**Модуль 14: Тестування та розгортання веб-додатків**

**Заняття 1: Тестування веб-додатків**

**Вступ**

Ключові вимоги до роботи будь-якого застосунку:

* надійність;
* відповідність специфікації;
* простота зміни застосунку відповідно до вимог, що змінюються (легкість підтримки);

Дотриматися цих вимог, коли застосунок розробляє одна людина, завдання непросте, але цілком посильне. Коли ж над "ПЗ" працює команда з кількох розробників, дотриматися всіх вимог стає дедалі складніше.

Зі зростанням команди і розміру кодової бази роль тестів у застосунку постійно зростає. Щоб застосунок працював саме так, як потрібно замовникам, потрібні приймальні тести. Щоб розробник міг вносити зміни в наявний код і бути впевненим, що його зміни не "ламають" застосунок, потрібні модульні тести (unit tests). Щоб бути впевненим, що застосунок коректно взаємодіє із зовнішніми системами (бази даних, зовнішні API, сервіси), потрібні інтеграційні тести.

**Види тестування**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/intro#%D0%B2%D0%B8%D0%B4%D0%B8-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)Існує безліч видів тестування. Вони бувають ручними, тобто виконуються вручну, або запускається деякий код для тестування — автоматизованими.

Перелічимо деякі типи тестування:

* Юніт-тестування (модульне) - перевіряє індивідуальні компоненти застосунку, такі як функції та методи;
* Інтеграційне тестування - перевіряє, як окремі компоненти працюють разом;
* Функціональне тестування - перевіряє, що застосунок працює відповідно до вимог та документації;
* Тестування на продуктивність - оцінює, як швидко та ефективно працює застосунок за певного навантаження;
* Тестування безпеки - перевіряє, наскільки застосунок захищений від різноманітних вразливостей та атак;
* Тестування на сумісність - перевіряє, як застосунок працює у різних браузерах, операційних системах і пристроях;
* Регресивне тестування - перевіряє, що зміни в коді не впливають на працездатність уже наявних функцій і компонентів застосунку. Фактично, це повторне тестування — перевірити, чи не зламали ми щось після додавання нового функціоналу;
* Наскрізне тестування (E2E) - тестує застосунок як ціле, перевіряючи функціональність і працездатність на всіх рівнях і в усіх сценаріях використання.

Мінімально необхідними є модульні-тести (Юніт-тести) і функціональне тестування, це має бути основою для тестування будь-якого вебзастосунку.

Юніт-тести допоможуть гарантувати, що окремі компоненти застосунку працюють коректно, а функціональне тестування допоможе переконатися, що застосунок працює відповідно до вимог та документації. Інші види тестування також є важливими, але їхня необхідність залежить від конкретного застосунку та його вимог.

Інтеграційне та функціональне тестування дають повне уявлення про те, як застосунок працює і як він виконує свою функцію. Іноді їх називають просто інтеграційні тести застосунку. Однак інтеграційне та функціональне тестування мають відмінності з точки зору, що ми тестуємо.

ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ

Перевіряє як окремі компоненти застосунку працюють разом. Це може включати перевірку інтеграції між базою даних, сервером і клієнтським інтерфейсом. Мета інтеграційного тестування полягає в тому, щоб переконатися, що всі компоненти працюють разом коректно.

ФУНКЦІОНАЛЬНЕ ТЕСТУВАННЯ

Перевіряє, що застосунок працює відповідно до вимог та документації. Це може включати перевірку різних функцій і можливостей застосунку, таких як вхід у систему, додавання і редагування даних тощо. Мета функціонального тестування полягає в тому, щоб переконатися, що застосунок виконує свою функцію коректно і відповідає вимогам кінцевих користувачів.

Головна мета тестів — впевненість. Тестування вебзастосунків необхідне, щоб переконатися, що вони працюють коректно і безпомилково. Це дає змогу виявити й виправити помилки та проблеми перед випуском застосунку на ринок. Також тестування допомагає збільшити надійність і якість застосунку, що зі свого боку привертає та задовольняє користувачів.

**Розробка через тестування**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/intro#%D1%80%D0%BE%D0%B7%D1%80%D0%BE%D0%B1%D0%BA%D0%B0-%D1%87%D0%B5%D1%80%D0%B5%D0%B7-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)Розробка через тестування — це наріжний камінь надійних, розширюваних і гнучких застосунків. Настільки важлива сфера розробки будь-якого ПЗ звісно автоматизується та існує безліч інструментів, що полегшують тестування коду на будь-яких мовах і рівнях.

Розробка через тестування (Test-Driven Development, **TDD**) — це методологія розробки ПЗ, у якій розробники виконують написання тестів для коду до написання самого коду. Це дає змогу розробникам визначати точно, як повинен працювати код, до того як його буде написано.

Процес розробки через тестування складається з трьох кроків:

1. Написання тесту для нереалізованої функціональності;
2. Написання коду, який проходить тест;
3. Рефакторинг коду для поліпшення його читабельності та підтримки;

Розробка через тестування допомагає розробникам створювати якісний код, який відповідає вимогам і проходить усі тести. Це також допомагає уникнути помилок і покращує подальшу підтримку коду. Розробка через тестування є популярною методологією в Agile розробці, яка допомагає командам створювати якісне ПЗ зі швидшим і гнучкішим процесом розробки.

**Пакети для тестування**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/intro#%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D0%B8-%D0%B4%D0%BB%D1%8F-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)

У Python екосистемі існує безліч фреймворків та інструментів для тестування. Перелічимо найпопулярніші серед них:

* [**unittest**](https://docs.python.org/3/library/unittest.html) (<https://docs.python.org/3/library/unittest.html>) — вбудований у Python фреймворк для тестування, який надає безліч функцій для створення та запуску юніт-тестів;
* [**pytest**](https://docs.pytest.org/en/7.2.x/) (https://docs.pytest.org/en/7.2.x/)— популярний фреймворк для тестування, який надає ширший функціонал та інструментарій для тестування, ніж unittest;
* [**nose**](https://nose.readthedocs.io/en/latest/) (https://nose.readthedocs.io/en/latest/) — популярний фреймворк для тестування, який розширює функціонал unittest;
* [**doctest**](https://docs.python.org/3/library/doctest.html) (https://docs.python.org/3/library/doctest.html) — модуль, що дозволяє включати тести в документацію коду;
* [**Selenium**](https://www.selenium.dev/) (https://www.selenium.dev/) — інструмент, що дає змогу автоматизувати тестування вебзастосунків, використовуючи браузери. Він може використовуватися практично з кожною сучасною мовою програмування;

Залежно від конкретної ситуації та вимог до проекту, розробники можуть використовувати один або кілька цих інструментів для тестування свого коду. Наприклад, можна використовувати unittest або pytest для юніт-тестування, а Selenium для автоматизації тестування веб-інтерфейсу. Важливо пам'ятати, що немає одного ідеального інструменту для всіх ситуацій, і вибір залежить від конкретних вимог і потреб до проекту.

**Документування коду**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/intro#%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BA%D0%BE%D0%B4%D1%83)Також важливою частиною якості коду є його документування. Документування коду в Python є важливою частиною розробки ПЗ, яка допомагає розробникам та іншим членам команди розуміти, як використовувати код і як він працює.

У Python існує кілька способів документування коду:

* Коментарі — текст, який знаходиться поза кодом, але використовується для пояснення того, що робить код. Коментарі можуть бути вставлені в будь-якому місці коду, і вони ігноруються інтерпретатором Python.
* Docstrings — спеціальний тип коментаря, який використовується для документування модулів, класів, функцій і методів. Docstrings знаходяться в потрійних лапках і можуть бути доступні через атрибут \_\_doc\_\_.
* [**Sphinx**](https://www.sphinx-doc.org/en/master/) (https://www.sphinx-doc.org/en/master/)— інструмент для генерації документації, який використовується для створення документації, що має професійний вигляд, на основі коду та docstrings.

Документування коду важливе для підтримки та розуміння коду як складової частини проекту, а також для полегшення розуміння коду з боку інших розробників та інших зацікавлених осіб.

**Документування коду**

Розробка хороших інструментів та утиліт може бути дуже непростим завданням. Потрібно добре розбиратися в потребах ваших користувачів і в тому, як саме користувачеві захочеться застосувати вашу утиліту.

Але головна складність у розробці допоміжних інструментів полягає в тому, що популярність і зручність використання інструменту тільки на половину визначає API (набір функцій, методів, класів). Дуже істотний внесок у зручність використання вносить наявність документації. Читати чужий код — це не проста навичка, читати "поганий" код може бути дуже виснажливо. Якщо ви змушуєте користувача читати ваш код, щоб зрозуміти, як саме ним користуватися, то ви робите помилку. Таким інструментом користуватися буде дуже незручно.

Є кілька інструментів для генерації документації з коду та коментарів у коді. Усі ці інструменти покликані дати можливість користувачеві розібратися в можливостях вашої утиліти без необхідності читати ваш код. У більшості випадків простого опису наявних функцій/методів та їхніх сигнатур достатньо для 95% користувачів.

**Робота PyDoc**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/doc#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-pydoc)[PyDoc](https://docs.python.org/3.7/library/pydoc.html) (https://docs.python.org/3.7/library/pydoc.html) — модуль для генерації та читання документації модулів і пакетів Python.

Приклад читання документації:

python -m pydoc [package name]

PyDoc бере документацію з:

* Атрибуту \_\_doc\_\_ об'єктів Python.
* Рядка коментарів відразу після оголошення методу/класу/функції.
* Рядка коментарів на самому початку модуля.

Але спочатку розберемося, як правильно писати коментарі до документації.

Нехай у нас є файл main.py з таким вмістом:

"""

Module of arithmetic operations

"""

def add(a: int, b: int) -> int:

"""

Returns the sum of two integers.

:param a: The first integer.

:type a: int

:param b: The second integer.

:type b: int

:return: The sum of a and b.

:rtype: int

"""

return a + b

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

print(add(2, 2))

Як бачимо, функція add містить docstring, яка надає інформацію про те, що робить функція, які параметри приймає, які значення повертає, і які можливі помилки можуть виникнути. Загальноприйнятий формат docstring, що називається [**docstring у стилі Google**](https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html) (https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example\_google.html), включає опис функції, параметри функції (їхню назву та тип) і значення, що повертається.

У прикладі docstring починається з потрійних лапок і закінчується потрійними лапками. Перший рядок зазвичай є коротким описом функції. Після цього слідує детальніший опис функції, включно з параметрами, значенням, що повертається, і можливими винятками. Наприкінці документації можна за бажанням навести приклад використання функції.

Ми використовуємо стиль reStructuredText, де кожен параметр, значення, що повертається, і сама функція описується в окремому рядку, що починається з двокрапки та ключового слова, наприклад, :param, :return, :type.

Наприклад, :param a: The first integer. описує параметр a як цілочисельне значення, :type a: int вказує тип даних параметра. Документація закінчується рядком :rtype: int, який описує тип значення, що повертається.

