МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:

«Програмний застосунок для використання технології DeepLink`s.»

Завідувач випускаючої кафедри — **Довбиш А.С.**

Керівник роботи — **Петров С.О.**

Студента групи ІН – 73 — **Бабича В.Ю.**

СУМИ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютерних наук

Затверджую_____

Зав. кафедрою Довбиш А.С.

"_____" _____2021 р.

ЗАВДАННЯ
до випускної роботи

Студента четвертого курсу, групи ІН-73 спеціальності "122 Комп'ютерні Науки" денної форми навчання Бабича Владислава Юрійовича.

Тема: "Програмний застосунок для використання технології DeepLink`s."

Затверджена наказом по СумДУ

№_____ от _____2021 р.

Зміст пояснювальної записки: 1) аналіз проблеми та постановка задачі; 2) вибір метода розв'язання задачі; 3) розробка інформаційного і програмного забезпечення системи

Дата видачі завдання "_____"_____2021 р.

Керівник випускної роботи _____ Петров С.О.

Завдання прийняв до виконання _____Бабич В.Ю.

РЕФЕРАТ

Записка: 65 стор., 28 рис., 1 табл., 1 додаток, 4 джерела.

Об’єкт дослідження — Технологія DeepLink’s

Мета роботи — Програмний застосунок для використання технології DeepLink’s

Методи дослідження — Технологія DeepLink, веб-сервери

Результати — розроблено web застосунок який працює за архітектурним стилем REST API і дає користувачам можливість створювати глибокі посилання для найбільш популярних додатків, Instagram та Facebook. Застосунок реалізовано з допомогою мови програмування Python та фреймворку aiohttp. В ході тестування проблем виявлено не було, застосунок працює добре

DeepLink, Instagram, Facebook, Python, aiohttp, Asynchronous programing, Posgresql, JWT, REST-full

ЗМІСТ

ВСТУП

В сучасному світі величезну ролі відіграють соціальні мережі. Люди мають змогу обмінювати новинами, слідкувати за цікавими особистями, компаніями, об’єднуватися в групи за інтересами, тощо. Тут як і всюди існує комерційна складова. Існує безліч способів продати продукт чи послугу через соціальні мережі. Також варто зазначати, що для просування цього компанії часто використовують рекламні майданчики, що надаються розробниками соціальних-мереж. Загалом варто зазначити, що левову частину прибутку соц-мереж складають саме рекламні оголошення.

Розглянемо ситуацію: маємо продавця Василя який торгує шкарпетками через профіль в Instagram. В певний момент часу він зрозумів, що хоче отримувати більше замовлень і налаштував рекламу, клік на яку веде в його профіль. Потенційному покупку реклама сподобалась і він захотів дізнатися більше деталей, та можливо зробити замовлення. Наразі послідовність його дій будет наступною: потенційний покупець тицяє на рекламне оголошення в Instagram, і відразу переходить на профіль продавця, але профіль відкривається не самим застосунком, а в внутрішньому браузері, користувач тут не залогінений, інтерфейс не настільки дружний, як в самому застосунку. І тут постає вибір: або знайти профіль вручну в застосунку, або авторизуватися в внутрішньому браузері. Виникає логічний висновок: якщо юзер недостатньо зацікавлений він не буде робити всі ці дії а пройде далі.

На щастя є вирішення цієї проблеми - використання технології DeepLinks. Для кожної мобільної операційної системи є можливість створити спеціальне посилання, яке буде відкриватися одразу в потрібному застосунку. Але ж ми не можемо знати які конкретно користувачі будуть перехожити за посилання. Тому це вирішується веб-сервером, який може дізнатися користувач з яким пристроєм надіслав запит, та на основі цього сформувати правильне посилання. Таким чином ми спрощуємо шлях від рекламного

оголошення до потенційного замовлення, а отже збільшуємо прибуток продавця. Це дуже просте пояснення, насправді ця технологія дає набагато більше гнучкості і можливостей, але основна мета - спростити шлях від посилання до певної сторінки в застосунку.

Метою роботи є розробка програмного застосунку який би міг формувати спеціальні посилання в залежності від операційної системи користувача та переадресовувати на їх. Об’єктом дослідження є механізми формування таких посилань, принцип визначення операційної системи клієнта на боці серверу. Предметом є розробка застосунку, який може поєднати ці дві функції.

На основі попередніх знань, для реалізації застосунку були поставлені наступні цілі

- Обрати найбільш підходящу мову програмування
- Провести аналіз інформації щодо формування DeepLink
- Провести дослідження, щодо визначення операційної системи користувача на сервері

1 АНАЛІЗ ВІДОМИХ РІШЕНЬ

Перед виконанням роботи дуже корисно вивчити вже готові подібні рішення. Це значно спростить проектування системи, допоможе одразу виявити потенційні помилки, проблеми, також варто звернути увагу на переваги, та врахувати це все при проектуванні та розробці.

1.1 Огляд схожих застосунків

1.1.1 WEB сайт appsflyer

На першій же сторінці бачимо пропозицію зареєструватися на демо період і короткий опис можливостей застосунку (Рисунок 1.1)

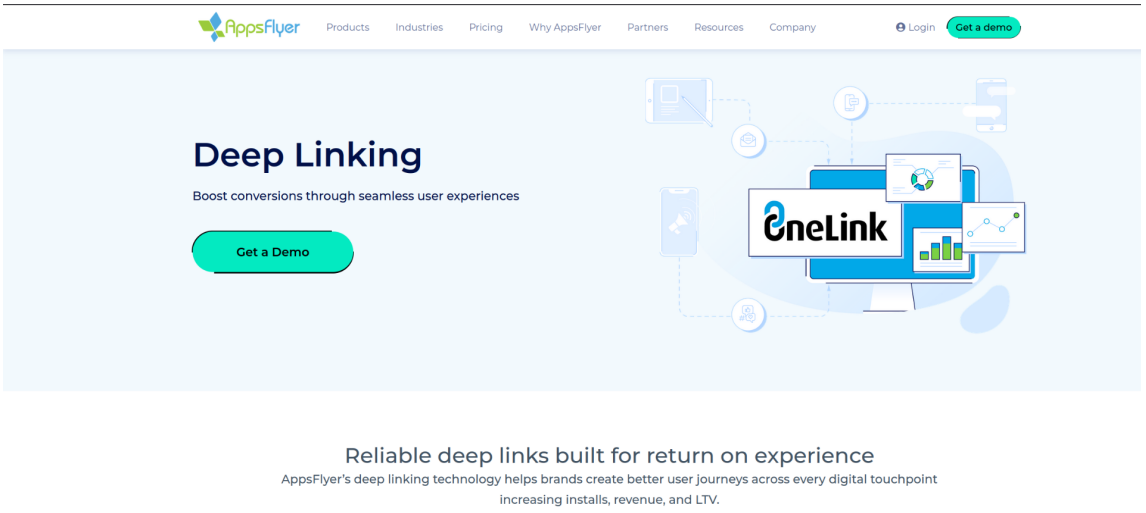

Рисунок 1.1

Далі йде блок з описом основних можливостей та випадків застосування технології (Рисунок 1.2)

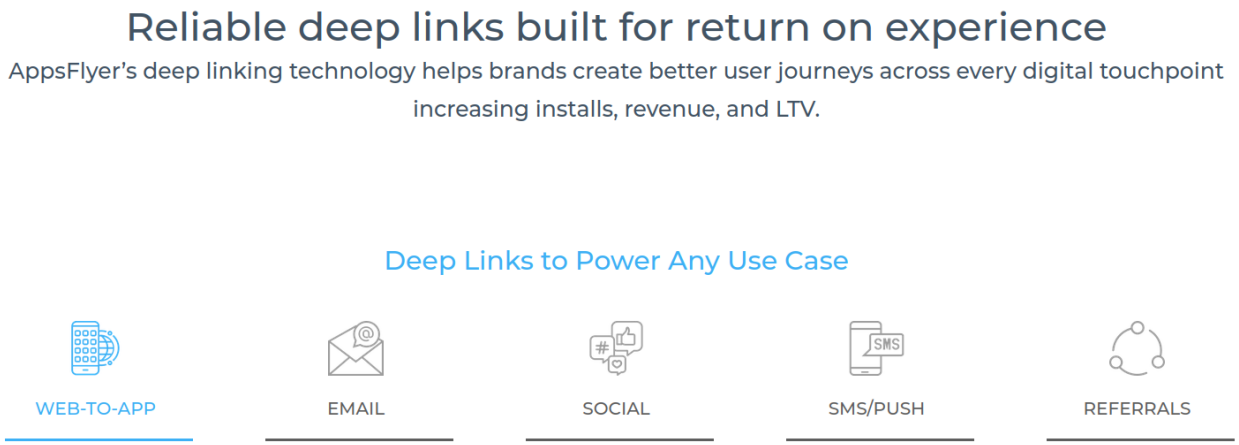

Рисунок 1.2 - Основні варіанти застосування

З точки зору маркетингу дуже важливим є розділ с перерахування компаній-гігантів які використовували цей застосунок, серед яких HBO, Nike, Ebay (Рисунок 1.3)

Рисунок 1.3 - Компанії-клієнти

Наступним блоком який позитивно впливає на вибір користувача є коротка статистика яка показує ефективність технології (Рисунок 1.5)

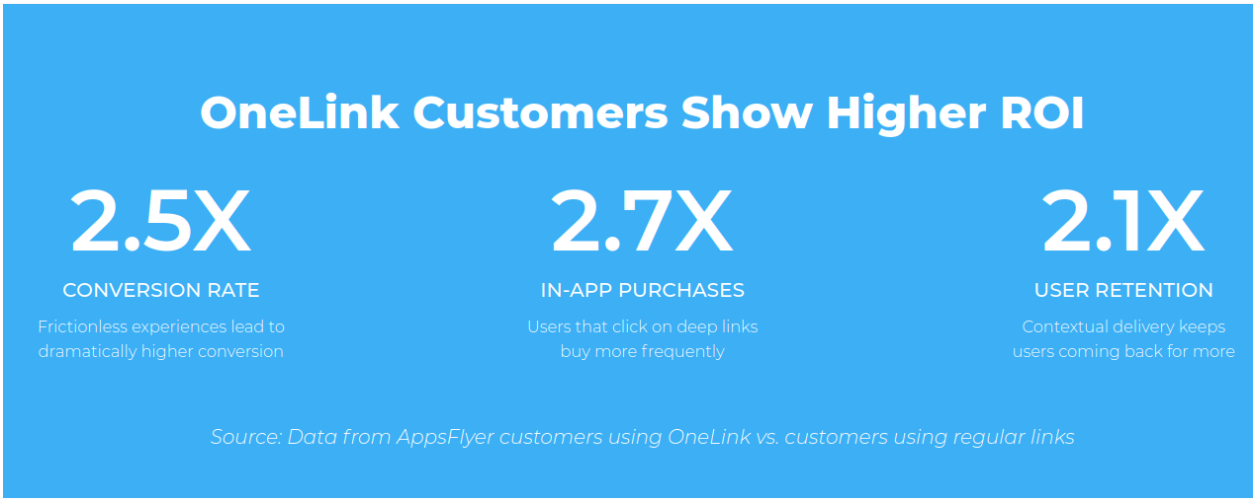

Рисунок 1.5 - Статистика, після використання сервісу

Також на сторінці є чудова ілюстрація для демонстрації принципу роботи застосунку. Для звичайного користувача це може стати дуже важливою інформацією

