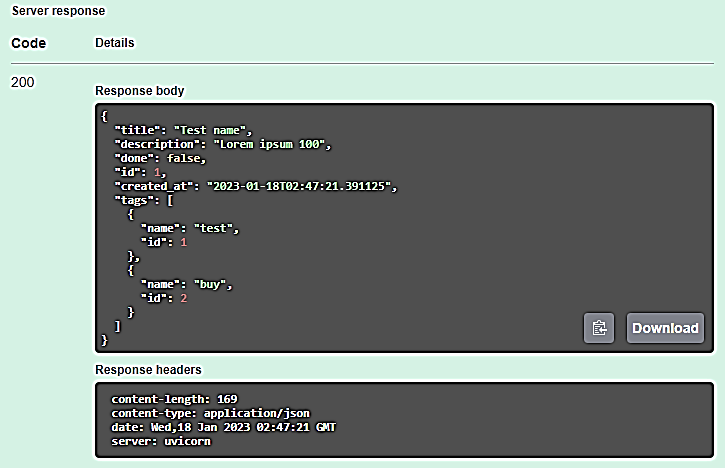
2

]

}

Де список tags містить id наших створених тегів.

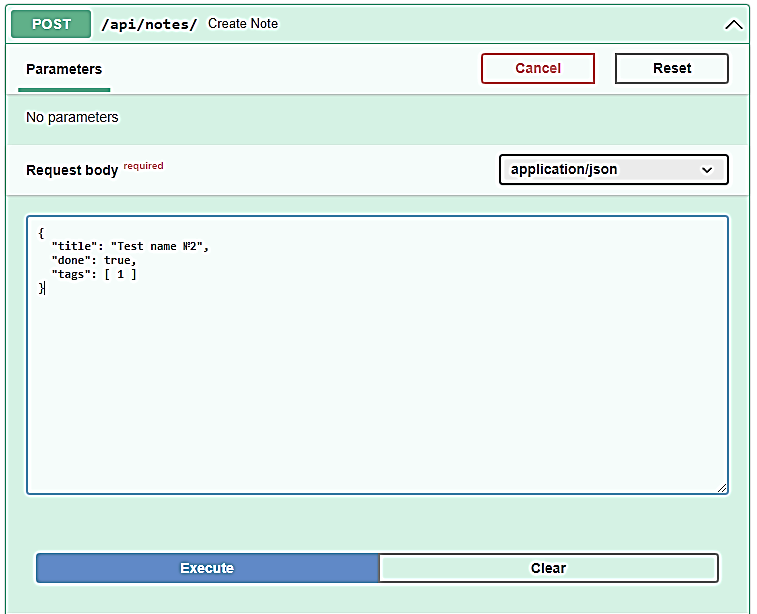
Отримаємо наступний результат:



**Результат створення нотатки**

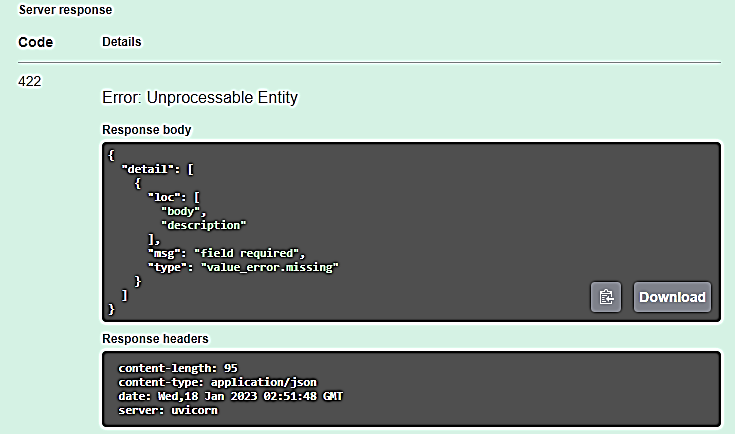
Тепер подивимося, чи працюють наші Pydantic схеми для валідації даних.

Спробуємо, наприклад, створити нотатку без обов'язкового поля description.



**Спроба створити неповну нотатку**

Отримаємо відповідь наступного виду зі статусом [422 Unprocessable Entity](https://developer.mozilla.org/ru/docs/Web/HTTP/Status/422) (https://developer.mozilla.org/ru/docs/Web/HTTP/Status/422):



**Результат спроби створити неповну нотатку**

Нам повідомляють, що пропущено "field required" обов'язкове поле "description" у тілі "body" запиту.

Як бачимо наш застосунок працює. Решту операцій CRUD та перевірку маршрутів виконайте самостійно.

**Висновок**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-11/module-11-02/fullexample#%D0%B2%D0%B8%D1%81%D0%BD%D0%BE%D0%B2%D0%BE%D0%BA)Ми отримали повноцінний**Rest API** веб-застосунок на FastAPI

Готовий результат повинен бути як у цьому живому прикладі, запустіть та подивіться код:

Щоб побачити Swagger документацію нашого застосунку, після запуску прикладу, натисніть [посилання](https://fastapi-rest-api.krabaton.repl.co/docs) (https://replit.com/@Krabaton/FastAPI-Rest-API)

**INFO**

Для полегшення живого прикладу було здійснено заміну бази даних на SQLite. Це пов'язано з тим, що сервіс Replit підтримує PostgeSQL тільки на платній основі. Також ми не використали міграції Alembic. Сам запуск сервера виконується прямо з коду:

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

uvicorn.run(app, host="0.0.0.0", port=8000)

Ви також можете [**завантажити повний приклад коду**](https://github.com/GoIT-Python-Web/REST-API) розділу для запуску на локальній машині.( https://github.com/GoIT-Python-Web/REST-API)