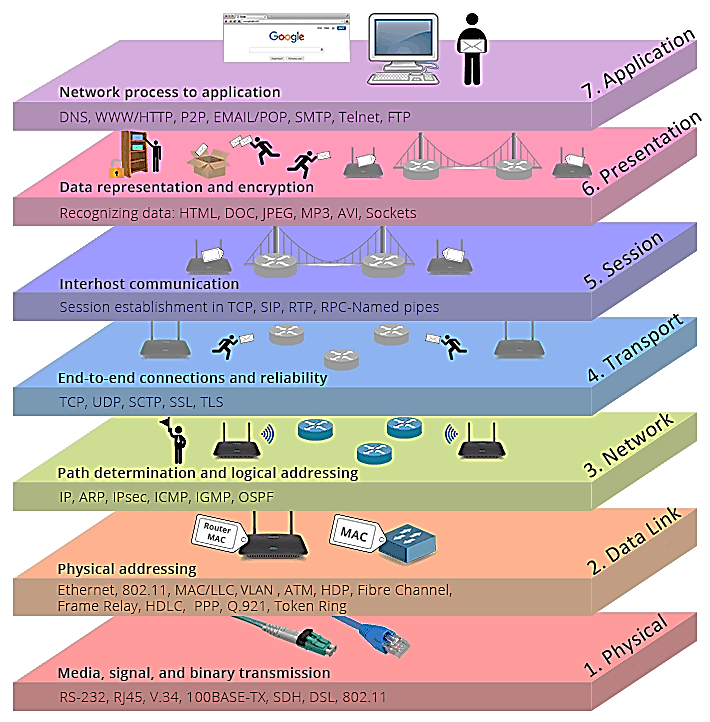
**Модуль 14: Додаткові матеріали**

**Взаємодія рівнів моделі OSI**

Завдання кожного рівня — надання послуг вищому рівню, «маскуючи» деталі реалізації цих послуг. При цьому кожен рівень на комп'ютері-відправнику працює так, ніби він безпосередньо пов'язаний з таким самим рівнем на комп'ютері-одержувачі. Цей логічний або віртуальний зв'язок між однаковими рівнями показаний на малюнку.



Однак, насправді зв'язок здійснюється між суміжними рівнями одного комп'ютера - програмне забезпечення, що працює на кожному рівні, реалізує певні мережеві функції відповідно до набору протоколів. Перед подачею в мережу дані розбиваються на пакети. Пакет (packet) - це одиниця інформації, що передається між пристроями мережі як єдине ціле. Пакет проходить послідовно через усі рівні програмного забезпечення. На кожному рівні до пакета додається деяка форматуюча інформація, або адресна, яка необхідна для успішної передачі даних мережею.

На приймаючій стороні пакет проходить через всі рівні в зворотному порядку. Програмне забезпечення на кожному рівні читає інформацію пакета, потім видаляє інформацію, додану до пакета на цьому ж рівні відправляючою стороною, і передає пакет на наступний рівень. Коли пакет дійде до Прикладного рівня, вся адресна інформація буде видалена та дані набудуть свого первісного вигляду.

Таким чином, за винятком нижнього рівня мережевої моделі, жодний інший рівень не може безпосередньо надіслати інформацію відповідному рівню іншого комп'ютера. Інформація на комп'ютері-відправнику повинна пройти через всі рівні. Потім вона передається по мережевому кабелю на комп'ютер-одержувач і знову проходить крізь усі шари, доки досягне того ж рівня, з якого вона була надіслана на комп'ютері-відправнику.

Наприклад, якщо мережевий рівень передає інформацію з комп'ютера А, вона спускається через Канальний та Фізичний рівні до мережевого кабелю, далі по ньому потрапляє в комп'ютер В, де піднімається через Фізичний та Канальний рівні та досягає Мережевого рівня.

Прикладом інформації, переданої мережевим рівнем комп'ютера А мережевому рівню комп'ютера, могла би слугувати адреса і, вочевидь, інформація контролю помилок, додані до пакета.

Взаємодія суміжних рівнів здійснюється через інтерфейс. Інтерфейс визначає послуги, які нижній рівень надає верхньому, і спосіб доступу до них. Кожному рівню одного комп'ютера "здається", що він безпосередньо взаємодіє з таким самим рівнем іншого комп'ютера.

Далі описується кожен із семи рівнів моделі OSI та визначаються послуги, які вони надають суміжним рівням.

**Прикладний рівень**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/web/osi#%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D0%B8%D0%B9-%D1%80%D1%96%D0%B2%D0%B5%D0%BD%D1%8C)**Рівень 7, Прикладний (Application)**, — найвищий рівень моделі OSI. Він є вікном для доступу прикладних процесів до мережевих послуг. Цей рівень забезпечує послуги, які безпосередньо підтримують застосунки користувача, такі як програмне забезпечення для передачі файлів, доступу до баз даних та електронна пошта. Нижчі рівні підтримують завдання, що виконуються на прикладному рівні. Прикладний рівень управляє загальним доступом до мережі, потоком даних та відновленням після збоїв зв'язку.

**Представницький рівень**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/web/osi#%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BD%D0%B8%D1%86%D1%8C%D0%BA%D0%B8%D0%B9-%D1%80%D1%96%D0%B2%D0%B5%D0%BD%D1%8C)**Рівень 6, Представницький (Presentation)**, визначає формат, який використовується для обміну даними між мережевими комп'ютерами. Цей рівень можна назвати перекладачем. На Представницькому рівні комп'ютера-відправника дані, що надійшли від Прикладного рівня, перетворюються на загальнозрозумілий проміжний формат.

На цьому ж рівні комп'ютера-отримувача відбувається зворотний переклад: із проміжного формату до того, що використовується Прикладним рівнем комп'ютера. Представницький рівень відповідає за перетворення протоколів, трансляцію даних, їх шифрування, зміну або перетворення набору символів (кодової таблиці), що застосовується, та розширення графічних команд. Представницький рівень, крім того, керує стисненням даних для зменшення загальної кількості переданих бітів.

На цьому рівні працює утиліта, яка називається редиректором (redirector). Її призначення – перенаправляти локальні операції введення/виведення на мережевий сервер.

**Сеансовий рівень**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/web/osi#%D1%81%D0%B5%D0%B0%D0%BD%D1%81%D0%BE%D0%B2%D0%B8%D0%B9-%D1%80%D1%96%D0%B2%D0%B5%D0%BD%D1%8C)

**Рівень 5, Сеансовий (Session)**, дозволяє двом застосункам на різних комп'ютерах встановлювати, використовувати і завершувати з'єднання, що називається сеансом. На цьому рівні виконуються такі функції, як розпізнавання імен та захист, необхідні для зв'язку двох застосунків у мережі.

Сеансовий рівень забезпечує синхронізацію між завданнями користувача за допомогою розміщення в потоці даних контрольних точок (checkpoints). Таким чином, у разі помилки потрібно знову передати лише дані останньої контрольної точки. Цей рівень управляє діалогом між взаємодіючими процесами, тобто регулює, яка зі сторін коли, як довго тощо, повинна здійснювати передачу.

**Транспортний рівень**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/web/osi#%D1%82%D1%80%D0%B0%D0%BD%D1%81%D0%BF%D0%BE%D1%80%D1%82%D0%BD%D0%B8%D0%B9-%D1%80%D1%96%D0%B2%D0%B5%D0%BD%D1%8C)**Рівень 4, Транспортний (Transport)**, розташовується нижче Сеансового рівня. Транспортний рівень гарантує доставку пакетів без помилок, у тій самій послідовності, без втрат та дублювання. На цьому рівні комп'ютера-відправника повідомлення перепаковуються: довгі розбиваються на кілька пакетів, а короткі об'єднуються в один. Це збільшує ефективність передачі пакетів мережею. На Транспортному рівні комп'ютера-отримувача повідомлення розпаковуються, відновлюються в початковому вигляді, і, зазвичай, надсилається сигнал підтвердження приймання.

Транспортний рівень управляє потоком повідомлень, перевіряє помилки та бере участь у вирішенні проблем, пов'язаних з відправленням та отриманням пакетів.

**Мережевий рівень**

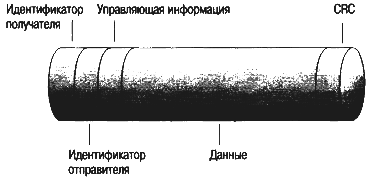
[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/web/osi#%D0%BC%D0%B5%D1%80%D0%B5%D0%B6%D0%B5%D0%B2%D0%B8%D0%B9-%D1%80%D1%96%D0%B2%D0%B5%D0%BD%D1%8C)**Рівень 3, Мережевий (Network)**, відповідає за адресацію повідомлень та перетворення логічних адрес та імен у фізичні адреси. Одним словом, виходячи з конкретних мережевих умов, пріоритету послуги та інших факторів, тут визначається маршрут від комп'ютера-відправника до комп'ютера-отримувача. На цьому рівні вирішуються також такі завдання та проблеми, пов'язані з мережевим графіком, як комутація пакетів, маршрутизація та перевантаження.

Якщо мережевий адаптер маршрутизатора не може передавати великі блоки даних, надіслані комп'ютером-відправником, на Мережевому рівні ці блоки розбиваються на менші. А Мережевий рівень комп'ютера-отримувача збирає ці дані у вихідний стан.

**Канальний рівень**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/web/osi#%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%B8%D0%B9-%D1%80%D1%96%D0%B2%D0%B5%D0%BD%D1%8C)**Рівень 2, Канальний (Data Link)**, здійснює передачу кадрів (frames) даних від Мережевого рівня до Фізичного. Кадри - це логічно організована структура, в яку можна розміщувати дані. Канальний рівень комп'ютера-отримувача упаковує «сирий» потік бітів, які надходять від Фізичного рівня, у кадри даних. На мал. представлений простий кадр даних, де ідентифікатор відправника - адреса комп'ютера-відправника, а ідентифікатор одержувача – адреса комп'ютера-отримувача.

Керуюча інформація використовується для маршрутизації, а також вказує на тип пакета та сегментацію. Дані — інформація, що власне передається. CRC (залишок надлишкової циклічної суми) - це відомості, які допоможуть виявити помилки, що, у свою чергу, гарантує правильне приймання інформації.



