МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет електроніки та інформаційних технологій

Кафедра комп‘ютерних наук

Кваліфікаційна робота магістра

Інформаційна технологія оцінки релевантності тестів навчання дистанційних курсів

Здобувач освіти гр. ІН.м – 13 Владислав Бабич

Науковий керівник,
доцент, к.т.н. Сергій Петров

В. о. завідувача кафедри
доцент, к.т.н. Ігор ШЕЛЕХОВ

Суми 2022

Сумський державний університет

(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*

Спеціальність *«122 - Комп'ютерні науки»*

Затверджую:

В.о. зав.кафедри _____

_____

"_____" _____ 20___р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Бабичу Владислав Юрійовичу

(прізвище, ім'я, по батькові)

1. Тема роботи *Інформаційна технологія оцінки релевантності тестів навчання дистанційних курсів*

затверджую наказом по інституту від "____" _____ 20 ___ р. № _____

2. Термін здачі здобувачем вищої закінченого роботи_____

3. Вхідні дані до роботи _____

_____

_____

_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)

1) Огляд технологій та підходів до створення серверного веб застосунку; 2) Постановка задачі та формування завдань дослідження; 3) Опис алгоритму рекомендування релевантних тестів; 4) Розробка додатку; 5) Аналіз результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____

_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата

		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	Огляд технологій, що застосовуються для розпізнавання об’єктів, машинного навчання, комп’ютерного зору		
2.	Постановка задачі та формування завдань дослідження.		
3.	Опис алгоритму розпізнавання велосипедного транспорту		
4.	Розробка додатку для розпізнавання велосипедного транспорту		
5.	Оформлення пояснювальної записки до кваліфікаційної магістерської роботи		

Студент – дипломник _____
(підпис)

Керівник проекту _____
(підпис)

РЕФЕРАТ

Записка: 32 стор., 14 рис., 1 додаток, 9 літературних джерел.

Об’єкт дослідження — Інформаційна технологія оцінки релевантності тестів навчання дистанційних курсів

Мета роботи — розробка серверного застосунку мовою програмування Python з використанням розумної технології пропонування навчальних тестів

Результати — проведений аналіз відомих методик проведення навчального тестування. З’ясовано основні проблеми та шляхи їх вирішення. Підготовано систему рекомендації навчальних тестів, яка орієнтується на найбільш ефективне навчання користувача, збирає та враховує статистику його відповідей. Система утримує інтерес користувача для підвищення його результатів. Застосунок реалізовано як серверний з використання мови програмування Python та фреймворку FastAPI

Інформаційна технологія оцінки релевантності тестів навчання дистанційних курсів, Information technology for assessing the relevance of distance learning tests, , PYTHON, FastAPI, pydantic, PostgreSQL

ЗМІСТ

ВСТУП

Починаючи приблизно з 2017 року все більше і більше навчальних програм та курсів переходять або ж засновуються з нуля на основі онлайн ресурсів. Підхід онлайн навчання став особливо актуальних з настанням епідемії covid, в зв'язку з карантинним режимом і іншими обмеженнями. Оскільки організацію онлайн навчання дає масу можливостей для покращення і оптимізації цього процесу все більше і більше теоретичних занять і тестів створюються онлайн на різних ресурсах, що дає можливість відразу отримати оцінку, побачити правильні\неправильні відповіді, пояснення та інше.

При проходженні тестів, для закріплення знань як правило є статичний список питань, який однаковий для всіх, та дуже рідко змінюється. Виходячи з цього часто студентам не комфортно проходити одні і ті ж питання, по темам в яких вони добре знаються, або ж навпаки в темах які для них занадто важкі. Що формує негативний досвід в вивченні дисципліни та знижує мотивацію студента.

Для того щоб оптимізувати процес проходження тестування мною була вигадана система, яка адаптується до рівня знань студента в різних темах, та формує список питань орієнтуючись на конкретного студента, та його знання.