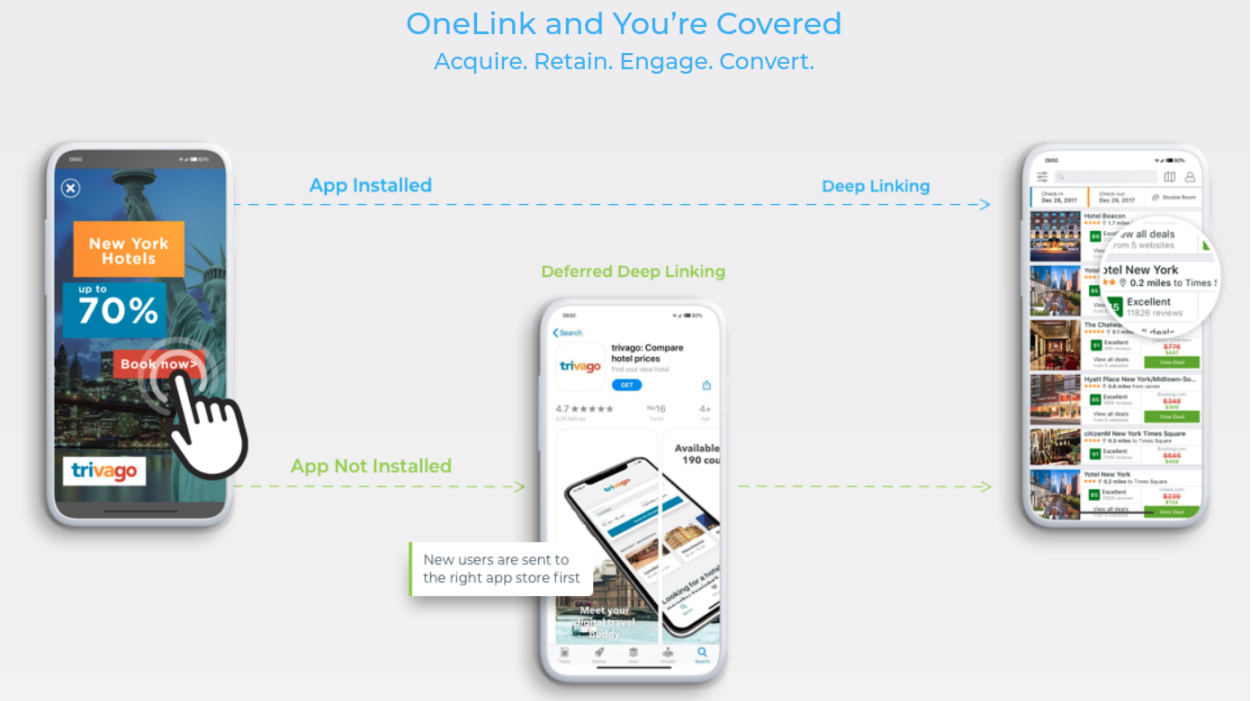

Рисунок 1.6 - Демонстрація принципу роботи

Сайт дуже масштабний на має багато функціоналу, користується попитом в світових компаній та є одним з лідерів галузі.

Основною перевагою цьому сервісу є його ж популярність, та багатий функціонал. Але з іншого боку інтерфейс користувача досить складний, та сервіс не дає можливості інтеграції з іншими застосунками. Він має свою фронтенд частину через яку і будується взаємодія з звичайними користувачами.

1.1.2 WEB сайт branch.io

Розглянемо найближчий аналог. На головній сторінці відразу бачимо перелік компаній які користуються сервісом, та короткий опис (Рисунок 1.7)

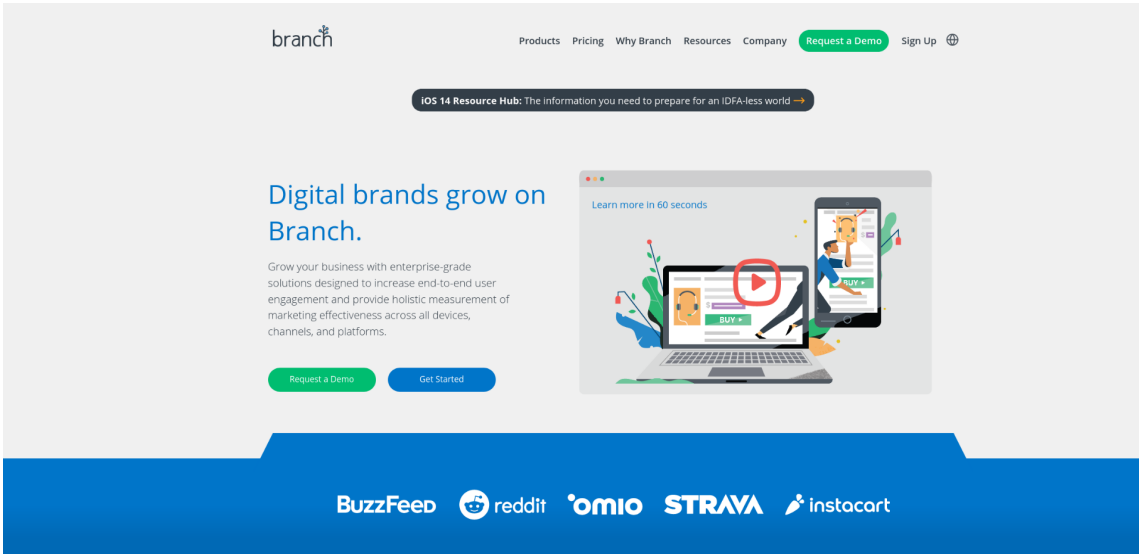

Рисунок 1.7 - Головна сторінка сайту

Спробуємо зареєструватися та створити посилання на профіль Instagram.

Після заповнення даних отримуємо на електронну пошту повідомлення для підтвердження (Рисунок 1.8)

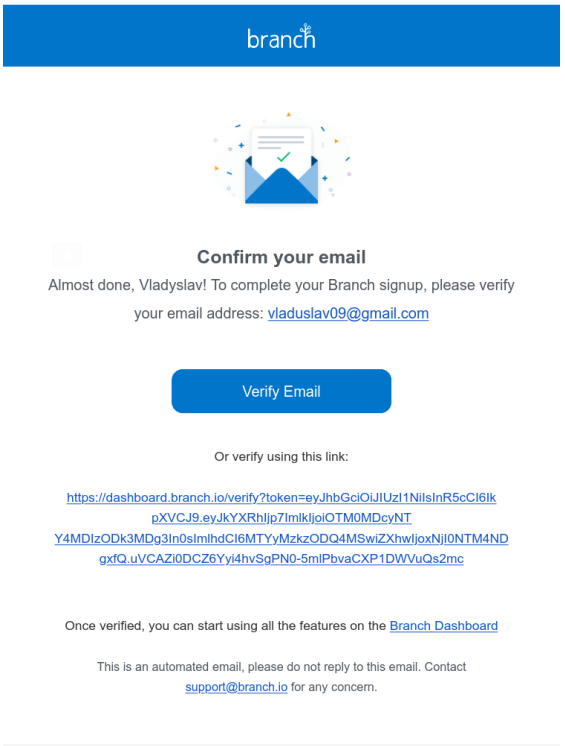

Рисунок 1.8 - Лист підтвердження реєстрації

Кабінет користувача виглядає наступним чином (Рисунок 1.9) Можна зробити попередні висновки, що сервіс має багатий функціонал та пропонує користувачам багато можливостей. Але з іншого боку він досить складний для початківця і потребує деякий час на вивчення

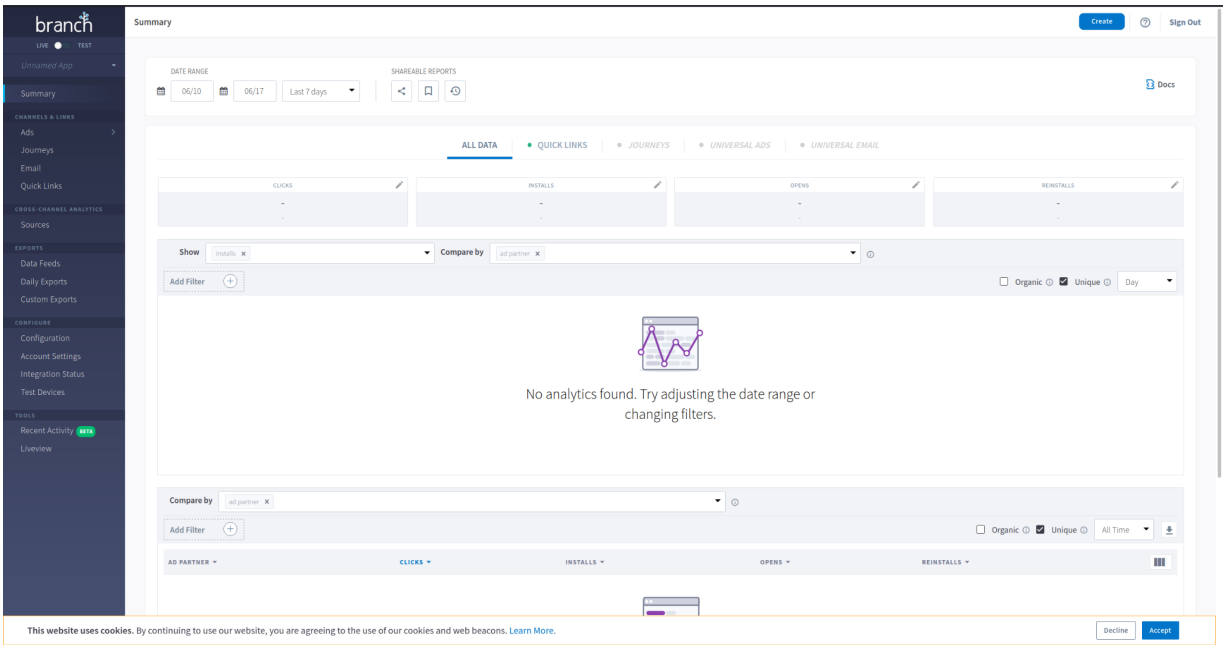

Рисунок 1.9 - Особистий кабінет користувача

При спробі створити посилання отримуємо спливаюче вікно, яке повідомляє про умови оплати (Рисунок 1.10) . Приблизно оцінюючи витрати на оренду обчислювальних машин та їх пропускну здатність можна зробити попередній висновок, що при правильному маркетингу подібні системи є прибутковими. Також варто звернути увагу на можливість підключити тестовий період. Це дуже корисна можливість, яка дозволяє користувачу спробувати сервіс, та з плином часу зробити більш обдуманий висновок, який базується вже й на досвіді використання сервісу.