INFO

Полегшити написання рядків документації може допомогти плагін для PyCharm [Trelent - AI Docstrings on Demand](https://plugins.jetbrains.com/plugin/18716-trelent--ai-docstrings-on-demand) (<https://plugins.jetbrains.com/plugin/> 18716-trelent--ai-docstrings-on-demand). Він за допомогою AI дозволяє створювати досить хороші рядки документації для популярних мов програмування.

Для нашої функції add ми отримаємо наступний рядок документації (комбінація клавіш Alt-D):

"""

The add function returns the sum of two integers.

:param a:int: Specify that the function expects an integer as input

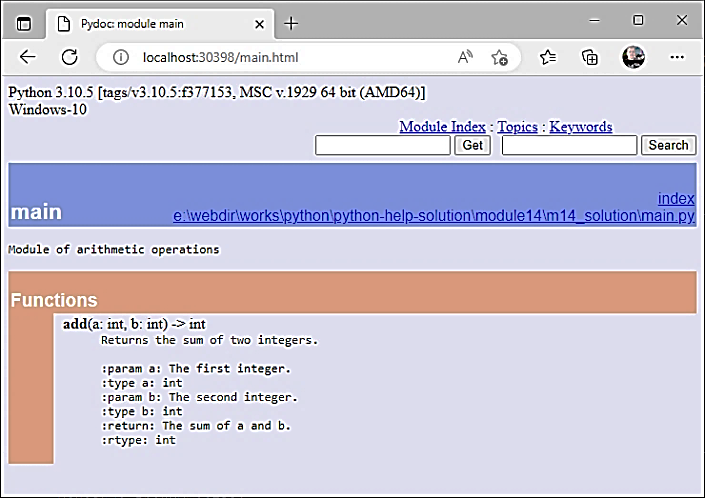
:param b:int: Specify that the type of b is an integer

:return: The sum of two integers

:doc-author: Trelent

"""

Щоб побачити документацію до модуля main.py, виконайте команду в проекті pydoc -b. Команда запустить сервер і додатково відкриє веб-браузер для сторінки індексу модуля. Виберіть посилання з модулем main і побачите нашу документацію.



**Документація модуля main.py**

**Sphinx**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/doc#sphinx)Але ширший функціонал і можливості для створення документації надає утиліта [**Sphinx**](https://www.sphinx-doc.org/en/master/) (https://www.sphinx-doc.org/en/master/)**.**

**Sphinx**- це інструмент для генерації документації на Python. Він використовує геStrukturdText як мову розмітки, яка надає багатий набір інструментів для написання документації, включно з форматуванням тексту, розбиттям на розділи, списками, таблицями, посиланнями, індексами та багато іншого. Дозволяє автоматично генерувати документацію на основі docstrings у форматах HTML, PDF, EPUB і, навіть LaTeX.

Основні переваги:

1. Спрощення процесу створення документації. Sphinx дозволяє автоматично згенерувати документацію для проекту на Python на основі docstrings, використовуючи різні форми форматування та розмітки.
2. Підтримка різних форматів. Sphinx підтримує безліч форматів документації, що дає змогу легко створювати документацію в необхідному форматі зі зрозумілим і привабливим зовнішнім виглядом.
3. Легкість у використанні. Синтаксис і форматування документації, що підтримуються Sphinx, засновані на стандартах, зрозумілих для розробників, і не вимагають спеціальних знань.

Встановлення:

poetry add sphinx

Основні команди Sphinx:

* sphinx-quickstart - це команда для створення нового проекту документації. Вона ставить низку запитань користувачеві, таких як назва проєкту, автор, мова тощо, а потім генерує структуру каталогів і файлів, які можуть бути використані для початку роботи над документацією.
* sphinx-build - це команда для складання документації. Вона приймає вхідний файл або директорію з вихідними файлами документації та генерує HTML, PDF, LaTeX або інший формат документації.
* sphinx-apidoc - це команда для автоматичної генерації документації з модулів Python. Вона приймає ім'я модуля Python і генерує документацію для всіх його функцій, класів і методів.
* sphinx-autogen - це команда, яку варто використовувати для генерації документації на основі шаблонів. Вона дозволяє визначити структуру документації та заповнити її автоматично на основі шаблонів.

Згенерована за допомогою Sphinx або PyDoc документація зручна тим, що вона дуже "близька" до коду. Типова проблема підтримки документації в актуальному стані — це не виправити опис класу/методу в документації, коли він змінився в коді. Дуже швидко документація приходить у непридатність і нею стає неможливо користуватися. Тому підтримка документації в належному стані це важливо.

Sphinx - це потужний інструмент для створення документації на Python, який полегшує процес створення та підтримки документації, а також дає змогу генерувати документацію в різних форматах. З використанням Sphinx ви можете створювати документацію для інших розробників, які використовують ваш проект.

Використаємо Sphinx для генерації документації до нашого проекту REST API.

**Документування коду проекту**

**Підготовка**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/sphinx#%D0%BF%D1%96%D0%B4%D0%B3%D0%BE%D1%82%D0%BE%D0%B2%D0%BA%D0%B0)Спочатку встановимо пакет Sphinx у наш проект у вигляді [dev залежності](https://python-poetry.org/docs/managing-dependencies/#dependency-groups) (https://python-poetry.org/docs/managing-dependencies/#dependency-groups)

poetry add sphinx -G dev

У файлі pyproject.toml повинна з'явитися окрема група залежностей зі встановленим пакетом Sphinx

[ tool.poetry.group.dev.dependencies ]

sphinx = "^6.1.3"

Потім в корені проекту виконаємо команду, яка після невеликого опитування створить нам директорію docs, в якій буде знаходитися наша документація.

sphinx-quickstart docs

Погоджуємося із запропонованими варіантами за замовчуванням і введемо, що назва проекту - Rest API, а автор - Krabat:

You have two options for placing the build directory for Sphinx output.

Either, you use a directory "\_build" within the root path, or you separate

"source" and "build" directories within the root path.

> Separate source and build directories (y/n) [n]:

The project name will occur in several places in the built documentation.

> Project name: Rest API

> Author name(s): Krabat

> Project release []:

If the documents are to be written in a language other than English,

you can select a language here by its language code. Sphinx will then

translate text that it generates into that language.

For a list of supported codes, see

https://www.sphinx-doc.org/en/master/usage/configuration.html#confval-language.

> Project language [en]:

Creating file E:\WebDir\Works\Python\python-help-solution\solution\_rest\_api\_tests\docs\conf.py.

Creating file E:\WebDir\Works\Python\python-help-solution\solution\_rest\_api\_tests\docs\index.rst.

Creating file E:\WebDir\Works\Python\python-help-solution\solution\_rest\_api\_tests\docs\Makefile.

Creating file E:\WebDir\Works\Python\python-help-solution\solution\_rest\_api\_tests\docs\make.bat.

Finished: An initial directory structure has been created.

**Налаштування конфігурації**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/sphinx#%D0%BD%D0%B0%D0%BB%D0%B0%D1%88%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BA%D0%BE%D0%BD%D1%84%D1%96%D0%B3%D1%83%D1%80%D0%B0%D1%86%D1%96%D1%97)Перейдемо в терміналі в створену папку docs і зробимо її поточною директорією

cd docs

У редакторі відкриємо файл docs/conf.py і помістимо в нього такий вміст.

**docs/conf.py**

import sys

import os

sys.path.append(os.path.abspath('..'))

project = 'Rest API'

copyright = '2023, Krabat'

author = 'Krabat'

extensions = ['sphinx.ext.autodoc']

templates\_path = ['\_templates']

exclude\_patterns = ['\_build', 'Thumbs.db', '.DS\_Store']

html\_theme = 'nature'

html\_static\_path = ['\_static']

Давайте розберемо детальніше вміст цього файлу. У список sys.path додається абсолютний шлях до батьківської директорії з поточної директорії. Це необхідно для того, щоб Sphinx міг знаходити модулі Python, що перебувають поза директорією проекту документації. Задаються наступні налаштування проекту:

* project - назва проекту. Ми ввели її раніше project = 'Rest API'
* copyright - авторські права copyright = '2023, Krabat'
* author - автори проекту. У нас тут псевдонім творця author = 'Krabat'.

Далі вказуються розширення, які будуть використовуватися при генерації документації. У цьому випадку це `sphinx.ext.autodoc - це розширення автоматично генерує документацію на основі docstrings у нашому коді. Вказується шлях до шаблонів документації templates\_path = ['\_templates'].

В exclude\_patterns вказуються директорії або файли, які не включатимуться до документації, що генерується. Задається тема оформлення документації html\_theme, в цьому випадку ми змінили її на привабливішу nature. Вказується шлях до директорії зі статичними файлами html\_static\_path = ['\_static'] (наприклад, зображеннями), які будуть використовуватися в документації.

**Описуємо структуру документації**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/sphinx#%D0%BE%D0%BF%D0%B8%D1%81%D1%83%D1%94%D0%BC%D0%BE-%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D1%83-%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D1%96%D1%97)

Всередині файлу docs/index.rst нам потрібно описати структуру документації нашого проекту. Додамо туди наступний вміст:

**docs/index.rst**

.. Rest API documentation master file, created by

sphinx-quickstart on Thu Mar 2 15:09:18 2023.

You can adapt this file completely to your liking, but it should at least

contain the root `toctree` directive.

Welcome to Rest API's documentation!

====================================

.. toctree::

:maxdepth: 2

:caption: Contents:

REST API main

===================

.. automodule:: main

:members:

:undoc-members:

:show-inheritance:

REST API repository Notes

=========================

.. automodule:: src.repository.notes

:members:

:undoc-members:

:show-inheritance:

REST API repository Tags

=========================

.. automodule:: src.repository.tags

:members:

:undoc-members:

:show-inheritance:

REST API repository Users

=========================

.. automodule:: src.repository.users

:members:

:undoc-members:

:show-inheritance:

REST API routes Notes

=========================

.. automodule:: src.routes.notes

:members:

:undoc-members:

:show-inheritance:

REST API routes Tags

=========================

.. automodule:: src.routes.tags

:members:

:undoc-members:

:show-inheritance:

REST API routes Users

=========================

.. automodule:: src.routes.users

:members:

:undoc-members:

:show-inheritance:

REST API routes Auth

=========================

.. automodule:: src.routes.auth

:members:

:undoc-members:

:show-inheritance:

REST API service Auth

=========================

.. automodule:: src.services.auth

:members:

:undoc-members:

:show-inheritance:

REST API service Email

=========================

.. automodule:: src.services.email

:members:

:undoc-members:

:show-inheritance:

Indices and tables

==================

\* :ref:`genindex`

\* :ref:`modindex`

\* :ref:`search`

Фактично ми прописали, до яких файлів у нашому проекті потрібно створити документацію. Розберемо на прикладі додавання документації до файлу src/repository/notes.py нашого проекту. За це відповідає наступний блок файлу docs/index.rst:

REST API repository Notes

=========================

.. automodule:: src.repository.notes

:members:

:undoc-members:

:show-inheritance:

Розберемо конструкції цих записів детальніше:

REST API repository Notes

=========================

Це заголовок, написаний у форматі ReStructuredText. Символи = під заголовком використовуються для створення розділової лінії. Вона не може бути меншою за ширину самого заголовка, інакше під час генерації документації ви отримаєте повідомлення про помилку.