Для прикладу уявимо абстрактного студента Петрика який вивчає математику молодших класів. Йому вдалося чудово засвоїти тему з прикладами на додавання\віднімання але у нього виникають труднощі з множенням та діленням. Очевидно що для цього випадку було б краще рекомендувати тести саме з тем множення та ділення, оскільки таким чином користувач зможе в більшій мірі приділити увагу саме питанням які для нього є проблемними та підтягнути знання в них, розв'язуючі саме ці завдання. Також варто не забувати що система має бути цікавою для користувача і ми не можемо рекомендувати тільки ті питання які є складними, оскільки там чином інтерес до проходження

тестів буде падати в зв’язку з негативним досвідом. Виходячи з цього ми маємо підтримувати середній показник вдало вирішених тестів на певному рівні, комбінуючі теми які є для користувача складними і ті, які є простими. Також варто зауважити що в кожній окремо взятій темі є питання різного рівня складності, отже система також має підлаштовуватися під рівень знань користувача та рекомендувати саме ті тести які йому під силу, поступово разом з прогресом користувача підвищуючи їх складність.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Дослідження актуальності проблеми

Тестування з статичним списком питань по окремій темі, які подаються одразу, порядок вирішення тестів самовільний, перевірка проходить після надання користувачем всіх відповідей.

Тестування з статичним списком питань по окремій темі, які подаються блоками, порядок вирішення тестів в блоці самовільний, перевірка проходить після надання користувачем всіх відповідей на блок.

Тестування з динамічним списком питань по окремій темі, які подаються одразу, порядок вирішення тестів самовільний, перевірка проходить після надання користувачем всіх відповідей. В разі повторного проходження список та порядок питань змінюється.

Тестування в якому кожне питання структурно винесено окремо, є можливість пропускати та повертатися до окремих запитать, результат проходження окремого тесту надається одразу після вибору в ньому варіанту відповіді.В разі повторного проходження список та порядок питань не змінюється змінюється.

Тестування в якому питання йдуть по черзі, можливості переходу між питаннями немає, результат проходження окремого питання одразу після вибору варіанта вибору. В разі повторного проходження список та порядок питань не змінюється.

Виходячи з описаних вище проблем, було зроблено висновок реалізувати власну систему рекомендації навчальних тестів. Фокус системи має бути направлений на утримання уваги і цікавості користувача та підвищення ефективності його навчання

1.2 Аналітичний огляд різних методологій навчального тестування

Перша проблему яку хочеться виділити це - постійний незмінний список питань та відповідей в тесті, після певної спроби, особливо якщо питання та відповіді подаються в тому ж порядку та форматі, в залежності від навичок користувача правильні відповіді просто запам’ятовуються орієнтуючись на розмір тексту, його місцезнаходження, ключові слова тощо. При цьому користувач не засвоює питання висвітлене в тесті, але тим не менш отримує за нього позитивну оцінку.

Друга проблема на яку було звернуто увагу в ході вивчення можливих проблем, та методів покращення тестування це великий список текстів між якими можна переключатися. В такому варіанті часто відповідь на одне питання може лежати в постановці іншого. Знову ж це не дає змогу гарно перевірити знання користувача та дає йому спосіб отримати бал вищий ніж реальна оцінка.

Третя проблема на мою думку полягає в відсутності пояснення до питань, це дуже сповільнює роботу над помилками та часто змушує користувача просто вгадувати методом виключення.

Орієнтуючись на наведені вище проблеми можна зробити висновок що найбільш ефективною системою тестування в розрізі закріплення та перевірки знань та навичок користувача є та що відповідає наступним критеріям:

1. Не дає користувачу можливості перемикатися між питаннями
2. Дає пояснення до відповідей
3. Має велику базу питань, список питань в тесті постійно змінюється
4. Порядок відповідей до окремого питання при кожній спробі проходження тесту змінюється
5. Пропонує користувачу релевантні питання з темами які є для нього проблемними

6. Утримує рівень цікавості користувача до тесту не даючи робити багато помилок поспіль

1.3 Постановка задачі

В ході огляду проблем та можливих покращень тестувальних систем, а також огляду доступних технологій та підходів було вирішено створити серверний застосунок який дає можливість створювати та редагувати список питань та тем, зберігати статистику відповідей, та на основі статистики відповідей користувача видавати релевантні питання по запиту.