**Мал. Кадр даних**

Канальний рівень забезпечує точність передачі кадрів між комп'ютерами через фізичний рівень. Це дозволяє Мережевому рівню зчитати передачу даних мережевим з'єднанням практично безпомилковою.

Зазвичай, коли Канальний рівень посилає кадр, він очікує з боку одержувача підтвердження приймання. Канальний рівень одержувача перевіряє наявність можливих помилок передачі. Кадри, пошкоджені під час передачі або не отримали підтвердження про приймання, надсилаються знову.

**Фізичний рівень**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/web/osi#%D1%84%D1%96%D0%B7%D0%B8%D1%87%D0%BD%D0%B8%D0%B9-%D1%80%D1%96%D0%B2%D0%B5%D0%BD%D1%8C)**Рівень 1, Фізичний (Physical)**, — найнижчий у моделі OSI. Цей рівень здійснює передачу неструктурованого, «сирого» потоку бітів фізичним середовищем (наприклад, по мережевому кабелю). Тут реалізуються електричний, оптичний, механічний та функціональний інтерфейси з кабелем. Фізичний рівень також формує сигнали, які переносять дані, що надійшли від усіх вищих рівнів.

На цьому рівні визначається спосіб з'єднання мережевого кабелю з платою мережевого адаптера, зокрема кількість контактів у роз'ємах та їх функції. Крім того, тут визначається спосіб передачі сигналів мережевим кабелем. Фізичний рівень призначений для передачі бітів (нулів і одиниць) від одного комп'ютера до іншого. Зміст самих бітів на цьому рівні значення немає. Цей рівень відповідає за кодування даних та синхронізацію бітів, гарантуючи, що передана одиниця буде сприйнята саме як одиниця, а не як нуль. Зрештою, Фізичний рівень встановлює тривалість кожного біта та спосіб перетворення біта у відповідні електричні або оптичні імпульси, передані по мережевому кабелю.

**Основи роботи з**[**Flask**](http://flask.pocoo.org/) (https://flask.palletsprojects.com/en/3.0.x/)**.**

Одним із найпопулярніших сучасних веб-фреймворків на Python можна назвати [Flask](http://flask.pocoo.org/) (https://flask.palletsprojects.com/en/3.0.x/). Flask досить простий для новачків і відмінно підходить для невеликих застосунків із не до кінця визначеним набором вимог. Flask надає мінімальний набір інструментів для створення вебзастосунків. Саме простота і гнучкість роблять Flask гарним варіантом для першого веб-фреймворку.

Почати знайомство з Flask можна зі створення простого застосунку, який виводить “Hello World”. Створюємо новий файл main.py і вводимо наступний код.

from flask import Flask

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

@app.route('/')

def index():

return 'Hello World'

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

app.run()

Ця програма “Hello World” створена за допомогою фреймворку Flask. Щоб запустити сервер, потрібно виконати:

python main.py

Запуск файлу main.py запускає локальний сервер для розробки на порту 5000. Залишилося відкрити улюблений браузер та зайти на <https://127.0.0.1:5000/>, щоб побачити застосунок Hello World у дії.

Зупинити сервер можна за допомогою комбінації CTRL+C.

У кожного Flask-застосунку повинен бути екземпляр класу. Екземпляр - це WSGI-застосунок (WSGI - це інтерфейс для взаємодії сервера з фреймворком), який показує, що сервер передає всі отримані запити екземпляру для подальшої обробки. Об'єкт класу Flask створюється наступним чином:

from flask import Flask

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

У другому рядку створюється об'єкт Flask. Для цього конструктору Flask призначається аргумент \_\_name\_\_. Конструктор Flask повинен мати обов'язковий аргумент. Ним слугує назва пакета. У більшості випадків значення \_\_name\_\_ підходить. Назва пакета застосунку використовується фреймворком Flask, щоб знаходити статичні файли, шаблони тощо.

Маршрут (або шлях) використовується у Flask для прив'язки URL до функції представлення. Ця функція відповідає на запит. У Flask декоратор **route** використовується, щоб зв'язати URL-адресу з функцією. Ось як створюється маршрут.

@app.route('/')

def index():

return 'Hello World'

Цей код призначає функцію index() обробником кореневого URL у застосунку. Іншими словами, щоразу, коли застосунок буде отримувати запит, де шлях - /, викликається функція index(), і на цьому запит завершується.

Як варіант, можна використовувати метод **add\_url\_rule()**, замість декоратора route для маршрутизації.**add\_url\_rule()** - це простий метод, а не декоратор. Крім URL, він приймає кінцеву точку та назву функції представлення. Кінцева точка належить до унікального імені маршруту. Зазвичай назва функції представлення — це і є кінцева точка. Flask може генерувати URL із кінцевої точки, але про це пізніше. Попередній код аналогічний наступному:

def index():

return 'Hello World'

app.add\_url\_rule('/', 'index', index)

Декоратор route використовується в більшості випадків, але **add\_url\_rule()** має свої переваги. Функція представлення повинна повернути рядок. Якщо намагатись повернути щось інше, сервер відповість помилкою 500 Internal Sever Error.

Наприклад, у наступному списку 3 шляхи.

@app.route('/')

def index():

return 'Home Page'

@app.route('/career/')

def career():

return 'Career Page'

@app.route('/feedback/')

def feedback():

return 'Feedback Page'

Коли URL у маршруті закінчується слешем (/), Flask перенаправляє запит без слеша на URL зі слешем. Так, запит до**/career**буде перенаправлений на**/career/.**

Для однієї функції представлення можна використовувати кілька URL.

Наприклад:

@app.route('/contact/')

@app.route('/feedback/')

def feedback():

return 'Feedback Page'

У цьому випадку, у відповідь на запити **/contact/** або **/feedback/**, буде викликана функція**feedback().**

Якщо перейти за адресою, для якої немає відповідної функції представлення, з'явиться помилка 404 Not Found. Ці маршрути статичні. Більшість сучасних застосунків мають динамічні URL-адреси. Динамічний URL — це адреса, яка складається з однієї або кількох змінних частин, що впливають на виведення сторінки. Наприклад, при створенні вебзастосунку зі сторінками профілів, у кожного користувача буде унікальний id. Профіль першого користувача буде на сторінці **/user/1**, другого — на **/user/2**і так далі. Дуже незручний спосіб досягти такого результату — створювати маршрути для кожного користувача окремо.

Натомість можна відзначити динамічні частини URL як <variable\_name>(змінні). Ці частини потім передаватимуть ключові слова функції відображення. Наступний код демонструє шлях із динамічним елементом.

@app.route('/user/<id>/')

def user\_profile(id):

return "Profile page of user #{}".format(id)

У цьому прикладі на місці <id> буде вказуватись частина URI, яка йде після /user/. Наприклад, якщо зайти на /user/100/, відповідь буде наступною.

Profile page of user #100

За замовчуванням динамічні частини URL-адреси передаються у функцію у вигляді рядків. Це можна змінити за допомогою конвертера, який вказується перед динамічними елементами URL за допомогою <converter:variable\_name>.

Наприклад, /user/<int:id>/ працюватиме з адресами /user/1/, /user/200/ та іншими. Але /user/cowboy/, /user/foobar10/ та /user/@@##/ не підійдуть.

Доступні у Flask конвертери:

* string (за замовчуванням)
* int
* float
* path (рядки та слеші)
* uuid (символьні id)

Для запуску сервера розробки необхідно використовувати метод **run()**об'єкта Flask.

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

app.run()

Сервер розробки Flask використовується виключно для тестування, тому його продуктивність невисока.

Аргументи URL-адреси, які додаються до рядка запиту, часто використовуються для передачі даних у вебзастосунок. Можливо, вам вже доводилося бачити рядок запиту на веб-сторінках.

example.com?arg1=value1&arg2=value2

Рядок запиту починається після знака (?) питання. Він містить пари ключ-значення, розділені символом амперсанда (&). У кожній парі після ключа йде знак дорівнює (=), а потім іде значення. Рядки запитів корисні для передачі даних, які вимагають дій з боку користувача. Ви можете згенерувати рядок запиту у своєму застосунку та додати його до URL так, щоб при запиті користувача дані передавалися автоматично. Рядок запиту може бути згенерований формами, що використовують метод GET.

Щоб обробляти аргументи запиту, можна скористатися проксі-об'єктом запиту request. Це спеціальний об'єкт Flask, який є унікальним для кожного запиту і містить всю інформацію про запит.

Вам потрібно буде програмувати частину, яка оброблятиме аргументи запиту.

Код, який зчитує ключ **name,**

використовуючи **request.args.get('name')** або **request.args['name'].**

from flask import request

@app.route('/find/user')

def find\_user():

user\_name = request.args.get('name')

return f"Searching for user by name '{user\_name}'"

Потім запустіть застосунок і перейдіть до URL:

http://127.0.0.1:5000/find/user?name=Test

Використання даних форм передбачає відправлення запиту POST з формою в тілі запиту.

Замість відображення даних в URL (крім випадків відправлення форм у вигляді запиту GET), дані форм передаються застосунком непомітно. Хоча ви не бачите дані форм, що передаються, ваш застосунок може їх зчитувати.

Щоб продемонструвати це, давайте створимо маршрут **form-example** такий, щоб приймав запити GET та POST, та повертав форму:

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

def form\_example():

return '''

<form method="POST">

<div><label>Language: <input type="text" name="language"></label></div>

<div><label>Framework: <input type="text" name="framework"></label></div>

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

</form>'''

Потім запустіть застосунок і перейдіть до URL:

http:*//127.0.0.1:5000/form-example*

Браузер повинен відображати форму з двома полями введення: одне — language та одне — для framework, а також кнопку відправлення.

Найважливіше знати про цю форму те, що вона виконує запит POST до того самого маршруту, який згенерував форму. Всі ключі, які зчитуються у застосунку, надходять від атрибутів name у полях введення форми. У цьому випадку language та framework є іменами полів введення, і тому ви матимете доступ до них у застосунку.

Всередині функції перегляду вам доведеться перевірити метод запиту: GET або POST. Якщо це запит GET, можна вивести форму. В іншому випадку - це запит POST, і вам потрібно обробити вхідні дані.