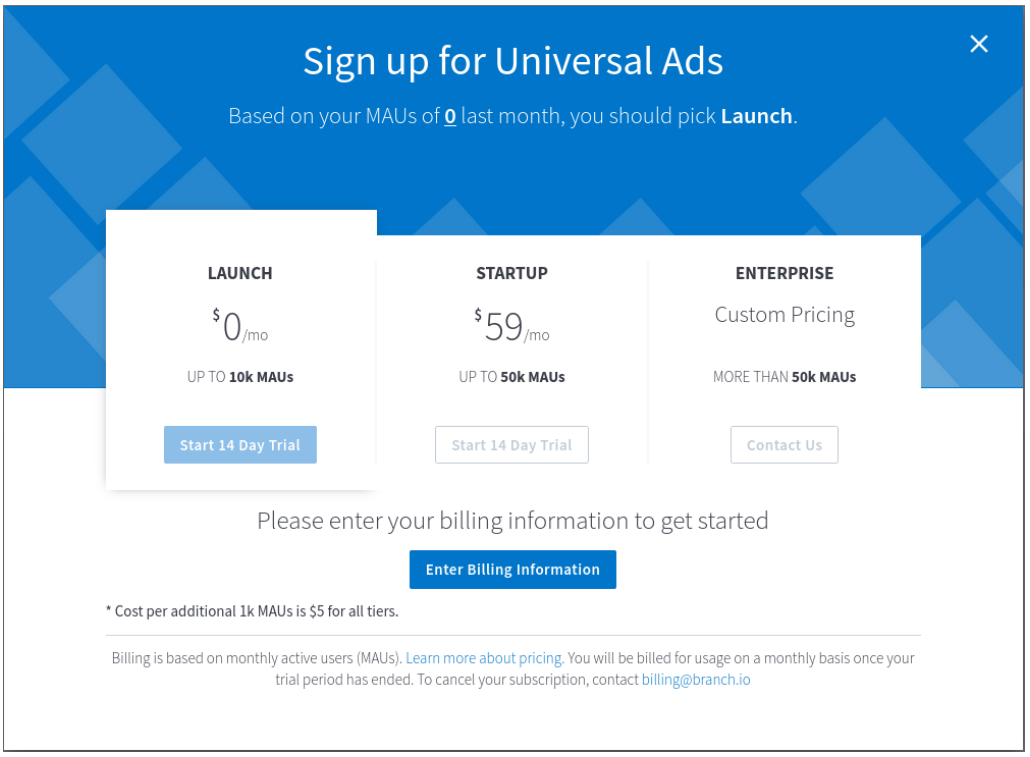

Рисунок 1.10 Вікно з вартістю користування послугами

На жаль ми не можемо протестувати функціонал без надання платіжних реквізитів. В цілях інформаційної безпеки доводиться пропустити цей крок.

У результаті огляду поданих вище двох сервісів було виявлено їх переваги та недоліки. Виходячи з цього ми можемо побудувати порівняльну характеристику поданих застосунків

Таблиця 1.1 Порівняльна характеристика застосунків

Параметри	appsflyer	branch.io
Головна сторінка	Доступна інформація для звичайного користувача, відмітки компаній гігантів які використовують застосунок	Багато потрібної інформації, коментарі клієнтів
Обширність функціоналу	Великий набір доступних можливостей,	Дуже обширний функціонал

	інтеграція з багатьма іншими сервісами компанії	
Доступність для інтеграції з зовнішніми сервісами	Відсутня	Відсутня.
Дружність до користувача	На високому рівні, але потребує зусиль на початку роботи з застосунком	Дуже зручний кабінет з безліччю статистики та можливостей, але досить важкий для розуміння новачками

1.2 Постановка задачі

Після вивчення двох найбільш популярних застосунків доступних широкому колу користувачів було виявлено ряд переваг та недоліків. Для того, щоб створити якомога затребуваний застосунок варто звернути увагу саме на можливість інтеграції з сторонніми сервісами. Для того, щоб отримати зручний та потрібен застосунок варто виконати наступні складові

1) Вивчення мови програмування Python та її асинхронних фреймворків для реалізація WEB-серверу який міг би виконувати максимальну кількість операцій в одиницю часу, порівняння з доступними аналогами
2) Вибір Системи Управління Баз Даних для використання в застосунку та проектування зв’язків між сутностями Бази Даних
3) Створення WEB-застосунку для реалізації технології на соціальній мережі Instagram
4) Запуск застосунку на сервері
5) Тестування коректної роботи застосунку

2 ВИБІР МЕТОДІВ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

2.1 Вибір мови програмування

На даний момент існує декілька основних мов програмування для реалізацізації застосунку. Визначимо основні вимоги до мови програмування та проведемо огляд основних претендентів.

Список основних критеріїв при виборі мови програмування:

1. Рейтинг за версію TIOBE (ресурс який періодично оцінює популярність мов програмування серед професійних розробників)
2. Поріг входу та підтримка спільнотою
3. Швидкодія
4. Доцільність використання для створення WEB - сервісу

Розглянемо основних претендентів для створення сервісу:

1. JavaScript - Займає 7 позицію за рейтингом TIOBE, потребує глибоких знань для ефективного використання, має розвинуту спільноту на непогану швидкодію, використовується в основному в Frontend та Backend розробці
2. C++ - Займає 4 позицію за рейтингом TIOBE, потребує глибоких знань в теорії алгоритмів, Комп’ютерних науках та самій мові, має достатньо велику спільноту, є одним з лідерів по швидкодії серед мов програмування. Застосовується зазвичай для розробки драйверів, операційних систем, комп'ютерних програм.
3. Python - займає 2 стрічку в рейтингу TIOBE, дружній до новачків, має велику та дружню спільноту яка підтримується компаніями-лідерами в сфері програмування, при використання асинхронного підходу дозволяє досягти високих результатів в продуктивності WEB-застосунків, має велику кількість зручних фреймворків для створення WEB-застосунків

Зважаючи на рейтинг серед професійних розробників, який точно оцінює зручність мови в розробці, підтримку спільноти та розвиток мови та його можливості зупинимося саме на ньому. Також варто зазначити, що Python дозволяє інтегрувати доповнення і бібліотеки написані на мовах програмування C\C++, дружність до новачків, адже він використовує інтуїтивно зрозумілий синтаксис. Має підтримку багатьох архітектурних стилів розробки як функціональне програмування і ООП(Об’єктно-Орієнтоване Програмування) та велику кількість зручних WEB фреймворків.

2.2 Вибір фреймворку

При виборі фреймворку варто відразу поділити їй за двома освноними принципа

1. Принцип виконання коду (Асинхронний або синхронний)
2. Масштаб (Мікрофреймворки та ті, що мають дуже великий функціонал)

   За принципом виконання коду перевагою асинхронного підходу є його швидкодія, завдяки так званому не блокуючому очікуванню операцій вводу/виводу вдається досягти високої продуктивності. Розглянемо різницю на найбільш типовому приколаді - роботою з базами даних. При синхронному варіанті клієнт(застосунок який працює з БД) робить запит, чекає відповіді, віддає відповідь користувачу і повторює знову. При асинхронному підході клієнт робить відразу декілька запитів, потім перевіряє доступність даних в запиті і при наявності віддає користувачу та створює новий запит керуючись максимально можливим числом одночасних запитів.

За масштабом можна приблизно розділити на два типи:

1. Мікрофреймворки - ті, що дають основні можливості для роботи з веб-запитами та залишають реалізацію додаткового функціоналу програмісту. Вони дають можливість створювати функціонал майже з нуля, та мають невелику кодову базу. Наприклад Django
2. Так звані Фулл-стак фреймворки одразу містять в собі деякі доповнення та розширення, наприклад ORM (Object-Relational-Mapping) - систему для зручної роботи з Базами Даних, вбудовані системи авторизації, тощо. Яскравими прикладами є Flask, aiohttp.

В цьому випадку вибираємо категорію мікрофреймворків, адже маємо не досить не типовий функціонал та потребуємо широких можливостей для самостійної розробки. Також при цьому загальний розмір сервісу буде менше і працювати він буде швидше зважаючи на кількість виконуваного коду.

Для створення WEB-серверу був обраний асинхронний мікрофреймворк aiohttp. Оскільки він створювався розробника Python, має велику спільноту, зручну інтеграцію з мовою програмування Python та зручну документацію

2.3 Інструменти для розробки

Зазвичай достатньо просто текстового редактору який зміг би відкривати файли з програмним кодом, але для підвищення ефективності були створені так звані IDE - В перекладі з англійської інтегровані середовища розробки. Для Python це Pycharm. Всередині він містить відразу декілька важливих інструментів:

1. Редактор тексту з автодоповнення, підказками і зручним кольоровим форматування програмного коду
2. Інтеграцію з інтерпретатором Python для зручного запуску скриптів

3. Відладчика - інструмента який дозволяє виконувати програмний код стрічка за стрічкою та одразу слідкувати як змінюються змінні. Дуже сильно спрощує пошук помилок.
4. Зручний для розробки, вбудований файловий менеджер.

2.4 Вибір Системи Управління Баз Даних

СУБД (Система Управління Базами Даних) - спеціальний програмний комплекс націлений на зручне збереження, додавання та обробку даних

Враховуючи вимоги до застосунку було обрано PostgreSQL. Вона працює з реляційною базою даних, що дозволяє поєднувати за допомогою ідентифікаторів різні сутності, будувати зв’язки між ними. Також вона є однією з найбільш популярних на даний момент та використовує SQL як мову написання запитів, яка є безумовним лідером. Ця СУБД дозволяє гнучко налаштовувати правда доступу, та має асинхронні драйвери доступи, які дозволяють виконувати одночасно декілька запитів, що є дуже важливо в нашому випадку

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Створення структури програмного коду та допоміжних інструментів

Для початку активуємо віртуальне оточення та встановимо всі потрібні нам бібліотеки та розширення. Віртуальне оточення потрібно для того, щоб всі версії бібліотек не конфліктували між собою на різних проектах. Тобто створюється спеціальне оточення всередині якого існують всі необхідні залежності та ніяким чином не впливають на інші такі ж оточення.
Це можна зробити командою : “. venv/bin/activate” (Рисунок 3.1)

```
vladw@vladw-UX430UNR:~/PycharmProjects/libero$ . venv/bin/activate
(venv) vladw@vladw-UX430UNR:~/PycharmProjects/libero$
```

Рисунок 3.1 - Активація віртуального оточення

В ході розробки було створено наступному структуру верхнього рівня (Рисунок 3.2)

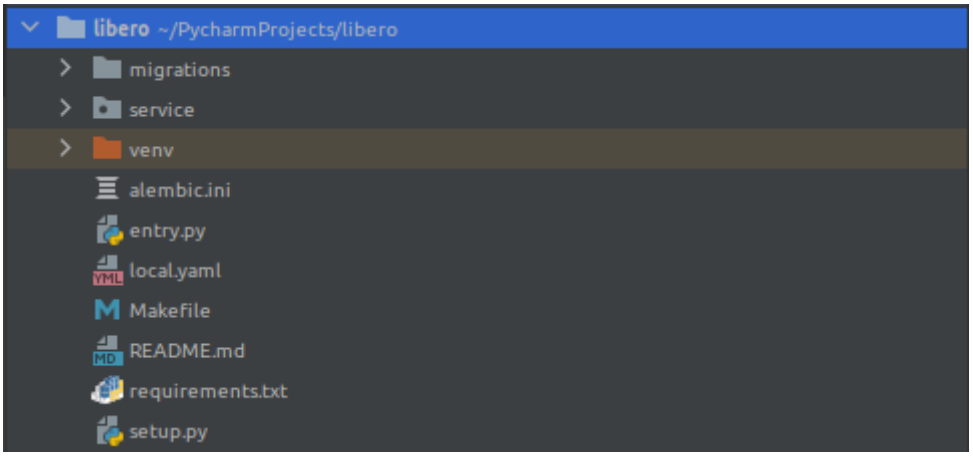

Рисунок 3.2 - Структура верхнього рівня

Розберемо складові:

