**Модуль 13: Розширені теми Beckend-розробки**

**Заняття 1: Робота з листами та кешування**

**Розсилка та надсилання електронних листів**

Розсилка та надсилання листів – це дві різні функції, які можуть бути використані для комунікації з клієнтами або одержувачами електронної пошти.

Загалом, розсилка призначена для масової комунікації, а надсилання листів – для індивідуальної.

**Розсилка електронних листів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/intro#%D1%80%D0%BE%D0%B7%D1%81%D0%B8%D0%BB%D0%BA%D0%B0-%D0%B5%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%B8%D1%85-%D0%BB%D0%B8%D1%81%D1%82%D1%96%D0%B2)Розсилка електронних листів – це процес масового надсилання електронних листів кільком або, навіть, безлічі одержувачів. Ії можна використовувати для різних цілей, наприклад, для надсилання новинних бюлетенів, маркетингових кампаній, інформаційних повідомлень тощо.

Механізм розсилки електронних листів може бути реалізований за допомогою різних інструментів і методів. Одним із найпоширеніших методів є використання спеціалізованих сервісів для розсилки електронних листів, таких як [Mailchimp](https://mailchimp.com/) (https://mailchimp.com/), [SendGrid](https://sendgrid.com/) (https://sendgrid.com/en-us) або інший популярний сервіс. Ці сервіси надають інтерфейс для створення та надсилання розсилок, а також безліч інструментів для аналітики та оптимізації розсилок.

Під час розсилок листів можуть виникнути деякі ситуації, про які необхідно знати розробнику-початківцю:

* **BOUNCE** (відмова) – це стан, коли лист не може бути доставлено кінцевому одержувачу через неправильну адресу електронної пошти або з іншої причини.
* **SPAM** (спам) – це стан, коли лист вважається небажаним або незаконним, і може бути автоматично видалений або поміщений у папку «Спам».
* **DELIVERY**(доставка) – це стан, коли лист було успішно доставлено кінцевому одержувачу.

Важливо відзначити, що кожен з цих станів може бути індивідуальним для кожного листа і може залежати від багатьох факторів, таких як якість списку одержувачів, вміст повідомлення, налаштування антиспам-фільтрів тощо. Таким чином, для успішного розсилання електронних листів необхідно забезпечити якісний список одержувачів, створити якісний вміст та відстежувати результати розсилки, щоб оптимізувати та доопрацьовувати її. Інакше, якщо під час розсилки буде багато станів BOUNCE і SPAM, сервіс просто припинить розсилки для вас.

Однак, є й інші методи розсилання електронних листів, наприклад, розсилання через SMTP-сервер або за допомогою кастомного коду, написаного на Python або іншою мовою. Але в цьому випадку ймовірність попадання вашого листа до папки «Спам» користувача різко зростає.

**Надсилання листів**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/intro#%D0%BD%D0%B0%D0%B4%D1%81%D0%B8%D0%BB%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BB%D0%B8%D1%81%D1%82%D1%96%D0%B2)Надсилання листів – це процес надсилання одного або декількох листів конкретним одержувачам. Це може містити особисті повідомлення, професійні контакти тощо.

Для надсилання електронного листа за допомогою Python можна використовувати вбудовану бібліотеку smtplib. Ця бібліотека містить клас SMTP, який дозволяє встановити з’єднання з SMTP-сервером та надіслати електронний лист. Однак, краще використовувати інші бібліотеки для надсилання електронної пошти, наприклад, yagmail, flanker або envelopes. Усі вони надають зручний інтерфейс для надсилання електронної пошти та можуть бути використані для вирішення різних завдань.

Найчастіші завдання, пов’язані з надсиланням листів на Backend, наступні:

* Реєстрація користувача — надсилати користувачеві лист із підтвердженням реєстрації та посиланням для активації облікового запису;
* Скидання паролю — надсилати користувачеві лист із посиланням для скидання паролю;
* Повідомлення про нові події — надсилати користувачеві електронного листа з повідомленням про нові події або оновлення;
* Повідомлення про замовлення — надсилати користувачеві електронний лист із повідомленням про статус його замовлення, включно з доставкою та оплатою;
* Операції OTP Email;

INFO

OTP Email, або One-Time Password via Email, – це спосіб підтвердження ідентифікації користувача шляхом надсилання йому унікального коду у вигляді цифрової послідовності (OTP) на його електронну пошту. Користувач повинен ввести отриманий код у форму вебсайту або застосунку, щоб підтвердити свою аутентифікацію. OTP Mail є простим і зручним способом підвищити безпеку процесу аутентифікації, оскільки код можна використовувати тільки один раз і термін його дії закінчується через певний час.

**Надсилання електронної пошти за допомогою FastAPI-Mail**

Для нашого REST API ми реалізуємо підтвердження реєстрації користувача. Для надсилання листів будемо використовувати багатофункціональний пакет [Fastapi-mail](https://sabuhish.github.io/fastapi-mail/) (https://sabuhish.github.io/fastapi-mail/).

Для початку визначимося, за допомогою чого ми можемо надіслати лист? Ми можемо виконати інтеграцію зі стороннім поштовим сервісом, таким як Gmail, SendGrid, Mailgun або іншим. Найпростішим способом надсилання було б використання сервісу Gmail, але з часом політики безпеки Google посилюються, і налаштувати надсилання листів стає нетривіальним завданням.

INFO

Починаючи з 30 травня 2022 року, Google більше не підтримує використання сторонніх програм або пристроїв, які просять вас увійти у свій обліковий запис Google, використовуючи тільки своє ім’я користувача та пароль.

**Створення поштової скриньки**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/fastapi-email#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BF%D0%BE%D1%88%D1%82%D0%BE%D0%B2%D0%BE%D1%97-%D1%81%D0%BA%D1%80%D0%B8%D0%BD%D1%8C%D0%BA%D0%B8)Для надсилання листів ми будемо використовувати сервіс [**META**](https://meta.ua/uk/) (https://meta.ua/uk/). Зареєструємо там поштову скриньку і вкажемо можливість використання нашого облікового запису сторонніми програмами відповідно до цієї [**інструкції**](https://help.meta.ua/ua/mail/mail_client/) (https://help.meta.ua/ua/mail/mail\_client/)**.** Потрібно перейти до налаштувань вашої поштової скриньки, знайти пункт «Налаштування POP3 / SMTP сервера» і вибрати в ньому дозвіл доступу за протоколом POP3/SMTP (будь-який варіант). Після цього у вас повинна бути пошта зі своїм ім’ям, яке необхідно буде підставляти в прикладах. Далі в конспекті ми використовуватимемо вигаданий email "example@meta.ua" та вигаданий пароль "secretPassword".

NOTE

Можна використовувати будь-який сервіс для надсилання листів. Описаний спосіб – найпростіший для реалізації та розуміння механізму надсилання листів.

**Код надсилання листа​**

Тепер ми можемо розібрати механізм надсилання листа за допомогою пакета Fastapi-mail. Не забудьте його встановити.

**main.py**

from pathlib import Path

import uvicorn

from fastapi import FastAPI, BackgroundTasks

from fastapi\_mail import FastMail, MessageSchema, ConnectionConfig, MessageType

from pydantic import EmailStr, BaseModel

from typing import List

class EmailSchema(BaseModel):

email: EmailStr

conf = ConnectionConfig(

MAIL\_USERNAME="example@meta.ua",

MAIL\_PASSWORD="secretPassword",

MAIL\_FROM="example@meta.ua",

MAIL\_PORT=465,

MAIL\_SERVER="smtp.meta.ua",

MAIL\_FROM\_NAME="Example email",

MAIL\_STARTTLS=False,

MAIL\_SSL\_TLS=True,

USE\_CREDENTIALS=True,

VALIDATE\_CERTS=True,

TEMPLATE\_FOLDER=Path(\_\_file\_\_).parent / 'templates',

)

app = FastAPI()

@app.post("/send-email")

async def send\_in\_background(background\_tasks: BackgroundTasks, body: EmailSchema):

message = MessageSchema(

subject="Fastapi mail module",

recipients=[body.email],

template\_body={"fullname": "Billy Jones"},

subtype=MessageType.html

)

fm = FastMail(conf)

background\_tasks.add\_task(fm.send\_message, message, template\_name="example\_email.html")

return {"message": "email has been sent"}

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

uvicorn.run('main:app', port=8000, reload=True)

Цей код – повнофункціональний застосунок для надсилання листа користувачеві. Маршрут /send-email приймає адресу електронної пошти у вигляді корисного навантаження JSON і надсилає електронні листи як фонове завдання.

Клас EmailSchema визначає очікувану структуру корисного навантаження для надсилання електронної пошти.