Змініть маршрут form-example в app.py, додавши наступний код:

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

def form\_example():

*# handle the POST request*

if request.method == 'POST':

language = request.form.get('language')

framework = request.form.get('framework')

return '''

<h1>The language value is: {}</h1>

<h1>The framework value is: {}</h1>'''.format(language, framework)

*# otherwise handle the GET request*

return '''

<form method="POST">

<div><label>Language: <input type="text" name="language"></label></div>

<div><label>Framework: <input type="text" name="framework"></label></div>

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

</form>'''

Потім запустіть застосунок та перейдіть на ту саму URL. Введіть у полі language значення Python, а в полі framework — значення Flask. Потім натисніть кнопку Submit.

Браузер повинен вивести наступне повідомлення:

The language value is: Python

The framework value is: Flask

**Практичний приклад**

**Підготовка проєкту**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/flask/example#%D0%BF%D1%96%D0%B4%D0%B3%D0%BE%D1%82%D0%BE%D0%B2%D0%BA%D0%B0-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D1%83)У цьому завданні ми з вами реалізуємо просту систему управління нотатками з використанням популярного фреймворку [**Flask**](https://flask.palletsprojects.com/en/2.0.x/) (<https://flask.palletsprojects>. com/en/2.0.x/)**.**

Для доступу до БД будемо використовувати ORM підхід на основі [SQLAlchemy](https://www.sqlalchemy.org/) (https://www.sqlalchemy.org/), з яким ви вже знайомі.

Встановимо потрібні нам пакети:

* poetry add flask
* poetry add sqlalchemy

**Створення шаблонів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/flask/example#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%96%D0%B2)Виведення даних робитимемо з використанням "шаблонізатора" [**Jinja2**](https://lectureswww.readthedocs.io/6.www.sync/2.codding/3.templates/jinja2.html) (https://lectureswww.readthedocs.io/6.www.sync/2.codding/3.templates/jinja2.html), який використовується в багатьох популярних web-фреймворках на Python. Для стилізації розмітки ми використовуватимемо фреймворк [**Pico.css**](https://picocss.com/) (https://picocss.com/)та підключимо його в нашому проекті. Усі файли шаблону повинні лежати в папці templates в корені проекту. У ній буде один файл index.html - точка входу, і кілька файлів html для допоміжних операцій. Ми будемо використовувати файл index.html як шаблон для виведення головної сторінки, а файли note.html, tag.html та detail.html - для виведення інформації про нотатку та форми для її редагування. Нагадаємо, що всі файли шаблонів повинні бути в одній папці templates.

**Головний шаблон**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/flask/example#%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BD%D0%B8%D0%B9-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD)Файл index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Organizer</title>

<link

rel="stylesheet"

href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"

/>

</head>

<body>

<main class="container">

<nav

style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px"

>

<ul>

<li><a href="/note/" role="button">Add note</a></li>

<li><a href="/tag/" role="button">Add Tag</a></li>

</ul>

</nav>

{% for note in notes%}

<ul>

<li>

{%if not note.done%}

<a href="/detail/{{note.id}}" role="button" class="outline"

>{{note.name}}</a

>

<a href="/done/{{note.id}}" role="button">Mark Done</a>

<a href="/delete/{{note.id}}" role="button" class="contrast"

>Delete note</a

>

{% else %}

<a href="/detail/{{note.id}}" role="button" class="secondary"

><s>{{note.name}}</s></a

>

{% endif %}

</li>

</ul>

{% endfor %}

</main>

</body>

</html>

**Шаблон створення тегу**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/flask/example#%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%B5%D0%B3%D1%83)

Файл tag.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8"/>

<title>Add new tag</title>

<link

rel="stylesheet"

href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"

/>

</head>

<body>

<main class="container">

<nav

style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px"

>

<ul>

<li><a href="/" role="button"> Go back to the main page </a></li>

</ul>

</nav>

<form method="POST" action="/tag">

<div style="padding: 10px">

<label

>Enter the tag name:

<input type="text" name="name" required/>

</label>

</div>

<div class="grid">

<button type="reset" class="secondary">Reset</button>

<button type="submit">Submit</button>

</div>

</form>

</main>

</body>

</html>

**Шаблон створення нотатки**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/flask/example#%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BD%D0%BE%D1%82%D0%B0%D1%82%D0%BA%D0%B8)Файл note.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Add note</title>

<link

rel="stylesheet"

href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"

/>

</head>

<body>

<main class="container">

<nav

style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px"

>

<ul>

<li><a href="/" role="button"> Go back to the main page </a></li>

</ul>

</nav>

<form method="POST" action="/note">

<div style="padding: 10px">

<label

>Note name:

<input type="text" name="name" required />

</label>

</div>

<div style="padding: 10px">

<label

>Note description:

<input type="text" name="description" required />

</label>

</div>

<div style="padding: 10px">

<label

>Select tags:

<select name="tags" multiple="multiple" required>

{% for tag in tags %}

<option value="{{tag.name}}">{{tag.name}}</option>

{% endfor %}

</select>

</label>

</div>

<div class="grid">

<button type="reset" class="secondary">Reset</button>

<button type="submit">Submit</button>

</div>

</form>

</main>

</body>

</html>

**Шаблон перегляду нотатки**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/flask/example#%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD-%D0%BF%D0%B5%D1%80%D0%B5%D0%B3%D0%BB%D1%8F%D0%B4%D1%83-%D0%BD%D0%BE%D1%82%D0%B0%D1%82%D0%BA%D0%B8)

Файл detail.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<title>Note information</title>

<link

rel="stylesheet"

href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"

/>

</head>

<body>

<main class="container">

<nav

style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px"

>

<ul>

<li><a href="/" role="button"> Go back to the main page </a></li>

</ul>

</nav>

<h1>{{note.name}}</h1>

<p>Description: <b>{{note.description}}</b></p>

<p>Time of creation: <kbd>{{note.created.date()}}</kbd></p>

{% if note.done %}

<p>Status: <ins>Completed</ins></p>

{% else %}

<p>Status: <i>Not implemented</i></p>

{% endif %}

<p>Tags: <small> {{ note.tags|join(', ', attribute='name') }} </small></p>

</main>

</body>

</html>

**Підключення до бази даних**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/flask/example#%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)Виконаємо вже знайоме підключення до бази даних. Будемо використовувати SQLite у якості бази даних.

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.engine import create\_engine

from sqlalchemy.orm import sessionmaker, scoped\_session

engine = create\_engine(

"sqlite:///mynotes.db", connect\_args={"check\_same\_thread": False}, echo=True

)

db\_session = scoped\_session(

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

)

Base = declarative\_base()

**Створення моделей**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/flask/example#%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)Виконаємо створення моделей для об'єктів бази даних. Моделі наших класів опишемо у файлі **models.py.**

from datetime import datetime

from sqlalchemy import Column, Integer, String, Boolean

from sqlalchemy.orm import relationship

from sqlalchemy.sql.schema import ForeignKey, Table, MetaData

from sqlalchemy.sql.sqltypes import DateTime

from db import Base, engine, db\_session

*# таблиця для зв'язку many2many*

note\_m2m\_tag = Table(

"note\_m2m\_tag",

Base.metadata,

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

Column("note", Integer, ForeignKey("notes.id")),

Column("tag", Integer, ForeignKey("tags.id")),

)

class Note(Base):

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

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

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

created = Column(DateTime, default=datetime.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)

def \_\_repr\_\_(self) -> str:

return self.name

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

Base.metadata.create\_all(engine)

**Основний модуль застосунку**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/flask/example#%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D0%B8%D0%B9-%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C-%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BA%D1%83)Створимо файл та назвемо його **main.py** - це буде основний файл нашого застосунку. Необхідний код для запуску сервера виглядатиме так:

from flask import Flask, render\_template, request, redirect

from models import Note, Tag, db\_session

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

app.debug = True

app.env = "development"

@app.route("/", strict\_slashes=False)

def index():

notes = db\_session.query(Note).all()

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

@app.route("/detail/<id>", strict\_slashes=False)

def detail(id):

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

return render\_template("detail.html", note=note)

@app.route("/note/", methods=["GET", "POST"], strict\_slashes=False)

def add\_note():

if request.method == "POST":

name = request.form.get("name")

description = request.form.get("description")

tags = request.form.getlist("tags")

tags\_obj = []

for tag in tags:

tags\_obj.append(db\_session.query(Tag).filter(Tag.name == tag).first())

note = Note(name=name, description=description, tags=tags\_obj)

db\_session.add(note)

db\_session.commit()

return redirect("/")

else:

tags = db\_session.query(Tag).all()

return render\_template("note.html", tags=tags)

@app.route("/tag/", methods=["GET", "POST"], strict\_slashes=False)

def add\_tag():

if request.method == "POST":

name = request.form.get("name")

tag = Tag(name=name)

db\_session.add(tag)

db\_session.commit()

return redirect("/")

return render\_template("tag.html")

@app.route("/delete/<id>", strict\_slashes=False)

def delete(id):

db\_session.query(Note).filter(Note.id == id).delete()

db\_session.commit()

return redirect("/")

@app.route("/done/<id>", strict\_slashes=False)

def done(id):

db\_session.query(Note).filter(Note.id == id).first().done = True

db\_session.commit()

return redirect("/")

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

app.run()

Якщо запустити проект python main.py, то доступ до проекту ми отримаємо за адресою <http://127.0.0.1:5000/>

Після запуску прикладу, сам проект можна відкрити за цим [посиланням](https://flask-example.krabaton.repl.co/) (https://replit.com/@Krabaton/Flask-Example).

**Основи роботи з Aiohttp**

[**AIOHTTP**](https://docs.aiohttp.org/en/stable/) (https://docs.aiohttp.org/en/stable/) — це асинхронний фреймворк для створення мережевих застосунків мовою Python. AIOHTTP дозволяє створювати клієнт-серверні застосунки з використанням протоколу HTTP і підтримує асинхронні запити та відповіді, що дає змогу створювати ефективні та швидкі застосунки.

AIOHTTP розроблено для використання в асинхронних застосунках, заснованих на подієвій моделі програмування. Він використовує asyncio бібліотеку Python для обробки запитів і відповідей, що дає змогу обробляти велику кількість запитів одночасно, без блокування застосунку на очікування відповіді.

