Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

Розробка веб-застосунку з використанням хмарних веб-сервісів

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
с.в. **Борозенний С.О**

_____
(підпис)
“____” _____ 2020 р.

Виконав студент
Крайнік Ю.П
“19” квітня 2020 р.

Київ 2020

Календарний план виконання роботи

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	02.11.2019	
2.	Вивчення предметної області та підходи при роботі з хмарними сервісами.	10.12.2019	
3.	Розробка прототипу застосування	25.12.2019	
4.	Створення системи авторизації та API для доступу до AWS S3	30.01.2020	
5.	Створення фронт-енду застосування	20.02.2020	
6.	Розміщення застосування на платформі Heroku	20.03.2020	
7.	Написання теоретичної частини курсової роботи.	25.03.2020	
8.	Остаточне оформлення курсової роботи та підготовка презентації.	09.04.2020	
9.	Надсилання роботи для перевірку на відсутність плагіату.	19.04.2020	

Зміст

Анотація

У цій роботі продемонстровано процес розробки веб-застосування файлового обмінника із системою авторизації на основі JSON Web Token (JWT), взаємодією із реляційною базою даних за допомогою технології Object-Relational Mapping (ORM) та інтеграцією з хмарним веб-сервісом сховища даних Amazon S3. Для забезпечення оптимальної роботи застосунку в умовах виробництва використано чергу завдань Redis Queue, що виконує ресурсозатратні операції у паралельних процесах. Файлові запити до S3 передаються за допомогою створеного REST Api, а саме застосування розміщено на платформі Heroku, попередньо контейнеризовано у Docker.

Ключові слова: Python Flask, веб-застосування, Amazon S3, реляційна база даних, PostgreSQL, файлообмінник, REST Api, jwt авторизація, Heroku, розміщення, redis queue, черга задач, ajax, Docker, сторонні веб-сервіси, full-stack розробка, хмарні технології.

Вступ

Наразі, розробка більшості веб-застосунків так чи інакше залучає використання хмарних сервісів, що дозволяє компаніям зосереджуватися безпосередньо на бізнес-логіці застосування, а такі проблеми, як масштабування ресурсів, розподіл навантаження та зміцнення захисту, делегувати частково чи повністю стороннім сервісам.

У той час, як утилізація хмарних технологій спрощує багато процесів, розробник повинен володіти необхідними знаннями, щодо доцільного вибору хмарного сервісу відповідно до потреб застосування, та вміти ефективно його використовувати.

Основним завданням цієї роботи є показати сучасні підходи до розробки веб-застосування файлового обмінника зі створенням REST Api та системи авторизації, інтеграції з хмарним сховищем Amazon S3 та розміщення застосунку у Docker контейнері на платформі (PAAS) Heroku.

У першому розділі буде зроблено огляд моделей хмарних обчислень, основних веб-сервісів Amazon (AWS), а зокрема пояснені механізми закладені у Amazon S3.

У другому розділі буде зосереджено увагу на принципах роботи використаних під час розробки технологій, зокрема фреймворку Python Flask як бази веб-застосування, JSON Web Token (JWT) для використання під час авторизації та програмного інструментарію SQLAlchemy для доступу до бази даних.

У останньому розділі будуть описані архітектура розробленого застосування, програмна реалізація, ефективне використання можливостей Amazon S3 за допомогою AWS SDK boto3 і розміщення застосування на платформі Heroku в контейнері Docker.

1. Хмарні технології та Amazon

1.1 Класифікація хмарних обчислень

З часом хмарні обчислення займали дедалі більше місця у світі Інтернет-технологій і відповідно з'явилися різні моделі та стратегії розміщення для задоволення конкретних потреб користувачів. Кожний тип хмарного сервісу і спосіб розміщення надає можливості з різним рівнем контролю та гнучкості, тому розуміння різниці між кожним з них допоможе зробити оптимальний вибір для наявних бізнес-потреб.

1.1.1 Інфраструктура як сервіс (IaaS)

Інфраструктура як сервіс є найбільш гнучкою із хмарних моделей, оскільки дає можливість мати повний масштабований контроль над розпорядженням ресурсів та налаштуванням інфраструктури.

У цій моделі надається безпосередній доступ до апаратного забезпечення, мережевих функцій, операційних систем та простору для сховища даних. Яскравим прикладом застосування цієї моделі є Amazon Web Service (AWS).

Переваги IaaS

1. Зменшення капітальних витрат

   За використання інфраструктури на основі хмари, зникає потреба витрачати кошти на розміщення власного фізичного апаратного забезпечення. На додаток, більшість IaaS пропонують динамічну модель оплати, тобто пропорційно до пройденого часу або кількості використаного машинного простору.

2. Гнучкість

   Ця модель зручна для підтримки тимчасового або нестабільного робочого навантаження. Оскільки виділення інфраструктури власними силами не виправдовує витрачених коштів, IaaS дає можливість задовольнити цю потребу.

3. Простота розміщення та масштабованість

   Делегувати процес розміщення серверів, сховища для даних та мережі хмарному провайдеру є набагато простіше і швидше, ніж робити це власними силами без попередньої основи. Як результат, система стане доступною для використання набагато швидше.

   Також за потреби кількість апаратних ресурсів може змінюватися автоматично на боці провайдера, що дає можливість швидко реагувати на зміну навантаження на систему.

Недоліки IaaS

1. Неясність

   Оскільки вся інфраструктура підтримується і керується хмарним провайдером, зазвичай користувачу не відомо більшості деталей конфігурації та ефективності. У результаті, робити моніторинг системи може бути досить складно, за умови відсутності необхідних інструментів у відповідного IaaS.

2. Залежність від провайдера

   Оскільки вся інфраструктура підтримується третьою стороною, то у разі виникнення проблем в IaaS, система користувача теж буде недоступною. Також клієнтська сторона є обмеженою у здійсненні низькорівневих змін до наданих ресурсів, тому що всі оновлення та підтримку здійснює провайдер.

1.1.2 Платформа як сервіс (PaaS)

Платформа як сервіс надає налаштоване середовище для розміщення власних застосувань. Це може бути “чиста” операційна система або мати встановлений веб-сервер, базу даних, тощо. У цій моделі вся відповідальність за інфраструктуру покладається на хмарного провайдера і користувачу лише потрібно керувати підтримкою та розміщенням застосування. Для прикладу, цю модель використовує Google App Engine.

Переваги PaaS

1. Оперативність розміщення

   З цією моделлю тестувати новий функціонал, різноманітні конфігурації застосування у різних середовищах є набагато швидше за традиційний спосіб. Також будучи онлайн-платформою колаборація між командами теж покращується, оскільки кожен може робити зміни віддалено.

2. Автоматизовані оновлення

   Немає необхідності слідкувати за актуальністю версій та підтримкою програмного забезпечення, плануванням ресурсів, тощо, оскільки PaaS це робить без участі користувача. Це дає можливість заощадити як кошти, так і потенційно витрачений час на додаткову роботу.