Клас FastMail ініціалізується об’єктом ConnectionConfig, який визначає конфігурацію для підключення до сервера електронної пошти, наприклад, адресу сервера MAIL\_SERVER="smtp.meta.ua", порт MAIL\_PORT=465 та інформацію про аутентифікацію MAIL\_USERNAME="example@meta.ua", MAIL\_PASSWORD="secretPassword".

Функція send\_in\_background приймає об’єкт BackgroundTasks та тіло запиту body з електронною поштою. Після цього створює об’єкт повідомлення message із зазначеною темою subject="Fastapi mail module", одержувачем recipients=[body.email] і шаблоном тіла template\_body={"fullname": "Billy Jones"}.

INFO

Тема електронної пошти – це короткий текст, який є коротким викладом основного змісту листа і, зазвичай, відображається в поштовому клієнті одержувача разом з ім’ям відправника та датою отримання листа. Тема листа – важливий аспект електронного повідомлення, оскільки вона допомагає одержувачу визначити релевантність листа та вирішити, відкривати його чи ні.

А поштова служба часто за темою листа може відправити лист просто в спам.

Потім об’єкт FastMail використовується для надсилання повідомлення, при цьому файл шаблону example\_email.html знаходиться в папці templates. Папку, де зберігатимуться шаблони Jinja2 для листів, ми вказали через властивість конфігурації TEMPLATE\_FOLDER=Path(\_\_file\_\_).parent / 'templates', об’єкта ConnectionConfig.

Сам файл шаблону email\_template.html досить простий:

**/templates/example\_email.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Hello</title>

</head>

<body>

<p>Hi {{fullname}},</p>

<p>We invite you to visit our beautiful site</p>

<p>

<a href="https://localhost:8000">

Click here

</a>

</p>

<p>Thanks,</p>

<p>The Our Team</p>

</body>

</html>

Зверніть увагу, що всередині тегу <p>Hi {{fullname}},</p> ми підставляємо дані з тіла шаблону template\_body={"fullname": "Billy Jones"}. Так виконується передача необхідних даних всередину шаблону листа.

Саме надсилання листа здійснюється за допомогою класу BackgroundTasks наступним рядком коду background\_tasks.add\_task(fm.send\_message, message, template\_name="email\_template.html"). Фактично ми визначаємо надсилання email як фонове завдання.

**Клас BackgroundTasks**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/fastapi-email#%D0%BA%D0%BB%D0%B0%D1%81-backgroundtasks)

Що таке клас [**BackgroundTasks**](https://fastapi.tiangolo.com/tutorial/background-tasks/) (<https://fastapi.tiangolo.com/tutorial/> background-tasks/)? Клас BackgroundTasks дозволяє виконувати тривалі завдання, такі як надсилання електронної пошти або обробка зображень, у фоновому режимі, щоб ваш API міг продовжувати обробляти інші запити. Це особливо корисно для завдань, виконання яких може зайняти багато часу і, в іншому разі, API не зможе відповідати на інші запити.

У своєму коді ви можете використовувати BackgroundTasks, включивши його як параметр у функцію обробника маршруту і додавши потім до неї завдання за допомогою методу add\_task. Завдання виконуватиметься у фоновому режимі, що дозволить вам негайно повернути відповідь клієнту, не чекаючи завершення завдання. Це підвищує загальну продуктивність вашого API та запобігає блокуванню іншими запитами завдань, що довго виконуються.

Функція повертає відповідь у форматі JSON, яка вказує на те, що лист був надісланий. Хоча саме надсилання може виконуватися ще й у фоновому режимі.

Зрештою, останній блок запускає застосунок, використовуючи бібліотеку uvicorn:

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

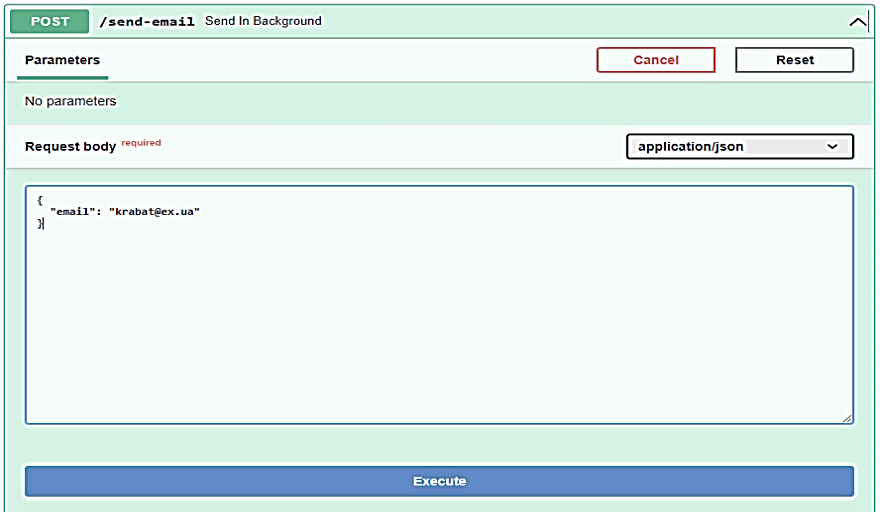
uvicorn.run('main:app', port=8000, reload=True)

Аргументи методу run:

* main:app вказує uvicorn, модуль – це файл main.py та об’єкт застосунку всередині коду app, який необхідно запустити;
* port=8000 визначає номер порту для прослуховування;
* reload=True дає змогу автоматично перезавантажувати застосунок у разі зміни коду;

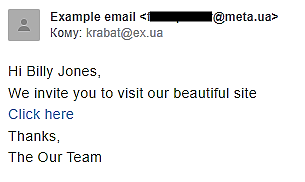
**Перевірка працездатності**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/fastapi-email#%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D1%96%D1%80%D0%BA%D0%B0-%D0%BF%D1%80%D0%B0%D1%86%D0%B5%D0%B7%D0%B4%D0%B0%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%96)Запустимо приклад і перейдемо за адресою документації Swagger <http://127.0.0.1:8000/docs>



**Надсилання листа**

Виконаємо надсилання листа на наявну адресу. Спочатку ми відразу отримаємо відповідь від сервера {"message": "email has been sent"}, а через деякий час побачимо лист у нашій поштовій скриньці.



**Результат надсилання листа**

Готовий результат повинен бути як у цьому живому прикладі нижче, запустіть і подивіться код. Надішліть лист на свій email, але, будь ласка, не зловживайте, оскільки сервіс надсилання має обмеження 200 листів на день.

Щоб побачити документацію Swagger нашого застосунку, після запуску прикладу натисніть [посилання](https://fastapi-mail-simple.krabaton.repl.co/docs) (https://replit.com/@Krabaton/FastAPI-Mail-simple). Спробуйте надіслати собі тестовий лист.

Тепер, коли ми зрозуміли механізм надсилання листа, реалізуймо підтвердження реєстрації користувача для нашого REST API.

**Верифікація email користувача**

Алгоритм перевірки електронної пошти – це процес, який використовується для перевірки приналежності адреси електронної пошти користувачеві. Він реалізується шляхом надсилання на цю адресу перевірчого електронного листа. Процес перевірки електронної пошти містить наступні кроки:

1. Користувач вказує адресу електронної пошти під час реєстрації або створення облікового запису.
2. Система генерує унікальний маркер перевірки та надсилає його на вказану адресу електронної пошти.
3. Користувач натискає на посилання перевірки в електронному листі, яке переводить його на сторінку підтвердження на вебсайті.
4. Сайт перевіряє, чи дійсний маркер перевірки, зазначений в URL-адресі, і чи відповідає він адресі електронної пошти.
5. Якщо перевірчий токен дійсний, сайт відзначає адресу електронної пошти як перевірену, і користувачеві надається повний доступ до функцій сайту.
6. Якщо перевірчий маркер недійсний, користувачеві пропонується повторити спробу або запросити ще один перевірчий email.

Цей процес допомагає гарантувати, що тільки власник адреси електронної пошти може отримати доступ до сайту та його функцій, запобігаючи створенню зловмисниками підроблених облікових записів і доступу до конфіденційної інформації.

**Змінюємо моделі та репозиторій**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/restapi-email#%D0%B7%D0%BC%D1%96%D0%BD%D1%8E%D1%94%D0%BC%D0%BE-%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%96-%D1%82%D0%B0-%D1%80%D0%B5%D0%BF%D0%BE%D0%B7%D0%B8%D1%82%D0%BE%D1%80%D1%96%D0%B9)

В модель User необхідно додати поле confirmed = Column(Boolean, default=False). Це логічний вираз, що визначає, чи був підтверджений email користувача. За замовчуванням значення поля дорівнює False; після того, як користувач підтвердить свою електронну пошту, його варто встановити в True.