Для реалізації системи було обрано наступні основні технології:

1. Мову програмування Python з версією 3.10.8 оскільки це моя основна мова програмування та найновіша стабільна її версія. Що дасть змогу швидко та якісно реалізувати застосунок уникнувши більшості незапланованих проблем
2. WEB-фреймворк FastAPI як один з найбільш поширених, та найбільш функціональних
3. poetry як систему контролю оточення при розробці та розгортанні застосунку, оскільки вона на даний момент є найбільш функціональною
4. Систему контролю баз даних PostrgreSQL як одну з найпопулярніших, стабільних та швидких

2 ОГЛЯД АРХІТЕКТУРИ ТА ВИБІР МЕТОДІВ РЕАЛІЗАЦІЇ

2.1 Архітектури реалізації мережевої взаємодії сервісів

1. REST (скор. англ. *Representational State Transfer*, «передача репрезентативного стану») - архітектурний підхід до організації між серверної взаємодії. Був створений Роєм Філдінгом одним з засновників HTTP. Підхід базується на наступних основних принципах: передача даних відбувається в обмеженій кількості основних форматів (XML, JSON, HTML), має підтримуватися кешування, відсутність залежності від стану між зв'язкою "запит-відповідь". Вважається що дотримання всіх правил забезпечує гарну масштабованність систем та дозволяє безпроблемно покращувати її в зв'язку з новими вимогами. REST використовує наступні типи повідомлень HTTP:
   1.1. POST - створення нового ресурсу
   1.2. GET - отримання інформації по ресурсу
   1.3. DELETE - видалення ресурсу
   1.4. PUT - зміна ресурсу
   1.5. PATCH - часткова зміна ресурсу