Далі йде блок, що починається з рядка:

.. automodule:: src.repository.notes

Ця директива Sphinx automodule автоматично створює документацію для модуля Python notes.py, розташованого за шляхом src/repository/notes.py. Ви повинні тут вказати шлях до файлу, як ми вказуємо для директиви import.

За automodule йдуть додаткові директиви:

* :members: вказує, що всі функції і класи в модулі повинні бути включені в документацію;
* :undoc-members: вказує, що документація для функцій і методів, які не мають docstrings, також повинна бути включена.
* :show-inheritance: вказує, що наслідування класів повинно бути відображено в документації.

Сукупність цих директив у блоці автоматично створює документацію для модуля notes.py з детальним описом усіх його функцій і класів, включаючи наслідування.

Таким чином у файлі index.rst ми описуємо увесь код, який хочемо додати в документацію.

Внизу файлу вже будуть інструкції, які стосуються автоматично згенерованих індексів і таблиць документації Sphinx.

* :ref:genindex генерує індекс, що містить список усіх термінів, згадуваних у документації.
* :ref:modindex генерує індекс, що містить список усіх модулів, документованих у документації.
* :ref:search створює форму пошуку, яку користувач може використовувати для пошуку за ключовими словами в документації. Так у нас буде пошук у документації.

Ці інструкції зазвичай розміщуються наприкінці документації у файлі index.rst, як зручний спосіб для користувачів швидко знайти цікаву для них інформацію в документації.

**Додаємо docstrings**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/sphinx#%D0%B4%D0%BE%D0%B4%D0%B0%D1%94%D0%BC%D0%BE-docstrings)

Перед генерацією документації нам необхідно додати до необхідних функцій і методів класів рядки docstrings. Покажемо на прикладі модуля src/repository/notes.py як це повинно виглядати:

**src/repository/notes.py**

from typing import List

from sqlalchemy import and\_

from sqlalchemy.orm import Session

from src.database.models import Note, Tag, User

from src.schemas import NoteModel, NoteUpdate, NoteStatusUpdate

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

"""

Retrieves a list of notes for a specific user with specified pagination parameters.

:param skip: The number of notes to skip.

:type skip: int

:param limit: The maximum number of notes to return.

:type limit: int

:param user: The user to retrieve notes for.

:type user: User

:param db: The database session.

:type db: Session

:return: A list of notes.

:rtype: List[Note]

"""

return db.query(Note).filter(Note.user\_id == user.id).offset(skip).limit(limit).all()

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

"""

Retrieves a single note with the specified ID for a specific user.

:param note\_id: The ID of the note to retrieve.

:type note\_id: int

:param user: The user to retrieve the note for.

:type user: User

:param db: The database session.

:type db: Session

:return: The note with the specified ID, or None if it does not exist.

:rtype: Note | None

"""

return db.query(Note).filter(and\_(Note.id == note\_id, Note.user\_id == user.id)).first()

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

"""

Creates a new note for a specific user.

:param body: The data for the note to create.

:type body: NoteModel

:param user: The user to create the note for.

:type user: User

:param db: The database session.

:type db: Session

:return: The newly created note.

:rtype: Note

"""

tags = db.query(Tag).filter(and\_(Tag.id.in\_(body.tags), Tag.user\_id == user.id)).all()

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

db.add(note)

db.commit()

db.refresh(note)

return note

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

"""

Removes a single note with the specified ID for a specific user.

:param note\_id: The ID of the note to remove.

:type note\_id: int

:param user: The user to remove the note for.

:type user: User

:param db: The database session.

:type db: Session

:return: The removed note, or None if it does not exist.

:rtype: Note | None

"""

note = db.query(Note).filter(and\_(Note.id == note\_id, Note.user\_id == user.id)).first()

if note:

db.delete(note)

db.commit()

return note

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

"""

Updates a single note with the specified ID for a specific user.

:param note\_id: The ID of the note to update.

:type note\_id: int

:param body: The updated data for the note.

:type body: NoteUpdate

:param user: The user to update the note for.

:type user: User

:param db: The database session.

:type db: Session

:return: The updated note, or None if it does not exist.

:rtype: Note | None

"""

note = db.query(Note).filter(and\_(Note.id == note\_id, Note.user\_id == user.id)).first()

if note:

tags = db.query(Tag).filter(and\_(Tag.id.in\_(body.tags), Note.user\_id == user.id)).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, user: User, db: Session) -> Note | None:

"""

Updates the status (i.e. "done" or "not done") of a single note with the specified ID for a specific user.

:param note\_id: The ID of the note to update.

:type note\_id: int

:param body: The updated status for the note.

:type body: NoteStatusUpdate

:param user: The user to update the note for.

:type user: User

:param db: The database session.

:type db: Session

:return: The updated note, or None if it does not exist.

:rtype: Note | None

"""

note = db.query(Note).filter(and\_(Note.id == note\_id, Note.user\_id == user.id)).first()

if note:

note.done = body.done

db.commit()

return note

Для інших файлів нашого проекту процес додавання буде аналогічним.

**Створюємо документацію**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/sphinx#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D1%8E%D1%94%D0%BC%D0%BE-%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D1%96%D1%8E)Всередині директорії docs Sphinx створив нам файл make.bat для генерації нашої документації. Після всіх необхідних приготувань виконаємо команду

.\make.bat html

Якщо все зробили правильно і не припустились помилок, то у вас повинно бути виведення, схоже на це:

Running Sphinx v6.1.3

loading pickled environment... done

building [mo]: targets for 0 po files that are out of date

writing output...

building [html]: targets for 1 source files that are out of date

updating environment: 0 added, 1 changed, 0 removed

reading sources... [100%] index

looking for now-outdated files... none found

pickling environment... done

checking consistency... done

preparing documents... done

writing output... [100%] index

generating indices... genindex py-modindex done

writing additional pages... search done

copying static files... done

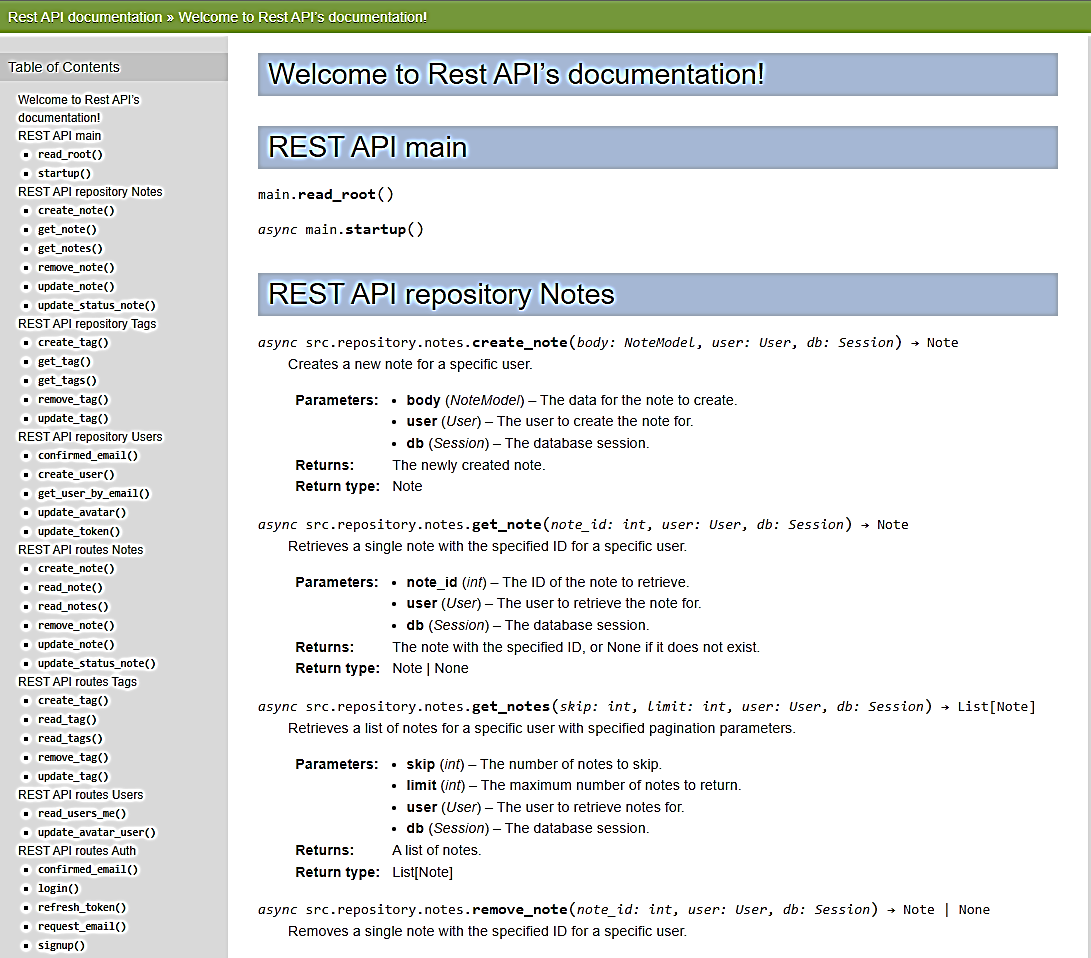
copying extra files... done

dumping search index in English (code: en)... done

dumping object inventory... done

build succeeded.

Результатом виконання команди повинен бути проект документації в папці docs/\_build/html. Точкою входу для нашої документації буде файл docs/\_build/html/index.html. Відкрийте його і ви повинні отримати результат як на зображенні нижче.



**Документація проекту**

**Висновок**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/sphinx#%D0%B2%D0%B8%D1%81%D0%BD%D0%BE%D0%B2%D0%BE%D0%BA)І пам'ятайте, що створення документації є важливою частиною розробки будь-якого проекту:

* Спрощення використання — документація полегшує використання програмного забезпечення, оскільки вона дає змогу користувачам зрозуміти, як працює програмне забезпечення і як ним користуватися. Це особливо важливо для складних проектів із безліччю функцій та опцій;
* Покращення підтримки — документація допомагає прискорити процес підтримки проекту, оскільки вона дає змогу програмістам швидко знайти відповіді на свої запитання і розв'язати проблеми, що виникають під час розвитку проекту;
* Покращення якості коду — створення документації допомагає програмістам краще розуміти їхній власний код, а отже, призводить до поліпшення якості коду і надійнішого програмного забезпечення загалом;
* Прискорення процесу розробки — створення документації прискорює процес розробки, оскільки документація дає змогу програмістам краще організовувати свій код і легше розуміти його;

**Фреймворк**[**Unittest**](https://docs.python.org/3/library/unittest.html) (https://docs.python.org/3/library/unittest.html)

Модульне тестування - це процес тестування окремих модулів або компонентів коду програми для виявлення і усунення помилок, і перевірки правильності їхньої роботи. Такі тести зазвичай створюються програмістами на ранніх етапах розробки продукту, до того, як код буде об'єднано з іншими модулями та застосунком загалом.