1. Папка migrations - місце де будуть зберігатися всі міграції (зміни структури БД)
2. Папка service - наш WEB сервіс з всією логікою
3. Папка venv - Місце де зберігаються всі встановлені нами бібліотеки та пакети (доповнення)
4. Файл .python-version - загальноприйнятий формат файлу з версію використаної мови програмування
5. Файл alembic.ini - налаштування для пакету alembic саме він буде генерувати за запускати міграції.
6. Файл entry.py - вхідна точка нашого застосунку, дозволяє запускасти сервіс з конфігурацією в залежності від переданих аргументів
7. Файл local.yaml - конфігурація для локальної розробки, завдяки цьому файлу ми зможемо перевизначти на різних машинах деякі глобальні змінні наприклад доступи до Бази Даних
8. Файл Makefile - дозволяє спростити використання, та описати довгі та важкі команди в більш зрозумілі та легкі
9. Файл README.md - Опис кроків для запуску та налаштування проекту
10. Файл requirements.txt - перелік всіх встановлених бібліотек з їх версіями
11. Файл setup.py - Так званий збирач проекту, дозволяє швидко налаштувати всі необхідні бібліотеки та пакети.

Тепер розглянемо файлову структуру самого сервісу (Рисунок 3.3)

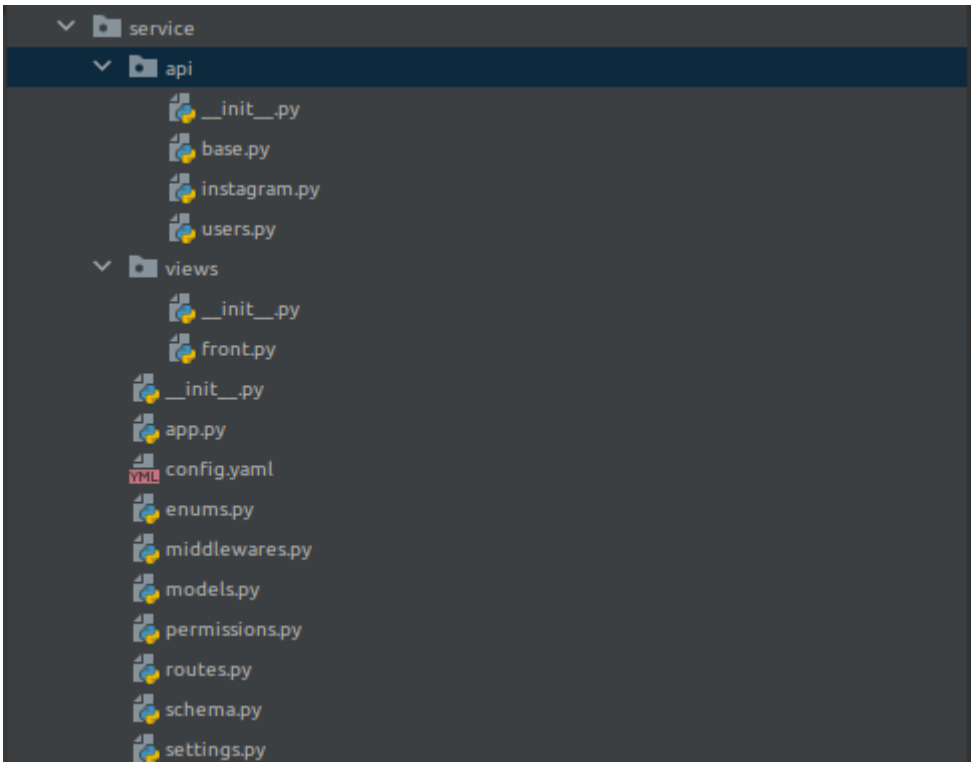

Рисунок 3.3 - Файлова структура сервісу

Коротко оглянемо елементи:

1. Папка api - каталог з усіма ендпоїнтами (ресурсами які віддають дані)
   1.1. __init__.py - спеціальний файл для ініціалізації папки як пакету Python
   1.2. base.py - Файл, який містить базовий клас ендпоїнту для генерування потрібних нам посилань
   1.3. instagram.py - містить нащадок базового класу посилань для роботи конкретно з посиланнями Instagram
   1.4. users.py - містить ендпоїнти для створення та перегляду профіля користувача, а також для генерації JWT-токену по авторизаційним даним користувача
2. Папка views
   2.1. __init__.py - спеціальний файл для ініціалізації папки як пакету Python

2.2. front.py - Спеціальний файл з ендпоїнтами для тестування доступу до БД та перевірки роботи застосунку

3. __init__.py - спеціальний файл для ініціалізації папки як пакету Python
4. app.py - містисть функцію create_app для ствроення застосунку aiohttp, підключення бази даних, налаштування шляхів для ендпоїнтів, підключення middlewares, конфігурування захисту та документації
5. config.yaml - файл з стандартними конфігурація, який повинен бути оновлений з local.yaml на кожній окремій машині
6. enums.py - Файл який містить класи колекції для даних які мають обмежену множину значень та часто використовуються в програмному коді
7. middlewares.py - визначає всіх middleware (спеціальні шари, які дозволяють обробляти запити перед обробкої їх напряму в функціях ендпоїнтів)
8. models.py - Файл з визначенням моделей таблиць бази даних
9. permissions.py - Файл з допоміжними функціями, які дозволяють перевіряти, чи має користувач доступ до певного функціоналу
10. routes.py - Файл який конфігуруєм який ендпоїнт яка функція/метод повинна виконувати
11. schema.py - Для валідації вхідних данних
12. settings.py - для додаткової конфігурації налаштуваня, дозволяє обновляти базовий конфіг з локального

Розглянемо основні пакети та бібліотеки які були встановлені для досягнення цілі

1. aiohttp - бібліотека яка надає інтерфейс асинхронного веб-серверу
2. aiohttp-middlewares - дозволяє підключати додаткові обробчити - middlewares
3. asyncpg - для асинхронної роботи з Базою Даних

4. gino - ОРМ яка дозволяє взаємодіяти з Базою Даних використовучи синтаксис мови програмування Python
5. itsdangerous - для генерації і перевірки зашифрованних даних
6. marshmallow - для створення схем валідації вхідних данних

3.2 Проектування Бази Даних

Завдяки використанню ORM Gino ми маємо можливість описати структури БД в файлі Python. Розглянемо реалізацію (Рисунок 3.4). Як може бачити було використано ООП (Об’єктно-Орієнтований Програмування) та наслідування класів для спрощення структури, та запобігання дублікуванню коду, так в класі *Base* ви визначили поля які мають бути в усіх таблицях наприклад:

1. id = Column(Integer, primary_key=True, autoincrement=True) - унікальний ідентифікатор запису, типу Integer.
2. created_at = Column(DateTime, server_default=text("NOW()")) - автоматично генерована дата створення запису
3. updated_at = Column(DateTime, server_default=text("NOW()")) - автоматично генерована дата оновлення запису, це доступно завдяки ORM яка всередині має функціонал для полей з такими іменами

Варто звернути що тут визначено *__abstract__ = True* цей параметр зазначає, що цей клас не є таблицей, він створений як інтерфейс для розширення

Розглянемо класс User, він наслідковується від Base, який ми попередньо розглянули раніше, всередині за допомогою *__tablename__ = "users"* визначено як повинна називатися таблиця пов’язана з цією моделлю (так прийнято називати подібний запис таблиці). Ця модель має наступні поля:

1. name = Column(String(50), nullable=False) - Ім’я користувача, з максимальною довжиною 50 типу Str, яка не може бути пустою

2. email = Column(String(50), nullable=False, unique=True) - Електронна пошта користувача з максимальною довжиною 50 типу Str, яка не може бути пустою, та не може повторюватися
3. password = Column(String(100), nullable=False) Поле з зашифрованим паролем користувача типу Str, до 100 символів, яке не може бути пустим

Також в цій моделі визначено метод to_dict(self), він повертає всі поля які мають в майбутньому серіалізуватися в json в форматі Dict - реалізація hash-таблиця в Python. Основним завдання цієї таблиці є збереження данних по користувачам.

Наступною розглянемо модель DeepLink, вона так само наслідкована від Base, задає назву таблиці *__tablename__ = "deep_links"* та має наступні поля:

1. slug = Column(String(30), nullable=False, unique=True) унікальний ідентифікатор посилання, за яким будуть переходити користувачі
2. target_link = Column(String(50), nullable=False) цільове посилання, куди має перейти користувач, в розумінні що це наприклад юзернейм користувача Instagram
3. user_id = Column(Integer, db.ForeignKey('users.id')) Посилання на користувача який створив посилання
4. clicked = Column(Integer, default=0) - лічильник кліків на посилання

Ця структура таблиці дозволяє реалізувати її функціонал, також можна переглянути ERD діаграму (Рисунок 3.4)

Рисунок 3.4 - ERD - діаграма бази даних

3.3 Опис Реалізації Ендпоїнтів

Розглянемо реалізацію реєстрації та авторизації користувача. Для авторизацій користувачів було вирішено використовувати JWT (JSON Web Token). Просто говорячи це зашифрований рядок який містить деякі дані про сутність. В нашому випадку це айді користувача та дата видачі токена, для того щоб обмежити час його життя. Розглянемо класс UserView (Рисунок 3.5), всередині його зареєстрований метод *post* і *get*. Метод post відповідає за реєстрацію користувачів. Варто звернути увагу, що в середині він перевіряє наявність юзерів з такою ж електронною поштою, та використовує сіль з конфіг файлу для шифрування паролів.

```python
class UserView(BaseView):

    @docs(
        tags=["Users"],
        summary="Info about current user",
        description="Get user info",
        security=[{"bearer": ["api"]}],
    )
    @response_schema(schema.UserResponse(), 200, description="Return user info")
    async def get(self):
        login_required(self.request)

        return web.json_response(self.user.to_dict())

    @docs(
        tags=["Users"],
        summary="Create user",
        description="Create new user",
    )
    @use_kwargs(schema.UserInput())
    @response_schema(schema.UserResponse(), 200, description="Creating new user")
    async def post(self):
        salt = self.request.app["config"].get('pass_salt').encode("utf-8")
        data = await self.input(schema=schema.UserInput)
        email_count = await db.select([1]).where(User.email == data["email"]).gino.scalar()
        if email_count:
            raise ValidationError("User with this email already exist")

        data["password"] = bcrypt.hashpw(data["password"].encode("utf-8"), salt).decode('utf-8')
        user = await User.create(**data)
        return web.json_response(user.to_dict())
```

Рисунок 3.5 - класс User

Також тут присутня валідація вхідних даних за допомогою schema.UserInput (Рисунок 3.6) Як можна побачити, marshmallow дозволяє дуже зручно створювати правила валідації вхідних даних, навіть має свій клас, для перевірки електронної пошти

```python
class UserInput(Schema):
    class Meta:
        ordered = True
        unknown = EXCLUDE

    name = fields.Str(required=True, validate=validate.Length(min=3, max=30))
    email = fields.Email(required=True)
    password = fields.Str(required=True, validate=validate.Length(min=7, max=15))
```

Рисунок 3.6 - Схема валідації вхідних даних для користувача

Метод *get* дозволяє отримати інформацію користувачу, який надіслав запит. Також тут присутня перевірка на авторизацію користувача.