INFO

Не забудьте виконати міграції для зміни таблиці користувача.

У репозиторії для роботи з даними користувача src/repository/users.py додамо наступну функцію.

async def confirmed\_email(email: str, db: Session) -> None:

user = await get\_user\_by\_email(email, db)

user.confirmed = True

db.commit()

Мета цієї функції – встановити атрибут confirmed користувача в значення True у базі даних.

**Сервіс надсилання листів для верифікації**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/restapi-email#%D1%81%D0%B5%D1%80%D0%B2%D1%96%D1%81-%D0%BD%D0%B0%D0%B4%D1%81%D0%B8%D0%BB%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BB%D0%B8%D1%81%D1%82%D1%96%D0%B2-%D0%B4%D0%BB%D1%8F-%D0%B2%D0%B5%D1%80%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%97)Тепер реалізуємо сервіс надсилання електронного листа користувачеві та помістимо його у файл src/services/email.py.

**src/services/email.py**

from pathlib import Path

from fastapi\_mail import FastMail, MessageSchema, ConnectionConfig, MessageType

from fastapi\_mail.errors import ConnectionErrors

from pydantic import EmailStr

from src.services.auth import auth\_service

conf = ConnectionConfig(

MAIL\_USERNAME="example@meta.ua",

MAIL\_PASSWORD="secretPassword",

MAIL\_FROM=EmailStr("example@meta.ua"),

MAIL\_PORT=465,

MAIL\_SERVER="smtp.meta.ua",

MAIL\_FROM\_NAME="Desired Name",

MAIL\_STARTTLS=False,

MAIL\_SSL\_TLS=True,

USE\_CREDENTIALS=True,

VALIDATE\_CERTS=True,

TEMPLATE\_FOLDER=Path(\_\_file\_\_).parent / 'templates',

)

async def send\_email(email: EmailStr, username: str, host: str):

try:

token\_verification = auth\_service.create\_email\_token({"sub": email})

message = MessageSchema(

subject="Confirm your email ",

recipients=[email],

template\_body={"host": host, "username": username, "token": token\_verification},

subtype=MessageType.html

)

fm = FastMail(conf)

await fm.send\_message(message, template\_name="email\_template.html")

except ConnectionErrors as err:

print(err)

В принципі, після попереднього розділу цей код не потребує особливих пояснень. Функція send\_email приймає три аргументи – адресу електронної пошти одержувача, ім’я користувача і хост, де працює наш застосунок.

Насамперед створюємо токен JWT для верифікації електронної пошти за допомогою методу auth\_service.create\_email\_token. Потім цей токен передається в шаблон електронної пошти як змінна за допомогою параметра template\_body.

Далі створюємо об’єкт MessageSchema із зазначеними темою, одержувачами та тілом шаблону, а потім передаємо його об’єкту FastMail. Наприкінці викликаємо метод send\_message, що передає повідомлення та ім’я шаблону email\_template.html. Сам шаблон знаходиться в папці templates.

Всередині сервісу Auth реалізуємо метод create\_email\_token:

**src/services/auth.py**

def create\_email\_token(self, data: dict):

to\_encode = data.copy()

expire = datetime.utcnow() + timedelta(days=7)

to\_encode.update({"iat": datetime.utcnow(), "exp": expire})

token = jwt.encode(to\_encode, self.SECRET\_KEY, algorithm=self.ALGORITHM)

return token

У результаті виклику функції auth\_service.create\_email\_token повертається закодований токен, що дійсний 7 днів. Саме стільки ми чекатимемо від користувача верифікації. А, оскільки під час створення в полі sub ми помістили email, він повністю персоналізований для користувача.

Реалізуємо шаблон листа у файлі src/services/templates/email\_template.html.

**src/services/templates/email\_template.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Password Reset</title>

</head>

<body>

<p>Hi {{username}},</p>

<p>Thank you for signing up for our service.</p>

<p>Please click the following link to verify your email address:</p>

<p>

<a href="{{host}}api/auth/confirmed\_email/{{token}}">

Verification

</a>

</p>

<p>If you did not sign up for our service, please ignore this email.</p>

<p>Thanks,</p>

<p>The Our Team</p>

</body>

</html>

Це Jinja2-шаблон електронного листа для перевірки адреси електронної пошти нового користувача, який зареєструвався в нашому застосунку. Лист вітає одержувача за ім'ям та дякує йому за реєстрацію. Потім він надає посилання для перевірки адреси електронної пошти, яка вказує на маршрут до REST API {{host}}api/auth/confirmed\_email/{{token}}. У листі також згадується, що якщо одержувач не підписувався на послугу, він може проігнорувати лист. Натиснувши на посилання у листі, користувач переходить за вказаним маршрутом і підтверджує свій email. Це ми реалізуємо трохи згодом.

У разі виникнення помилок під час встановлення з’єднання перехоплюється виняток ConnectionErrors, і повідомлення про помилку виводиться на консоль.

**Змінюємо роботу маршрутів**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/restapi-email#%D0%B7%D0%BC%D1%96%D0%BD%D1%8E%D1%94%D0%BC%D0%BE-%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D1%83-%D0%BC%D0%B0%D1%80%D1%88%D1%80%D1%83%D1%82%D1%96%D0%B2)

**Реєстрація користувача**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/restapi-email#%D1%80%D0%B5%D1%94%D1%81%D1%82%D1%80%D0%B0%D1%86%D1%96%D1%8F-%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%87%D0%B0)

**src/routes/auth.py**

from fastapi import APIRouter, HTTPException, Depends, status, Security, BackgroundTasks, Request

...

from src.services.email import send\_email

...

@router.post("/signup", response\_model=UserResponse, status\_code=status.HTTP\_201\_CREATED)

async def signup(body: UserModel, background\_tasks: BackgroundTasks, request: Request, db: Session = Depends(get\_db)):

exist\_user = await repository\_users.get\_user\_by\_email(body.email, db)

if exist\_user:

raise HTTPException(status\_code=status.HTTP\_409\_CONFLICT, detail="Account already exists")

body.password = auth\_service.get\_password\_hash(body.password)

new\_user = await repository\_users.create\_user(body, db)

background\_tasks.add\_task(send\_email, new\_user.email, new\_user.username, request.base\_url)

return {"user": new\_user, "detail": "User successfully created. Check your email for confirmation."}

В обробці маршруту реєстрації користувача "/signup" з’явився новий рядок коду: background\_tasks.add\_task(send\_email, new\_user.email, new\_user.username, request.base\_url). Створюємо фонове завдання надсилання листа за допомогою функції background\_tasks.add\_task. Функція send\_email – це асинхронна функція, яка надсилає лист на адресу, вказану в new\_user.email. Також ми передаємо параметр request.base\_url. Це атрибут об’єкта запиту Request, який є базовим URL запиту. Базовий URL включає схему (наприклад, http або https), ім’я хосту та порт, але не включає шлях або рядок запиту. У нашому випадку повний URL запиту буде http://127.0.0.1:8000/api/auth/signup, тоді request.base\_url буде містити http://127.0.0.1:8000/. Цей атрибут ми використовуємо для генерації URL у шаблоні електронної пошти.

**Аутентифікація користувача**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/restapi-email#%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%8F-%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%87%D0%B0)

**src/routes/auth.py**

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

async def login(body: OAuth2PasswordRequestForm = Depends(), db: Session = Depends(get\_db)):

user = await repository\_users.get\_user\_by\_email(body.username, db)

if user is None:

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Invalid email")

if not user.confirmed:

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Email not confirmed")

if not auth\_service.verify\_password(body.password, user.password):

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Invalid password")

# Generate JWT

access\_token = await auth\_service.create\_access\_token(data={"sub": user.email})

refresh\_token = await auth\_service.create\_refresh\_token(data={"sub": user.email})

await repository\_users.update\_token(user, refresh\_token, db)

return {"access\_token": access\_token, "refresh\_token": refresh\_token, "token\_type": "bearer"}

У цьому маршруті тепер потрібно додати додаткову перевірку, чи є email користувача верифікованим? Ми перевіряємо властивість user.confirmed, і якщо email не підтверджений – генеруємо виняток з описом "Email not confirmed".