   Візуалізація взаємодії на рисунку 2.1

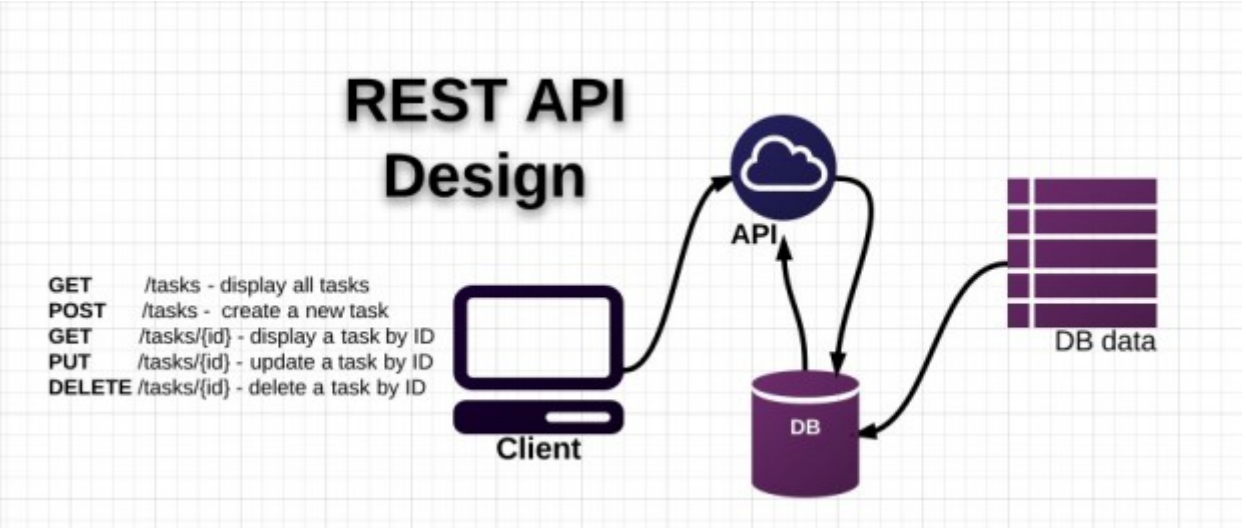

Рисунок 2.1 Візуалізація взаємодії REST підходу

2. GraphQL (скор. англ. Graph Query Language) - Підхід створений та реалізований компанією Facebook. Дозволяє вказувати які саме поля потрібні користувачу, спрощує інтеграцію декількох сервісів в один, використовує власну систему типів для опису даних, полегшує формування документації та її розуміння користувачами сервісу. Обмін даними проходить за протоколом HTTP, але при цьому використовується лише один метод POST

   Візуалізація взаємодії на рисунку 2.2

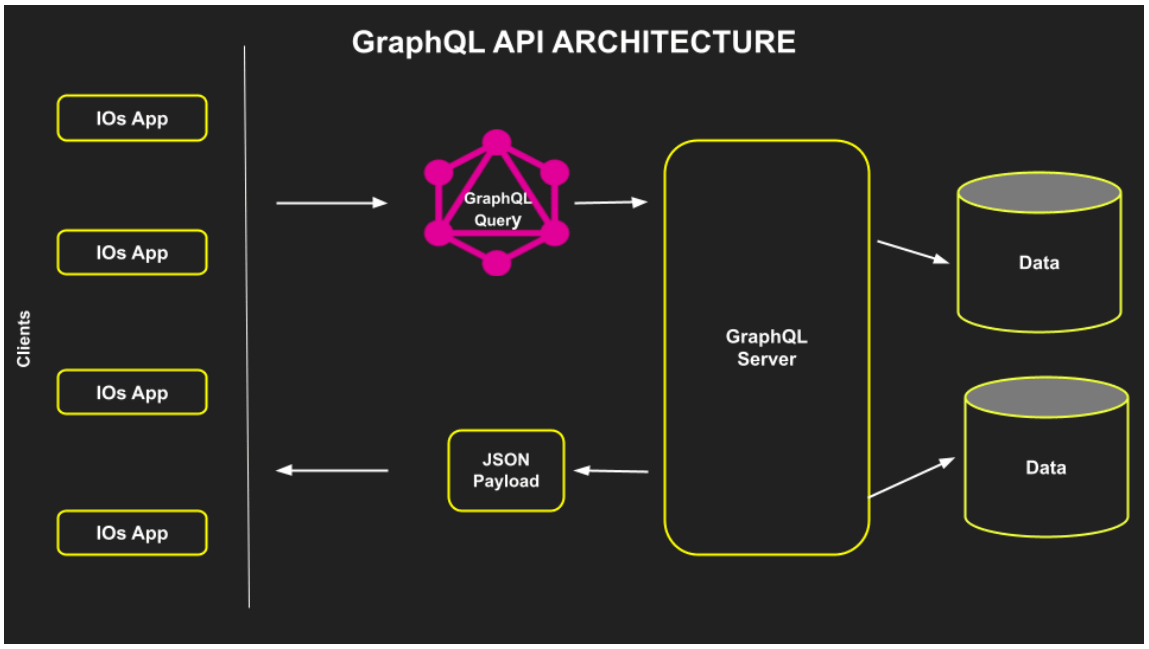

Рисунок 2.2 - Візуалізація взаємодії GraphQL підходу

3. gRPC (скор. англ. gRPC Remote Procedure Call), або Google Remote Procedure Call. Позиціонується як система віддаленого виклику процедур. Розроблена компанією Google в 2015 році. Архітектурний стиль використовує складні протоколи через що неможлива інтеграція з ним з браузеру, для цього конфігуруються окремі проксі-сервери