Варто зазначити, що для обох методів написана базова документація, яка в майбутньому значно допоможе в розробці нового функціоналу

Спробуємо створити нового користувача і отримати інформацію по ньому

Створення користувача (Рисунок 3.7)

Рисунок 3.7 - Запит на створення користувача

Після відправки запиту отримаємо наступну відповідь (Рисунок 3.8)

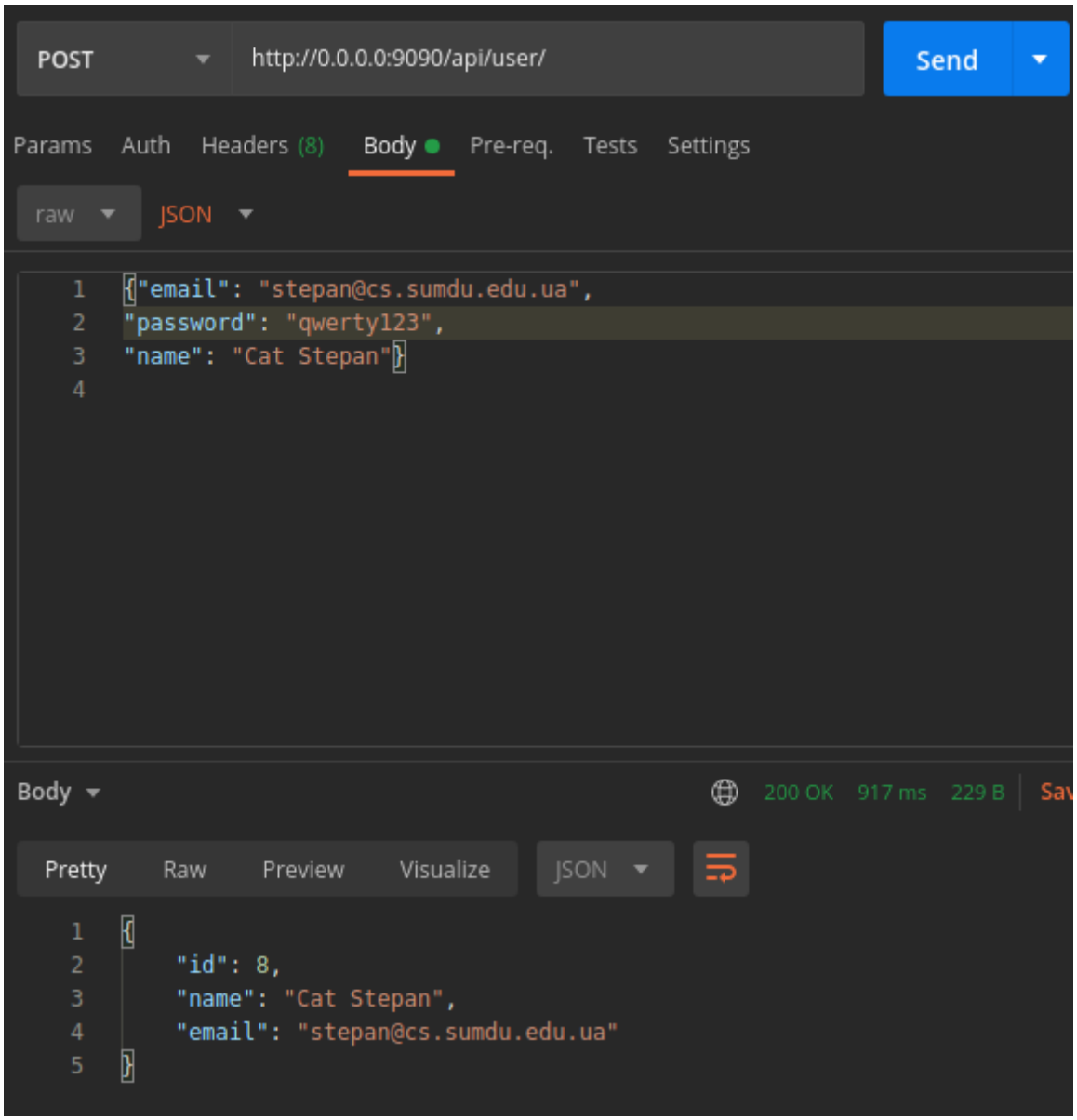

Рисунок 3.8 - Відповідь серверу на створення користувача

Як можемо бачити користувач успішно утворився на сервері

Тепер розглянемо функціонал авторизації, за нього відповідає класс LoginView який має один метод post та приймає на вхід пошту на пароль користувача, перевіряє наявність юзера, і в разі успіху повертає JWT токен в якому зашифровано айді та дата створення токена (Рисунок 3.9)

```python
class LoginView(BaseView):

    @docs(
        tags=["Users", "Auth"],
        summary="User Login",
        description="Return jwt token for user if email and password passed correctly",
    )
    async def post(self):
        data = await self.data
        user = await User.query.where(User.email == data["email"]).gino.first()
        jwt_secret = self.request.app["config"].get('jwt_secret')

        if user is None:
            raise web.HTTPBadRequest

        if bcrypt.checkpw(data["password"].encode("utf-8"), user.password.encode("utf-8")):
            return web.json_response(
                {"token": jwt.encode({"user_id": user.id, "created_at": time()}, jwt_secret, algorithm="HS256")})

```

Рисунок 3.9 - Класс авторизації користувача

Спробуємо надіслати запит на авторизацію (Отримання JWT-токена) (Рисунок 3.10)

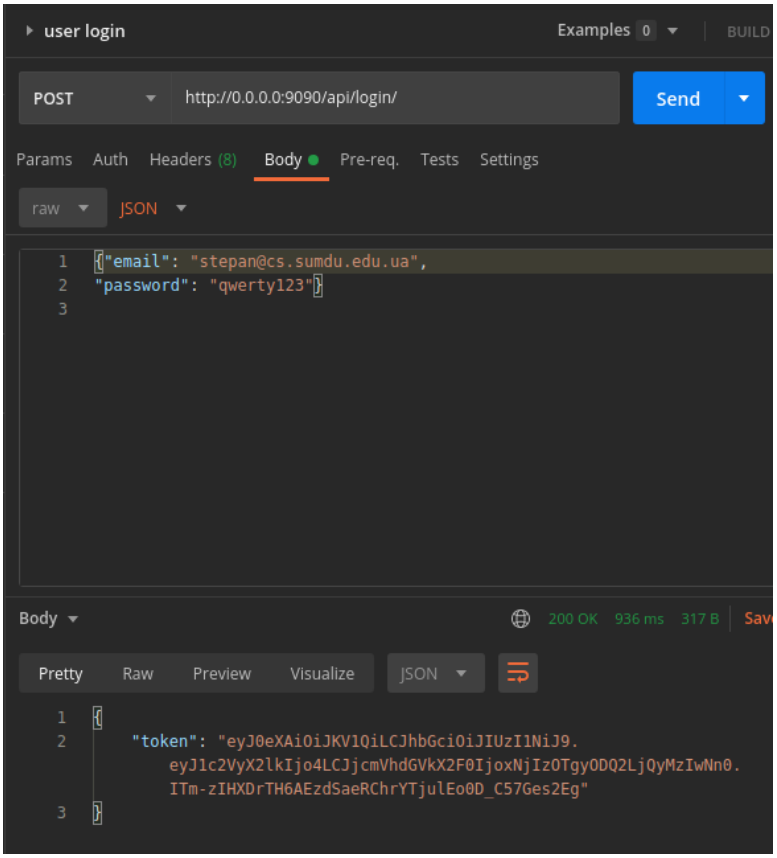

Рисунок 3.10 - Авторизація користувача

Авторизація проходить успішно, токен повертається. Роглянемо основний функціонал формування глибинних посилань на сторінку Instagram. Для цього був реалізований класс InstagramLinkView (Рисунок 3.11)

```python
class InstagramLinkView(BaseDeepLink):
    android_link = 'intent://www.instagram.com/{}/#Intent;package=com.instagram.android;scheme=https;end'
    ios_link = 'instagram://user?username={}'
    desktop_link = "https://www.instagram.com/{}/"
    service: str

    @docs(
        tags=["DeepLink", "Instagram"],
        summary="Create new deep link for instagram username",
        description="Create new deep link for instagram username",
        security=[{"bearer": ["api"]}],
    )
    @response_schema(schema.InstagramLinkCreate(), 200, description="Create deep link for instagram username")
    async def post(self):
        login_required(self.request)
        input_data = schema.InstagramLinkCreate().load(await self.data)

        deep_link = await DeepLink.create(slug=input_data.get("slug") or self.generate_slug(),
                                          target_link=input_data["user_name"],
                                          user_id=self.user.id)

        return web.json_response(deep_link.to_dict())

    async def get(self):
        slug = self.request.match_info.get("slug", None)
        link_obj = await DeepLink.query.where(DeepLink.slug == slug).gino.first()

        if not link_obj:
            raise web.HTTPForbidden

        await link_obj.update(clicked=link_obj.clicked + 1).apply()
        raise web.HTTPFound(location=await self.format_deep_link(link_obj.target_link))

    async def format_deep_link(self, user_name):
        device = await self.get_device(await self.get_user_agent())
        if device == OperationalSystem.ANDROID:
            return self.android_link.format(user_name)
        elif device == OperationalSystem.AIOS:
            return self.ios_link.format(user_name)
        return self.desktop_link.format(user_name)
```

Рисунок 3.11 - Клас для формування глибинних посилань

Він має 2 методы get який зчитує слаг з запиту, отримує по ньому об’єкт посилання з БД, в разі його відсутності повертає статус 404, в разі наявності формує посилання, та перенаправляє туди користувача. Метод post дозволяє створити нове прави для перенаправлення та зберегти його в базі даних.

Важливо відмітити, що тут перевіряється чи авторизований користувач, з допомогою схеми (Рисунок 3.12)

```python
class InstagramLinkCreate(Schema):
    class Meta:
        unknown = EXCLUDE

    user_name = fields.Str(required=True, validate=validate.Regexp(r"^[a-zA-Z0-9._]+$"))
    slug = fields.Str(required=False, validate=[validate.Regexp(r"^[a-zA-Z0-9_-]+$"), validate.Length(min=3, max=30)])
```

Рисунок 3.12 - Схема валідації вхідних даних для нового правила перенаправлення

Потім ці дані зберігаються в БД, та повертаються в вигляді JSON.