**Підтвердження email**

[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/restapi-email#%D0%BF%D1%96%D0%B4%D1%82%D0%B2%D0%B5%D1%80%D0%B4%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-email)**src/routes/auth.py**

@router.get('/confirmed\_email/{token}')

async def confirmed\_email(token: str, db: Session = Depends(get\_db)):

email = await auth\_service.get\_email\_from\_token(token)

user = await repository\_users.get\_user\_by\_email(email, db)

if user is None:

raise HTTPException(status\_code=status.HTTP\_400\_BAD\_REQUEST, detail="Verification error")

if user.confirmed:

return {"message": "Your email is already confirmed"}

await repository\_users.confirmed\_email(email, db)

return {"message": "Email confirmed"}

Додамо новий маршрут /confirm email/{token} для реалізації підтвердження електронної пошти. Маршрут визначає операцію GET і приймає токен як параметр шляху.

Функція confirmed\_email викликає метод auth\_service.get\_email\_from\_token з параметром token, який ми розмістили в шаблоні листа для верифікації користувача. Ми отримуємо електронну пошту користувача з токена. Потім викликаємо метод repository\_users.get\_user\_by\_email з отриманим email для отримання користувача з бази даних. Якщо користувача з отриманим email немає у базі даних, що вже підозріло, ми генеруємо помилку 400 Bad request. Потім перевіряємо ситуацію, яким чином електронну пошту користувача вже підтверджено. Тоді повертаємо відповідь у форматі JSON із повідомленням "Your email is already confirmed". Якщо email не підтверджено, функція викличе repository\_users.confirmed\_email для оновлення статусу email у базі даних і поверне відповідь у форматі JSON: {"message": "Email confirmed"}.

Додамо реалізацію методу auth\_service.get\_email\_from\_token:

**src/services/auth.py**

async def get\_email\_from\_token(self, token: str):

try:

payload = jwt.decode(token, self.SECRET\_KEY, algorithms=[self.ALGORITHM])

email = payload["sub"]

return email

except JWTError as e:

print(e)

raise HTTPException(status\_code=status.HTTP\_422\_UNPROCESSABLE\_ENTITY,

detail="Invalid token for email verification")

**Повторне надсилання листа**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/restapi-email#%D0%BF%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%BD%D0%B5-%D0%BD%D0%B0%D0%B4%D1%81%D0%B8%D0%BB%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BB%D0%B8%D1%81%D1%82%D0%B0)Залишилося реалізувати останній, шостий пункт алгоритму верифікації email користувача – повторне надсилання листа користувачеві.

**src/routes/auth.py**

...

from src.schemas import UserModel, UserResponse, TokenModel, RequestEmail

...

@router.post('/request\_email')

async def request\_email(body: RequestEmail, background\_tasks: BackgroundTasks, request: Request,

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

user = await repository\_users.get\_user\_by\_email(body.email, db)

if user.confirmed:

return {"message": "Your email is already confirmed"}

if user:

background\_tasks.add\_task(send\_email, user.email, user.username, request.base\_url)

return {"message": "Check your email for confirmation."}

Функція-обробник request\_email реалізує операцію POST для запиту повторної перевірки електронної пошти. Вона приймає тіло JSON з адресою електронної пошти. Тому потрібно визначити модель RequestEmail у файлі src/schemas.py.

**src/schemas.py**

class RequestEmail(BaseModel):

email: EmailStr

Робота функції request\_email починається з отримання користувача з бази даних із зазначеною адресою електронної пошти. Якщо користувач існує, перевіряється статус підтвердження електронної пошти. Якщо електронну пошту вже підтверджено, функція повертає відповідь, яка вказує, що електронну пошту вже підтверджено і повторна верифікація не потрібна.

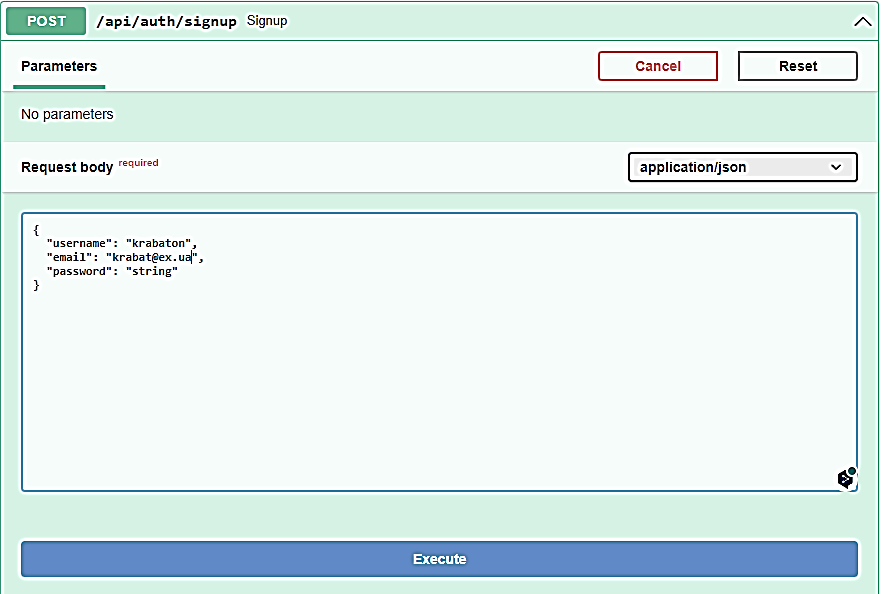
Якщо верифікація необхідна, то функція додає завдання у фоновий менеджер завдань BackgroundTasks для надсилання користувачеві електронного листа з посиланням для підтвердження. Нарешті, функція повертає відповідь, що вказує на те, що електронний лист надіслано для підтвердження.

Зверніть увагу, що листа надсилаємо, тільки якщо користувач існує в базі даних, а відповідь про успішне надсилання даємо завжди. Це не помилка, так ми страхуємося від випадку, якщо зловмисник вирішив перевірити, чи існує в системі незавершена реєстрація для електронного листа.

**Перевірка працездатності**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/restapi-email#%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D1%96%D1%80%D0%BA%D0%B0-%D0%BF%D1%80%D0%B0%D1%86%D0%B5%D0%B7%D0%B4%D0%B0%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%96)

Після додавання необхідного функціоналу перевірмо працездатність отриманого процесу.

1. Зареєструємо нового користувача:



**Реєстрація**

Відповідь:

{

"user": {

"id": 11,

"username": "krabaton",

"email": "krabat@ex.ua",

"created\_at": "2023-02-12T11:25:39.443947",

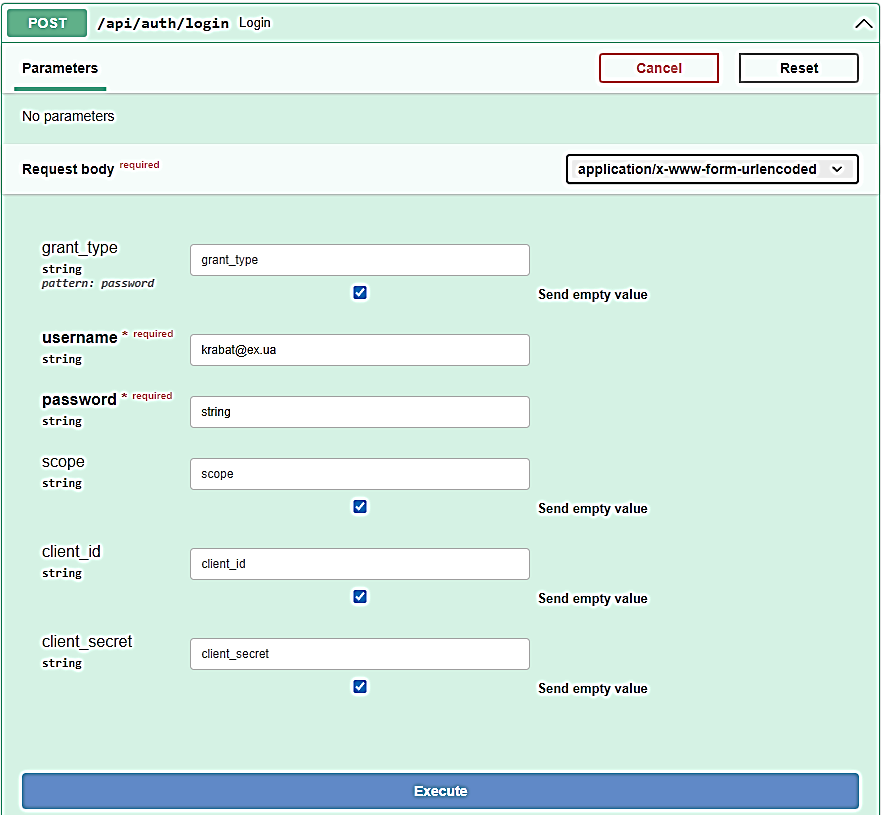
"avatar": "https://www.gravatar.com/avatar/b0005e4ae62475e666fc92fa35c3701a"

},

"detail": "User successfully created. Check your email for confirmation."