3. Масштабованість

   Як і в більшості хмарних сервісів, PaaS динамічно масштабує необхідні компоненти відповідно до навантаженості системи, тому за різкого збільшення кількості користувачів розміщеного застосування, спроможність мережі, наприклад, буде автоматично розширена.

Недоліки PaaS

1. Несумісність інфраструктури

   Якщо в компанії вже існує певна інфраструктура, то є імовірність, що не кожна її складова може бути налаштована під певний хмарний сервіс. У такому випадку доведеться замінити існуючі програми на хмаросумісні аналоги для повної інтеграції або залишити їх поза хмарою взагалі.

2. Прив'язка до вендора

   Зазвичай, робити міграцію сервісів наданих однією платформою до іншої є досить складно. У результаті, час простою системи збільшиться і з'являться додаткові витрати, коли буде здійснюватися перехід до іншої PaaS, тому варто ретельно обирати хмарного провайдера, який буде покривати усі наявні потреби.

3. Ризики безпеки

   Як правило, більшість даних зберігається в PaaS провайдера, тому їхньою обробкою можуть займатися сторонні сервіси і оцінити рівень безпеки даних може бути складно. Якщо застосування має сувору політику безпеки, це може бути проблемою.

1.1.3 Програмне забезпечення як сервіс (SaaS)

У цій моделі безпосередньо застосування доставляється до кінцевого користувача через Інтернет. SaaS надає закінчений продукт, який запущений і підтримується сервісним провайдером, тобто немає потреби слідкувати за інфраструктурою, яка необхідна для роботи застосування. Переваги та недоліки цієї моделі є аналогічними до попередньо описаних,

включаючи менші попередні витрати, швидкість розміщення та конфігурації, доступність, масштабованість, а також обмежений контроль, ризики безпеки і можливі уповільнення в роботі пов’язані із мережею.
Яскравими прикладами SaaS застосування є Gmail, Dropbox, Google Docs і подібні.

1.2 Amazon веб-сервіси (AWS)

Амазон веб-сервіси пропонує широкий набір хмарних продуктів, що включає в себе обчислювальні потужності, сховище для даних, бази даних, аналітику, мережеві технології, інструменти для розробки, менеджменту та безпеки. Усі ці сервіси доступні на вимогу за лічені секунди, без потреби попередньої оплати. [1] Нижче буде зроблено огляд кількох веб-сервісів.

1.2.1 Amazon EC2

Amazon Elastic Compute Cloud — це веб-сервіс, що надає захищені, змінні у розмірі хмарні обчислювальні потужності. EC2 має простий у користуванні веб-інтерфейс, що дозволяє отримати бажану конфігурацію машини за лічені хвилини, маючи повний контроль над обчислювальними ресурсами. Amazon пропонує на вибір екземпляри (EC2 instances) із різною потужністю та ємністю для різних потреб. Також створені віртуальні сервери можна конфігурувати через Secure Shell (SSH) протокол.
Модель оплати є досить гнучкою і залежно від типу навантаження та потреб, можна обрати економічно вигідні плани.

1.2.2 Amazon Step Functions

Сервіс покрокових функцій надає можливість комбінувати декілька Amazon веб-сервісів у безсерверні робочі процеси для пришвидшення створення та оновлення застосувань. Ці процеси виконуються у покроковій послідовності, де результат одного кроку йде на вхід наступному.

Завдяки відображенню процесу у вигляді діаграми скінченного автомата, процес розробки стає набагато інтуїтивнішим та простий у модифікації, оскільки кожен крок виконання відслідковується і у разі виникнення помилок автоматично перезапускається. AWS Step Functions підходить до таких довготривалих задач як: тренування моделі машинного навчання, генерація звітів або автоматизація девопс процесів. Також можливо створювати короткотривалі, багато обсяжні процеси як потокова обробка даних. [1]

1.2.3 Amazon S3

Amazon Simple Storage Service — це сервіс об’єктного сховища даних, що пропонує масштабованість, доступність даних, захист та ефективність найвищого рівня. Найкраще підходить для зберігання статичного або медіа контенту. Має витримку 99.999999999% (11 дев’яток), тобто іншими словами Amazon: “Якщо ви збережете 10 тис. об’єктів у нас, в середньому ми можливо втрачатимо 1 з них кожні 10 мільйонів років.”

S3 надає віртуально еластичний та необмежений простір для сховища, до якого можна доступитися через набір простих інтерфейсів. Модель оплати включає в себе декілька компонентів: розмір і тривалість збереження даних, GET, PUT та LIST запити і пересування даних на вихід та вхід.

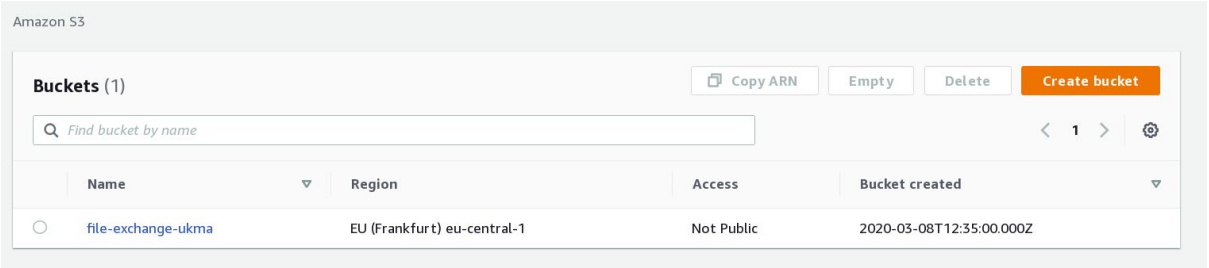

Рис 1. *Вікно з AWS S3 Management console. Перелік створених відер*

Принцип роботи

Можемо підсумувати основні механізми роботи Amazon S3 у такий спосіб:

- Ключовими термінами S3 є відро та об’єкт, де відро - це контейнер, а об’єкт - файл із метаданими, що включають ім’я, розмір та дату створення. Відра, створені користувачем, можуть містити будь-які об’єкти, що не перевищують 5 Терабайт у розмірі з метаданими не більше 2 Кілобайт.
- У кожного відра є глобально унікальний ідентифікатор, а кожному об’єкту користувач присвоює унікальний всередині відра ключ через REST або SOAP API. Amazon S3 підтримує концепт папки всередині відра, назва якої входить до складу ключа об’єкта, розділена з “/”. Відповідно, для кожного об’єкта існує унікальний url, що має форму <bucketName>.s3.amazonaws.com/<objectKey>
- Є можливість встановлювати політику доступу до відер та об’єктів за допомогою Bucket Policy та списків контролю доступу (ACL). Із першим методом власник може надавати кастомізований доступ конкретному користувачу, акаунту чи ip-адресі. Натомість списки контролю доступу можуть тільки надавати дозволи на індивідуальні об’єкти та відра.
- Для здійснення автентифікованих запитів через REST Api використовується кастомізована HTTP схема на основі Hash Message

Authentication Code (HMAC). Оскільки при створенні акаунта користувач отримує id ключа доступу та секретний ключ доступу, кожен запит підписується HMAC алгоритмом, використовуючи секретний ключ користувача та обрані елементи запиту.