Основними перевагами модульних тестів є:

1. Забезпечення працездатності окремих компонентів коду;
2. Запобігання появі помилок у коді;
3. Прискорення процесу розробки та зменшення кількості помилок, завдяки швидкій перевірці змін у коді.

Використання модульних тестів допомагає створити стабільніший і надійніший застосунок, полегшує його підтримку та оновлення, покращує розуміння коду програми, а також прискорює процес відлагодження та усунення помилок.

Вбудований у ядро Python фреймворк модульного тестування Unittest має досить потужний функціонал, також підтримує всі можливості Python і гарантовано працює очікувано без зайвої "магії".

Приклад тестування

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/unitest#%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)import unittest

def multiply\_numbers(x, y):

return x \* y

class TestMultiplication(unittest.TestCase):

def test\_multiply\_two\_positive\_numbers(self):

result = multiply\_numbers(2, 3)

self.assertEqual(result, 6)

def test\_multiply\_positive\_and\_negative\_numbers(self):

result = multiply\_numbers(2, -3)

self.assertEqual(result, -6)

def test\_multiply\_two\_negative\_numbers(self):

result = multiply\_numbers(-2, -3)

self.assertEqual(result, 6)

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

unittest.main()

Щоб написати тест, достатньо створити клас, ім'я якого починається з Test, інакше unittest проігнорує цей клас. Також він має бути успадкований від класу unittest.TestCase.

У цьому прикладі створюється клас TestMultiplication, який успадковується від unittest.TestCase. Кожен метод починається з префікса test\_, щоб модуль unittest міг автоматично визначити, що цей метод є тестом. Кожен метод викликає функцію multiply\_numbers з різними аргументами і перевіряє очікуваний результат за допомогою методу assertEqual.

Пакет unittest в результаті своєї роботи розділяє всі тести на 3 групи:

* пройдені успішно;
* провалені;
* ті, що викликають помилку.

Різниця між останніми двома в тому, що провалений тест сигналізує про неправильну поведінку застосунку, а тест, що викликає помилку, призводить до винятку Exception у коді застосунку, і застосунок "падає" в такому разі.

**Методи перевірки**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/unitest#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D1%96%D1%80%D0%BA%D0%B8)

Щоб вказати unittest, що саме потрібно перевірити, у класу TestCase є набір допоміжних методів і ось деякі з найпопулярніших:

* assertEqual(a, b) - перевіряє, що значення a і b повинні бути рівними;
* assertNotEqual(a, b) - перевіряє, що значення a і b не рівні.
* assertTrue(x) - перевіряє, що x правдиве (тобто не дорівнює False, 0, '', None).
* assertFalse(x) - перевіряє, що x хибне (тобто дорівнює False, 0, '', None).
* assertIs(a, b) - перевіряє, що a і b посилаються на один і той самий об'єкт.
* assertIsNot(a, b) - перевіряє, що a і b не посилаються на один і той самий об'єкт.
* assertIsNone(x) - перевіряє, що x дорівнює None.
* assertIsNotNone(x) - перевіряє, що x не дорівнює None.
* assertRaises - код всередині цього контексту повинен викликати виняток, переданий у першому аргументі;

**Test fixture**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/unitest#test-fixture)У загальному випадку Test fixture - це підготовка, необхідна для виконання тестів і необхідні дії з очищення після виконання тестів. Наприклад, створення тимчасових баз даних перед тестами.

Щоб виконати якийсь код, перед кожним тестом всередині Test suite ви можете визначити методи setUp і tearDown. Ці методи викликаються до та після кожного тестового методу в тестовому наборі. Метод setUp використовується для налаштування оточення тестування перед запуском тестів, а tearDown використовується для очищення або знищення ресурсів після виконання кожного тестового методу.

import unittest

class MyTest(unittest.TestCase):

def setUp(self):

self.my\_list = [1, 2, 3]

def tearDown(self):

del self.my\_list

def test\_list\_length(self):

self.assertEqual(len(self.my\_list), 3)

def test\_list\_contents(self):

self.assertListEqual(self.my\_list, [1, 2, 3])

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

unittest.main()

У цьому прикладі метод setUp створює список my\_list, який буде використовуватися в кожному тестовому методі. Метод tearDown видаляє список після виконання кожного тестового методу.

Тестовий набір містить два тестових методи: test\_list\_length і test\_list\_contents, які перевіряють довжину і вміст списку, відповідно.

Методи assertEqual і assertListEqual використовуються для перевірки очікуваних результатів.

Виклик unittest.main() запускає всі тести в цьому наборі.

Якщо потрібно виконати деякі операції, спільні для всього класу, можна визначити класові методи setUpClass і tearDownClass. Ці методи будуть викликані перед початком виконання всіх тестів з TestCase і після завершення їх всіх, відповідно.

Давайте розглянемо детальніший приклад. Нехай є набір функцій, що виконує прості числові обчислення:

**ops.py**

def add(a, b):

return a + b

def sub(a, b):

return a-b

def mul(a, b):

return a \* b

def div(a, b):

return a / b

Створимо набір тестів для перевірки працездатності коду:

**test\_ops.py**

import unittest

from ops import add, sub, mul, div

class TestExamples(unittest.TestCase):

@classmethod

def setUpClass(cls):

print('Start before all test')

@classmethod

def tearDownClass(cls):

print('Start after all test')

def setUp(self):

print('Start before each test')

def tearDown(self):

print('Start after each test')

def test\_add(self):

print("Add function test")

self.assertEqual(add(2, 3), 5)

def test\_sub(self):

print("Sub function test")

self.assertEqual(sub(2, 3), -1)

def test\_mul(self):

print("Mul function test")

self.assertEqual(mul(2, 3), 6)

def test\_div(self):

print("Div function test")

self.assertAlmostEqual(div(2, 3), 0.66666666)

with self.assertRaises(ZeroDivisionError) as cm:

div(3, 0)

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

unittest.main()

Якщо ми виконаємо код, то отримаємо наступне виведення:

Start before all test

Start before each test

Add function test

Start after each test

Start before each test

Div function test

Start after each test

Start before each test

Mul function test

Start after each test

Start before each test

Sub function test

Start after each test

Start after all test

Ran 4 tests in 0.008s

OK

Як бачимо, тут чітко простежується виконання функції setUp перед кожним тестом, а функції tearDown після кожного тесту. Функція setUpClass виконується перед усіма тестами, а функція tearDownClass після всіх тестів.

Зауважимо, що в тесті test\_div для функції div здійснюються дві перевірки. У першій перевіряється, що результат виконання функції при передачі аргументів 2 і 3 дорівнює очікуваному результату 0.6666666666 з точністю до 8 знаків після коми. Ця перевірка здійснюється спеціальною функцією assertAlmostEqual, вона використовується, коли потрібно виконати порівняння двох дійсних чисел. У другій перевіряється, що при передачі аргументів 3 і 0 функція викликає виняток ZeroDivisionError. Якщо функція div викликає цей виняток, тоді тест проходить успішно, інакше тест завершиться невдало.

**Тестування асинхронного коду**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/unitest#%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B0%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE-%D0%BA%D0%BE%D0%B4%D1%83)Пакет unittest містить інструментарій для тестування асинхронного коду IsolatedAsyncioTestCase.

import asyncio

import unittest

async def async\_add(a, b):

await asyncio.sleep(1)

return a + b

class TestAsyncMethod(unittest.IsolatedAsyncioTestCase):

async def test\_add(self):

"""Add function test"""

r = await async\_add(2, 3)

self.assertEqual(r, 5)

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

unittest.main()

У нас є асинхронна функція async\_add, яка обчислює суму двох аргументів і повертає результат через одну секунду. Ми визначаємо клас TestAsyncMethod, що наслідується від unittest.IsolatedAsyncioTestCase, який містить метод test\_add, що тестує роботу функції async\_add. Всередині методу test\_add викликається асинхронна функція async\_add і перевіряється, що її результат дорівнює очікуваному значенню.

**Mock-об'єкти. Заглушки для тестування**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/unitest#mock-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D0%B8-%D0%B7%D0%B0%D0%B3%D0%BB%D1%83%D1%88%D0%BA%D0%B8-%D0%B4%D0%BB%D1%8F-%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)**Mock**-об'єкт — це об'єкт-імітатор. Основне завдання **mock**-об'єктів дати можливість ізолювати код від будь-яких зовнішніх залежностей, створивши повністю контрольоване вами середовище, замінивши залежності **mock**-об'єктами. Вони дають змогу імітувати поведінку реальних об'єктів і використовуються в тестах для створення необхідних умов.

Mock-об'єкти зазвичай використовуються в ситуаціях, коли тестування вимагає створення умов, які важко або неможливо відтворити в справжньому оточенні.

Наприклад, це може бути необхідно, коли:

* код, що тестується, взаємодіє із зовнішніми ресурсами, такими як бази даних, веб-сервіси, файлові системи тощо;
* код, що тестується, залежить від інших компонентів, які ще не були реалізовані або не готові для тестування;
* код, що тестується, виконує деякі дії, які не можуть бути відтворені в цьому середовищі, наприклад, виклик певного винятку.

from unittest.mock import MagicMock

from my\_module import ProductionClass

thing = ProductionClass()

thing.method = MagicMock(return\_value=3)

thing.method(1, 2, 3, key='value')

thing.method.assert\_called\_with(1, 2, 3, key='value')

Об'єкт thing відіграє роль деякої зовнішньої залежності. Ми можемо створити **mock**-реалізацію цілого об'єкта або окремого його методу. У прикладі це метод method. Створюючи **mock**-реалізацію, ми можемо вказати, що вона повинна повертати, коли її викликають, значення 3: return\_value=3. Потім необхідно перевірити, що код викликав цей **mock** -метод і з певними аргументами: thing.method.assert\_called\_with. Таким чином, можна перевіряти, що ваш код повинен або не повинен звертатися до якихось залежностей і як саме.

Нерідко для тестування потрібно замінити "на гарячу" якийсь метод, для цього є декоратор і менеджер контексту patch:

with patch.object(ProductionClass, 'method', return\_value=3) as mock\_method:

thing = ProductionClass()

thing.method(1, 2, 3)

mock\_method.assert\_called\_once\_with(1, 2, 3)

У цьому прикладі ми всередині контексту замінили метод method класу ProductionClass на mock реалізацію.