}

2.Спробуємо аутентифікуватися без підтвердження електронної пошти



**Логін без верифікації**

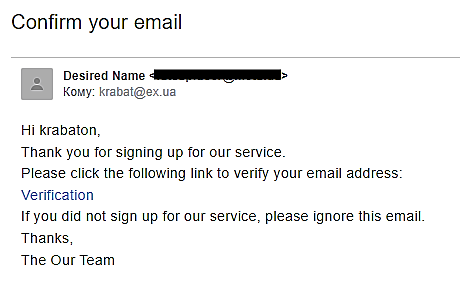
Відповідь:

{

"detail": "Email not confirmed"

}

3.Заходимо на пошту, зазначену під час реєстрації, знаходимо електронний лист і натискаємо на посилання верифікації:



**Лист верифікації**

Нам необхідно потрапити до браузера з таким вмістом:

{

"message": "Email confirmed"

}

Все, процес верифікації email користувача завершено.

4.Виконуємо аутентифікацію знову, але цього разу відповідь уже буде успішною.

{

"access\_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJrcmFiYXRAZXgudWEiLCJpYXQiOjE2NzYyMDE4NjgsImV4cCI6MTY3NjIxMDg2OCwic2NvcGUiOiJhY2Nlc3NfdG9rZW4ifQ.rU\_jh1FVtwIhieYn61VaJ6\_xYwDh2O9hYxyu13SJZCU",

"refresh\_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJrcmFiYXRAZXgudWEiLCJpYXQiOjE2NzYyMDE4NjgsImV4cCI6MTY3NjgwNjY2OCwic2NvcGUiOiJyZWZyZXNoX3Rva2VuIn0.RQ0opWXk6Teu07TNbRuJGpuqwiYyeMAILW1SRkS7Vdk",

"token\_type": "bearer"

}

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

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

Наступним кроком розберемо процес скидання паролю за допомогою email на прикладі проекту Django.

**Механізм скидання пароля**

Іншим частим використанням надсилання листів є операція скидання пароля, якщо користувач його забув. Цей сценарій ми розглянемо на прикладі фреймворку Django для проекту з модуля **"Модуль 10: Основи роботи з Django".**

Оскільки Django має готові представлення, які дають змогу реалізувати механізм скидання пароля для користувача, якщо він його забуде. Подивімось, як ми можемо додати цю функціональність до нашого застосунку.

**Робочий процес**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/django-email#%D1%80%D0%BE%D0%B1%D0%BE%D1%87%D0%B8%D0%B9-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81)Спочатку розберемо процес відновлення пароля:

1. Користувач запитує скидання пароля, вказуючи свою адресу електронної пошти у формі скидання пароля.
2. Django генерує унікальний токен скидання пароля і надсилає лист із посиланням на скидання пароля на адресу електронної пошти користувача. Посилання вмикає маркер скидання пароля як параметр.
3. Коли користувач натискає на посилання скидання пароля, Django перевіряє маркер скидання пароля, і з’являється форма для введення нового пароля.
4. Користувач вводить новий пароль та надсилає форму.
5. Django оновлює пароль користувача та повідомляє про це користувача.

**Налаштовуємо параметри електронної пошти**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/django-email#%D0%BD%D0%B0%D0%BB%D0%B0%D1%88%D1%82%D0%BE%D0%B2%D1%83%D1%94%D0%BC%D0%BE-%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8-%D0%B5%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D0%BE%D1%97-%D0%BF%D0%BE%D1%88%D1%82%D0%B8)Щоб надсилати листи у Django, ми повинні додати всередину файлу налаштувань settings.py нашого застосунку наступні константи:

**notes/notes/settings.py**

EMAIL\_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

EMAIL\_HOST = 'smtp.meta.ua'

EMAIL\_PORT = 465

EMAIL\_STARTTLS = False

EMAIL\_USE\_SSL = True

EMAIL\_USE\_TLS = False

EMAIL\_HOST\_USER = 'example@meta.ua'

EMAIL\_HOST\_PASSWORD = 'secretPassword'

DEFAULT\_FROM\_EMAIL = EMAIL\_HOST\_USER

Розберімо, що означають ці параметри:

* EMAIL\_BACKEND — backends, який буде використовуватися для надсилання електронної пошти. У цьому випадку використовується backends простого протоколу передачі пошти SMTP: django.core.mail.backends.smtp.EmailBackend;
* EMAIL\_HOST — ім’я хоста SMTP-сервера, який буде використовуватися для надсилання електронної пошти. У нашому випадку встановлено значення smtp.meta.ua;
* EMAIL\_PORT — порт, який буде використовуватися у разі підключення до SMTP-сервера зі встановленим значенням 465;
* EMAIL\_STARTTLS — прапорець, який вказує, чи використовувати безпеку транспортного рівня TLS у разі підключення до SMTP-сервера. Ми будемо використовувати протокол SSL і ставимо значення False;
* EMAIL\_USE\_SSL — прапорець, який вказує, чи варто використовувати SSL у разі підключення до SMTP-сервера. У нашому випадку ми встановлюємо значення True;
* EMAIL\_USE\_TLS — прапорець, який вказує, чи варто використовувати TLS у разі підключення до SMTP-сервера;
* EMAIL\_HOST\_USER — ім’я користувача, що використовується під час аутентифікації на SMTP-сервері;
* EMAIL\_HOST\_PASSWORD — пароль, який буде використовуватися під час аутентифікації на SMTP-сервері;
* DEFAULT\_FROM\_EMAIL — адреса електронної пошти за замовчуванням, яка буде використовуватися як адреса "from" для всіх листів, що надсилаються застосунком. Ми встановлюємо значення EMAIL\_HOST\_USER.

**Під час реєстрації додаємо поле email**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/django-email#%D0%BF%D1%96%D0%B4-%D1%87%D0%B0%D1%81-%D1%80%D0%B5%D1%94%D1%81%D1%82%D1%80%D0%B0%D1%86%D1%96%D1%97-%D0%B4%D0%BE%D0%B4%D0%B0%D1%94%D0%BC%D0%BE-%D0%BF%D0%BE%D0%BB%D0%B5-email)

Під час реєстрації нового користувача необхідно додати поле email:

**notes/users/templates/users/signup.html**

{% extends "noteapp/base.html" %}

{% block content %}

<div>

<h2>Sign Up</h2>

</div>

<form method="post" action="{% url 'users:signup' %}">

{% csrf\_token %}

<div>

<label> Username:

{{ form.username }}

</label>

<span>{{ form.errors.username }}</span>

</div>

<div>

<label> Email:

{{ form.email }}

</label>

<span>{{ form.errors.email }}</span>

</div>

<div>

<label> Password:

{{ form.password1 }}

</label>

<span>{{ form.errors.password1 }}</span>

</div>

<div>

<label> Password confirmation:

{{ form.password2 }}

</label>

<span>{{ form.errors.password2 }}</span>

</div>

<div class="grid">

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

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

</div>

</form>

{% endblock %}

Та змінити клас RegisterForm:

**notes/users/forms.py**

from django import forms

from django.contrib.auth.forms import UserCreationForm, AuthenticationForm

from django.contrib.auth.models import User

from .models import Profile

class RegisterForm(UserCreationForm):

username = forms.CharField(max\_length=100,

required=True,

widget=forms.TextInput())

email = forms.CharField(max\_length=100,

required=True,

widget=forms.TextInput())

password1 = forms.CharField(max\_length=50,

required=True,

widget=forms.PasswordInput())

password2 = forms.CharField(max\_length=50,

required=True,

widget=forms.PasswordInput())

class Meta:

model = User

fields = ['username', 'email', 'password1', 'password2']

class LoginForm(AuthenticationForm):

class Meta:

model = User

fields = ['username', 'password']

class ProfileForm(forms.ModelForm):

avatar = forms.ImageField(widget=forms.FileInput())

class Meta:

model = Profile

fields = ['avatar']

**Додаємо маршрути**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/django-email#%D0%B4%D0%BE%D0%B4%D0%B0%D1%94%D0%BC%D0%BE-%D0%BC%D0%B0%D1%80%D1%88%D1%80%D1%83%D1%82%D0%B8)Додамо наступні маршрути, щоб реалізувати механізм скидання пароля.