Основні переваги AIOHTTP включають:

* фреймворк модульний з добре продуманою архітектурою, містить і клієнтську, і серверну частину;
* фреймворк дуже гнучкий, один і той самий функціонал можна реалізувати кількома способами, обравши найвідповідніший;
* продумана архітектура спрощує тестування застосунку;
* фреймворк містить мінімальний набір інструментів, що робить його легким в освоєнні;
* легко розширюється за допомогою готових доповнень або використання інтерфейсів, що надаються самим фреймворком;
* добре документований;
* завдяки асинхронній архітектурі фреймворк демонструє високий рівень продуктивності та утилізації ресурсів;
* у фреймворку реалізований сервер високої якості, який підходить для застосування на реальних проектах, що значно спрощує запуск застосунку.

AIOHTTP також має безліч додаткових функцій, як-от підтримка cookies, авторизація та обробка форм даних, що робить його корисним інструментом для створення вебзастосунків будь-якого рівня складності.

До недоліків можна віднести:

* екосистема фреймворка ще не розвинена достатньо, розширень не багато і не всі з них хорошої якості;
* гнучкість фреймворка дає можливість наробити помилок недосвідченим розробникам, варто витратити час на вивчення документації та не копіювати готові рішення необдумано;

**Серверна частина**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/intro#%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%BD%D0%B0-%D1%87%D0%B0%D1%81%D1%82%D0%B8%D0%BD%D0%B0)Фреймворк доступний у репозиторії пакетів [**PyPI**](https://pypi.org/project/aiohttp/) (https://pypi.org/project/aiohttp/) і встановлюється:

pip install aiohttp

Реалізуємо для початку просту серверну частину:

from aiohttp import web

async def handle(request):

name = request.match\_info.get('name', "Anonymous")

text = "Hello, " + name

return web.Response(text=text)

app = web.Application()

app.add\_routes([web.get('/', handle),

web.get('/{name}', handle)])

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

web.run\_app(app, host='localhost')

Цей приклад сервера взято з документації. Розберемо, що відбувається в цьому прикладі.

Усі запити до сервера повинні оброблятися асинхронними функціями-обробниками запитів. Функція-обробник у нашому випадку називається handle.

Функції-обробники отримують як аргумент об'єкт класу Request — це змінна request у нашому прикладі. Об'єкт Request містить всю необхідну інформацію про запит.

Повертати функція-обробник handle повинна об'єкт Response або будь-який інший клас, що наслідує Response. Поки що пропустимо етап опрацювання запиту і рушимо далі кодом до створення застосунку та реєстрації функції-обробника.

Застосунок AIOHTTP — це об'єкт класу Application. Кожен застосунок містить свій контекст, до якого у функцій-обробників є доступ. Контекстом зазвичай буває: інструмент логування, з'єднання з базою даних, набір middleware (універсальні проміжні обробники) тощо. Коли ми створили застосунок, потрібно зареєструвати в ньому функції-обробники, які відповідатимуть за обробку запитів на зазначені ресурси. Для цього можна викликати метод add\_routes, який отримує список функцій-обробників.

Щоб зі звичайної асинхронної функції, що реалізує інтерфейс обробника Request на вході, а Response на виході, зробити функцію-обробник, можна скористатися допоміжними утилітами в модулі web.

У цьому прикладі ми створюємо обробники GET запитів і використовуємо метод-утиліту get з об'єкта web.

Метод get отримує два обов'язкові параметри:

* адресу (URL), за якою буде зареєстрований обробник;
* функцію, що реалізує протокол обробника (Request на вході, Response на виході);

INFO

У модулі web є утиліти для створення обробників POST, UPDATE, DELETE та інших типів запитів.

У прикладі ми створили обробники запитів GET на кореневий маршрут / і на кореневий маршрут із параметром /{name}. Тут змінна name - це параметр GET запиту.

app.add\_routes([web.get('/', handle),

web.get('/{name}', handle)])

Для запуску сервера достатньо викликати метод run\_app, передавши йому як аргумент об'єкт класу Application — у нашому прикладі змінна app.

**Обробка параметрів запиту**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/intro#%D0%BE%D0%B1%D1%80%D0%BE%D0%B1%D0%BA%D0%B0-%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D1%96%D0%B2-%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D1%83)Об'єкт Request містить всю інформацію про запит, а більшість атрибутів реалізують інтерфейс словника. Звернення до елементів словника виконується через синтаксис квадратних дужок або метод get. Пам'ятайте, що якщо елемент за ключем не знайдений, це викличе виняток KeyError:

* match\_info — словник із параметрами URL.
* headers — містить оброблені поля HTTP заголовка;
* query — містить поля з рядка запиту URL;
* cookies — набір полів із кук (cookie);
* `scheme — рядок, що вказує на схему запиту (http або https);
* method — рядок із типом запиту (POST, GET, DELETE, ...);

Оскільки тіло запиту може бути великим, для поліпшення продуктивності в AIOHTTP ви можете прочитати тільки параметри із заголовка запиту перед тим, як вирішувати або читати далі тіло запиту і що з ним робити. Для цього у класу Request доступ до тіла запиту реалізовано через виклик асинхронних методів:

* read повертає тіло запиту (тип bytes);
* text повертає тіло запиту, але намагається декодувати його (тип str);
* json якщо в запиті вказано, що в тілі міститься JSON-документ, то документ буде декодований, і ви отримаєте Python словник із вмістом тіла запиту;
* multipart повертає тіло запиту, передане за типом multipart/form-data, коли запит із великим об'єктом у тілі розбивається на кілька запитів (зазвичай так передають файли на сервер).

**Формування відповіді**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/intro#%D1%84%D0%BE%D1%80%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B2%D1%96%D0%B4%D0%BF%D0%BE%D0%B2%D1%96%D0%B4%D1%96)Для повернення відповіді клієнту використовується клас Response та його нащадки. Об'єкти класу Response реалізують інтерфейс словника і до їхніх атрибутів можна звертатися як до елементів словника. Атрибутами можуть бути:

* status (за замовчуванням 200, якщо не вказати інший);
* headers — заголовок;
* cookies кукі;

Залежно від типу відповіді тіло може формуватися по-різному.

* Для класу Response можна передати в конструктор (створення об'єкта класу) body — byte рядок з тілом відповіді.
* Або для класу Response можна передати в конструктор (створення об'єкта класу) text — str рядок із тілом відповіді.
* Для створення сокетів використовується клас WebSocketResponse, об'єкт цього класу використовується для комунікації з клієнтом через методи send\_str, receive.

Щоб спростити створення JSON відповідей, можна використовувати допоміжну функцію json\_response, вона перетворює аргумент data в JSON і поверне Response об'єкт з потрібними заголовками.

Окремо ви можете викликати винятки спадкоємці web.HTTPException, AIOHTTP коректно опрацює такий виняток і поверне відповідну HTTP відповідь клієнту.

**Контекст запиту**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/intro#%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82-%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D1%83)Кожен запит до сервера від клієнта обробляється в окремій корутині. Для кожного запиту створюється свій контекст запиту. Але у вас також є можливість створювати глобально доступні об'єкти. Зазвичай такими об'єктами є з'єднання з базою даних, налаштування тощо.

Для цього ви можете використовувати застосунок, об'єкт Application, як словник:

app['my\_private\_key'] = data

async def handler(request):

data = request.app['my\_private\_key']

У цьому прикладі ми зберегли деякий об'єкт data у нашому застосунку й отримуємо у функції-обробнику доступ до цього об'єкту, використовуючи поле app у request.

Для багатошарових застосунків, коли є застосунок і в нього кілька дочірніх, ви можете скористатися атрибутом config\_dict в об'єкті request.

Це ChainMap об'єкт, який об'єднує всі налаштування всіх батьківських застосунків, включно з поточним.

**Middleware**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/intro#middleware)Що робити, якщо потрібно виконати якісь дії для всіх запитів застосунку, незалежно від запитуваного ресурсу? Найчастіше, це завдання логування, перевірки правильності запиту тощо. У такому разі використовують механізм middleware.

Сама middleware безпосередньо викликає функцію-обробник і повертає відповідь. Протокол middleware відрізняється від протоколу функції-обробника тільки другим обов'язковим аргументом, самою функцією-обробником. Це дає можливість виконати якісь дії із запитом або відповіддю від функції-обробника всередині middleware.

from aiohttp import web

async def test(request):

print('Handler function called')

return web.Response(text="Hello")

@web.middleware

async def middleware1(request, handler):

print('Middleware 1 called')

response = await handler(request)

print('Middleware 1 finished')

return response

@web.middleware

async def middleware2(request, handler):

print('Middleware 2 called')

response = await handler(request)

print('Middleware 2 finished')

return response

app = web.Application(middlewares=[middleware1, middleware2])

app.router.add\_get('/', test)

web.run\_app(app, host = 'localhost')

Для додавання middleware у застосунок, при створенні об'єкта застосунку можна передати йому список middleware у параметрі middlewares.

Кожен middleware повинен приймати обов'язково два аргументи: сам запит і функцію-обробник. Щоб створити middleware в AIOHTTP, можна задекорувати асинхронну функцію декоратором middleware з модуля web.

Усі middleware об'єднуються в ланцюжок викликів і виконуються від першого в списку до останнього. Якщо в цьому прикладі зробити запит на кореневий ресурс, то в консолі побачите:

Middleware 1 called

Middleware 2 called

Handler function called

Middleware 2 finished

Middleware 1 finished

**Підсумок**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/intro#%D0%BF%D1%96%D0%B4%D1%81%D1%83%D0%BC%D0%BE%D0%BA)Загалом, AIOHTTP надає потужний і гнучкий інструментарій для створення високопродуктивних вебзастосунків на Python, даючи змогу розробникам ефективно працювати із запитами, відповідями та контекстом запиту. Використання асинхронності та корутин в AIOHTTP дає змогу досягти високої продуктивності застосунків і ефективно використовувати ресурси системи. Давайте розглянемо практичний приклад застосування фреймворка.

**Підготовка проєкту**

**Створюємо базовий застосунок**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/create#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D1%8E%D1%94%D0%BC%D0%BE-%D0%B1%D0%B0%D0%B7%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)Реалізуємо систему управління нотатками з використанням фреймворка [**AIOHTTP**](https://docs.aiohttp.org/en/stable/) (https://docs.aiohttp.org/en/stable/)**.**

Для доступу до БД будемо використовувати ORM підхід на основі [**SQLAlchemy**](https://www.sqlalchemy.org/) (https://www.sqlalchemy.org/)**.**

Створимо віртуальне оточення за допомогою [pipenv](https://pipenv.pypa.io/en/latest/) (https://pipenv.pypa.io/en/latest/). Простіше це зробити через IDE PyCharm.