Під час отримання автентифікованого запиту, Amazon S3 теж обчислює “підпис” для отриманого повідомлення і порівнює їх на еквівалентність. У разі успіху, доступ надається і виконується вказана операція, а інакше повертається помилка.

- Amazon S3 використовує тип узгодженості даних у кінцевому результаті (eventual consistency). Тобто у разі оновлення об’єкта, нові дані стануть доступними після успішної повної реплікації.

2. Використані технології

2.1 Python Flask

Flask це фреймворк мови програмування Python, призначений для розробки веб-застосувань. Його ціль - бути мінімалістичним без втрати функціональності. Flask є розширюваним та гнучким, що дає можливість розробнику обирати такі компоненти, як доступ до бази даних, систему авторизації, валідацію веб-форм та інші.

Ініціалізація

Усі Flask застосування для роботи повинні створити екземпляр застосунку. Веб-сервер передає усі запити від клієнта цьому об'єкту для подальшої обробки, використовуючи протокол Web Server Gateway Interface (WSGI). Мінімалістична природа Flask дозволяє створити базове застосування з однієї веб-сторінки, залучаючи малу кількість рядків коду. У Зразку 1 продемонстрована програма, що повертає рядок "Simple app" на кореневій сторінці.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Simple app!'

if __name__ == ' __main__ ':
    app.run()
```

Зразок 1. *Зразок базового веб-застосування на Flask*

Маршрутизація

Декоратор (обгортка над функцією в Python) app.route використовується для зв'язування функції із заданим відносним url. У Зразку 1 функція index() зв'язується із кореневою сторінкою, що відповідає значенню аргумента "/". Також, цей декоратор може приймати список дозволених HTTP методів для цього шляху. За замовчуванням, дозволений лише GET.

Досить часто ми хочемо отримати об'єкт із певної колекції, тому зазвичай url буде мати вигляд "collectionName/<object_id>". У цьому випадку, ідентифікатор конкретного об'єкта буде змінним і створювати вручну всі можливі шляхи є неефективно. Flask надає можливість передавати цей динамічний компонент як аргумент функції, яку обгорнуто з app.route(). Значення цього аргументу автоматично береться із вказаного url, вкладеного в кутові дужки. За замовчуванням, динамічні компоненти мають тип даних string, проте його можна уточнити через ":" у url шляху.
У прикладі нижче за шляхом /user/ можна отримати відповідь лише за умови, що динамічний компонент user_id має тип int.

```
@app.route('/user/<user_id:int>')
def user(user_id):
    return f'<h1>Your id is {user_id} </h1>'
```

Зразок 2. *Використання динамічних компонентів url*

Запити

Взаємодія із запитами від клієнта відбувається з допомогою об'єкта контекстного запиту. Flask використовує контексти, щоб зробити певні об'єкти глобально доступними у застосуванні, але лише для одного потоку, щоб коректно опрацьовувати запити кількох клієнтів одночасно.

Існує 2 типи контекстів у Flask: контекст застосування та запиту. Кожен з них має декілька змінних, до яких надається доступ. Для першого типу це:

1. **current_app** — екземпляр активного застосування.
2. **g** — об'єкт, який слугує тимчасовим сховищем під час опрацювання запиту. Обнуляється після кожного запиту.

До змінних контексту запиту належать:

1. **request** — об'єкт, що містить в собі дані надісланого клієнтом HTTP запиту, такі як атрибути, заголовки, аргументи чи тіло форми.
2. **session** — об'єкт сесії користувача у формі словника (ключ-значення), дані якого зберігаються між запитами.

Коли застосування отримує запит від клієнта, йому необхідно знайти відповідну функцію, яка повинна його опрацювати. Для цього завдання Flask здійснює пошук наданого в запиті url в url мапі застосування, що зберігає url, http метод та відповідну їм функцію, визначена розробником для обробки. Flask наповнює цю структуру даних із допомогою декораторів app.route, як було показано у Зразку 1.

Відповіді

Коли Flask викликає view функцію, то її результатом має бути HTTP відповідь. Для підвищення гнучкості цього процесу Flask попронує декілька можливостей для різних потреб. Для прикладу, якщо

повертається рядок, то він автоматично конвертується у валідну html сторінку. Також існують спеціальні функції для створення відповідей, такі як send_file(path, name), що надсилає файл клієнту, або render_template(path), що використовує двигун шаблонів, який буде детальніше описано нижче.

Також є функція make_response(*args), що надає можливість явно конструювати об'єкт відповіді. Це може бути потрібно для більш детальної конфігурації, як встановлення додаткових заголовків відповіді чи cookies.

У Flask також можна легко робити переадресацію (redirect) із допоміжною функцією. Цей вид відповіді особливий тим, що не включає в собі документ сторінки, а лише надає браузеру новий url. Він позначається з кодом http відповіді 302 і часто використовується у веб-формах.

За замовчуванням, view функції повертають статус код 200, що означає успіх, проте розробник може встановити власний, як аргумент при створенні відповіді.

Шаблони

Шаблони надають можливість відділити презентацію від бізнес-логіки. Фактично, це файл, що містить статичний текст http відповіді разом із змінними, що будуть містити динамічні дані, які стануть відомі у контексті запиту. Процес, під час якого в змінні підставляються фактичні значення і повертається кінцевий текст відповіді, називається рендерингом.

Для завдання рендерингу Flask використовує шаблонізатор Jinja2. Jinja2 допомагає розробнику без зайвих зусиль забезпечити безпеку

застосування, уникаючи потенційно небезпечний ввід користувача. Нижче наведено приклад шаблона з динамічними даними у змінній “name”.