Або ми можемо замінити цілий модуль або частину його вмісту для тесту:

from unittest.mock import patch

@patch('module.ClassName2')

@patch('module.ClassName1')

def test\_class(MockClass1, MockClass2):

module.ClassName1()

module.ClassName2()

assert MockClass1 is module.ClassName1

assert MockClass2 is module.ClassName2

assert MockClass1.called

assert MockClass2.called

У цьому прикладі ми замінили в модулі module два класи ClassName2 і ClassName1 на **mock**-реалізації. Звісно, що модуль module повинен існувати для цього.

У тесті ми використовуємо декоратор @patch, щоб замінити вихідні класи ClassName1 і ClassName2 у модулі module на мок-об'єкти MockClass1 і MockClass2 відповідно. Декоратори передадуть їх як параметри функції test\_class.

Потім ми викликаємо вихідні класи ClassName1 і ClassName2 всередині функції test\_class, але насправді вони замінені на мок-об'єкти. Ми перевіряємо, що заміна відбулася успішно, порівнюючи значення MockClass1 і MockClass2 з оригінальними класами ClassName1 і ClassName2.

Нарешті, ми перевіряємо, чи були викликані мок-об'єкти, використовуючи атрибут called.

Мок-об'єкти використовуються у тестуванні для заміни реальних об'єктів, які можуть бути недоступні під час тестування, наприклад, через зовнішні залежності, тривалі операції або непередбачувані стани. Замість цього, ми можемо використовувати мок-об'єкти для імітації поведінки цих об'єктів і перевірки, як наш код реагує на цю поведінку.

Використання Mock-об'єктів дає змогу створювати умови, які необхідні для тестування коду, і контролювати їхню поведінку під час виконання тестів.

**Модульні тести для репозиторію**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/unitest#%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D1%96-%D1%82%D0%B5%D1%81%D1%82%D0%B8-%D0%B4%D0%BB%D1%8F-%D1%80%D0%B5%D0%BF%D0%BE%D0%B7%D0%B8%D1%82%D0%BE%D1%80%D1%96%D1%8E)Кінцева мета модульного тестування - це перевірити правильність роботи кожної функції в модулі та переконатися, що вони повертають правильний результат для всіх можливих вхідних даних. Як приклад ми покриємо модульними тестами функції нашого модуля src/repository/notes.py для роботи з нотатками. Створимо в корені проекту папку tests і файл із тестами test\_unit\_repository\_notes.py

**tests/test\_unit\_repository\_notes.py**

import unittest

from unittest.mock import MagicMock

from sqlalchemy.orm import Session

from src.database.models import Note, Tag, User

from src.schemas import NoteModel, NoteUpdate, NoteStatusUpdate

from src.repository.notes import (

get\_notes,

get\_note,

create\_note,

remove\_note,

update\_note,

update\_status\_note,

)

class TestNotes(unittest.IsolatedAsyncioTestCase):

def setUp(self):

self.session = MagicMock(spec=Session)

self.user = User(id=1)

async def test\_get\_notes(self):

notes = [Note(), Note(), Note()]

self.session.query().filter().offset().limit().all.return\_value = notes

result = await get\_notes(skip=0, limit=10, user=self.user, db=self.session)

self.assertEqual(result, notes)

async def test\_get\_note\_found(self):

note = Note()

self.session.query().filter().first.return\_value = note

result = await get\_note(note\_id=1, user=self.user, db=self.session)

self.assertEqual(result, note)

async def test\_get\_note\_not\_found(self):

self.session.query().filter().first.return\_value = None

result = await get\_note(note\_id=1, user=self.user, db=self.session)

self.assertIsNone(result)

async def test\_create\_note(self):

body = NoteModel(title="test", description="test note", tags=[1, 2])

tags = [Tag(id=1, user\_id=1), Tag(id=2, user\_id=1)]

self.session.query().filter().all.return\_value = tags

result = await create\_note(body=body, user=self.user, db=self.session)

self.assertEqual(result.title, body.title)

self.assertEqual(result.description, body.description)

self.assertEqual(result.tags, tags)

self.assertTrue(hasattr(result, "id"))

async def test\_remove\_note\_found(self):

note = Note()

self.session.query().filter().first.return\_value = note

result = await remove\_note(note\_id=1, user=self.user, db=self.session)

self.assertEqual(result, note)

async def test\_remove\_note\_not\_found(self):

self.session.query().filter().first.return\_value = None

result = await remove\_note(note\_id=1, user=self.user, db=self.session)

self.assertIsNone(result)

async def test\_update\_note\_found(self):

body = NoteUpdate(title="test", description="test note", tags=[1, 2], done=True)

tags = [Tag(id=1, user\_id=1), Tag(id=2, user\_id=1)]

note = Note(tags=tags)

self.session.query().filter().first.return\_value = note

self.session.query().filter().all.return\_value = tags

self.session.commit.return\_value = None

result = await update\_note(note\_id=1, body=body, user=self.user, db=self.session)

self.assertEqual(result, note)

async def test\_update\_note\_not\_found(self):

body = NoteUpdate(title="test", description="test note", tags=[1, 2], done=True)

self.session.query().filter().first.return\_value = None

self.session.commit.return\_value = None

result = await update\_note(note\_id=1, body=body, user=self.user, db=self.session)

self.assertIsNone(result)

async def test\_update\_status\_note\_found(self):

body = NoteStatusUpdate(done=True)

note = Note()

self.session.query().filter().first.return\_value = note

self.session.commit.return\_value = None

result = await update\_status\_note(note\_id=1, body=body, user=self.user, db=self.session)

self.assertEqual(result, note)

async def test\_update\_status\_note\_not\_found(self):

body = NoteStatusUpdate(done=True)

self.session.query().filter().first.return\_value = None

self.session.commit.return\_value = None

result = await update\_status\_note(note\_id=1, body=body, user=self.user, db=self.session)

self.assertIsNone(result)

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

unittest.main()

Перед кожним тестом виконується функція setUp, де ми створюємо об'єкт MagicMock для заміни об'єкта Session у модульних тестах self.session = MagicMock(spec=Session). У цьому випадку MagicMock використовується для створення "фіктивного" об'єкта Session, а використання параметра spec=Session у конструкторі MagicMock вказує, що створюваний об'єкт матиме ті самі атрибути і методи, що й об'єкт Session.

Кожен метод класу TestNotes тестує поведінку функцій модуля src/repository/notes.py.

1. Тест async def test\_get\_notes(self) перевіряє коректність роботи функції get\_notes, яка повертає список нотаток користувача. Для перевірки використовується мок-об'єкт сесії бази даних self.session, який імітує отримання списку нотаток із бази даних self.session.query().filter().offset().limit().all.return\_value = notes. Створюються три нотатки notes = [Note(), Note(), Note(), Note()], які очікуються в результаті виконання функції get\_notes. Тест порівнює отриманий результат result з очікуваним notes і перевіряє, що вони збігаються self.assertEqual(result, notes).
2. Тест async def test\_get\_note\_found(self) перевіряє коректність роботи функції get\_note, яка повертає одну нотатку користувача за її ідентифікатором. Для перевірки використовується мок-об'єкт сесії бази даних self.session, який імітує отримання нотатки з бази даних self.session.query().filter().first.return\_value = note. Створюється одна нотатка note = Note(), яка очікується в результаті виконання функції get\_note. Тест порівнює отриманий результат з очікуваним і перевіряє, що вони збігаються self.assertEqual(result, note).
3. Ми також перевіряємо ситуацію пошуку нотатки, яка не існує. Тест async def test\_get\_note\_not\_found(self) перевіряє коректність роботи функції get\_note, коли нотатка із зазначеним ідентифікатором не знайдена. Для перевірки використовується мок-об'єкт сесії бази даних, який імітує відсутність нотатки в базі даних. Очікується, що функція поверне значення None. Тест перевіряє, що функція повертає саме None.
4. Тест async def test\_create\_note(self) перевіряє коректність роботи функції create\_note, яка створює нову нотатку користувача. Ми імітуємо отримання вхідних даних за допомогою моделі валідації body = NoteModel(title="test", description="test note", tags=[1, 2]). Для перевірки використовується мок-об'єкт сесії бази даних, який імітує отримання списку тегів користувача з бази даних.

tags = [Tag(id=1, user\_id=1), Tag(id=2, user\_id=1)]

self.session.query().filter().all.return\_value = tags

Створюється нова нотатка із зазначеними параметрами, яка очікується в результаті виконання функції create\_note. Тест перевіряє, що отриманий результат відповідає очікуваному і порівнює поля result з переданими полями body. Також перевіряємо, що створена нотатка отримує унікальний ідентифікатор id.

self.assertEqual(result.title, body.title)

self.assertEqual(result.description, body.description)

self.assertEqual(result.tags, tags)

self.assertTrue(hasattr(result, "id"))

5.Тест async def test\_remove\_note\_found(self) перевіряє коректність роботи функції remove\_note, яка видаляє нотатку користувача за її ідентифікатором. Створюється одна нотатка, яка очікується в результаті функції remove\_note.

note = Note()

self.session.query().filter().first.return\_value = note

Тест порівнює отриманий результат з очікуваним і перевіряє, що вони збігаються self.assertEqual(result, note).

Решта тестів класу TestNotes використовують аналогічний підхід і їхнє функціонування повинно вже бути зрозумілим. Таким самим способом ми можемо написати модульні тести для решти модулів.

ЗАВДАННЯ

Спробуйте самостійно написати тести для модулів src/repository/tags.py та src/repository/users.py.

**Фреймворк PyTest**

Один із найпопулярніших фреймворків для тестування — це [PyTest](https://docs.pytest.org/en/6.2.x/) (https://docs.pytest.org/en/6.2.x/). Фреймворк дає можливість швидше писати складніші сценарії тестування застосунку. При цьому основний наголос PyTest робить на спрощення написання маленьких тестів. Є дуже велика кількість розширень для PyTest, що спрощують написання тестів. PyTest вміє запускати тести, написані з використанням інших популярних фреймворків, таких як unittest і nose. Ще однією ключовою особливістю PyTest є його зручне та зрозуміле виведення результатів тестування.

Щоб почати використовувати pytest, потрібно встановити його за допомогою poetry:

poetry add pytest

В той час як unittest наголошує на класах і методах, PyTest дотримується функціонального підходу. Ви, звичайно, можете об'єднувати тести в класи, але в цьому немає необхідності.

**Створення тестів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/pytest#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%B5%D1%81%D1%82%D1%96%D0%B2)Тести в pytest зазвичай знаходяться у файлах з іменами, що починаються з test\_ або з ім'ям файлу, що закінчується на \_test. Функції тестування повинні починатися з test\_, щоб pytest міг їх знайти і запустити. Наприклад:

**main.py**

def inc(x):

return x + 1

def test\_answer():

assert inc(3) == 5

У цьому прикладі ми створили функцію inc і тест, який тестує цю функцію: test\_answer. Щоб запустити тести, достатньо виконати в консолі: pytest main.py.