Також варто зазначити, що в класі визначені шаблони для формування посилань (Рисунок 3.13)

```python
    android_link = 'intent://www.instagram.com/{}/#Intent;package=com.instagram.android;scheme=https;end'
    ios_link = 'instagram://user?username={}'
    desktop_link = "https://www.instagram.com/{}/"
    service: str
```

Рисунок 3.13 - Шаблони для формування посилань Instagram

Давайте спробуємо створити нове правило (Рисунок 3.14)

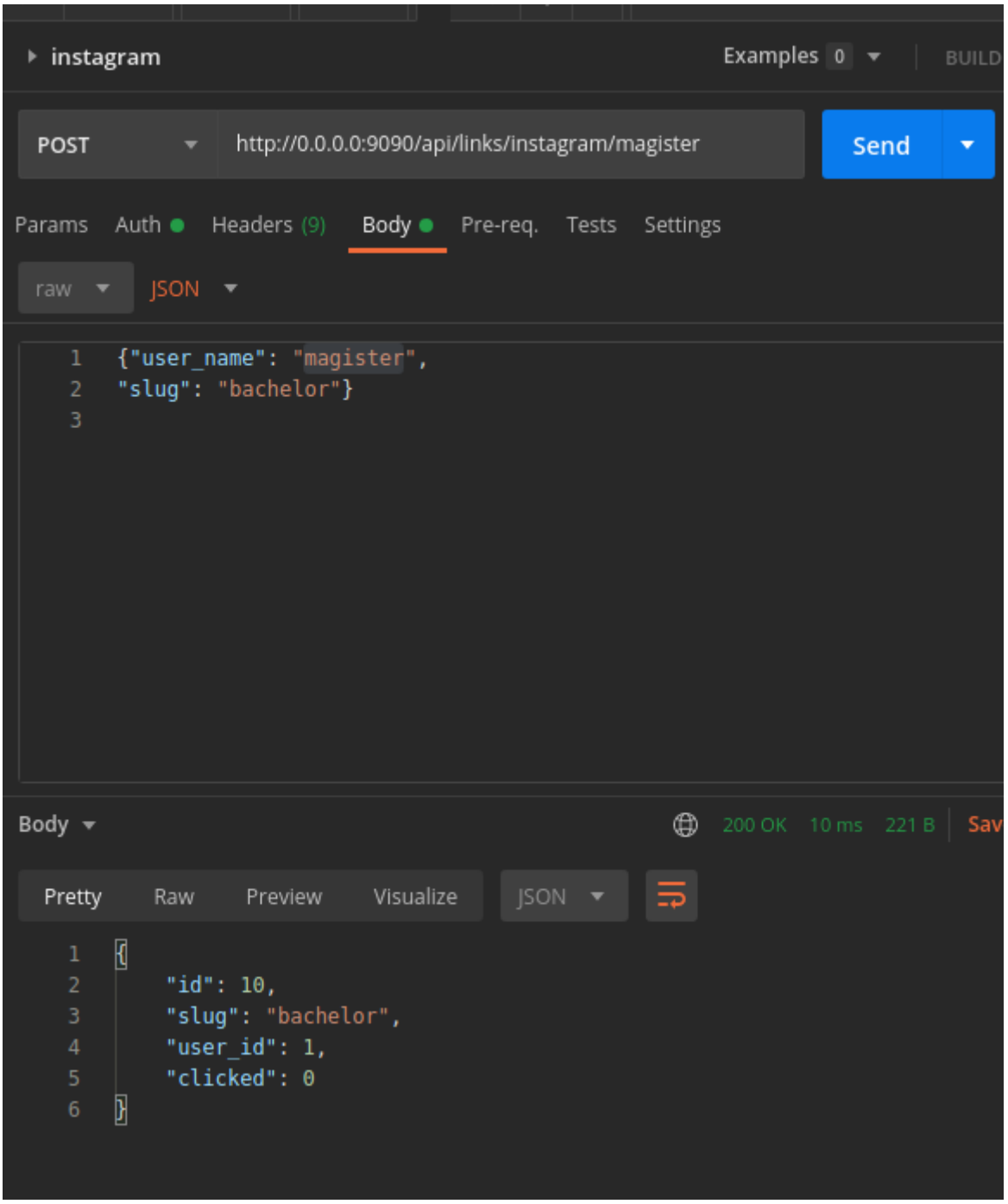

Рисунок 3.14 - Створення правила для перенаправлення

Також розглянемо перехід за згенерованим слагом (Рисунок 3.15)

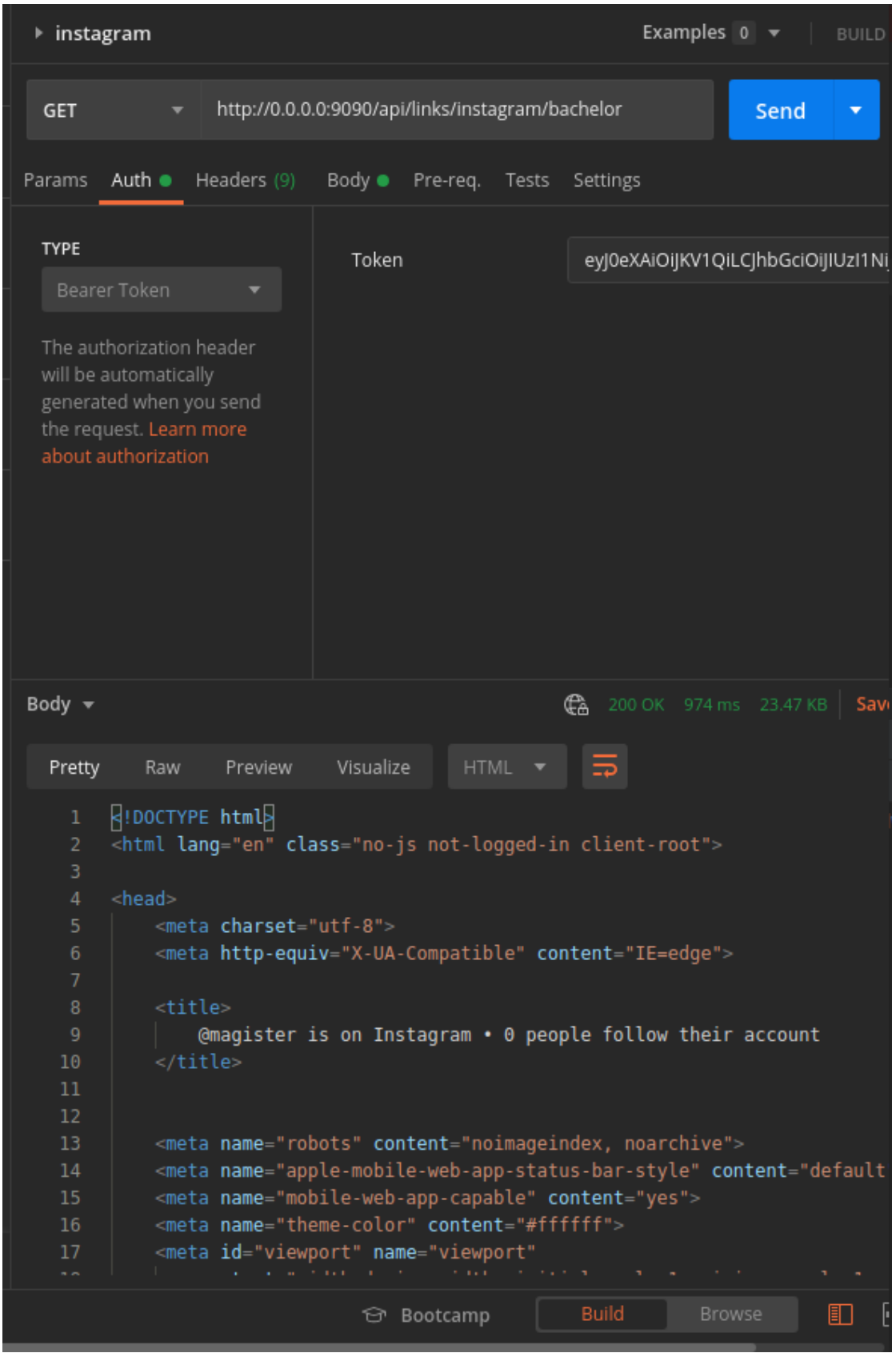

Рисунок 3.15 - Перехід за посиланням з слагом

Функціонал працює, нас перенаправлено на сторінку в інстаграм.

Якщо ми спробуємо підмінити заголовок User-Agent на ios варіант то отримуємо редірект на застосунок інстаграм (Рисунок 3.16)

Рисунок 3.16 - Перенаправлення на застосунок

Можна помітити, що нас спрямували на протокол Instagram, це означає, що сервіс правильно визначив користувача і переправив його!

Тепер треба розглянути базовий клас для формування посилань (Рисунок 3.17)

```python
class BaseDeepLink(BaseView):
    android_link: str
    ios_link: str
    desktop_link: str
    service: str

    async def get_user_agent(self):
        return self.request.headers.get('User-Agent')

    @staticmethod
    async def get_device(user_agent):
        if 'android' in user_agent.lower():
            return OperationalSystem.ANDROID
        elif 'iphone' in user_agent.lower():
            return OperationalSystem.AIOS
        elif 'ipad' in user_agent.lower():
            return OperationalSystem.AIOS
        else:
            return OperationalSystem.DESKTOP

    async def format_deep_link(self, user_name):
        raise NotImplementedError

    @classmethod
    def generate_slug(cls):
        """Generating random slugs"""
        random_str = ''.join(random.choices(string.ascii_uppercase + string.digits, k=4))
        random_str += str(int(time()))[:3]
        return random_str

```

Рисунок 3.17 - Базовий клас для формування посилань

Насамперед він містить заготовку для визначення шаблонів для формування. Потім маємо метод *get_user_agent* який зчитує інформацію про користувача з заголовків запиту.Наступний метод - *get_device,* Він в залежності від даних *user_agent* повертає потенційну операційну систему які

позначені як: *android, iphone, ipad, DESKTOP.* Метод format_deep_link - це заготовка, яка має бути визначена в кожному класі наслідникy. Метод *generate_slug* генерує випадковий слаг, якщо він не був заданий юзером. Для цього ми генеруємо рядок з 4 букв, а потім додаємо до нього останні 3 цифри з числа, яке характеризує кількість секунд, що пройшли від початку епохи.