```
<!doctype html>
<title> Hello from Flask </title>
<h1> Hello {{ name }}! </h1>
```

Зразок 3. *Jinja2 шаблон із змінною “name”*

2.1.1 SQLAlchemy

SQLAlchemy — це інструментарій для роботи з SQL базами даних безпосередньо через мову Python. Бібліотека надає стандартизований інтерфейс, що дозволяє не турбуватися про діалект бази даних, із якою відбуватиметься комунікація. Для використання SQLAlchemy всередині Flask застосувань, існує розширення, яке має назву Flask-SQLAlchemy.

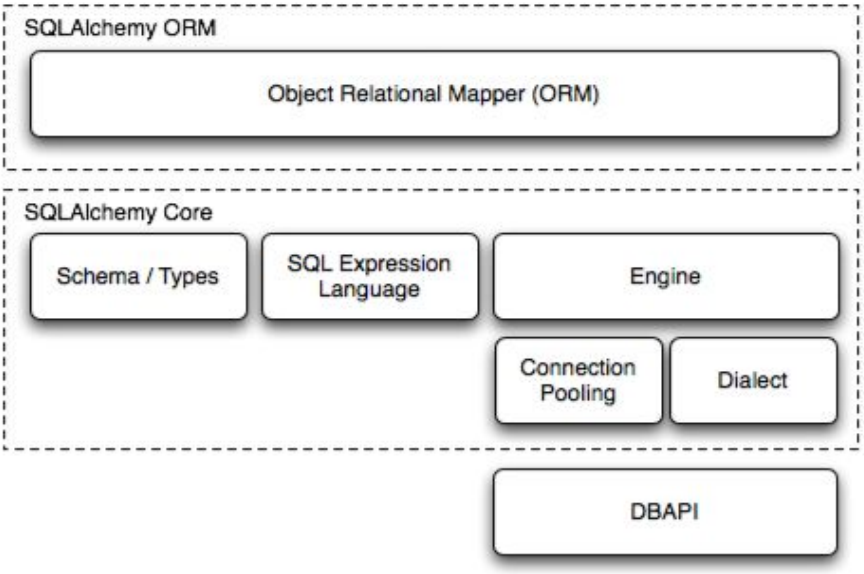

Рис. 2. *Компоненти SQLAlchemy[3]*

Моделі

Термін модель відповідає довготривалим сутностям, що використовуються в застосуванні. У контексті об'єктно-реляційного відображення (ORM), модель має форму Python класу із атрибутами, що збігаються із колонками відповідної таблиці бази даних. Екземпляр бази даних, у вигляді Python об'єкта, із Flask-SQLAlchemy надає базовий клас для розширення моделями, а також набір допоміжних класів та функцій, що використовуються для визначення їхньої структури.