Виведення буде:

pytest main.py

====================== test session starts ======================

platform win32 -- Python 3.10.5, pytest-7.2.1, pluggy-1.0.0

rootdir: E:\WebDir\Works\Python\python-help-solution\module14\m14\_

solution

collected 1 item

main.py F [100%]

=========================== FAILURES ============================

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**\_\_ test*\_answer* ***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_****\_*\_

def test\_answer():

> assert inc(3) == 5

E assert 4 == 5

E + where 4 = inc(3)

main.py:6: AssertionError

==================== short test summary info ====================

FAILED main.py::test*\_answer - assert 4 == 5*

*======================= 1 failed in 0.10s =======================*

Як бачимо PyTest трактує будь-яке виключення AssertionError як тест, що не пройшов перевірку. Це відрізняє його від unittest, де є спеціальний механізм, щоб розрізняти винятки, що трапляються в застосунку, від винятків, викликаних "неправильною" поведінкою застосунку.

**Запуск тестів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/pytest#%D0%B7%D0%B0%D0%BF%D1%83%D1%81%D0%BA-%D1%82%D0%B5%D1%81%D1%82%D1%96%D0%B2)Для запуску тестів у Pytest потрібно викликати команду pytest з необхідними параметрами з терміналу. Ось найнеобхідніші параметри, які можна використовувати в командному рядку Pytest:

* -v або --verbose - показує додаткову інформацію про тести, наприклад, імена тестів, які були пройдені.
* -s - вимикає перехоплення виведення, щоб виведення тестових функцій було спрямовано в консоль.
* -x - зупиняє виконання тестів за першої помилки або невдалого тесту.
* -q або --quiet - зменшує кількість виведених повідомлень про прогрес виконання тестів.
* -h або --help - виводить довідкову інформацію щодо використання команди.

Для запуску тестів потрібно запустити скрипт pytest і передати йому шлях до файлу з тестами:

pytest test\_example.py

Можна також передати директорію з тестами:

pytest tests/

PyTest також підтримує безліч опцій командного рядка. Для отримання списку опцій можна виконати команду:

pytest *--help*

Також для запуску тестів, за допомогою pytest, необхідно додати в файл pyproject.toml наступну секцію:

[tool.pytest.ini\_options]

pythonpath = ["."]

Це дозволяє pytest зрозуміти, що імпорт модулів в тестах потрібно проводити з кореня проекту.

**Початкові умови тестів**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/pytest#%D0%BF%D0%BE%D1%87%D0%B0%D1%82%D0%BA%D0%BE%D0%B2%D1%96-%D1%83%D0%BC%D0%BE%D0%B2%D0%B8-%D1%82%D0%B5%D1%81%D1%82%D1%96%D0%B2)

INFO

Fixture (англ. "арматура") у тестуванні - це фрагмент коду, який потрібен для підготовки тестового оточення, виконання тестів і очищення тестового оточення після тестування.

Щоб не повторювати код у кожній тестовій функції, можна визначити функцію fixture, яка виконуватиметься перед кожним тестом. Для цього використовується декоратор @pytest.fixture. Fixture може повертати будь-який об'єкт, зокрема об'єкти класів, словники, рядки тощо.

**pex02.py**

import pytest

@pytest.fixture

def some\_data():

return {"a": 1, "b": 2, "c": 3}

def test\_addition(some\_data):

assert some\_data["a"] + some\_data["b"] == 3

def test\_subtraction(some\_data):

assert some\_data["c"] - some\_data["b"] == 1

Фікстура some\_data повертає словник із ключами a, b і c, кожен із яких має своє значення. У першому тесті test\_addition(some\_data) ми отримуємо словник як параметр і перевіряємо, що сума значень "a" і "b" дорівнює 3. У другому тесті test\_subtraction(some\_data) перевіряється, що різниця значень "c" і "b" дорівнює 1.

У результаті виконання тесту ми отримаємо таке виведення:

pytest .\pex02.py -v

================================== test session starts ===================================

platform win32 -- Python 3.10.5, pytest-7.2.1, pluggy-1.0.0 -- C:\Users\Krabat\AppData\Loca

l\pypoetry\Cache\virtualenvs\m14-solution-NwB1pYkt-py3.10\Scripts\python.exe

cachedir: .pytest\_cache

rootdir: E:\WebDir\Works\Python\python-help-solution\module14\m14\_solution

plugins: anyio-3.6.2

collected 2 items

pex02.py::test\_addition PASSED [ 50%]

pex02.py::test\_subtraction PASSED [100%]

=================================== 2 passed in 0.02s ====================================

У цьому тесті ми створили фікстуру some\_data, яка створює словник. Зверніть увагу, щоб скористатися цією фікстурою, ми передаємо її в тест як аргумент тесту. Це непрозорий момент, якщо припуститися помилки в імені фікстури або аргументі тесту, то PyTest викличе виняток, будьте акуратними.

Найчастіше фікстуру використовують для створення з'єднання з базою даних. Наприклад, створимо файл зі з'єднанням із базою даних:

**db.py**

import sqlite3

from contextlib import contextmanager

@contextmanager

def create\_connection():

conn = sqlite3.connect(':memory:')

yield conn

conn.rollback()

conn.close()

І сам файл із тестами:

import pytest

import db

data = ("John", "john@example.com")

@pytest.fixture

def db\_connection():

with db.create\_connection() as connection:

yield connection

def test\_db\_operation(db\_connection):

cursor = db\_connection.cursor()

cursor.execute("""

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,

fullname STRING,

email STRING

);

""")

cursor.execute("INSERT INTO users (fullname, email) VALUES (?, ?)", data)

db\_connection.commit()

r = cursor.execute("SELECT fullname, email FROM users")

assert r.fetchone() == data

Ця фікстура створює з'єднання з базою даних і закриває його після завершення тесту. Далі ми використовуємо фікстуру в тестовій функції test\_db\_operation, яка потребує з'єднання з базою даних.

Виконання тесту:

pytest .\pex03.py -v

================================== test session starts ===================================

platform win32 *-- Python 3.10.5, pytest-7.2.1, pluggy-1.0.0 -- C:\Users\Krabat\AppData\Loca*

l\pypoetry\Cache\virtualenvs\m14-solution-NwB1pYkt-py3.10\Scripts\python.exe

cachedir: .pytest\_cache

rootdir: E:\WebDir\Works\Python\python-help-solution\module14\m14\_solution

plugins: anyio-3.6.2

collected 1 item

pex03.py::test\_db\_operation PASSED [100%]

=================================== 1 passed in 0.01s ====================================

**Параметризовані тести**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/pytest#%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D1%96-%D1%82%D0%B5%D1%81%D1%82%D0%B8)Параметризовані тести - це техніка в тестуванні, яка дає змогу запускати один і той самий тест із різними вхідними параметрами. Це дає змогу значно спростити процес написання та підтримки тестів, а також збільшити покриття коду.

Можна визначити кілька параметризованих тестів в одній функції, передаючи різні набори аргументів. Для цього використовується декоратор @pytest.mark.parametrize. Він дає змогу передати кілька значень параметрів у тестову функцію у вигляді кортежу або списку.

import pytest

@pytest.mark.parametrize("a, b, result", [

(1, 2, 3),

(2, 3, 5),

(3, 4, 7)

])

def test\_addition(a, b, result):

assert a + b == result

@pytest.mark.parametrize('test\_input, expected\_output', [

('3+5', 8),

('2+4', 6),

('6\*9', 54),

])

def test\_eval(test\_input, expected\_output):

assert eval(test\_input) == expected\_output

Функція pytest.mark.parametrize() автоматично запускає тести для кожної комбінації параметрів, передаючи їх у тестову функцію.

Виведення:

pex04.py::test\_addition*[1-2-3]* PASSED *[ 16%]*

pex04.py::test\_addition*[2-3-5]* PASSED *[ 33%]*

pex04.py::test\_addition*[3-4-7]* PASSED *[ 50%]*

pex04.py::test\_eval*[3+5-8]* PASSED *[ 66%]*

pex04.py::test\_eval*[2+4-6]* PASSED *[ 83%]*

pex04.py::test\_eval*[6\*9-54]* PASSED *[100%]*

=================================== 6 passed in 0.02s ====================================

Окрім цього, pytest.mark.parametrize можна використовувати з фікстурами, щоб передавати параметри з фікстури в тестову функцію. Приклад:

import pytest

@pytest.fixture

def add\_fixture():

return lambda x, y: x + y

@pytest.mark.parametrize('a, b, expected', [

(1, 2, 3),

(3, 4, 7),

(-2, 5, 3),

])

def test\_addition(add\_fixture, a, b, expected):

assert add\_fixture(a, b) == expected

Тут ми використовуємо фікстуру add\_fixture, яка повертає лямбда-функцію додавання двох чисел. Декоратор pytest.mark.parametrize дає змогу передати кілька значень параметрів a, b і expected у тестову функцію test\_addition. Функція add\_fixture використовується як параметр add\_fixture для тестової функції.

**Mock об'єкти**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/pytest#mock-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D0%B8)PyTest також підтримує моки, які дозволяють замінити об'єкти на фіктивні об'єкти. Для цього можна використовувати бібліотеку unittest.mock або бібліотеку [pytest-mock](https://pytest-mock.readthedocs.io/en/latest/index.html) (https://pytest-mock.readthedocs.io/en/latest/index.html):

import requests

from unittest.mock import Mock, patch

def send\_request(url):

response = requests.get(url)

return response.status\_code

def test\_send\_request():

mock\_get = Mock(return\_value=Mock(status\_code=200))

with patch('requests.get', mock\_get):

status\_code = send\_request('http://example.com')

assert status\_code == 200

mock\_get.assert\_called\_once\_with('http://example.com')

У цьому тесті ми створюємо мок-об'єкт mock\_get, який імітує реальну функцію requests.get і завжди повертає об'єкт, що містить статус-код 200. Потім ми використовуємо patch з unittest.mock, щоб тимчасово замінити справжню функцію requests.get на наш мок об'єкт. Викликаємо нашу функцію `send\_request з довільною URL-адресою, і вона повинна повернути статус-код 200, який ми перевіряємо за допомогою твердження assert. Нарешті, ми переконуємося, що наш мок-об'єкт mock\_get був викликаний один раз із потрібною URL-адресою, використовуючи метод assert\_called\_once\_with.

Для використання pytest-mock у прикладі необхідно встановити цей пакет:

poetry add pytest-mock

І переписати функцію тесту наступним чином:

import requests

def send\_request(url):

response = requests.get(url)

return response.status\_code

def test\_send\_request(mocker):

mock\_get = mocker.patch('requests.get')

mock\_get.return\_value.status\_code = 200

status\_code = send\_request('http://example.com')

assert status\_code == 200

mock\_get.assert\_called\_once\_with('http://example.com')

**Пишемо тести**

Наразі для нашого REST API є модульні тести для модуля src/repository/notes.py. Як приклад, використовуючи фреймворк pytest, покриємо тестами наступні маршрути:

* src/routes/auth.py - аутентифікація та авторизація
* src/routes/tags.py - робота з тегами

**Налаштування та конфігурація тестів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/write_test#%D0%BD%D0%B0%D0%BB%D0%B0%D1%88%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%82%D0%B0-%D0%BA%D0%BE%D0%BD%D1%84%D1%96%D0%B3%D1%83%D1%80%D0%B0%D1%86%D1%96%D1%8F-%D1%82%D0%B5%D1%81%D1%82%D1%96%D0%B2)В середині папки з тестами tests створіть файл conftest.py. Він використовується фреймворком pytest для налаштування та конфігурації тестів. Цей файл дозволяє задавати фікстури (fixtures) для тестів, які створюють і надають необхідні об'єкти та дані для їхнього виконання.