Встановимо потрібний пакет фреймворку:

pipenv install aiohttp

У директорії проекту створимо файл app.py і помістимо в нього наступний код:

**app.py**

from aiohttp import web

app = web.Application()

web.run\_app(app)

У коді ми імпортуємо клас web з модуля aiohttp. Потім створюємо екземпляр класу Application() і присвоюємо його змінній app. Клас Application - це об'єкт застосунку aiohttp і надає базову структуру для нашого майбутнього вебзастосунку.

Потім ми викликаємо метод run\_app нашого екземпляра Application. Метод запускає наш застосунок, та готовий до обробки вхідних запитів.

Запустимо проект - файл app.py, і відкриємо посилання <http://localhost:8080/> у браузері. Зараз він повинен повертати помилку 404: Not Found. Наш код не визначає обробників маршрутів для вхідних запитів, тому застосунок нічого не робить і просто запускає порожній веб-сервер. Для роботи застосунку необхідно встановити обробники маршрутів, які оброблятимуть вхідні запити та повертатимуть відповіді.

Створімо маршрут і представлення для обробки вхідних запитів.

У корені проекту створіть папку src і помістіть у неї файл представлення src/views.py

**src/views.py**

from aiohttp import web

async def index(request):

return web.Response(text='Hello Aiohttp!')

Ця функція index є найпростішим представлення, можливим в Aiohttp. Ми визначаємо функцію index, яка приймає як аргумент об'єкт request, що представляє вхідний HTTP-запит. Наша функція повертає об'єкт web.Response, який є HTTP-відповіддю. У цьому випадку це HTTP-відповідь із текстом "Hello Aiohttp!". Зверніть увагу, що ключове слово async вказує, що функція index є асинхронною.

Щоб цей обробник маршруту був використаний в застосунку, необхідно додати відповідний маршрут, пов'язаний із цією функцією, в об'єкт застосунку web.Application(). Ми повинні створити маршрут для відображення index і помістити наступний код у файл src/routes.py.

**src/routes.py**

from src.views import index

def setup\_routes(app):

app.router.add\_get('/', index)

За допомогою методу app.router.add\_get ми визначаємо маршрут '/' (кореневий маршрут) і пов'язуємо його з функцією index, яка буде викликатися при отриманні HTTP-запитів на цей маршрут.

INFO

Доброю практикою є поділ представлень, маршрутів, моделей тощо. У такому разі буде більше файлів кожного типу, і їх зручно групувати за різними місцями.

Тепер потрібно додати виклик функції setup\_routes у файлі app.py:

from aiohttp import web

from src.routes import setup\_routes

app = web.Application()

setup\_routes(app)

web.run\_app(app)

Перезапустимо сервер і цього разу, коли ми відкриємо браузер, ми побачимо повідомлення 'Hello Aiohttp!', яке нам повертає функція index для кореневого маршруту.

**Конфігураційні файли**

[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/create#%D0%BA%D0%BE%D0%BD%D1%84%D1%96%D0%B3%D1%83%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D1%96-%D1%84%D0%B0%D0%B9%D0%BB%D0%B8)Фреймворк aiohttp не залежить від конфігурації і не має вбудованої підтримки будь-якої схеми конфігурації.

Ми будемо використовувати yaml файли для зберігання конфігурації застосунку. Конфіги у вигляді файлів json або ini також хороші, але yaml краще.

Тому створимо папку conf у корені проекту і помістимо туди файл config.yaml з наступним вмістом

postgres:

database: postgres

user: postgres

password: 567234

host: localhost

port: 5432

minsize: 1

maxsize: 5

Цей код є конфігураційним файлом для підключення до бази даних PostgreSQL і містить наступні параметри:

* database - ім'я бази даних, до якої ми хочемо підключитися;
* user - ім'я користувача, під яким ми хочемо підключитися до бази даних;
* password - пароль користувача;
* host - ім'я хоста або IP-адреса, на якому запущений сервер бази даних;
* port - порт, на якому запущений сервер бази даних;
* minsize - мінімальна кількість підключень до бази даних, яка повинна бути встановлена в пулі з'єднань;
* maxsize - максимальна кількість підключень до бази даних, яка може бути встановлена в пулі з'єднань.

Щоб читати файли yaml у нашому застосунку, необхідно встановити пакет pyyyaml, :

pipenv install pyyaml

Щоб файл app.py залишався чистим і коротким, підключення конфігураційних змінних помістимо в окремий файл src/conf/settings.py.

**src/settings.py**

import pathlib

import yaml

BASE\_DIR = pathlib.Path(\_\_file\_\_).parent.parent

config\_path = BASE\_DIR / 'config' / 'app\_cfg.yaml'

def get\_config(path):

with open(path) as f:

config = yaml.safe\_load(f)

return config

config = get\_config(config\_path)

Далі завантажуємо конфігурацію у файлі app.py:

**app.py**

from aiohttp import web

from src.routes import setup\_routes

from src.settings import config

app = web.Application()

setup\_routes(app)

app['config'] = config

web.run\_app(app)

Ми додаємо об'єкт config до нашого об'єкта застосунку app, використовуючи ключ 'config', щоб передати конфігураційні налаштування в наш застосунок.

**Підключаємо базу даних до проєкту**

**Створюємо Docker контейнер**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/db#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D1%8E%D1%94%D0%BC%D0%BE-docker-%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%B9%D0%BD%D0%B5%D1%80)Для запуску бази даних [postgres](https://hub.docker.com/_/postgres) (<https://hub.docker.com/_/postgres>) використовуємо Docker. У командному рядку необхідно виконати наступну команду, щоб запустити контейнер із нашою базою даних.

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

**Створюємо моделі**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/db#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D1%8E%D1%94%D0%BC%D0%BE-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%96)Встановимо необхідні пакети для роботи з базою даних - це ORM sqlalchemy і драйвер для роботи з базою даних PostgreSQL psycopg2.

pipenv install sqlalchemy psycopg2

Створимо файл src/models.py з нашими моделями:

**src/models.py**

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

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import relationship

from sqlalchemy.sql.schema import ForeignKey, Table

from sqlalchemy.sql.sqltypes import DateTime

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)

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

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

done = Column(Boolean, default=False)

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

updated\_at = Column(DateTime, default=func.now(), onupdate=func.now())

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)

Щоб представляти зв'язок багато-до-багатьох між нотатками та тегами, ми створили таблицю note\_m2m\_tag за допомогою класу Table. Ми визначаємо стовпці для ідентифікатора запису id, ідентифікатора нотатки note\_id та ідентифікатора тегу tag\_id, а також вказуємо, що стовпці з ідентифікаторами нотаток і тегів є зовнішніми ключами, які посилаються на відповідні таблиці та повинні використовувати операцію CASCADE при видаленні записів.

Далі ми визначили клас Note, який представляє нотатки в нашій базі даних. Тут ми визначаємо відношення tags, яке представляє зв'язок багато-до-багатьох між нотатками та тегами за допомогою аргументу secondary у функції relationship, який вказує на таблицю note\_m2m\_tag. За допомогою аргументу backref створюємо зв'язки для зворотного посилання з тегами.

Нарешті клас Tag представляє теги в нашій базі даних. Тут вказуємо, що ім'я тегу повинно бути унікальним за допомогою аргументу unique=True.

**Підключення до бази даних**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/db#%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 виконання запитів до БД ми створимо асинхронну функцію управління контекстом, яка створює сесію бази даних PostgreSQL і підключає її до вебзастосунку.

**src/db.py**

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

async def pg\_context(app):

conf = app['config']['postgres']

url\_db = f"postgresql://{conf['user']}:{conf['password']}@{conf['host']}/{conf['database']}"

DBSession = sessionmaker(bind=create\_engine(url\_db))

session = DBSession()

app['db\_session'] = session

yield

app['db\_session'].close()

Функція починає роботу з отримання параметрів конфігурації PostgreSQL зі словника app['config'] і створення URL, який далі використовується для створення рушія бази даних.

conf = app['config']['postgres']

url\_db = f"postgresql://{conf['user']}:{conf['password']}@{conf['host']}/{conf['database']}"

Потім створюємо об'єкт sessionmaker, який прив'язується до рушія бази даних.

Цей об'єкт sessionmaker використовується для створення об'єкта session, який потім приєднується до вебзастосунку як app['db\_session'].

Нарешті, функція використовує ключове слово yield для створення корутини, яка може бути використана як асинхронний менеджер контексту. Ця корутина виконується при вході в контекст і створює сесію бази даних. При виході з контексту сесія закривається за допомогою методу close об'єкта session.

Тепер необхідно додати у файл застосунку app.py рядок app.cleanup\_ctx.append(pg\_context).

**app.py**

from aiohttp import web

from src.db import pg\_context

from src.routes import setup\_routes

from src.settings import config

app = web.Application()

setup\_routes(app)

app['config'] = config

app.cleanup\_ctx.append(pg\_context)

web.run\_app(app)

**Виконуємо міграції**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/db#%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)Встановлюємо пакет alembic.

pipenv install alembic

У корені проекту необхідно ініціалізувати оточення alembic за допомогою команди:

alembic init migrations

Цей виклик створить папку з ім'ям migrations всередині нашого проекту, а також файл alembic.ini з налаштуваннями.

Наступним кроком буде зміна файлу env.py. Оскільки ми хочемо використовувати автогенерацію SQL скриптів у міграціях alembic, нам необхідно повідомити про це оточення у файлі env.py, який розташований у папці migrations.

Модифікуємо файл migrations/env.py наступним чином. Спочатку додамо імпорт необхідних модулів

...

from alembic import context

from src.settings import config as our\_config

from src.db import Base

...

Далі знайдемо в коді змінну target\_metadata = None і змінимо її на:

target\_metadata = Base.metadata

user = our\_config['postgres']['user']

password = our\_config['postgres']['password']

host = our\_config['postgres']['host']

database = our\_config['postgres']['database']

config.set\_main\_option('sqlalchemy.url', f'postgresql://{user}:{password}@{host}/{database}')

Нижче додамо код, який витягує значення для user, password, host і database зі словника our\_config['postgres']. Потім метод set\_main\_option об'єкта config використовується для встановлення параметра sqlalchemy.url на сконструйований URL. Змінна URL будується за допомогою f-рядків для інтерполяції значень користувача, пароля, хоста і бази даних у рядок, що має формат postgresql://{user}:{password}@{host}/{database}.