```
class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, index=True)
```

Зразок 4. *Приклад створення моделі з Flask-SQLAlchemy*

Зв'язки

Як відомо, у реляційних базах даних є можливість встановлювати зв'язки між записами різних таблиць. SQLAlchemy підтримує 4 типи зв'язків: один-до-одного, один-до-багатьох, багато-до-одного та багато-до-багатьох (вимагає створення додаткової таблиці для асоціації). Ці зв'язки можна встановити за допомогою методу екземпляра бази даних db.relationship(). Нижче будуть описані найпоширеніші опції конфігурації зв'язку.

1. backref — додає зворотний напрямок зв'язку, створюючи атрибут моделі із вказаним ім'ям. Цей атрибут дозволяє доступитися до зв'язаної моделі через об'єкт, а не зовнішній ключ.
2. primaryjoin — вказує умову для join операції в явному вигляді. Необхідне лише у випадку двозначних зв'язків, наприклад існує декілька зовнішніх ключів.

3. lazy — вказує, яким способом завантажувати зв’язні об’єкти. Серед можливих варіантів можна обрати: **select** (об’єкти вантажаться на вимогу, тобто коли до них вперше доступаються), **joined** (пов’язані об’єкти завантажуються одразу ж в одній результуючій множині після виконання операції join), **subquery** (теж саме, що і в joined, але як підзапит), **noload** (пов’язані об’єкти не завантажуються) і **dynamic** (замість завантаження об’єктів повертається окремий запит, який можна конфігурувати)
4. userlist — якщо має значення False, то використовує скалярне значення замість списку (наприклад для перетворення із один-до-багатьох в один-до-одного)
5. cascade — визначає стратегію для каскадних операцій. Можливі значення: **save-update** (коли батьківський об’єкт оновлюється, то його дитина теж), **delete** (батьківський об’єкт видаляється разом із дитиною), **delete-orphan** (коли об’єкт втрачає посилання на батьківський об’єкт, то він буде видалений), **merge** (коли об’єкт додається до сесії бази даних, то усі асоційовані з ним об’єкти теж будуть додані; стоїть за замовчуванням) і **all** (включає усі каскадні стратегії, окрім delete-orphan)

2.2 JSON Web Token (JWT)

JSON Web Token — це відкритий стандарт[5], що визначає компактний і автономний метод для безпечної передачі інформації між сторонами у вигляді JSON об'єкта. Оскільки ця інформація містить цифровий підпис, вона може бути верифікована. Цей підпис може бути зроблений за допомогою секретної фрази (використовуючи HMAC алгоритм) або пари ключів згенерованих RSA або ECDSA.

Застосування JWT

Найбільш поширений випадок використання цього токену — це під час авторизації. Як тільки користувач виконує логін у систему, кожен наступний http запит буде включати в себе JWT, дозволяючи йому доступитися до захищених сервісів та ресурсів веб-застосування. Цей метод зручний тим, що є простим у налаштуванні і дозволяє використання у різних доменах.

2.2.1. Структура JWT

JSON Web Token складається із трьох частин: header, payload та signature.

Header

Заголовок JWT містить в собі інформацію про те, як повинен обчислюватись цифровий підпис. Представлений у формі json об'єкту і містить поля **alg**, що відповідає за алгоритм підпису (HMAC SHA256 або RSA), та **typ**, який має значення "JWT" для ідентифікації. Після створення, цей json закодований у Base64Url[11].

Payload

Друга частина токену містить в собі корисне наповнення або, як його ще називають, claims (заявки). Ці заявки несуть інформацію про сутність, зазвичай користувача, наприклад userID. Також існує список стандартних заявок для JWT Payload, серед яких є **iss** (визначає застосування, із якого відправляється токен), **sub** (визначає тему токена) та **exp** (тривалість життя токена). Далі так само кодується у Base64Url.

Signature

Для створення підпису необхідно взяти попередньо закодовані header та payload і з'єднати їх через крапку. Після цього результат хешується алгоритмом, вказаним у заголовку, на основі заданого секретного ключа.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Зразок 5. *Приклад створення signature із алгоритмом HMAC SHA256*

Останнім кроком необхідно закодувати підпис в Base64URL і сконкатенувати через крапку з попередніми компонентами. У такому форматі досить просто передавати токен у середовищах HTML та HTTP, завдячуючи компактності, на відміну від стандартів на основі XML.

2.2.2 Принцип роботи із JWT

Використовуючи JSON Web Token для перевірки автентифікації користувача, у процесі зазвичай бере участь 3 сторони: користувач, сервер застосування та сервер авторизації. Останній видає користувачу токен, завдяки якому він зможе взаємодіяти із застосунком.

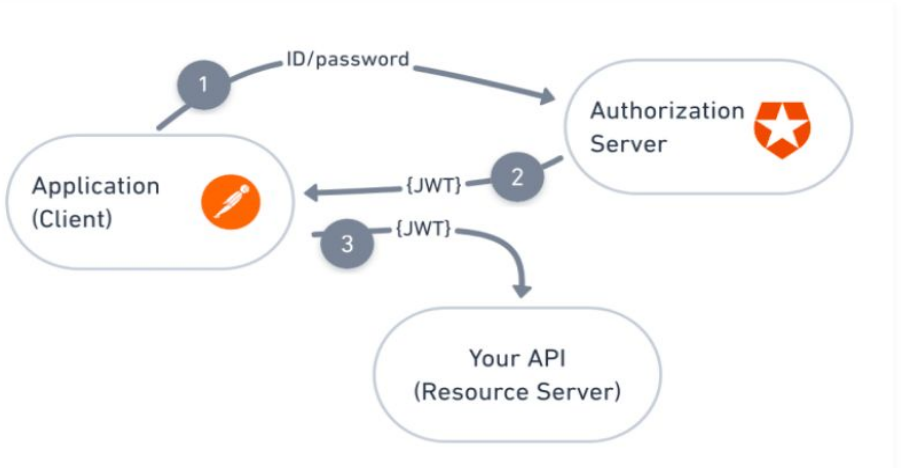

Рис. 3. *Принцип роботи JWT[6]*

Послідовність кроків використання JWT така:

1. Користувач заходить на сервер авторизації з допомогою ключа автентифікації, наприклад пари логін/пароль.
2. Сервер авторизації генерує веб-токен за отриманими даними і надсилає його користувачу.
3. Коли користувач робить запит до серверу застосування, він причіпляє попередньо отриманий JWT і застосування перевіряє за токеном чи користувач є дійсно той, за кого себе видає, і виконує запит у разі успіху.

Застереження

Важливо розуміти, що використання JWT не приховує дані автоматично, оскільки його основна мета — це верифікація авторизованості джерела даних. Це означає, що не рекомендується включати секретну інформацію в тіло токена.

3. Веб-застосування файлового обмінника

3.1 Опис проекту

Розроблене веб-застосування є платформою для завантаження та поширення персональних файлів. Даний програмний продукт надає необхідний базовий функціонал, такий як:

- створення персонального сховища з авторизацією
- завантаження будь-якого файла у сховище через Amazon S3
- отримання переліку завантажених файлів із розміром та датою модифікації
- видалення вибраного файла зі сховища
- отримання тимчасового посилання на файл, яке можна поширити

У додаток, безпосередньо в кабінеті AWS S3 було створено конфігурацію на тривалість життя створених файлів, після якої файл буде автоматично видалено, для чистки від застарілих файлів.

Розглядаючи можливості для розширення функціоналу, можна виділити додавання обмеження по загальному розміру сховища для кожного користувача, а також відображення тривалості життя для кожного файлу.

Нижче будуть продемонстровані знімки користувацького інтерфейсу розробленого застосування.

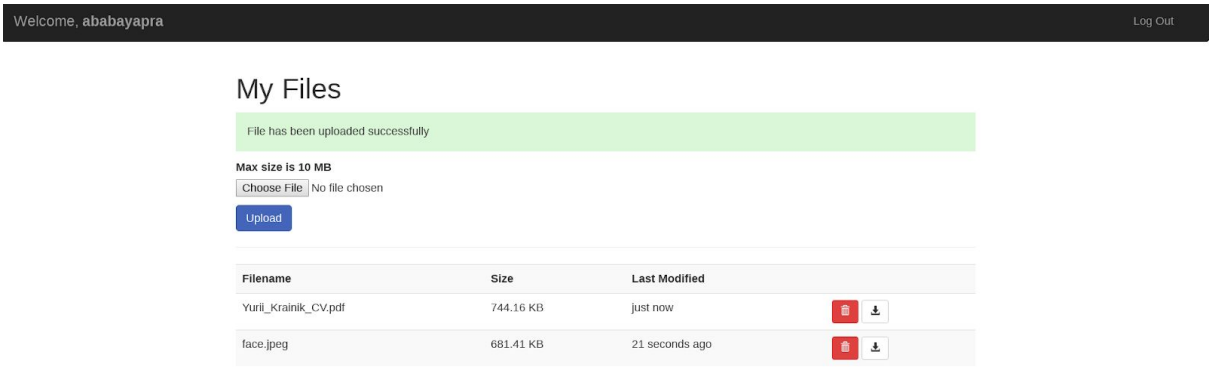

Рис. 5. *Особистий кабінет користувача. Після завантаження файлу у сховище*

Рис. 6. *Вікно реєстрації нового користувача*

3.2 Архітектура застосування

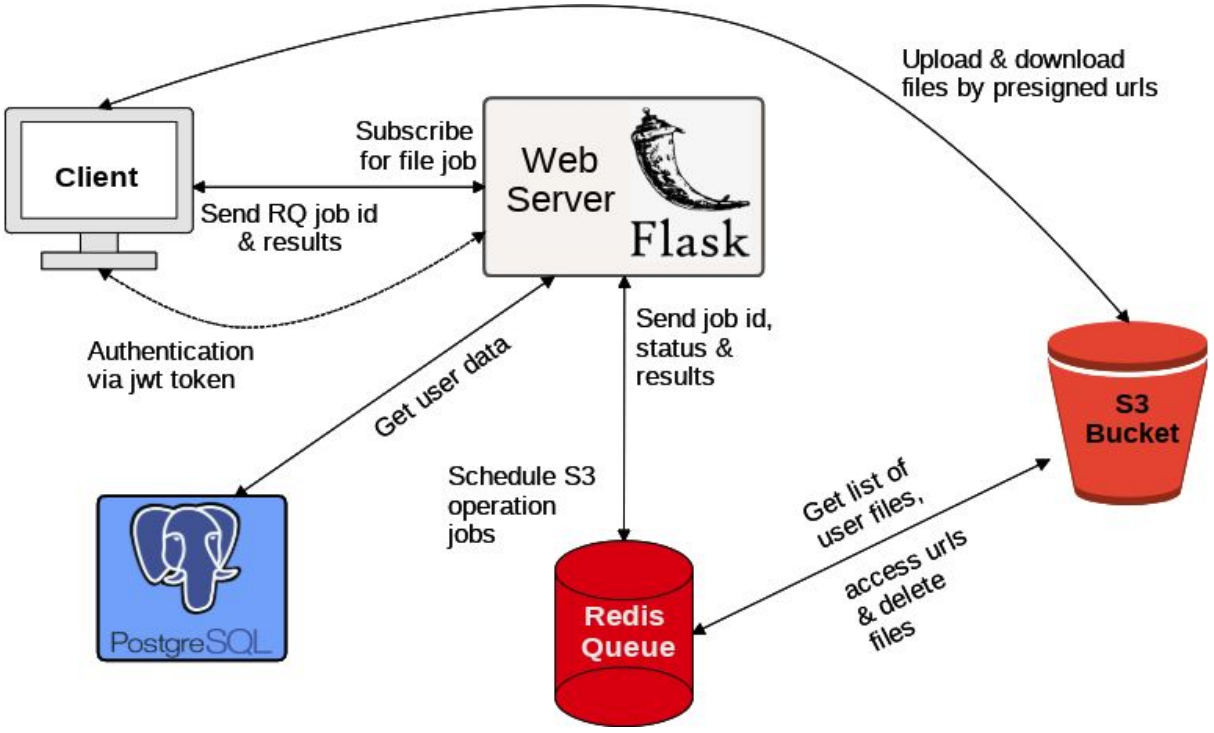

Рис. 7. *Архітектура застосування. Здійснення запитів/відповідей між компонентами*

Процес роботи циклу запит/відповідь складається з таких кроків:

1. Клієнт надсилає запит на Flask веб-сервер і перевіряється на присутність і валідність токену авторизації. Детальніше про сесію користувача буде далі в роботі.
2. Якщо запит дозволено, на сервері створюється фонове Redis Queue[9] завдання для здійснення обробки файла і комунікації із AWS S3 Api. У цій ситуації асинхронність необхідна для зменшення навантаженості з веб-сервера. Після створення, клієнту повертається job_id.
3. Після отримання айді створеного завдання, клієнт з допомогою AJAX кожну секунду робить запит за url *tasks/<job_id>*,

перевіряючи готовність результатів фонової задачі. У разі успіху фронт-енд обробляє ці результати залежно від їхнього призначення. Варто зауважити, що для завдання перевірки готовності даних, краще підходить технологія WebSocket, але оскільки основний фокус цієї роботи на бек-енд розробці, для спрощення застосовано AJAX.

4. Заплановані фонові задачі здійснюють запити до створеного REST Api за url префіксом */api/files* для здійснення комунікації із AWS S3. Безпосередньо для доступу до Amazon сховища використовується бібліотека boto3, детальніше про яку буде далі в роботі.

3.2.1 Контролер сесії користувача

Цей компонент відповідає за створення, видалення і оновлення JWT користувача, а також захист маршрутів застосування від неавторизованого доступу. Для здійснення вказаних операцій використовується open-source бібліотека flask_jwt_extended[7], що є розширенням для фреймворка Flask.
Під час здійснення клієнтом запиту на веб-сервер, можливі 3 сценарії: користувач не залогінений, закінчився термін життя токена доступу або користувач має валідний jwt.

Відсутній токен доступу

Якщо користувач не залогінений, тобто не має JWT, то він переадресований на сторінку з логіном, де він може зайти в акаунт за email і паролем або зареєструватися на іншій сторінці. Спершу розглянемо процес здійснення логіну.

Якщо хеш наданого клієнтом паролю збігається із хешом паролю користувача, отриманого за заданим email, то формується сесія для цього користувача. У випадку невдачі, повертається помилка у вигляді flash

повідомлення. Далі за допомогою функцій *create_access_token()* та *create_refresh_token()* з бібліотеки flask_jwt_extended формуються access та refresh токени. Вони приймають на вхід параметр identity для наповнення payload, що у цьому випадку є user_id з сутності користувача (про це далі). Код функції логіну можна переглянути в Додатку А.

Здійснити реєстрацію нового користувача може тільки не залогінений користувач. Спершу надані дані для реєстрації валідуються і серіалізуються в об'єкт моделі User за допомогою бібліотеки marshmallow[8]. Перед створенням нового користувача у базі даних, пароль нового користувача в методі серіалізації хешується функцією *generate_password_hash()* із модуля werkzeug.security, що доступний для Flask застосування за замовчуванням. Після успішної реєстрації, користувач переадресовується на сторінку з логіном.

Прострочений access токен

За замовчуванням, access токен має 24 години життя після його створення, тому після закінчення цього терміну, необхідно його оновити. Для уникнення повторного логіну користувача, використовують refresh токен. Він існує безпосередньо для оновлення токена доступу і має тривалість життя 30 днів за замовчуванням.

Для цієї задачі ми створюємо функцію, що буде опрацьовувати подію простроченості jwt, і обгортаємо в декоратор *@jwt.expired_token_loader* із бібліотеки flask_jwt_extended. У тілі функції здійснюється запит за локальним шляхом */token/refresh*, де присвоюється новий access токен, згенерований аналогічним шляхом як під час логіну.

Дійсний токен доступу

У цій ситуації користувач, який здійснює запити на шляхи, що захищені декоратором *@jwt_required,* і має валідний токен доступу, отримує відповідь від веб-сервера залежно від результату виконання бізнес-логіки.

3.2.2 Сутність користувача в базі даних

Для створення таблиць в базі даних було використано моделі ORM розширення Flask-SQLAlchemy, яке було описано в Розділі 2.1. Для цього проекту була необхідність лише в створенні моделі для користувача, оскільки файли безпосередньо зберігаються в AWS S3, а зв'язок із користувачем закладений безпосередньо в ключ файла під час завантаження у сховище (детальніше про цей процес далі в роботі).

З метою спрощення роботи було створено допоміжний класові методи *lookup()* та *identify()*, що витягують об'єкт користувача з бази даних за email та id відповідно. Також до екземплярів моделі додано метод *check_password()*, що порівнює хеші паролів на еквівалентність. Повний код моделі User в Додатку Б.

3.2.3 Контролер роботи з файлами

Цей компонент є, так би мовити, проміжним інтерфейсом між AWS S3 і клієнтом. Серед доступних шляхів є: повернення списку файлів користувача, отримання посилання на завантаження до та зі сховища Amazon, видалення конкретного файлу та отримання результату фонової операції. Кожна з цих файлових операцій, реєструє завдання в Redis Queue[9], результат якого можна отримати за шляхом /tasks/<task_id>.

Більш детально варто розглянути функцію підготовки файла до завантаження на AWS S3.