Фікстури можуть використовуватися в декількох тестових файлах і модулях, і визначати їх щоразу вручну для кожного тесту неефективно.

Файл conftest.py вирішує цю проблему, дозволяючи визначити фікстури один раз і використовувати їх у кількох тестових файлах і модулях. Функції-фікстури в цьому файлі визначаються за допомогою декоратора @pytest.fixture. Коли тест запускається, pytest автоматично шукає файл conftest.py і використовує функції-фікстури, визначені в ньому, для надання даних і об'єктів, які необхідні для виконання наших тестів.

**tests/conftest.py**

import pytest

from fastapi.testclient import TestClient

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

from main import app

from src.database.models import Base

from src.database.db import get\_db

SQLALCHEMY\_DATABASE\_URL = "sqlite:///./test.db"

engine = create\_engine(

SQLALCHEMY\_DATABASE\_URL, connect\_args={"check\_same\_thread": False}

)

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

@pytest.fixture(scope="module")

def session():

*# Create the database*

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

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

db = TestingSessionLocal()

try:

yield db

finally:

db.close()

@pytest.fixture(scope="module")

def client(session):

*# Dependency override*

def override\_get\_db():

try:

yield session

finally:

session.close()

app.dependency\_overrides[get\_db] = override\_get\_db

yield TestClient(app)

@pytest.fixture(scope="module")

def user():

return {"username": "deadpool", "email": "deadpool@example.com", "password": "123456789"}

Наш файл conftest.py містить набір фікстур, які ми будемо використовувати в тестах.

Фікстура session використовується для створення тестової бази даних SQLite і сесії SQLAlchemy. Для заміни бази даних на тестову ми використовуємо підхід, [**описаний у документації**](https://fastapi.tiangolo.com/advanced/testing-database/) (https://fastapi.tiangolo.com/advanced/testing-database/) фреймворка FastAPI.

INFO

Ми використовуємо тестову базу даних, щоб ізолювати тести від реальної бази даних. Якби ми використовували реальну базу даних для тестів, це могло б призвести до несподіваних змін у даних та інших проблем, аж до знищення реальних даних. Використання тестової бази даних дає нам змогу створювати і змінювати дані, не зачіпаючи реальну базу даних. Тестова база даних також може бути очищена перед кожним тестом, щоб переконатися, що тести виконуються в ізольованому середовищі.

Ця фікстура має область дії модуля scope="module", тобто буде виконана тільки один раз для всього модуля тестів.

Рядок Base.metadata.drop\_all(bind=engine) видаляє всі наявні таблиці з бази даних, потім Base.metadata.create\_all(bind=engine) створює всі таблиці, зазначені в моделях бази даних. Оператор yield повертає цю сесію, а блок finally забезпечує правильне закриття сесії після завершення всіх тестів.

Загалом, фікстура session встановлює чисту сесію бази даних для використання тестами і гарантує, що база даних буде видалена і відтворена до та після тестів.

Область видимості у фікстурах визначає, наскільки довго і де буде доступна фікстура під час виконання тестів. В pytest є чотири рівні областей видимості фікстур:

* function фікстура створюється для кожної функції тесту і знищується після її завершення;
* class фікстура створюється для кожного класу тестів і знищується після виконання всіх методів тестів у цьому класі;
* module фікстура створюється один раз для всього модуля і знищується після виконання всіх тестів у модулі
* session фікстура створюється один раз для всієї сесії тестування і знищується після виконання всіх тестів

Фікстура client використовується для створення об'єкта тестового клієнта TestClient(app), який, в свою чергу, буде використовуватися в тестах для виконання запитів до нашого REST API. У цій фікстурі визначено залежність get\_db — її ми замінюємо на створену раніше фікстуру session за допомогою операції app.dependency\_overrides[get\_db] = override\_get\_db. Тепер під час виконання тестів скрізь, де буде вираз db: Session = Depends(get\_db), у нашому застосунку, замість функції get\_db, буде використовуватися функція override\_get\_db. І застосунок під час тестів буде використовувати нашу тестову базу даних. Ця фікстура також має область дії модуля.

INFO

TestClient - це клас із бібліотеки FastAPI, який дає змогу створювати клієнти для тестування FastAPI застосунків. Він надає зручний спосіб надсилання HTTP-запитів до застосунку, а потім виконати перевірку правильності відповіді. За допомогою TestClient можна виконати HTTP-запити, наприклад GET, POST, PUT і DELETE, на певну URL-адресу і отримати відповідь від застосунку.

У тестах TestClient зазвичай використовується для надсилання запитів до застосунку та перевірки відповіді. Він дозволяє тестувати реакцію застосунку на певні запити та сценарії використання, і переконатися, що вони працюють правильно. TestClient заснований на пакеті httpx і для запусків тестів нам необхідно буде встановити цей пакет: poetry add httpx -G test

Фікстура user використовується для створення тестового користувача, який буде використовуватися в деяких тестах.

**Тести для маршрутів аутентифікації та авторизації**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/write_test#%D1%82%D0%B5%D1%81%D1%82%D0%B8-%D0%B4%D0%BB%D1%8F-%D0%BC%D0%B0%D1%80%D1%88%D1%80%D1%83%D1%82%D1%96%D0%B2-%D0%B0%D1%83%D1%82%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%97-%D1%82%D0%B0-%D0%B0%D0%B2%D1%82%D0%BE%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D1%96%D1%97)Створимо файл tests/test\_route\_auth.py для тестування функціональності процесів аутентифікації вебзастосунку. Кожна функція файлу - це тест, який перевірятиме різні аспекти аутентифікації:

* test\_create\_user: створення нового користувача та надсилання йому електронного листа для підтвердження email;
* test\_repeat\_create\_user: перевірка того, що під час повторної спроби створити користувача з тим самим email, повертається код помилки 409;
* test\_login\_user\_not\_confirmed: перевірка того, що користувач не може увійти в систему, якщо його email не був підтверджений;
* test\_login\_user: перевірка того, що зареєстрований і підтверджений користувач може увійти в систему з коректними даними;
* test\_login\_wrong\_password: перевірка того, що користувач не може увійти в систему з невірним паролем;
* test\_login\_wrong\_email: перевірка того, що користувач не може увійти в систему з невірним email.

Це частина необхідних тестів, які дозволять нам зрозуміти сам процес тестування.

**tests/test\_route\_auth.py**

from unittest.mock import MagicMock

from src.database.models import User

def test\_create\_user(client, user, monkeypatch):

mock\_send\_email = MagicMock()

monkeypatch.setattr("src.routes.auth.send\_email", mock\_send\_email)

response = client.post(

"/api/auth/signup",

json=user,

)

assert response.status\_code == 201, response.text

data = response.json()

assert data["user"]["email"] == user.get("email")

assert "id" in data["user"]

def test\_repeat\_create\_user(client, user):

response = client.post(

"/api/auth/signup",

json=user,

)

assert response.status\_code == 409, response.text

data = response.json()

assert data["detail"] == "Account already exists"

def test\_login\_user\_not\_confirmed(client, user):

response = client.post(

"/api/auth/login",

data={"username": user.get('email'), "password": user.get('password')},

)

assert response.status\_code == 401, response.text

data = response.json()

assert data["detail"] == "Email not confirmed"

def test\_login\_user(client, session, user):

current\_user: User = session.query(User).filter(User.email == user.get('email')).first()

current\_user.confirmed = True

session.commit()

response = client.post(

"/api/auth/login",

data={"username": user.get('email'), "password": user.get('password')},

)

assert response.status\_code == 200, response.text

data = response.json()

assert data["token\_type"] == "bearer"

def test\_login\_wrong\_password(client, user):

response = client.post(

"/api/auth/login",

data={"username": user.get('email'), "password": 'password'},

)

assert response.status\_code == 401, response.text

data = response.json()

assert data["detail"] == "Invalid password"

def test\_login\_wrong\_email(client, user):

response = client.post(

"/api/auth/login",

data={"username": 'email', "password": user.get('password')},

)

assert response.status\_code == 401, response.text

data = response.json()

assert data["detail"] == "Invalid email"

Розгляньмо декілька тестів детальніше для кращого розуміння їхнього функціонування. Тест на правильність створення користувача test\_create\_user:

def test\_create\_user(client, user, monkeypatch):

mock\_send\_email = MagicMock()

monkeypatch.setattr("src.routes.auth.send\_email", mock\_send\_email)

response = client.post(

"/api/auth/signup",

json=user,

)

assert response.status\_code == 201, response.text

data = response.json()

assert data["user"]["email"] == user.get("email")

assert "id" in data["user"]

Під час тестів ми не хочемо надсилати лист користувачу для верифікації email. Для цього необхідно виконати мок функції send\_email. Ми будемо використовувати MagicMock, який автоматично створить мок-об'єкт з усіма необхідними методами та атрибутами, включно з await, який використовується у функції send\_email. Для підміни функції send\_email ми використовуємо monkeypatch.

INFO

Цей [**механізм**](https://docs.pytest.org/en/7.1.x/how-to/monkeypatch.html) (https://docs.pytest.org/en/7.1.x/how-to/ monkeypatch.html) в pytest дозволяє замінити об'єкт у процесі виконання тестів. Об'єкт monkeypatch використовують для заміни значення змінної оточення, функції або класу на фіктивні значення під час виконання тестів.

Метод monkeypatch.setattr підміняє виклик функції send\_email з модуля src.routes.auth на мок-об'єкт mock\_send\_email. Після цього ми не будемо відправляти реальні електронні листи під час тестування. Цей механізм дуже корисний при створенні тестів, коли необхідно змінити якісь глобальні або системні налаштування, щоб уникнути впливу на реальну систему або, щоб змінити поведінку коду, що тестується, під час тестування.

Далі client.post виконує POST-запит на вказану URL-адресу /api/auth/signup, передаючи в тілі запиту JSON-представлення даних користувача user. Після відповіді нашого застосунку за допомогою assert переконуємося, що код стану відповіді сервера дорівнює 201 (успішне створення ресурсу). Якщо це не так, повертаємо помилку з текстом відповіді response.text. Перевіряємо дані, що повертаються застосунком data = response.json(), витягуючи JSON-дані з відповіді сервера. Перевіряємо assert data["user"]["email"] == user.get("email"), що електронна пошта нового користувача дорівнює очікуваній.