Розберемо реалізовані middlewares (Обробчики, що дозволяють опрацювати запити, до і після основних обробчиків). Це чудовий засіб щоб реалізувати як приклад авторизацію, middleware опрацьовує запит перед основним обробчиком, перевіряє авторизацію, якщо юзер авторизований додає його об’єкт в запит. Розглянемо auth_middleware (Рисунок 3.18)

```python
@web.middleware
async def auth_middleware(request: web.Request, handler: Handler) -> web.StreamResponse:
    jwt_secret = request.app["config"].get('jwt_secret')
    jwt_lifetime = request.app["config"].get('jwt_lifetime')

    token = None

    auth_header = request.headers.get('Authorization')
    if auth_header:
        _type, token = auth_header.split(" ")
        if _type.lower() == "bearer":
            token = token

    if token is None:
        request['user'] = None
        return await handler(request)
    try:
        token_data = jwt.decode(token, jwt_secret, algorithms=["HS256"])
    except InvalidSignatureError:
        raise web.HTTPForbidden(reason="Invalid token")

    if time() - token_data["created_at"] < jwt_lifetime:
        user = await User.get(token_data["user_id"])
        request["user"] = user
        return await handler(request)
    else:
        raise web.HTTPForbidden(reason="Token expired. Login please")
```

Рисунок 3.18 - Авторизаційна middleware

Перш за все, вона читає допустимий час життя токена та сіль для його розшифрування. Потім перевіряє заголовки запиту на предмет правильно форматованого для JWT-токену. В разі відсутності токену на місці користувача повертається None - пустота. Якщо токен знайдено, здійснюється спроба розшифрувати його, і відловлюється помилка, в разі якої збуджується помилка HTTPForbidden. Останнім кроком є перевірка, чи є токен актуальним за часом, якщо ні, то знову ж збуджується така помилка, але з іншим текстом. В разі успіху запит йде до основного обробчика з об’єктом користувача.

Іншим прикладом middleware у нас є error_middleware (Рисунок 3.19). Вона кардинально відрізняється за принципом від попередньої. Якщо авторизація працює перед основним обробчиком, то middleware спрацьовую після збудження якоїсь помилка в програмному коді. Вдалося виділити основні типи помилок які сповіщають корисну інформацію користувачу та написати для їх окрему обробку. Загалом основне її призначення форматування будь-яких помилок, які можуть виникнути в ході роботи сервісу.

```python
@web.middleware
async def error_middleware(request: web.Request, handler: Handler) -> web.StreamResponse:
    try:
        return await handler(request)
    except HTTPFound as e:
        raise e
    except ValidationError as e:
        # Formatting marshmallow errors
        error = {
            "message": "Validation error", "extensions": {
                "code": "BAD_USER_INPUT",
                "validationErrors": {
                    k: ",".join(v)
                    for k, v in e.normalized_messages().items()
                },
            }}
        return web.json_response(text=ujson.dumps(error), status=400)

    except HTTPClientError as e:
        error = {"message": e.reason, "extensions": {
            "code": "HTTP_EXCEPTION",
            "status_code": e.status_code,
        }}
        return web.json_response(text=ujson.dumps(error), status=e.status_code)

    except Exception as e:
        logger.error(e)
        return web.json_response({"message": "You are found a bug! Report and get + respect"}, status=500)

```

Рисунок 3.19 - middleware для обробки помилок.

ВИСНОВКИ

Після виконання роботи можна зробити наступні висновки:

6) Вивчено мову програмування Python та асинхронний WEB-фреймворк aiohttp
7) Створено WEB застосунок який дозволяє створювати посилання в застосунок Instagram
8) Протестували роботу застосунку
9) Запустили на сервері

СПИСОК ЛІТЕРАТУРИ

1. TIOBE Index for June 2021

   [Електронний ресурс] – Режим доступу:
   https://www.tiobe.com/tiobe-index/
   Дата доступу 7.04.2021
2. Документація Python [Електронний ресурс] – Режим доступу : https://docs.python.org/3/ t – Дата доступу 6.04.2021
3. Документація Gino [Електронний ресурс] – Режим доступу : https://python-gino.org/docs/en – Дата доступу 9.04.2021
4. Документація aiohttp [Електронний ресурс] – Режим доступу : https://wibe.team/sozdanie-bota-v-telegram/ - Дата доступу : 27.04.2021

ДОДАТОК А

Програмна реалізація

base.py

```
import random
import string
from time import time

from aiohttp import web

from service.enums import OperationalSystem
from service.models import User


class BaseView(web.View):

@property
def user(self) -> User:
        if self.request.get("user"):
        return self.request.get("user")
        raise web.HTTPUnauthorized

@property
async def data(self):
        return await self.request.json()

async def input(self, schema) -> dict:
        data = await self.data
        return schema().load(data)
```

```
class BaseDeepLink(BaseView):
android_link: str
ios_link: str
desktop_link: str
service: str

async def get_user_agent(self):
        return self.request.headers.get('User-Agent')

@staticmethod
async def get_device(user_agent):
        if 'android' in user_agent.lower():
        return OperationalSystem.ANDROID
        elif 'iphone' in user_agent.lower():
        return OperationalSystem.AIOS
        elif 'ipad' in user_agent.lower():
        return OperationalSystem.AIOS
        else:
        return OperationalSystem.DESKTOP

async def format_deep_link(self, user_name):
        raise NotImplementedError

@classmethod
def generate_slug(cls):
        """Generating random slugs"""
```

```
            random_str = ''.join(random.choices(string.ascii_uppercase +
string.digits, k=4))
            random_str += str(int(time()))[:3]
            return random_str
```

instagram.py

```
from aiohttp import web
from aiohttp_apispec import response_schema, docs

from service import schema
from service.api.base import BaseDeepLink
from service.enums import OperationalSystem
from service.models import DeepLink
from service.permissions import login_required


class InstagramLinkView(BaseDeepLink):
android_link =
'intent://www.instagram.com/{}/#Intent;package=com.instagram.android;scheme=
https;end'
ios_link = 'instagram://user?username={}'
desktop_link = "https://www.instagram.com/{}/"
service: str

@docs(
        tags=["DeepLink", "Instagram"],
        summary="Create new deep link for instagram username",
        description="Create new deep link for instagram username",
        security=[{"bearer": ["api"]}],
)
```

```
        @response_schema(schema.InstagramLinkCreate(), 200,
description="Create deep link for instagram username")
        async def post(self):
                login_required(self.request)
                input_data = schema.InstagramLinkCreate().load(await self.data)

                deep_link = await DeepLink.create(slug=input_data.get("slug") or
self.generate_slug(),
                                        target_link=input_data["user_name"],
                                        user_id=self.user.id)

                return web.json_response(deep_link.to_dict())

        async def get(self):
                slug = self.request.match_info.get("slug", None)
                link_obj = await DeepLink.query.where(DeepLink.slug ==
slug).gino.first()

                if not link_obj:
                raise web.HTTPNotFound

                await link_obj.update(clicked=link_obj.clicked + 1).apply()
                raise web.HTTPFound(location=await
self.format_deep_link(link_obj.target_link))

        async def format_deep_link(self, user_name):
                device = await self.get_device(await self.get_user_agent())
                if device == OperationalSystem.ANDROID:
                return self.android_link.format(user_name)
```

```
            elif device == OperationalSystem.AIOS:
            return self.ios_link.format(user_name)
            return self.desktop_link.format(user_name)
```

users.py

```
from time import time

import bcrypt
import jwt
from aiohttp import web
from aiohttp_apispec import response_schema, docs, use_kwargs
from marshmallow import ValidationError

from service import schema
from service.api.base import BaseView
from service.models import User, db
from service.permissions import login_required


class UserView(BaseView):

    @docs(
            tags=["Users"],
            summary="Info about current user",
            description="Get user info",
            security=[{"bearer": ["api"]}],
    )
    @response_schema(schema.UserResponse(), 200, description="Return user
info")
    async def get(self):
```

```
            login_required(self.request)

            return web.json_response(self.user.to_dict())

    @docs(
            tags=["Users"],
            summary="Create user",
            description="Create new user",
    )
    @use_kwargs(schema.UserInput())
    @response_schema(schema.UserResponse(), 200, description="Creating new user")
    async def post(self):
            salt = self.request.app["config"].get('pass_salt').encode("utf-8")
            data = await self.input(schema=schema.UserInput)
            email_count = await db.select([1]).where(User.email == data["email"]).gino.scalar()
            if email_count:
            raise ValidationError("User with this email already exist")

            data["password"] = bcrypt.hashpw(data["password"].encode("utf-8"), salt).decode('utf-8')
            user = await User.create(**data)
            return web.json_response(user.to_dict())


    class LoginView(BaseView):

    @docs(
```

```
            tags=["Users", "Auth"],
            summary="User Login",
            description="Return jwt token for user if email and password passed
correctly",
        )
        async def post(self):
            data = await self.data
            user = await User.query.where(User.email ==
data["email"]).gino.first()
            jwt_secret = self.request.app["config"].get('jwt_secret')

            if user is None:
            raise web.HTTPBadRequest

            if bcrypt.checkpw(data["password"].encode("utf-8"),
user.password.encode("utf-8")):
            return web.json_response(
                    {"token": jwt.encode({"user_id": user.id, "created_at": time()},
jwt_secret, algorithm="HS256")})
```

front.py

```
        from aiohttp import web

        from service.models import db


        async def frontpage(request):
        return web.json_response({"hello": "world"})
```

```
async def healthcheck(request):
await db.status("select 1;")
return web.json_response({"status": "ok"})
```

app.py

```
import logging
from typing import Dict, Any

import ujson
from aiohttp import web
from aiohttp.web_app import Application

__all__ = ("create_app",)

from aiohttp_apispec import setup_aiohttp_apispec

from service.middlewares import get_middlewares
from service.models import db
from service.routes import setup_routes

logger = logging.getLogger(__name__)



async def create_app(config: Dict[str, Any]) -> Application:
# Create application with setting to allow to upload files up to 10 MB
app = web.Application(
        client_max_size=config["uploads_max_size"],
        middlewares=get_middlewares(),
```

```
)
# Store config
app["config"] = config

# Setup for development
if config["debug"]:
        import aiohttp_debugtoolbar

        aiohttp_debugtoolbar.setup(app, intercept_redirects=False)

setup_routes(app)

security = {
        "bearer": {
        "type": "oauth2",
        "flow": "password",
        "tokenUrl": "/auth/token",
        # "authorizationUrl": "/auth/authorize",
        "scopes": {
                "api": "Basic personal access",
                "author": "Grants access to user operations",
        },
        }
}

setup_aiohttp_apispec(
        app=app,
        title="API Documentation",
        version="v1",
```

```
        securityDefinitions=security,
        url="/api-doc/swagger.json",
)

# setup db connection
await db.set_bind(
        app["config"]["database_uri"],
        echo=app["config"]["debug"],
        json_serializer=ujson.dumps,
        json_deserializer=ujson.loads,
)

return app
```

config.yaml

```
uploads_max_size: 10485760 # 10 MiB

debug: false

database_uri: postgresql://postgres:postgres@localhost:5432/linker
secret: jT14CKi8nMadyf8OFNfZDzsbeuNriUXGMHmg15W1g0w=
pass_salt: "$2b$14$2Rc.y4fWmvKaxg5MoW95D."



jwt_secret: llgbOIgUlC
jwt_lifetime: 7200 # life time fo jwt token

token_max_age: 360000000000000000000000000000000000000000
```

enums.py

```
import enum


class Services(enum.Enum):
INSTAGRAM = "INSTAGRAM"
FACEBOOK = "FACEBOOK"


class OperationalSystem(enum.Enum):
AIOS = enum.auto()
ANDROID = enum.auto()
DESKTOP = enum.auto()
```