**notes/users/urls.py**

from django.contrib.auth.views import PasswordResetDoneView, PasswordResetConfirmView, PasswordResetCompleteView

from django.urls import path

from . import views

app\_name = 'users'

urlpatterns = [

path('signup/', views.signupuser, name='signup'),

path('login/', views.loginuser, name='login'),

path('logout/', views.logoutuser, name='logout'),

path('profile/', views.profile, name='profile'),

path('reset-password/', views.ResetPasswordView.as\_view(), name='password\_reset'),

path('reset-password/done/', PasswordResetDoneView.as\_view(template\_name='users/password\_reset\_done.html'),

name='password\_reset\_done'),

path('reset-password/confirm/<uidb64>/<token>/',

PasswordResetConfirmView.as\_view(template\_name='users/password\_reset\_confirm.html',

success\_url='/users/reset-password/complete/'),

name='password\_reset\_confirm'),

path('reset-password/complete/',

PasswordResetCompleteView.as\_view(template\_name='users/password\_reset\_complete.html'),

name='password\_reset\_complete'),

]

Наприклад, користувач намагається залогінитися, але забув пароль. Користувач натискає на посилання «Forgot Password?» у шаблоні login.html

**notes/users/templates/users/login.htm**l

{% extends "noteapp/base.html" %}

{% block content %}

{% if messages %}

<div>

{% for message in messages %}

<div {% if message.tags %} class="alert-{{ message.tags }}" {% endif %}>{{ message }}</div>

{% endfor %}

</div>

{% endif %}

<form action="{% url 'users:login' %}" method="post">

{% csrf\_token %}

<div>

<label> Username:

{{ form.username }}

</label>

</div>

<div>

<label>Password:

{{ form.password }}

</label>

</div>

<div class="grid">

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

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

</div>

<div class="small text-center">

<a href="{% url "users:password\_reset" %}" class="text-center"><i>Forgot Password?</i></a>

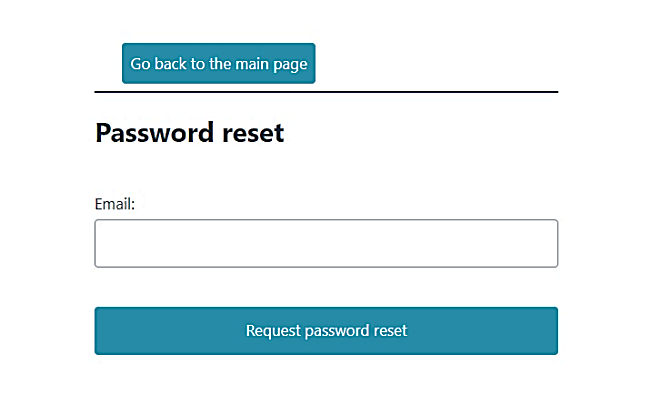
</div>

</form>

{% endblock %}

**Представлення для скидання пароля**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/django-email#%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B4%D0%BB%D1%8F-%D1%81%D0%BA%D0%B8%D0%B4%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BF%D0%B0%D1%80%D0%BE%D0%BB%D1%8F)Це приведе його до маршруту reset-password/ зі сторінкою, де йому буде запропоновано ввести свою адресу електронної пошти. Це представлення ми будемо обробляти шляхом класу ResetPasswordView. Це представлення демонструє форму, в якій користувач надсилає свою адресу електронної пошти.



**Сторінка маршруту /reset-password**

Django перевіряє, існує чи ні користувач із наданою адресою електронної пошти в базі даних. Якщо користувач існує, то генерує посилання для скидання пароля, яке буде використовуватися лише один раз. Після цього він надсилає це посилання на адресу електронної пошти користувача.

Клас ResetPasswordView є класом-представленням для скидання пароля користувача Django. Він наслідується від PasswordResetView з django.contrib.auth.views та SuccessMessageMixin з django.contrib.messages.views.

**notes/users/views.py**

from django.contrib import messages

from django.contrib.auth import authenticate, login, logout

from django.contrib.auth.decorators import login\_required

from django.contrib.auth.views import PasswordResetView

from django.contrib.messages.views import SuccessMessageMixin

from django.shortcuts import render, redirect

from django.urls import reverse\_lazy

...

class ResetPasswordView(SuccessMessageMixin, PasswordResetView):

template\_name = 'users/password\_reset.html'

email\_template\_name = 'users/password\_reset\_email.html'

html\_email\_template\_name = 'users/password\_reset\_email.html'

success\_url = reverse\_lazy('users:password\_reset\_done')

success\_message = "An email with instructions to reset your password has been sent to %(email)s."

subject\_template\_name = 'users/password\_reset\_subject.txt'

Клас SuccessMessageMixin надає можливість показувати повідомлення про успішне виконання дії в користувацькому інтерфейсі. Клас PasswordResetView є класом-представленням, який вже реалізує функціональність скидання пароля.

Наше представлення ResetPasswordView встановлює кілька атрибутів для налаштування процесу скидання пароля:

* template\_name — ім’я шаблону, який буде використовуватися для сторінки скидання пароля;
* email\_template\_name — ім’я шаблону, який буде використовуватися для листа про скидання пароля;
* html\_email\_template\_name — ім’я шаблону, який буде використовуватися для HTML-версії листа про скидання пароля;
* success\_url — URL-адреса для перенаправлення після успішного надсилання листа про скидання пароля;
* success\_message — повідомлення, яке буде показано користувачеві після успішного надсилання листа про скидання пароля;
* subject\_template\_name — ім’я шаблону, який буде використовуватися для теми листа про скидання пароля;

Визначимо шаблон для сторінки скидання пароля.

**notes/users/templates/users/password\_reset.html**

{% extends "noteapp/base.html" %}

{% block content %}

<h2>Password reset</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Request password reset</button>

</form>

{% endblock %}

Визначимо шаблон для HTML-версії листа про скидання пароля

**notes/users/templates/users/password\_reset\_email.html**

{% autoescape off %}

<body>

<p>Hi {{ user.username }},</p>

<p>You requested a password reset for your account. Please follow this link to set a new password:</p>

<p>

<a href="{{ protocol }}://{{ domain }}{% url 'users:password\_reset\_confirm' uidb64=uid token=token %}">

{{ protocol }}://{{ domain }}{% url 'users:password\_reset\_confirm' uidb64=uid token=token %}

</a>

</p>

<p>If you didn't request a password reset, please ignore this email.</p>

<p>Thanks,</p>

<p>The Our Team</p>

{% endautoescape %}

Лист містить кілька абзаців тексту, які надають користувачеві інформацію про процес скидання пароля. Посилання на сторінку скидання пароля генерується за допомогою тегу шаблону {% url %} та шаблону URL users:password\_reset\_confirm. Ми також використовуємо наступні змінні в шаблоні:

* user.username — ім’я користувача, який запросив скидання пароля;
* protocol — протокол (наприклад, "http" або "https"), який використовується в URL-адресі сторінки скидання пароля;
* domain — домен вебсайту;
* uid — закодоване в base64 представлення ідентифікатора користувача;
* token — токен скидання пароля, який був згенерований для користувача;

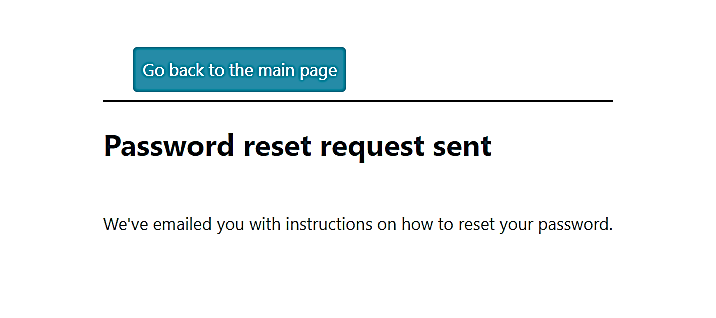
Шаблон для теми листа – це простий текстовий файл:

**notes/users/templates/users/password\_reset\_subject.txt**

Password Reset

**Представлення успішного відправлення листа**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/django-email#%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D1%83%D1%81%D0%BF%D1%96%D1%88%D0%BD%D0%BE%D0%B3%D0%BE-%D0%B2%D1%96%D0%B4%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BB%D0%B8%D1%81%D1%82%D0%B0)

Після надсилання листа Django, відповідно до параметра success\_url = reverse\_lazy('users:password\_reset\_done') класу ResetPasswordView, виконає редірект на маршрут reset-password/done/. Цей маршрут обробляється представленням PasswordResetDoneView.as\_view(template\_name='users/password\_reset\_done.html').