Цей URL потім буде використовувати alembic, щоб під'єднатися до нашої бази даних.

Після цього ми можемо виконати міграцію, виконавши наступну консольну команду в корені проекту.

alembic revision --autogenerate -m 'Init'

У разі успішної операції отримаємо виведення:

INFO [alembic.runtime.migration] Context impl PostgresqlImpl.

INFO [alembic.runtime.migration] Will assume transactional DDL.

INFO [alembic.autogenerate.compare] Detected added table 'notes'

INFO [alembic.autogenerate.compare] Detected added table 'tags'

INFO [alembic.autogenerate.compare] Detected added table 'note\_m2m\_tag'

Generating E:\WebDir\Works\Python\python-help-solution\app\_add\_aoihttp\migrations\versions\afdfdbcecded\_init.py ... done

І файл міграцій у папці migrations/versions наступного змісту:

"""'Init'

Revision ID: afdfdbcecded

Revises:

Create Date: 2023-05-05 17:30:57.827890

"""

from alembic import op

import sqlalchemy as sa

*# revision identifiers, used by Alembic.*

revision = 'afdfdbcecded'

down\_revision = None

branch\_labels = None

depends\_on = None

def upgrade() -> None:

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

op.create\_table('notes',

sa.Column('id', sa.Integer(), nullable=False),

sa.Column('name', sa.String(length=50), nullable=False),

sa.Column('description', sa.String(length=150), nullable=False),

sa.Column('done', sa.Boolean(), nullable=True),

sa.Column('created\_at', sa.DateTime(), nullable=True),

sa.Column('updated\_at', sa.DateTime(), nullable=True),

sa.PrimaryKeyConstraint('id')

)

op.create\_table('tags',

sa.Column('id', sa.Integer(), nullable=False),

sa.Column('name', sa.String(length=25), nullable=False),

sa.PrimaryKeyConstraint('id'),

sa.UniqueConstraint('name')

)

op.create\_table('note\_m2m\_tag',

sa.Column('id', sa.Integer(), nullable=False),

sa.Column('note\_id', sa.Integer(), nullable=True),

sa.Column('tag\_id', sa.Integer(), nullable=True),

sa.ForeignKeyConstraint(['note\_id'], ['notes.id'], ondelete='CASCADE'),

sa.ForeignKeyConstraint(['tag\_id'], ['tags.id'], ondelete='CASCADE'),

sa.PrimaryKeyConstraint('id')

)

# ### end Alembic commands ###

def downgrade() -> None:

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

op.drop\_table('note\_m2m\_tag')

op.drop\_table('tags')

op.drop\_table('notes')

# ### end Alembic commands ###

Для застосування міграції з отриманого файлу і створення таблиць наших моделей у БД виконаємо команду:

alembic upgrade head

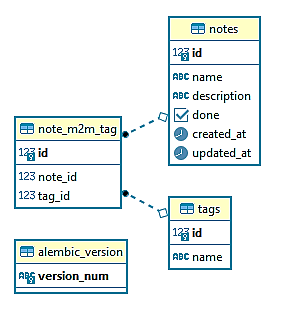
Виведення:

INFO [alembic.runtime.migration] Context impl PostgresqlImpl.

INFO [alembic.runtime.migration] Will assume transactional DDL.

INFO [alembic.runtime.migration] Running upgrade -> afdfdbcecded, 'Init'

Після виконання, міграція створить таблиці в нашій БД. І ми отримаємо наступну ER діаграму.



Ми можемо переходити до написання бізнес-логіки нашого застосунку.

**Пишемо застосунок**

**Підключення шаблонів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/app#%D0%BF%D1%96%D0%B4%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%BD%D1%8F-%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%96%D0%B2)У нашому проекті ми будемо використовувати Jinja2 для рендеру наших шаблонів. Для цього нам необхідно встановити бібліотеку aiohttp\_jinja2:

pipenv install aiohttp\_jinja2

Підключаємо обробку наших майбутніх шаблонів у файл app.py. Самі шаблони ми опишемо трохи згодом:

**app.py**

import asyncio

import sys

from aiohttp import web

import aiohttp\_jinja2

import jinja2

from src.db import pg\_context

from src.routes import setup\_routes

from src.settings import config, BASE\_DIR

app = web.Application()

loader = jinja2.FileSystemLoader(str(BASE\_DIR / 'src' / 'templates'))

aiohttp\_jinja2.setup(app, loader=loader)

setup\_routes(app)

app['config'] = config

app.cleanup\_ctx.append(pg\_context)

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

if sys.platform == 'win32':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

web.run\_app(app)

Ми налаштували шаблонізатор Jinja2, встановивши завантажувач loader для завантаження шаблонів з каталогу src/templates і приєднавши його до застосунку за допомогою методу aiohttp\_jinja2.setup.

loader = jinja2.FileSystemLoader(str(BASE\_DIR / 'src' / 'templates'))

aiohttp\_jinja2.setup(app, loader=loader)

Далі, якщо код виконується як головний модуль:

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

if sys.platform == 'win32':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

web.run\_app(app)

У коді виконується перевірка, що він запущений під операційною системою Windows, і якщо це так, то встановлює політику циклу подій для використання WindowsSelectorEventLoopPolicy. Це необхідно для коректного завершення циклу подій фреймворка aiohttp. Потім застосунок запускається за допомогою методу web.run\_app.

**Відображення головної сторінки**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/app#%D0%B2%D1%96%D0%B4%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B3%D0%BE%D0%BB%D0%BE%D0%B2%D0%BD%D0%BE%D1%97-%D1%81%D1%82%D0%BE%D1%80%D1%96%D0%BD%D0%BA%D0%B8)Щоб підключити відображення головної сторінки, змінимо файл src/views.py:

**src/views.py**

import aiohttp\_jinja2

import aiohttp.web

from src.models import Note, Tag

@aiohttp\_jinja2.template('index.html')

async def index(request):

notes = request.app['db\_session'].query(Note).all()

return {"notes": notes}

Пояснимо доданий код. Декоратор @aiohttp\_jinja2.template використовується для зазначення того, що функція index повинна відображати шаблон index.html.

Використовуючи збережену сесію підключення до бази даних request.app['db\_session'], ми запитуємо модель Note, що зберігається в атрибуті db\_session об'єкта app. Після запиту змінна notes містить список усіх нотаток, що зберігаються в базі даних. Потім функція index повертає словник, який включає список нотаток як значення, пов'язане з ключем notes. Цей словник буде використовуватися шаблонізатором Jinja2 для рендерингу шаблону index.html.

Тепер розберемо код шаблону index.html

**src/templates/index.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8"/>

<title>Organizer</title>

<link rel="stylesheet" href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"/>

</head>

<body>

<main class="container">

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

<ul>

<li><a href="/note/" role="button">Add note</a></li>

<li><a href="/tag/" role="button">Add Tag</a></li>

</ul>

</nav>

{% for note in notes%}

<ul>

<li>

{%if not note.done%}

<a href="/detail/{{note.id}}" role="button" class="outline">{{note.name}}</a>

<a href="/done/{{note.id}}" role="button">Mark Done</a>

<a href="/delete/{{note.id}}" role="button" class="contrast">Delete note</a>

{% else %}

<a href="/detail/{{note.id}}" role="button" class="secondary"><s>{{note.name}}</s></a>

{% endif %}

</li>

</ul>

{% endfor %}

</main>

</body>

</html>

Для базової стилізації тегів шаблону ми використовуємо фреймворк Pico CSS.

<link rel="stylesheet" href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"/>

Для навігації ми використовуємо панель навігації з двома кнопками: "Add note" і "Add Tag". Кнопки - це гіперпосилання, які при натисканні направляють користувача на відповідні сторінки. Атрибут role=" button" використовується для зазначення того, що гіперпосилання повинні поводитися як кнопки, а атрибут style використовується для застосування деяких базових стилів до панелі навігації:

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

<ul>

<li><a href="/note/" role="button">Add note</a></li>

<li><a href="/tag/" role="button">Add Tag</a></li>

</ul>

</nav>

Всередині шаблону ми використовуємо синтаксис Jinja2 для ітерації за списком нотаток notes, переданих із функції відображення index, і відображення подробиць кожної нотатки у вигляді HTML елемента списку — тег li.

{% for note in notes%}

<ul>

<li>

{%if not note.done%}

<a href="/detail/{{note.id}}" role="button" class="outline">{{note.name}}</a>

<a href="/done/{{note.id}}" role="button">Mark Done</a>

<a href="/delete/{{note.id}}" role="button" class="contrast">Delete note</a>

{% else %}

<a href="/detail/{{note.id}}" role="button" class="secondary"><s>{{note.name}}</s></a>

{% endif %}

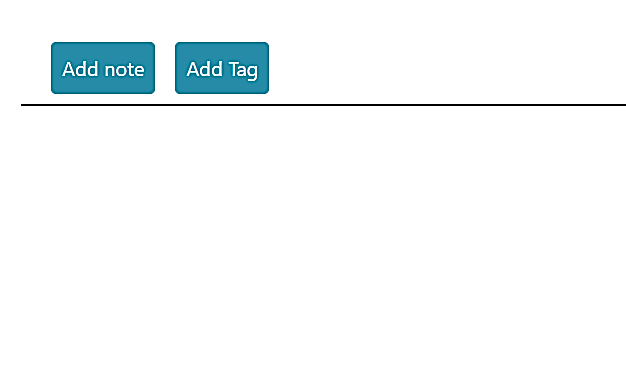
</li>

</ul>

{% endfor %}

Всередині циклу ми включаємо посилання для позначення нотатки як виконаної <a href="/done/{{note.id}}" role="button">Mark Done</a>, видалення нотатки <a href="/delete/{{note.id}}" role="button" class="contrast">Delete note</a> і перегляду подробиць нотатки <a href="/detail/{{note.id}}" role="button" class="secondary"><s>{{note.name}}</s></a>.

Під час запуску сервера ми повинні побачити наступне відображення для головної сторінки нашого застосунку.