```
@files_bp.route('/tasks/<task_id>')
@jwt_required
def get_task_result(task_id):
    job = q.fetch_job(task_id)
    if job.is_finished:
        resp = {'data': job.result, 'csrf_token': (get_raw_jwt()
or {}).get("csrf")}
        return jsonify(resp), 200
    return {}, 202
```

Зразок 6. *Функція отримання результатів задачі з Redis Queue*

Підготовка файла до завантаження у сховище

Спершу перевіряється валідність наданої інформації про файл і отримується безпечна версія його назви для уникнення ін'єкції програмних інструкцій. Далі формується унікальний ключ об'єкта сховища за назвою файла та даними залогіненого користувача:

1. Генерується рядок uuid[10] версії 4 в 16-ому форматі.
2. Згенерований uuid, назва файлу та айді користувача з'єднуються в один рядок у форматі *{user_id}/{file_id}_{file_name}*.
3. Результат закодований в base64URL[11]

Далі сформований ключ використовується в шляху запиту до локального Api, а дані з файлової форми передаються в тілі запиту незмінними. Код підготовки файлу до завантаження можна знайти в Додатку С.

3.2.4 REST Api для доступу до AWS S3

За наявними файловими операціями було створено 2 ресурси: FileListRes із url */api/files* та FileRes із url */api/files/<key>*. Для списку файлів визначений лише метод GET, а для файлового ресурсу GET, PUT і DELETE. Для безпосереднього доступу до S3 було використано AWS SDK із назвою boto3[12]. Особливості його використання будуть описані далі.

Перед початком роботи, необхідно створити сесію для доступу до веб-сервісів Amazon за даними, отриманими при створенні AWS акаунту: назва регіону, айді ключа доступу та сам ключ доступу.