middlewares.py

```
import logging
from time import time

import jwt
import ujson
from aiohttp import web
from aiohttp.web_exceptions import HTTPClientError, HTTPFound
from aiohttp.web_middlewares import normalize_path_middleware
from aiohttp_middlewares.annotations import Handler
from jwt import InvalidSignatureError
from marshmallow import ValidationError

from service.models import User

logger = logging.getLogger(__name__)
```

```
    @web.middleware
    async def auth_middleware(request: web.Request, handler: Handler) ->
web.StreamResponse:
    jwt_secret = request.app["config"].get('jwt_secret')
    jwt_lifetime = request.app["config"].get('jwt_lifetime')

    token = None

    auth_header = request.headers.get('Authorization')
    if auth_header:
        _type, token = auth_header.split(" ")
        if _type.lower() == "bearer":
        token = token

    if token is None:
        request['user'] = None
        return await handler(request)
    try:
        token_data = jwt.decode(token, jwt_secret, algorithms=["HS256"])
    except InvalidSignatureError:
        raise web.HTTPForbidden(reason="Invalid token")

    if time() - token_data["created_at"] < jwt_lifetime:
        user = await User.get(token_data["user_id"])
        request["user"] = user
        return await handler(request)
    else:
```

```
                raise web.HTTPForbidden(reason="Token expired. Login please")


        @web.middleware
        async def error_middleware(request: web.Request, handler: Handler) ->
web.StreamResponse:
        try:
                return await handler(request)
        except HTTPFound as e:
                raise e
        except ValidationError as e:
                # Formatting marshmallow errors
                error = {
                "message": "Validation error", "extensions": {
                        "code": "BAD_USER_INPUT",
                        "validationErrors": {
                        k: ",".join(v)
                        for k, v in e.normalized_messages().items()
                        },
                }}
                return web.json_response(text=ujson.dumps(error), status=400)

        except HTTPClientError as e:
                error = {"message": e.reason, "extensions": {
                "code": "HTTP_EXCEPTION",
                "status_code": e.status_code,
                }}
                return web.json_response(text=ujson.dumps(error),
status=e.status_code)
```

```
        except Exception as e:
                logger.error(e)
                return web.json_response({"message": "You are found a bug! Report
and get + respect"}, status=500)


        def get_middlewares():
        return (normalize_path_middleware(append_slash=True,
merge_slashes=True),
                error_middleware,
                auth_middleware)
```

models.py

```
        from gino import Gino
        from sqlalchemy import Column, Integer, DateTime, text, String

        db = Gino()


        class Base(db.Model):
        __abstract__ = True
        id = Column(Integer, primary_key=True, autoincrement=True)
        created_at = Column(DateTime, server_default=text("NOW()"))
        updated_at = Column(DateTime, server_default=text("NOW()"))


        class User(Base):
        __tablename__ = "users"
```

```
name = Column(String(50), nullable=False)
email = Column(String(50), nullable=False, unique=True)
password = Column(String(100), nullable=False)

def to_dict(self):
        return {
        "id": self.id,
        "name": self.name,
        "email": self.email
        }


class DeepLink(Base):
__tablename__ = "deep_links"

slug = Column(String(30), nullable=False, unique=True)
target_link = Column(String(50), nullable=False)
user_id = Column(Integer, db.ForeignKey('users.id'))
clicked = Column(Integer, default=0)

def to_dict(self):
        return {
        "id": self.id,
        "slug": self.slug,
        "user_id": self.user_id,
        "clicked": self.clicked
        }
```

permissions.py

```
from aiohttp import web
```

```
from service.models import User


def login_required(request: web.Request):
"""Check client authorization"""
if request.get("user") is None:
        raise web.HTTPForbidden(reason="Login required")


def is_owner(obj, user: User):

if obj.user_id != user.id:
        raise web.HTTPForbidden(reason="You are not owner of this file")
```

routes.py

```
from aiohttp import web

__all__ = ('setup_routes',)

from yarl import URL

from service.api import users, instagram
from service.views import front


def setup_routes(app: web.Application) -> None:
# frontend
app.router.add_route("GET", "/", front.frontpage, name="index")
app.router.add_route("GET", "/status/", front.healthcheck)
```

```
    # API
    base_api_url = URL("/api/")
    app.router.add_view(to_path(base_api_url / "user"), users.UserView)
    app.router.add_view(to_path(base_api_url / "login"), users.LoginView)

    # Deep Links
    app.router.add_view(to_path(base_api_url / "links" / "instagram" / "{slug}"),
instagram.InstagramLinkView)


    def to_path(url: URL, *, has_trailing_slash: bool = True) -> str:
    """Convert URL instance into string path, suitable for aiohttp.web router.

    When `has_trailing_slash` is `True` - append trailing slash for URL, if it not
    already appended.
    """
    # Do not append trailing slash if it already added
    if url.parts[-1]:
        url = url / "" if has_trailing_slash else url
    return url.human_repr()
```

schema.py

```
    from marshmallow import Schema, fields, EXCLUDE, validate


    class UserResponse(Schema):
    class Meta:
        ordered = True
```

```
        id = fields.Int()
        name = fields.Str()
        email = fields.Email()


        class UserInput(Schema):
        class Meta:
                ordered = True
                unknown = EXCLUDE

        name = fields.Str(required=True, validate=validate.Length(min=3, max=30))
        email = fields.Email(required=True)
        password = fields.Str(required=True, validate=validate.Length(min=7,
max=15))


        class InstagramLinkCreate(Schema):
        class Meta:
                unknown = EXCLUDE

        user_name = fields.Str(required=True,
validate=validate.Regexp(r"^[a-zA-Z0-9._]+$"))
        slug = fields.Str(required=False,
validate=[validate.Regexp(r"^[a-zA-Z0-9_-]+$"), validate.Length(min=3,
max=30)])
```

settings.py

```
import logging
import os
from pathlib import Path

from yaml import load

try:
from yaml import CLoader as Loader
except ImportError:
from yaml import Loader  # noqa:F401,WPS440

__all__ = ("load_config",)

logger = logging.getLogger(__name__)

CONFIG = None


def load_config(additional_conf_file: str = None):
"""Load config from YAML file(s)"""
# Load default config
default_conf_path = Path(__file__).parent / "config.yaml"
with open(default_conf_path, "r") as load_file:
        config = load(load_file, Loader=Loader)

# Load additional config
additional_config = {}
if additional_conf_file:
```

```
            additional_conf_path = Path(__file__).parent.parent /
additional_conf_file.name
            logger.info(f"Loaded additional config {additional_conf_file.name}")
            with open(additional_conf_path, "r") as load_additional_file:
            additional_config = load(load_additional_file, Loader=Loader)

    # Update default config with additional
    if additional_config:
            config |= additional_config

    # Update config with environ variables
    for env_key in config.keys():
            environ_value = os.environ.get(env_key.upper())
            if environ_value:
            config |= {env_key: environ_value}

    # add database uri to env variables
    os.environ.setdefault('database_uri', config['database_uri'])

    global CONFIG
    CONFIG = config

    return config
```

entry.py

```
import argparse
import asyncio
import logging

from alembic import command
```

```
        from alembic.config import Config
        from aiohttp import web

        from service.app import create_app
        from service.settings import load_config

        try:
        import uvloop

        asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
        except ImportError:
        logging.warning("Uvloop is not available")

        parser = argparse.ArgumentParser(
        description="Service for deep links",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        )
        parser.add_argument("--host", help="Host to listen")
        parser.add_argument("--port", help="Port to accept connections")
        parser.add_argument(
        "--reload",
        action="store_true",
        help="Reload server when code is changed. For development purposes",
        )
        parser.add_argument("--migrate", action="store_true", help="Migrate
database")
        parser.add_argument(
        "--revision", action="store_true", help="Create new migration revision"
        )
```

```
        parser.add_argument("--show", action="store_true", help="Show
migrations")
        parser.add_argument("--downgrade", action="store_true", help="Downgrade
database")
        parser.add_argument(
        "-c",
        "--config",
        type=argparse.FileType("r"),
        dest="config_file",
        help="path to configuration file",
        )

        args = parser.parse_args()
        config = load_config(args.config_file)

        if config.get("debug"):
        logging.basicConfig(level=logging.DEBUG)

        # auto reload for better developing
        if args.reload:
        logging.info("Start with autoreload")
        import aioreloader

        aioreloader.start()

        if __name__ == "__main__":
        alembic_cfg = Config("alembic.ini")

        if args.migrate:
```

```
            command.upgrade(config=alembic_cfg, revision="head")
        elif args.revision:
            message = input("Revision comment: ")
            command.revision(config=alembic_cfg, message=message,
autogenerate=True)

        elif args.show:
            command.show(config=alembic_cfg, rev="head")

        elif args.downgrade:
            revision = input("Downgrade revision (-1 for previous, Enter to skip):
")
            if revision:
            command.downgrade(alembic_cfg, revision)
        else:
            app = create_app(config=config)
            web.run_app(
            app=app,
            host=config.get("host", args.host),
            port=config.get("port", args.port),
            access_log_format=" :: %r %s %T %t",  # noqa: WPS323
            )
```

Makefile

```
# alias for virtual environment
venv/bin/activate:
   python3 -m venv venv
```

```
setup: venv/bin/activate install migrate ## project setup

install: venv/bin/activate ## install requirements
    . venv/bin/activate; pip install pip==21.0.1 wheel setuptools
    . venv/bin/activate; pip install --exists-action w -Ur requirements.txt

run: venv/bin/activate ## Local Run
    . venv/bin/activate; python entry.py --host=0.0.0.0 --port=5000



# Migration commands
revision: venv/bin/activate ## Create new db revision
    . venv/bin/activate; python entry.py --revision --config=local.yaml
migrate: venv/bin/activate ## Apply migrations
    . venv/bin/activate; python entry.py --migrate --config=local.yaml
show: venv/bin/activate ## Show migrations
    . venv/bin/activate; python entry.py --show --config=local.yaml
downgrade: venv/bin/activate ## Downgrade db migration
    . venv/bin/activate; python entry.py --downgrade --config=local.yaml

# For local development
dev: venv/bin/activate # Local run in dev mode
    . venv/bin/activate; python -X dev entry.py --host=0.0.0.0 --port=8080
--reload
```

requirements.txt

```
aiohttp==3.7.4.post0
aiohttp-apispec==2.2.1
aiohttp-middlewares==1.1.0
alembic==1.6.5
```

apispec==3.3.2

async-timeout==3.0.1

asyncpg==0.23.0

attrs==21.2.0

chardet==4.0.0

gino==1.0.1

greenlet==1.1.0

idna==3.2

itsdangerous==2.0.1

Jinja2==2.11.3

Mako==1.1.4

MarkupSafe==2.0.1

marshmallow==3.12.1

multidict==5.1.0

psycopg2==2.8.6

python-dateutil==2.8.1

python-editor==1.0.4

PyYAML==5.4.1

six==1.16.0

SQLAlchemy==1.3.24

typing-extensions==3.10.0.0

ujson==4.0.2

webargs==5.5.3

yarl==1.6.3

setup.py

```
from setuptools import setup

setup(
        name="oxm",
```

```
version="0.0.1",
description="oxm",
platforms=["POSIX"],
install_requires=[
"aiohttp>=3,<4",
],
extras_require={
"test": [
"pytest-aiohttp",
],
},
include_package_data=True,
zip_safe=False,)
```