**Сторінка маршруту /reset-password/done/**

Визначаємо шаблон password\_reset\_done.html:

**notes/users/templates/users/password\_reset\_done.html**

{% extends "noteapp/base.html" %}

{% block content %}

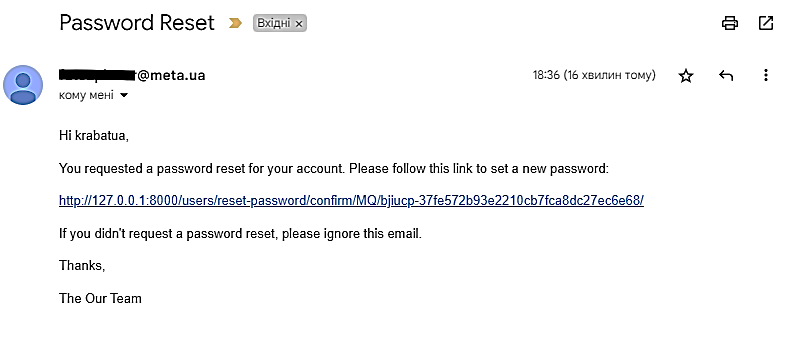
<h2>Password reset request sent</h2>

<p>We've emailed you with instructions on how to reset your password.</p>

{% endblock %}

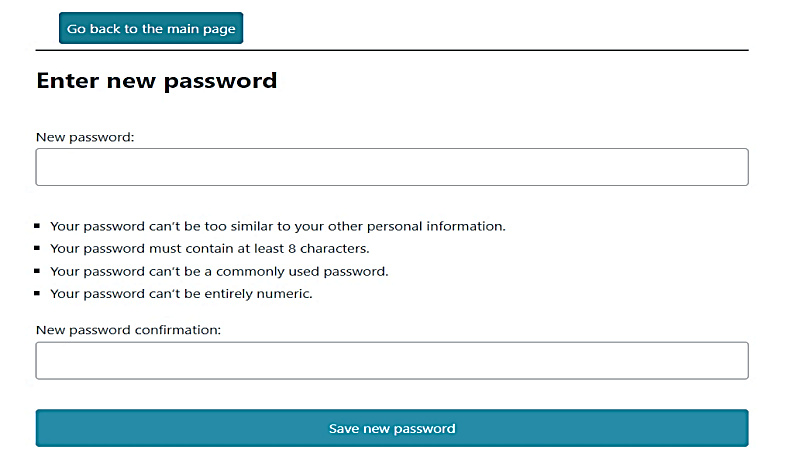
**Скидання пароля**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/django-email#%D1%81%D0%BA%D0%B8%D0%B4%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BF%D0%B0%D1%80%D0%BE%D0%BB%D1%8F)Необхідно зайти до своєї поштової скриньки та знайти лист з інструкцією скидання пароля.



**Лист скидання пароля**

Після натискання на вказане посилання ми перейдемо на маршрут /reset-password/confirm/<uidb64>/<token>/. Де ми повинні створити новий пароль.



**Створення нового пароля**

За обробку цієї форми відповідає представлення PasswordResetConfirmView. Представлення буде використовувати шаблон users/password\_reset\_confirm.html для відображення форми підтвердження скидання пароля і, після успішного підтвердження, перенаправлятиме користувача на URL /users/reset-password/complete/.

Метод as\_view використовується для перетворення представлення на основі класу на об’єкт, що викликається, який ми використовуємо в конфігурації маршруту.

**notes/users/templates/users/password\_reset\_confirm.html**

{% extends "noteapp/base.html" %}

{% block content %}

<h2>Enter new password</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Save new password</button>

</form>

{% endblock %}

**Представлення успішного підтвердження**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/django-email#%D0%BF%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D1%83%D1%81%D0%BF%D1%96%D1%88%D0%BD%D0%BE%D0%B3%D0%BE-%D0%BF%D1%96%D0%B4%D1%82%D0%B2%D0%B5%D1%80%D0%B4%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F)Після того, як ми введемо новий пароль і відправимо його на збереження в базу даних, Django повідомить користувача про це за допомогою класу PasswordResetCompleteView.

...

path('reset-password/complete/',

PasswordResetCompleteView.as\_view(template\_name='users/password\_reset\_complete.html'),

name='password\_reset\_complete'),

...

Цей код створює шаблон URL для сторінки завершення скидання пароля.

Шаблон прив’язується до класу PasswordResetCompleteView із вбудованих представлень аутентифікації Django і використовує шаблон users/password\_reset\_complete.html для відображення сторінки завершення.



**Успішне скидання пароля**

Ось так виглядає шаблон:

**notes/users/templates/users/password\_reset\_complete.html**

{% extends "noteapp/base.html" %}

{% block content %}

<h2>Password reset complete!</h2>

{% endblock %}

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

Щоб побачити наш застосунок, після запуску прикладу, натисніть на [посилання](https://django-project-email.krabaton.repl.co/) (https://replit.com/@Krabaton/Django-project-email)

Ми реалізували скидання пароля користувача для застосунку Django. Для застосунку FastAPI логіка поведінки буде точно така сама, як і для застосунку Django. За необхідності ми можемо самостійно реалізувати верифікацію email користувача в застосунку REST API. Пропонуємо вам виконати це самостійно.

**Кешування вебзастосунків**

Ми вже мали справу з основами кешування, коли розглядали базу даних Redis. Нагадаємо, що кешування — це механізм, який використовується для зберігання даних у пам’яті, щоб уникнути їх повторного витягування із джерела даних на кожен запит.

Правильний підхід до кешування вебзастосунків у Python залежить від умов проекту та інфраструктури. Але загалом варто дотримуватися наступних принципів:

1. Використання кешування для тих частин застосунку, які часто запитуються і нечасто змінюються.
2. Кешування на рівні застосунку або на рівні бази даних, залежно від складності завдання.
3. Використання pub/sub-системи для інвалідації кешу, якщо вимагається динамічне оновлення даних в кеші.
4. Тестування і відлагодження кешування з урахуванням завдання і навантаження на застосунок.
5. Обмеження розміру кешу і видалення старих записів за потреби.

Але загальне правило — використовуйте кешування лише в тому випадку, якщо воно збільшує продуктивність вашого застосунку!

Залежно від вимог до застосунку, ви можете вибрати один із наступних підходів:

* Повний кеш: зберігати всі відповіді вебзастосунку в кеші, щоб уникнути доступу до бази даних. Цей підхід підходить для статичних або мало змінюваних даних.
* Частковий кеш: зберігати лише деякі часто запитувані дані в кеші. Цей підхід є прийнятним для застосунків з високим рівнем звернення до бази даних.
* Кеш з тайм-аутом: зберігати дані в кеші на певний період, після закінчення якого, дані будуть оновлені. Цей підхід є прийнятним для застосунків з даними, що динамічно змінюються.

Кешування може суттєво збільшити продуктивність вебзастосунку, особливо, якщо ваші дані не змінюються часто або, якщо ви маєте велику кількість повторюваних запитів. Однак, важливо пам’ятати, що кешування також може призвести до неактуальних даних, якщо дані змінюються в джерелі даних, тому необхідно правильно налаштувати час життя кешу і механізми оновлення кешу.

**Способи налаштування часу життя кешу**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/cache#%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D0%B8-%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%87%D0%B0%D1%81%D1%83-%D0%B6%D0%B8%D1%82%D1%82%D1%8F-%D0%BA%D0%B5%D1%88%D1%83)

Існує кілька способів налаштувати час життя кешу:

1. Використання тайм-ауту: Це найпростіший спосіб налаштування часу життя кешу. Ви можете встановити час життя кешу в секундах, за якого дані будуть видалені з кешу і знову завантажені з джерела даних. Наприклад, у випадку Redis, можна використовувати метод expire(key, seconds)
2. Використання подій інвалідації: У цьому випадку ви можете встановити події, які повинні відбуватися під час зміни даних у джерелі даних, і використовувати їх для інвалідації відповідних даних у кеші. Наприклад, якщо дані продукту змінилися в базі даних, ви можете надіслати подію інвалідації для кешу цього продукту, щоб видалити його з кешу і знову завантажити свіжі дані з бази даних.

Вибір методу залежить від вашої конкретної ситуації та потреб, але найважливіше — мати механізм для оновлення кешу та забезпечення актуальності даних.

**Встановлення тайм-ауту**

[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/cache#%D0%B2%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%B0%D0%B9%D0%BC-%D0%B0%D1%83%D1%82%D1%83)Методами налаштування часу життя кешу є:

1. Least Recently Used (LRU): Цей метод використовує метадані про час останнього звернення до кожного ключа в кеші. Коли кеш заповнюється, дані, які використовувалися давно і є такими, що найрідше використовуються, видаляються.
2. Time To Live (TTL): Цей метод використовує метадані про час життя кожного ключа в кеші. Кожен ключ має час життя, після якого він видаляється з кешу.

Наприклад, якщо у вас є REST API, який повертає дані продукту з бази даних за ідентифікатором продукту, ви можете кешувати ці дані під час їхнього першого запиту, щоб уникнути повторного звернення до бази даних під час наступних запитів до цього самого продукту.