**Головна сторінка застосунку**

**Додавання тегу**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/app#%D0%B4%D0%BE%D0%B4%D0%B0%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%82%D0%B5%D0%B3%D1%83)Додамо дві функції у файл views.py. Перша функція tag відповідатиме за відображення шаблону tag.html з формою для введення тегу.

Друга create\_tag призначена для опрацювання POST запиту форми та збереження тегу в базі даних.

**src/views.py**

@aiohttp\_jinja2.template('tag.html')

async def tag(request):

return {}

async def create\_tag(request):

data = await request.post()

name = data["name"]

session = request.app['db\_session']

tag = session.query(Tag).filter(Tag.name == name).first()

if tag is None:

tag = Tag(name=name)

request.app['db\_session'].add(tag)

request.app['db\_session'].commit()

return aiohttp.web.HTTPFound(location=request.app.router['index'].url\_for())

Рядок коду await request.post() зчитує дані форми, відправлені в POST-запиті, і повертає об'єкт MultiDict.

INFO

Об'єкт MultiDict - це словник, який може мати кілька значень для одного і того самого ключа. Це корисно при роботі з полями HTML-форми, які мають однакове ім'я, наприклад, прапорець checkbox.

Ви можете отримати доступ до значень data, використовуючи імена ключів як рядки. У нас є поле введення з ім'ям name, тепер можна отримати доступ до його значення за допомогою data['name'].

Перед створенням нового тегу необхідно перевірити, чи існує тег із заданим ім'ям у базі даних. Один зі способів зробити це - використати метод запиту сесії для пошуку тегу за ім'ям і створювати новий тег, тільки якщо тегу з таким ім'ям не існує.

tag = session.query(Tag).filter(Tag.name == name).first()

if tag is None:

tag = Tag(name=name)

request.app['db\_session'].add(tag)

request.app['db\_session'].commit()

Цей код спочатку шукає тег із заданим ім'ям за допомогою query і filter, і якщо він не знаходить такий тег, то створює новий тег і додає його в сесію за допомогою методу add. Нарешті, він фіксує зміни в базі даних за допомогою commit. Якщо тег із заданим ім'ям уже існує, він пропускає етап створення й одразу повертає відповідь із перенаправленням на індексну сторінку.

Далі для функцій відображення додамо необхідні маршрути в обробнику.

**src/routes.py**

from src.views import index, tag, create\_tag

def setup\_routes(app):

app.router.add\_get('/', index, name='index')

app.router.add\_get('/tag/', tag, name='tag')

app.router.add\_route('POST', '/tag', create\_tag)

За відображення шаблону відповідає рядок коду app.router.add\_get('/tag/', tag, name='tag'), а за обробку POST запиту — app.router.add\_route('POST', '/tag', create\_tag).

Останній штрих - це сам шаблон tag.html для відображення форми введення тегу:

**src/templates/tag.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8"/>

<title>Add new tag</title>

<link rel="stylesheet" href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"/>

</head>

<body>

<main class="container">

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

<ul>

<li><a href="/" role="button"> Go back to the main page </a></li>

</ul>

</nav>

<form method="POST" action="/tag">

<div style="padding: 10px">

<label>

Enter the tag name:

<input type="text" name="name" required/>

</label>

</div>

<div class="grid">

<button type="submit">Submit</button>

<button type="reset" class="secondary">Reset</button>

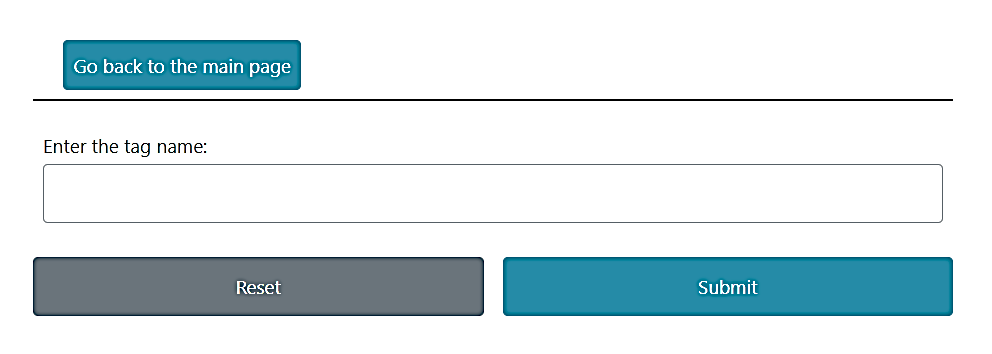
</div>

</form>

</main>

</body>

</html>



**Сторінка введення тега**

Створимо пару тегів перед реалізацією нотаток, наприклад test і python.

**Додавання нотатки**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/app#%D0%B4%D0%BE%D0%B4%D0%B0%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BD%D0%BE%D1%82%D0%B0%D1%82%D0%BA%D0%B8)Після реалізації механізму створення тегів, ми можемо приступити до реалізації створення нотатки.

Першим ми реалізуємо обробник маршруту для відображення сторінки створення нотатки. Всередині файлу src/views.py визначаємо асинхронну функцію з іменем note, яка повертає об'єкт словника, що містить усі теги з бази даних. Це ті теги, які ми додамо до нашої нотатки. Не забуваємо задекорувати функцію @aiohttp\_jinja2.template('note.html'), щоб вказати, що відповідь повинна бути відображена з використанням шаблону Jinja2 з ім'ям note.html.

**src/views.py**

@aiohttp\_jinja2.template('note.html')

async def note(request):

tags = request.app['db\_session'].query(Tag).all()

return {"tags": tags}

Сам шаблон має такий вигляд:

**src/templates/note.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8"/>

<title>Add note</title>

<link rel="stylesheet" href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"/>

</head>

<body>

<main class="container">

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

<ul>

<li><a href="/" role="button"> Go back to the main page </a></li>

</ul>

</nav>

<form method="POST" action="/note">

<div style="padding: 10px">

<label>

Note name:

<input type="text" name="name" required/>

</label>

</div>

<div style="padding: 10px">

<label>

Note description:

<input type="text" name="description" required/>

</label>

</div>

<div style="padding: 10px">

<label>

Select tags:

<select name="tags" multiple="multiple" required>

{% for tag in tags %}

<option value="{{tag.name}}">{{tag.name}}</option>

{% endfor %}

</select>

</label>

</div>

<div class="grid">

<button type="submit">Submit</button>

<button type="reset" class="secondary">Reset</button>

</div>

</form>

</main>

</body>

</html>

Ми додали панель навігації з посиланням на головну сторінку:

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

<ul>

<li><a href="/" role="button"> Go back to the main page </a></li>

</ul>

</nav>

Далі описуємо форму з трьома обов'язковими полями: назва нотатки, опис нотатки та теги. Тут варто виділити особливо поле форми tags - це елемент select, який дає змогу користувачеві вибрати один або кілька тегів для нотатки за допомогою атрибута multiple="multiple". Доступні теги заповнюються з переданого в шаблон словника {"tags": tags} за допомогою циклу for Jinja2.

Для обробки POST-запиту форми необхідно створити обробник створення нотатки:

**src/views.py**

async def create\_note(request):

data = await request.post()

tags = data.getall("tags")

tags\_obj = []

for tag in tags:

tags\_obj.append(request.app['db\_session'].query(Tag).filter(Tag.name == tag).first())

note = Note(name=data["name"], description=data["description"], tags=tags\_obj)

request.app['db\_session'].add(note)

request.app['db\_session'].commit()

return aiohttp.web.HTTPFound(location=request.app.router['index'].url\_for())

Ми обробляємо дані, які були відправлені на сервер, внаслідок надсилання форми. Дані форми отримуємо за допомогою data = await request.post(). Окремо витягуємо теги, які були обрані користувачем, оскільки вони відправлені нам списком tags = data.getall("tags"). Як ви розумієте, в tags зараз зберігаються імена тегів, а нам потрібні об'єкти тегів Tag для створення нотатки. Тому створюємо список об'єктів тегів tags\_obj, використовуючи об'єкти тегів з бази даних, де імена тегів збігаються з іменами обраних тегів filter(Tag.name == tag).

tags\_obj = []

for tag in tags:

tags\_obj.append(request.app['db\_session'].query(Tag).filter(Tag.name == tag).first())

Потім створюємо новий об'єкт нотатки note із зазначеним ім'ям, описом і тегами. Зберігаємо його в базу даних. Якщо операція виконана успішно, функція виконує перенаправлення на головну сторінку.

Залишилося тільки додати відповідні обробники маршрутів у нашу функцію setup\_routes:

**src/views.py**

from src.views import index, tag, create\_tag, note, create\_note

def setup\_routes(app):

app.router.add\_get('/', index, name='index')

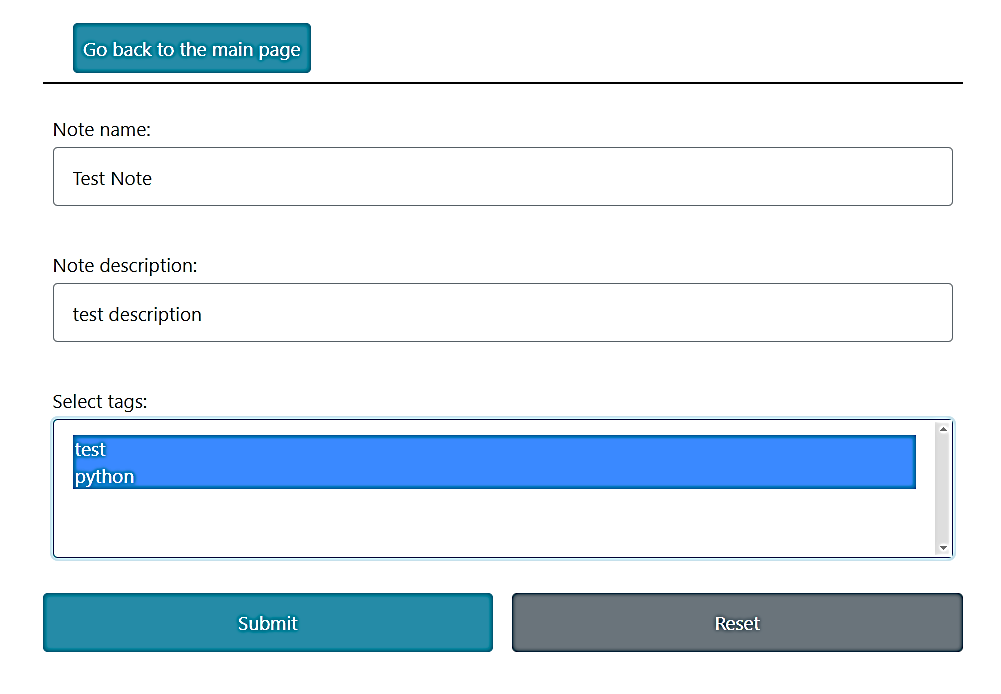
app.router.add\_get('/tag/', tag, name='tag')

app.router.add\_route('POST', '/tag', create\_tag)

app.router.add\_get('/note/', note, name='note')

app.router.add\_route('POST', '/note', create\_note)