Перевіряємо assert "id" in data["user"], що повернуті дані містять ідентифікатор нового користувача, а отже, користувач був створений в базі даних.

Розберемо ще тест, який перевіряє, чи може користувач увійти в систему після реєстрації та підтвердження своєї адреси електронної пошти.

def test\_login\_user(client, session, user):

current\_user: User = session.query(User).filter(User.email == user.get('email')).first()

current\_user.confirmed = True

session.commit()

response = client.post(

"/api/auth/login",

data={"username": user.get('email'), "password": user.get('password')},

)

assert response.status\_code == 200, response.text

data = response.json()

assert data["token\_type"] == "bearer"

Спочатку тест витягує об'єкт користувача з бази даних current\_user, використовуючи його адресу електронної пошти, потім встановлює властивість підтвердження користувача в True і зберігає зміни в базі даних session.commit().

Нагадаємо, що, оскільки ми встановили параметр scope="module" для session, то під час виконання тестів у нашому модулі всі дані залишаються в базі даних, а, значить, ми маємо доступ до користувача, який створили в тесті test\_create\_user.

Після цього тест відправляє POST-запит на кінцеву точку /api/auth/login з електронною адресою та паролем користувача в якості тіла запиту. Зверніть увагу, що зараз ми передаємо об'єкт data, оскільки наш сервер очікує відповідь від форми з MIME-тип вмістом application/x-www-form-urlencoded.

Потім він перевіряє, що код стану відповіді дорівнює 200 і що JSON відповіді містить поле token\_type зі значенням "bearer", яке вказує на те, що користувач успішно пройшов аутентифікацію і отримав токен доступу.

Решта тестів логічно зрозумілі та схожі.

Виконаємо тести pytest tests/test\_route\_auth.py -v:

tests/test\_route\_auth.py::test\_create\_user PASSED [ 16%]

tests/test\_route\_auth.py::test\_repeat\_create\_user PASSED [ 33%]

tests/test\_route\_auth.py::test\_login\_user\_not\_confirmed PASSED [ 50%]

tests/test\_route\_auth.py::test\_login\_user PASSED [ 66%]

tests/test\_route\_auth.py::test\_login\_wrong\_password PASSED [ 83%]

tests/test\_route\_auth.py::test\_login\_wrong\_email PASSED [100%]

================================ 6 passed in 2.05s ================================

**Тести для роботи з тегами**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/write_test#%D1%82%D0%B5%D1%81%D1%82%D0%B8-%D0%B4%D0%BB%D1%8F-%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B8-%D0%B7-%D1%82%D0%B5%D0%B3%D0%B0%D0%BC%D0%B8)Тепер розглянемо тестовий модуль test\_route\_tags.py з тестами всіх CRUD операцій над тегами. Під час запуску тестів з тегами у нас вже нічого немає в тестовій базі даних, оскільки вона була очищена операцією Base.metadata.drop\_all(bind=engine) у фікстурі session.

Для цього модуля нам необхідна фікстура token, яка імітує аутентифікацію і повертає нам access\_token для авторизації. Вона створює користувача, виконує процедуру підтвердження його email-адреси, а потім авторизується через API і повертає токен доступу.

**tests/test\_route\_tags.py**

from unittest.mock import MagicMock, patch

import pytest

from src.database.models import User

from src.services.auth import auth\_service

@pytest.fixture()

def token(client, user, session, monkeypatch):

mock\_send\_email = MagicMock()

monkeypatch.setattr("src.routes.auth.send\_email", mock\_send\_email)

client.post("/api/auth/signup", json=user)

current\_user: User = session.query(User).filter(User.email == user.get('email')).first()

current\_user.confirmed = True

session.commit()

response = client.post(

"/api/auth/login",

data={"username": user.get('email'), "password": user.get('password')},

)

data = response.json()

return data["access\_token"]

def test\_create\_tag(client, token):

with patch.object(auth\_service, 'r') as r\_mock:

r\_mock.get.return\_value = None

response = client.post(

"/api/tags",

json={"name": "test\_tag"},

headers={"Authorization": f"Bearer {token}"}

)

assert response.status\_code == 201, response.text

data = response.json()

assert data["name"] == "test\_tag"

assert "id" in data

def test\_get\_tag(client, token):

with patch.object(auth\_service, 'r') as r\_mock:

r\_mock.get.return\_value = None

response = client.get(

"/api/tags/1",

headers={"Authorization": f"Bearer {token}"}

)

assert response.status\_code == 200, response.text

data = response.json()

assert data["name"] == "test\_tag"

assert "id" in data

def test\_get\_tag\_not\_found(client, token):

with patch.object(auth\_service, 'r') as r\_mock:

r\_mock.get.return\_value = None

response = client.get(

"/api/tags/2",

headers={"Authorization": f"Bearer {token}"}

)

assert response.status\_code == 404, response.text

data = response.json()

assert data["detail"] == "Tag not found"

def test\_get\_tags(client, token):

with patch.object(auth\_service, 'r') as r\_mock:

r\_mock.get.return\_value = None

response = client.get(

"/api/tags",

headers={"Authorization": f"Bearer {token}"}

)

assert response.status\_code == 200, response.text

data = response.json()

assert isinstance(data, list)

assert data[0]["name"] == "test\_tag"

assert "id" in data[0]

def test\_update\_tag(client, token):

with patch.object(auth\_service, 'r') as r\_mock:

r\_mock.get.return\_value = None

response = client.put(

"/api/tags/1",

json={"name": "new\_test\_tag"},

headers={"Authorization": f"Bearer {token}"}

)

assert response.status\_code == 200, response.text

data = response.json()

assert data["name"] == "new\_test\_tag"

assert "id" in data

def test\_update\_tag\_not\_found(client, token):

with patch.object(auth\_service, 'r') as r\_mock:

r\_mock.get.return\_value = None

response = client.put(

"/api/tags/2",

json={"name": "new\_test\_tag"},

headers={"Authorization": f"Bearer {token}"}

)

assert response.status\_code == 404, response.text

data = response.json()

assert data["detail"] == "Tag not found"

def test\_delete\_tag(client, token):

with patch.object(auth\_service, 'r') as r\_mock:

r\_mock.get.return\_value = None

response = client.delete(

"/api/tags/1",

headers={"Authorization": f"Bearer {token}"}

)

assert response.status\_code == 200, response.text

data = response.json()

assert data["name"] == "new\_test\_tag"

assert "id" in data

def test\_repeat\_delete\_tag(client, token):

with patch.object(auth\_service, 'r') as r\_mock:

r\_mock.get.return\_value = None

response = client.delete(

"/api/tags/1",

headers={"Authorization": f"Bearer {token}"}

)

assert response.status\_code == 404, response.text

data = response.json()

assert data["detail"] == "Tag not found"

Кожна тестова функція виконує HTTP-запит до відповідної кінцевої точки з дійсним маркером доступу, включеним у заголовок запиту. Ми використовуємо функцію patch.object для імітації об'єкта r, який є екземпляром Redis у модулі auth\_service. Ми ставимо цю заглушку, щоб не було необхідності запускати Redis під час виконання тестів.

* test\_create\_tag тестує створення нового тегу за маршрутом POST /api/tags. Тест перевіряє, що код стану відповіді дорівнює 201, дані відповіді містять правильне ім'я тегу і що в даних відповіді присутнє поле id.
* test\_get\_tag перевіряє отримання існуючого тегу за маршрутом GET /api/tags/{id}. Тест перевіряє, що код стану відповіді дорівнює 200, дані відповіді містять правильне ім'я тегу, і що поле id присутнє в даних відповіді.
* test\_get\_tag\_not\_found тестує отримання неіснуючого тегу за маршрутом GET /api/tags/{id}. Тест перевіряє, що код стану відповіді при цьому дорівнює 404, а дані відповіді містять правильне повідомлення про помилку.
* test\_get\_tags перевіряє отримання всіх тегів за маршрутом GET /api/tags. Тест перевіряє, що код стану відповіді дорівнює 200, дані відповіді - це список, перший елемент списку містить правильне ім'я тегу, і що поле id присутнє в першому елементі даних відповіді.
* test\_update\_tag перевіряє оновлення існуючого тегу за маршрутом PUT /api/tags/{id}. Тест перевіряє, що код стану відповіді дорівнює 200, дані відповіді містять правильне ім'я тегу і що поле id присутнє в даних відповіді.
* test\_update\_tag\_not\_found перевіряє оновлення неіснуючого тегу за маршрутом PUT /api/tags/{id}. Тест перевіряє, що код стану відповіді дорівнює 404, а дані відповіді містять правильне повідомлення про помилку.
* test\_delete\_tag перевіряє видалення існуючого тегу за маршрутом DELETE /api/tags/{id}. Тест перевіряє, що код стану відповіді дорівнює 200, дані відповіді містять правильне ім'я тегу і що поле id присутнє в даних відповіді.
* test\_repeat\_delete\_tag перевіряє видалення неіснуючого тегу за маршрутом DELETE /api/tags/{id}. Тест перевіряє, що код стану відповіді дорівнює 404, а дані відповіді містять правильне повідомлення про помилку.

Виконаємо тести pytest tests/test\_route\_tags.py -v:

tests/test\_route\_tags.py::test\_create\_tag PASSED [ 12%]

tests/test\_route\_tags.py::test\_get\_tag PASSED [ 25%]

tests/test\_route\_tags.py::test\_get\_tag\_not\_found PASSED [ 37%]

tests/test\_route\_tags.py::test\_get\_tags PASSED [ 50%]

tests/test\_route\_tags.py::test\_update\_tag PASSED [ 62%]

tests/test\_route\_tags.py::test\_update\_tag\_not\_found PASSED [ 75%]

tests/test\_route\_tags.py::test\_delete\_tag PASSED [ 87%]

tests/test\_route\_tags.py::test\_repeat\_delete\_tag PASSED [100%]

================================ 8 passed in 4.21s ================================

**Висновок**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-14/module-14-01/write_test#%D0%B2%D0%B8%D1%81%D0%BD%D0%BE%D0%B2%D0%BE%D0%BA)

Ми провели тестування маршрутів нашого REST API, пов'язаних з авторизацією і тегами. Усі методи працюють коректно, а також, у разі помилок, REST API повертають правильні HTTP-статуси і повідомлення про помилки. Ми використовували мок-об'єкти для заміни залежностей всередині коду тестів, таких як сервіс аутентифікації.

ЗАВДАННЯ

Тепер спробуйте самостійно доповнити і розширити тести, наприклад, додавши додаткові перевірки на коректність даних, що повертаються, а також покрити тестами роути users.py і notes.py. Можна також додати модульні тести на інші модулі репозиторію.

Ви також можете [**завантажити повний приклад коду**](https://github.com/GoIT-Python-Web/rest-api-tests)**(https://github.com/GoIT-Python-Web/rest-api-tests)** цього модуля для запуску на локальній машині.