```
session = boto3.Session(
    region_name=app.config['AWS_REGION'],
    aws_access_key_id=app.config['AWS_ACCESS_KEY'],
    aws_secret_access_key=app.config['AWS_SECRET_KEY']
)
```

Зразок 7. *Приклад створення сесії для доступу до AWS*

Важливо відмітити, що дані для доступу до сторонніх сервісів категорично не рекомендується вставляти безпосередньо в код, а використовувати змінні оточення або окремі файли для зчитування.

Отримання списку файлів користувача

Для доступу до файлів сховища, необхідно спершу отримати інформацію про відро, в якому вони зберігаються. Як було попередньо сказано, ключ об'єкта сховища містить в собі айді поточного користувача, тому з використанням функції *my_bucket.objects.filter()* за заданим префіксом можемо отримати інформацію про файли користувача. Результат є списком з об'єктів типу ObjectSummary[13], з якого далі вилучаються необхідні атрибути в окремому json'і: ключ, назва файлу, розмір та дата модифікації.

Формування посилання для завантаження файлу до AWS S3

З точки зору ефективності, використовується метод завантаження файлу через підписаний url безпосередньо з боку клієнта. Адже, пропускати файл через веб-сервер кардинально уповільнює його роботу, тому цей спосіб є оптимальним. Суть генерації підписаного url полягає в формуванні тимчасового доступу до AWS, використовуючи дані для входу розробника

та інформацію про наданий файл. Результатом виклику функції *s3.generate_presigned_post()* є шлях до файлу на AWS S3 та набір необхідних полів для заповнення клієнтом в HTML формі, які необхідні для верифікації авторизованості POST запиту.

Формування посилання для завантаження файлу з AWS S3

У цьому випадку додаткової взаємодії від клієнта не потребується, оскільки для отримання доступу до завантаження файлу необхідно знати лише його ключ. Для здійснення цієї операції використовується функція *s3.generate_presigned_url()*, як і в попередньо згаданій функції один з аргументів якої вказує на тривалість життя посилання в секундах. У разі успіху, користувач буде переадресований на цей url для завантаження файлу. Повний код роботи із AWS S3 можна знайти в додатку D.

Варто відмітити, що згенерований url можна поширювати третім сторонам, оскільки в аргументах GET запиту містяться усі необхідні поля для верифікації.

3.3 Розміщення веб-застосування

Для розміщення розробленого застосування було використано платформу-як-сервіс Heroku[14], попередньо контейнеризувавши його в Docker[15]. Для побудови Docker образу разом із встановленням змін оточення та додаткових Heroku плагінів було використано маніфест heroku.yml[16]. У додаток, застосовано плагін heroku-manifest, який дозволяє відразу ініціалізувати нове застосування на платформі.

Для ролі бази даних було обрано СКБД PostgreSQL[17], а як HTTP сервер обрано gunicorn[18]. У секції **run**, маніфесту heroku.yml, вказані інструкції для запуску веб-сервера, директивою *web*, а як *worker* у цьому випадку працює Redis Queue, яка буде запущена в окремому процесі. Вміст Dockerfile та heroku.yml можна знайти в Додатку Е.

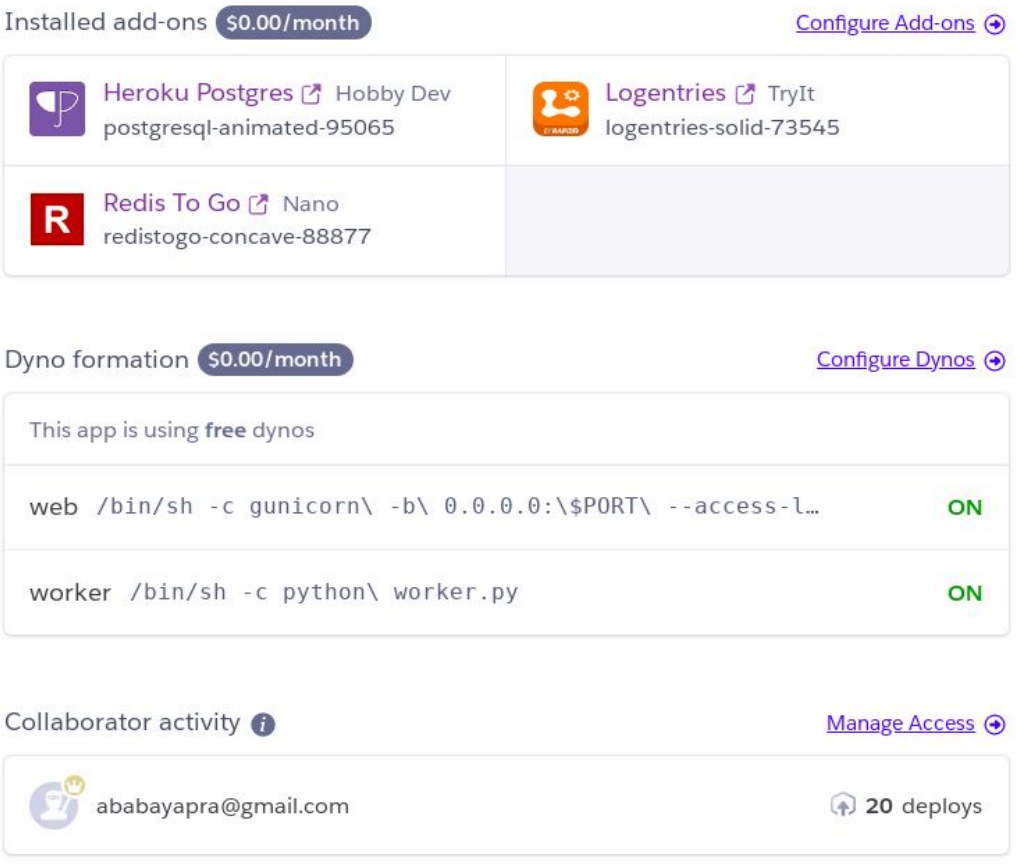

Рис. 8. *Вікно з інформацією про розміщене Heroku застосування*

Висновки

Фокусом цієї роботи було навчитися знаходити та опановувати інструменти для ефективної розробки веб-застосувань. Для наближення до умов виробництва, було залучено використання хмарних сервісів, як платформа Heroku та AWS S3. Наявність системи free-tier для використання Amazon веб-сервісів надала можливість для вільного експериментування.

Під час розробки багатокомпонентної системи важливо розуміти, які підходи використовувати та як ефективно налаштовувати комунікацію між мікросервісами. Завчасно дослідивши предметну область та існуючі рішення конкретних проблем, можна на етапі архітектури застосування зрозуміти доцільність вибору технологій у контексті поточної задачі.

У результаті виконання цієї роботи були розібрані та застосовані підходи до розробки веб-застосування із системою авторизації, ефективної роботи з файлами та сторонніми API, використовуючи фреймворк Flask та його розширення. Завдяки його мінімалістичності, не виникало обмеженості у виборі та була можливість випробувати різні технології, підібравши найбільш доцільні до кожної наявної бізнес-проблеми.

Розмістивши веб-застосування на платформі Heroku, я побачив неоптимальність обраних методів вирішення проблеми ефективного завантаження файлів до хмарного сховища і в результаті подальшого дослідження зміг досягти бажаної оптимальності. Саме тому, залучення сторонніх сервісів та розміщення в Інтернеті дає зрозуміти ефективність програмних рішень завдяки існуючим обмеженням та рекомендаціям під час використання цих сервісів.

Список використаної літератури

1. Overview of Amazon Web Services
2. Amazon Simple Storage Service - API Reference
3. SQLAlchemy Documentation — SQLAlchemy 1.3 Documentation
4. Welcome to Flask — Flask Documentation (1.1.x)
5. RFC 7519 - JSON Web Token (JWT)
6. Using JWT to authenticate and authorize requests in Postman
7. Flask-JWT-Extended's Documentation — flask-jwt-extended 3.24.1 documentation
8. marshmallow: simplified object serialization — marshmallow 3.5.0 documentation
9. RQ: Simple job queues for Python
10. Universally unique identifier
11. Base64URL Wiki
12. Boto 3 Documentation — Boto 3 Docs 1.12.41 documentation
13. ObjectSummary Boto3 Reference
14. Heroku: Cloud Application Platform
15. Docker: Empowering App Development for Developers
16. Building Docker Images with heroku.yml
17. PostgreSQL: The world's most advanced open source database
18. Gunicorn - Python WSGI HTTP Server for UNIX

Додаток А (обов’язковий)

“Функція логіну користувача”