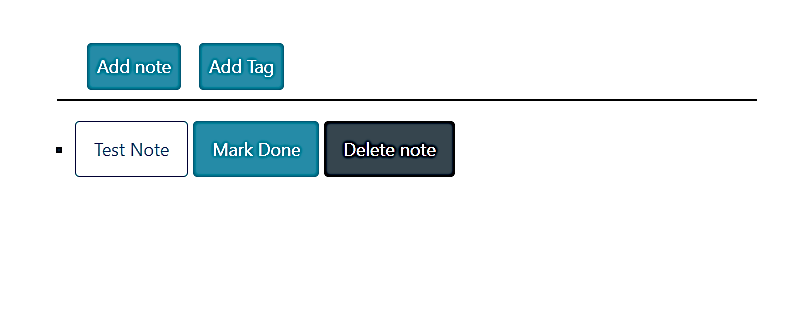
Тепер ми можемо створити нашу першу нотатку



**Сторінка створення нотатки**

Відображення інформації про нотатку[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/app#%D0%B2%D1%96%D0%B4%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D1%96%D1%97-%D0%BF%D1%80%D0%BE-%D0%BD%D0%BE%D1%82%D0%B0%D1%82%D0%BA%D1%83)

Після створення нотатки головна сторінка повинна відображати створену нотатку.



**Головна сторінка застосунку**

Реалізуємо функціонал першої кнопки для показу інформації про нотатку. Створимо функцію-обробник detail для відображення детальної інформації про нотатку.

**src/views.py**

@aiohttp\_jinja2.template('detail.html')

async def detail(request):

note\_id = request.match\_info.get('note\_id')

note = request.app['db\_session'].query(Note).filter(Note.id == note\_id).first()

if not note:

return aiohttp.web.HTTPFound(location=request.app.router['index'].url\_for())

return {"note": note}

Ми отримуємо унікальний ідентифікатор нотатки note\_id зі словника match\_info.

Він містить параметри, які будуть передані в запиті URL. Потім ми шукаємо нотатку із заданим note\_id у базі даних. Якщо нотатку не знайдено, ми перенаправляємо користувача на головну сторінку. Якщо нотатку знайдено, ми повертаємо словник з об'єктом нотатки, який буде передано в шаблон для відображення детальної інформації.

Додаємо обробник маршруту, не забувши вказати в маршруті параметр запиту note\_id:

**src/routes.py**

from src.views import index, tag, create\_tag, note, create\_note, detail

def setup\_routes(app):

app.router.add\_get('/', index, name='index')

app.router.add\_get('/tag/', tag, name='tag')

app.router.add\_route('POST', '/tag', create\_tag)

app.router.add\_get('/note/', note, name='note')

app.router.add\_route('POST', '/note', create\_note)

app.router.add\_get('/detail/{note\_id}', detail, name='detail')

І реалізуємо сам шаблон для відображення сторінки детальної інформації про нотатку:

**src/templates/detail.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8"/>

<title>Note information</title>

<link rel="stylesheet" href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"/>

</head>

<body>

<main class="container">

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

<ul>

<li><a href="/" role="button"> Go back to the main page </a></li>

</ul>

</nav>

<h1>{{note.name}}</h1>

<p>Description: <b>{{note.description}}</b></p>

<p>Time of creation: <kbd>{{note.created\_at.date()}}</kbd></p>

{% if note.done %}

<p>Status:

<ins>Completed</ins>

</p>

{% else %}

<p>Status: <i>Not implemented</i></p>

{% endif %}

<p>Tags:

<small>

{{ note.tags|join(', ', attribute='name') }}

</small>

</p>

</main>

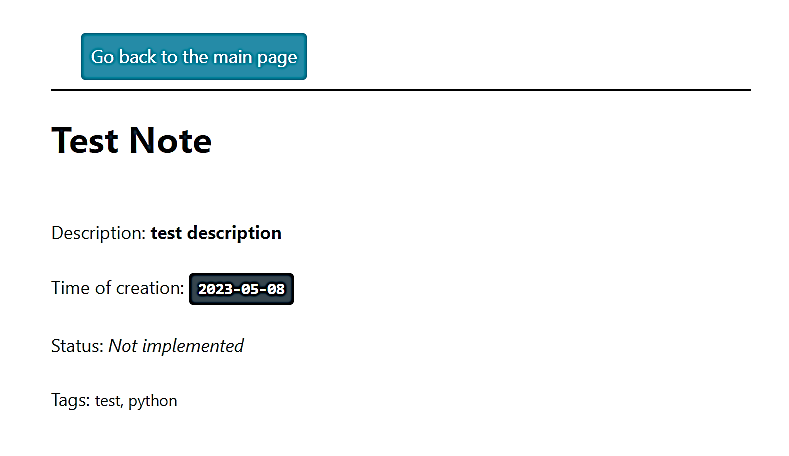
</body>

</html>

Як і скрізь використовуємо бібліотеку [Picocss](https://picocss.com/) (https://picocss.com/) для оформлення сторінки. На сторінці виводиться назва нотатки, опис, дата створення, статус виконання і пов'язані теги.

При виведенні тегів ми використовуємо фільтр join, який з'єднує значення списку note.tags через кому і додає пробіл після кожної коми для кращої читабельності. Аргумент attribute задає ім'я атрибута моделі Tag, значення якого буде використовуватися для кожного елемента списку при об'єднанні. У цьому випадку ми використовуємо значення атрибута name моделі для відображення назви тегу.

Тепер під час кліку на першу кнопку з назвою нотатки, ми потраплятимемо на сторінку з детальною інформацією про нотатку:



**Сторінка відображення нотатки**

**Видалення нотатки**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/app#%D0%B2%D0%B8%D0%B4%D0%B0%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BD%D0%BE%D1%82%D0%B0%D1%82%D0%BA%D0%B8)Додамо у функціональність нашого застосунку можливість видалення ще не виконаної нотатки. Функція delete\_note видаляє нотатку із зазначеним ідентифікатором note\_id з бази даних.

**src/views.py**

async def delete\_note(request):

note\_id = request.match\_info.get('note\_id')

session = request.app['db\_session']

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

if note:

session.query(Note).filter(Note.id == note\_id).delete()

session.commit()

return aiohttp.web.HTTPFound(location=request.app.router['index'].url\_for())

Для цього ми отримуємо об'єкт db\_session із застосунку та перевіряємо існування нотатки з ідентифікатором note\_id. Якщо нотатка існує, то виконуємо видалення нотатки із зазначеним note\_id із бази даних, використовуючи метод delete об'єкта запитуsession.query(Note). Зберігаємо зміни в базі даних і перенаправляємо користувача на головну сторінку застосунку.

У функцію setup\_routes файлу src/routes.py необхідно додати обробку маршруту app.router.add\_get('/delete/{note\_id}', delete\_note, name='delete').

Так просто ми додали механізм видалення непотрібних нотаток.

**Змінюємо статус нотатки**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/additional/aiohttp/app#%D0%B7%D0%BC%D1%96%D0%BD%D1%8E%D1%94%D0%BC%D0%BE-%D1%81%D1%82%D0%B0%D1%82%D1%83%D1%81-%D0%BD%D0%BE%D1%82%D0%B0%D1%82%D0%BA%D0%B8)Останнім ми реалізуємо зміну статусу нотатки на виконану. Функція відображення done\_note оновлює атрибут done нотатки на True, вказуючи, що нотатка була завершена.

**src/views.py**

async def done\_note(request):

note\_id = request.match\_info.get('note\_id')

session = request.app['db\_session']

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

if note:

note.done = True

session.commit()

return aiohttp.web.HTTPFound(location=request.app.router['index'].url\_for())

Додаємо обробник маршруту:

**src/routes.py**

from src.views import index, tag, create\_tag, note, create\_note, detail, delete\_note, done\_note

def setup\_routes(app):

app.router.add\_get('/', index, name='index')

app.router.add\_get('/tag/', tag, name='tag')

app.router.add\_route('POST', '/tag', create\_tag)

app.router.add\_get('/note/', note, name='note')

app.router.add\_route('POST', '/note', create\_note)

app.router.add\_get('/detail/{note\_id}', detail, name='detail')

app.router.add\_get('/delete/{note\_id}', delete\_note, name='delete')

app.router.add\_get('/done/{note\_id}', done\_note, name='done')

Тепер під час кліку на кнопці "Mark Done" ми вважатимемо нотатку виконаною і візуально це матиме наступний вигляд:



**Виконана нотатка**

Ми отримали повноцінний вебзастосунок і, якщо запустити скрипт python app.py, то доступ до проекту ми отримаємо за адресою <http://127.0.0.1:8080/>

**Висновок**

Наш застосунок Organizer дає змогу користувачеві створювати, редагувати та видаляти нотатки. Кожна нотатка містить назву, опис і теги, які допомагають легко знаходити потрібну інформацію. Користувач може позначити нотатку як виконану.

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

Щоб побачити роботу нашого застосунку, після запуску прикладу, натисніть [посилання](https://aiohttp-server.krabaton.repl.co/) (https://aiohttp-server.krabaton.repl.co/).

При створенні застосунку ми розглянули базовий функціонал фреймворку aiohttp, що забезпечує швидку та ефективну взаємодію між сервером і клієнтом. Для роботи з базою даних у застосунку використовується SQLAlchemy, що дає змогу працювати з різними СУБД.

Застосунок загалом працює коректно, але для поліпшення його функціональності пропонуємо самостійно додати можливість редагування нотаток і пошук нотаток за тегами.