За допомогою Redis ми можемо використовувати метод set, щоб зберегти дані в кеші, та метод get, щоб витягти дані з кешу.

import redis

import json

import uvicorn

from fastapi import FastAPI

app = FastAPI()

# Connecting to Redis

r = redis.Redis(host='localhost', port=6379, db=0)

database = {

"10": 10,

"11": 11,

"12": 12

}

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

def read\_product(product\_id: int):

product = r.get(str(product\_id))

if product is None:

product = fetch\_product\_from\_db(product\_id)

r.set(str(product\_id), json.dumps(product))

r.expire(str(product\_id), 3600)

return product

return json.loads(product)

def fetch\_product\_from\_db(product\_id: int):

data = database.get(str(product\_id), None)

print(data)

return data

Ми визначили функцію read\_product, яка обробляє маршрут /product/{product\_id}. У функції виконується спроба отримати дані продукту з Redis за допомогою команди r.get(str(product\_id)). Якщо дані не знайдені, викликається функція fetch\_product\_from\_db для витягування даних із внутрішньої бази даних database. Потім дані серіалізуються у формат JSON і зберігаються в Redis за допомогою команди r.set(str(product\_id), json.dumps(product)). Також встановлюється час життя даних в Redis за допомогою команди r.expire(str(product\_id), 3600) на 3600 секунд. При повторному зверненні до продукту з {product\_id} ми вже, напевно, отримаємо його з кешу.

Це лише приклад того, як можна налаштувати час життя кешу, залежно від конкретної ситуації можна використовувати й інші методи. Важливо розуміти, що вибір методу залежить від вашої конкретної ситуації та потреб, наприклад, якщо дані змінюються часто, то метод LRU буде прийнятнішим, а якщо дані рідко змінюються, то прийнятнішим буде метод TTL.

INFO

Для реалізації використання подій інвалідації кешу в попередньому прикладі, можна використовувати механізм [подій та підписки](https://redis-py.readthedocs.io/en/stable/examples/asyncio_examples.html#Pub/Sub-Mode) (https://redis-py.readthedocs.io/en/stable/examples/asyncio\_examples.html#Pub/Sub-Mode) з Redis. Але сам механізм не тривіальний і навряд чи його можна застосувати в наших умовах, але якщо виникне така необхідність, то краще використовувати сторонні пакети. Наприклад, пакет [python-redis-cache](https://pypi.org/project/python-redis-cache/) (https://pypi.org/project/python-redis-cache/) вже реалізований з можливістю інвалідації закешованих значень.

**Кешуємо користувача в нашому застосунку**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/cache#%D0%BA%D0%B5%D1%88%D1%83%D1%94%D0%BC%D0%BE-%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%87%D0%B0-%D0%B2-%D0%BD%D0%B0%D1%88%D0%BE%D0%BC%D1%83-%D0%B7%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BA%D1%83)У нашій функції для авторизації get\_current\_user з класу Auth ми постійно запитуємо користувача з бази даних user = await repository\_users.get\_user\_by\_email(email, db). Це досить «вартісна» операція, причому вона виконується для кожного захищеного маршруту. Перепишемо метод з використанням кешу на Redis.

import pickle

...

class Auth:

pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

SECRET\_KEY = "secret\_key"

ALGORITHM = "HS256"

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="/api/auth/login")

r = redis.Redis(host='localhost', port=6379, db=0)

...

async def get\_current\_user(self, token: str = Depends(oauth2\_scheme), db: Session = Depends(get\_db)):

credentials\_exception = HTTPException(

status\_code=status.HTTP\_401\_UNAUTHORIZED,

detail="Could not validate credentials",

headers={"WWW-Authenticate": "Bearer"},

)

try:

# Decode JWT

payload = jwt.decode(token, self.SECRET\_KEY, algorithms=[self.ALGORITHM])

if payload['scope'] == 'access\_token':

email = payload["sub"]

if email is None:

raise credentials\_exception

else:

raise credentials\_exception

except JWTError as e:

raise credentials\_exception

user = self.r.get(f"user:{email}")

if user is None:

user = await repository\_users.get\_user\_by\_email(email, db)

if user is None:

raise credentials\_exception

self.r.set(f"user:{email}", pickle.dumps(user))

self.r.expire(f"user:{email}", 900)

else:

user = pickle.loads(user)

return user

Як бачимо, ми спочатку перевіряємо користувача в кеші user = self.r.get(f"user:{email}"). Якщо запис відсутній, то ми звертаємося до бази даних user = await repository\_users.get\_user\_by\_email(email, db) та серіалізуємо об’єкт користувача self.r.set(f"user:{email}", pickle.dumps(user)) за допомогою pickle.dumps, а потім зберігаємо серіалізований рядок у кеш self.r.set(f"user:{email}", pickle.dumps(user)) за допомогою методу r.set.

Встановлюємо час зберігання ключа user:{email} в кеші на 900 секунд self.r.expire(f"user:{email}", 900). Якщо користувача знайдено в кеші, ми десеріалізуємо його user = pickle.loads(user) за допомогою pickle.loads, але в будь-якому разі повертаємо з функції get\_current\_user об'єкт User.

INFO

Для довідки тут описано [просунутіший спосіб](просунутіший%20спосіб) (https://developer.redis.com/develop/python/fastapi/) використання Redis у застосунках FastAPI від розробників.

**Використання Docker Compose**

На поточний момент ми в проекті почали використовувати відразу дві бази даних: PostgreSQL та Redis. Які запускаємо в окремих контейнерах. Зі зростанням проекту і збільшенням служб це стає, скажімо так, незручно. Тут нам допоможе такий інструмент, як Docker Compose.

**Docker Compose** — це інструмент, який дозволяє вам визначати та запускати багатоконтейнерні застосунки Docker. Він використовує файл YAML з ім’ям docker-compose.yml для налаштування служб, мереж і томів, що складають застосунок. З Compose ви можете запускати та зупиняти всі служби, визначені у файлі docker-compose.yml, за допомогою однієї команди, а не запускати кілька команд запуску docker.

Ось реалізація файлу docker-compose.yml для нашого застосунку:

**docker-compose.yml**

version: '3'

services:

redis:

image: redis:alpine

ports:

- "6379:6379"

postgres:

image: postgres:12

environment:

POSTGRES\_DB: rest\_app

POSTGRES\_USER: postgres

POSTGRES\_PASSWORD: 567234

ports:

- "5432:5432"

volumes:

- ./postgres-data:/var/lib/postgresql/data

Цей файл docker-compose.yml визначає конфігурацію Docker Compose, яка складається з двох сервісів: redis та postgres.

Служба redis використовує образ Docker redis:alpine, зіставляє внутрішній порт 6379 з портом хосту 6379 і не має певні змінні середовища або томи.

Служба postgres використовує образ Docker postgres:12, зіставляє внутрішній порт 5432 з портом хосту 5432 та має змінні середовища, визначені для імені бази даних rest\_app, імені користувача postgres та паролю 567234. Крім того, для даних postgres визначений том, що відображає локальний каталог ./postgres-data на каталог контейнера /var/lib/postgresql/data.

З такою конфігурацією ми можемо запустити служби Redis та PostgreSQL за допомогою команди docker-compose up -d, зробивши їхні дані доступними в зазначених портах на вашій локальній машині.

Ось найпоширеніші команди для роботи з файлом docker-compose.yml:

* docker-compose up – запуск служб, визначених у файлі docker-compose.yml. Ця команда створить і запустить контейнери для кожного сервісу, а також створить мережу для взаємодії контейнерів.
* docker-compose down – зупинити і видалити контейнери, мережі і томи, створені командою docker-compose up.
* docker-compose ps – виведе список контейнерів, створених docker-compose up, а також їхній статус та іншу інформацію.
* docker-compose logs – перегляд логів контейнерів, створених docker-compose up.
* docker-compose exec <ім’я служби> <команда> – виконати команду в запущеному контейнері, створеному docker-compose up. Наприклад, docker-compose exec redis redis-cli запустить команду redis-cli в контейнері redis.
* docker-compose build – перебудувати образи, визначені у файлі docker-compose.yml. Це може бути корисно, якщо ви внесли зміни до своїх сервісів і хочете переконатися, що образи актуальні.

Це лише кілька команд, що найчастіше використовуються, повний набір команд docker-compose можна знайти в [офіційній документації](https://docs.docker.com/engine/reference/commandline/compose/) (https://docs.docker.com/reference/cli/docker/compose/) Docker Compose.

Тепер перед початком роботи ми однією командою docker-compose up запускаємо одразу всі сервіси, що використовуються в проекті.