```
auth_bp.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'GET':
        if get_jwt_identity() is None:
            return render_template('login.html')
        return redirect('/')

    email = request.form.get('email')
    password = request.form.get('password')
    user = authenticate(email, password)
    if not user:
        flash('No user was found with the given credentials.' , 'danger')
        return redirect(url_for('auth.login'))
    access_token = create_access_token(identity=user.identity)
    refresh_token = create_refresh_token(identity=user.identity)
    resp = make_response(redirect('/'))

    set_access_cookies(resp, access_token)
    set_refresh_cookies(resp, refresh_token)
    return resp
```

Додаток Б (обов’язковий)

“Визначення моделі User”

```
class User(db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String, nullable=False, unique=True)
    password = db.Column(db.String, nullable=False)

    @classmethod
    def lookup(cls, email):
        return cls.query.filter_by(email=email).one_or_none()

    @classmethod
    def identify(cls, id):
        return cls.query.get(id)

    @property
    def identity(self):
        return self.id

    def check_password(self, password):
        return check_password_hash(self.password, password)

    def __repr__(self):
        return f'User<email={self.email}>'
```

Додаток С (обов’язковий)

“Функція підготовки файлу до завантаження на AWS S3”

```
@files_bp.route('/upload', methods=['POST'])
@jwt_required
def upload_file():
    # There is no file selected to upload
    filename = request.form['fileName']
    if filename == "":
        return jsonify({'error': 'No file has been selected'}), 400

    if int(request.form['fileSize']) > app.config["MAX_FILE_SIZE"]:
        return jsonify({'error': 'The file is too large'}), 422

    filename = secure_filename(filename)
    object_name = utils.encode_key(
        utils.generate_obj_name(generate_uuid_str(), filename,
get_jwt_identity())
    )
    job = q.enqueue(utils.request_json,'PUT',
                    url=f'{app.config["BASE_URL"]}{api.url_for(FileRes,
key=object_name)}',
                    data=request.form, cookies=request.cookies)
    return jsonify({'task_id': job.get_id()}), 202
```

Додаток D (обов'язковий)

"REST Api для роботи з AWS S3"

```
@api.resource('/files')
class FileListRes(Resource):
    method_decorators = [jwt_required]

    def get(self):
        s3_resource = session.resource('s3')
        my_bucket = s3_resource.Bucket(app.config['S3_BUCKET'])
        fl = [utils.file_summary(f) for f in

my_bucket.objects.filter(Prefix=f'{get_jwt_identity()}/')]
        fl_sorted = [obj for obj in
                     sorted(fl,
                            key=operator.itemgetter('last_modified'),
                            reverse=True)]
        return {'file_list': json.dumps(fl_sorted, default=str)}


@api.resource('/files/<string:key>')
class FileRes(Resource):
    method_decorators = [jwt_required]

    def get(self, key):
        # Generate a presigned URL for the S3 object
        object_name = utils.decode_key(key)
        s3_client = session.client('s3')
        try:
            response = s3_client.generate_presigned_url(
                'get_object',
                Params={'Bucket': app.config['S3_BUCKET'],
                        'Key': object_name},
                ExpiresIn=3600
            )
        except ClientError:
            return {'error': 'File not found.'}, 404
```

```
        # The response contains the presigned URL
        return {'url': response}, 200

    def put(self, key):
        object_name = utils.decode_key(key)
        s3 = session.client('s3',
config=Config(signature_version='s3v4'))
        fields = {
                'acl': 'private',
                'Content-Type': request.form['fileType'],
                'Content-Disposition': f'attachment; '
                f'filename="{request.form["fileName"]}"'
            }
        response = s3.generate_presigned_post(
            Bucket=app.config['S3_BUCKET'],
            Key=object_name,
            Fields=fields,
            Conditions=[{item[0]: item[1]} for item in
fields.items()],
            ExpiresIn=3600
        )
        print(response)
        return response


    def delete(self, key):
        object_name = utils.decode_key(key)
        s3 = session.resource('s3')
        try:
            obj = s3.Object(app.config['S3_BUCKET'], object_name)
            obj.delete()
        except ClientError:
            return {'error': 'File not found.'}, 404

        return {}, 204
```

Додаток Е (обов’язковий)

“Файли для розміщення веб-застосування”

Dockerfile

```
FROM python:3.7.2-alpine

WORKDIR /usr/src/app
COPY requirements.txt requirements.txt
RUN apk update && apk add postgresql-dev gcc python3-dev musl-dev
RUN pip install -r requirements.txt
RUN pip install gunicorn psycopg2

COPY flaskapp flaskapp
COPY config.py manage.py worker.py ./
ENV FLASK_APP manage.py

CMD gunicorn -b 0.0.0.0:$PORT --access-logfile - --error-logfile -
manage:app
```

Heroku.yml

```
setup:
 addons:
   - plan: heroku-postgresql
build:
 docker:
   web: ./Dockerfile
run:
  web: gunicorn -b 0.0.0.0:$PORT --access-logfile - --error-logfile -
manage:app
  worker:
    command:
      - python worker.py
    image: web
```