   Візуалізацію взаємодії можна побачити на рисунку 2.3

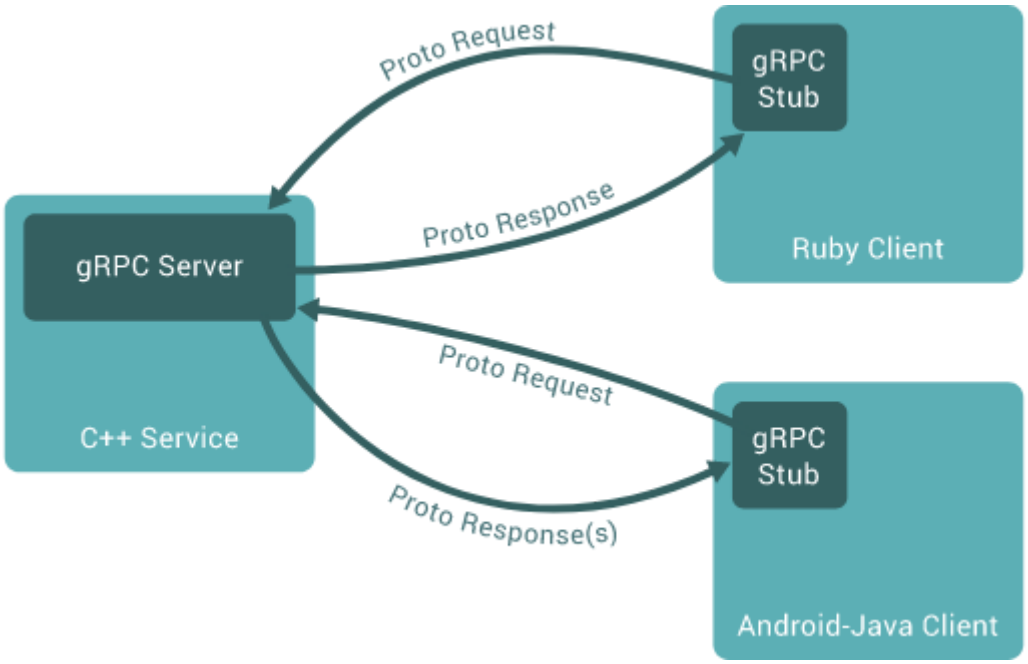

Рисунок 2.3 - Візуалізація взаємодії при gRPC підході

Кожний архітектурний підхід має свої переваги і недоліки. Для нашої системи буде доцільно обрати REST оскільки наразі він є найбільш поширеним та зрозумілим майбутнім користувачам.

2.2 Огляд доступних фреймворків для створення системи

Для початку було б доцільно дати визначення поняттю фреймворк.

Фреймворк - програмна платформа, для створення застосунків, визнающа структуру, архітектуру та підходи до створення ПЗ, що має влаштовані засоби для покращення та пришвидшення створення продукту

Розглянемо найбільш популярні для мови Python:

1. Django - найбільш поширений фреймворк для створення серверних веб застосунків, має багато вбудованих можливостей та функціоналу, суворо задає свою архітектуру та складно для нетипового переналаштування
2. flask - міні фрейворк, не диктує свою архітектуру, має мінімум вбудованих можливостей
3. aiohttp - асинхронний веб фреймворк з мінімумом вбудованого функціоналу, простий до розширення, переналаштування, відрізняється своєю швидкістю.

4. FastAPI - фреймворк з оптимальною кількістю влаштованого функціоналу, відрізняється використання pydantic для валідації та користування типами даних, автоматичною вбудованою документацією для застосунку

   Виходячи з описаних варіантів та вимог до кінцевого продукту варто зупинити свій вибір на FastAPI оскільки документація проекта значно полегшить інтеграцію нових користувачів. А швидкодія та поширеність фреймворку допоможуть в створенні застосунку

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Архітектурна модель

Для майбутнього покращення системи варто встановити наступні критерії до створеного застосунку

1. Система допускає масштабування
2. Система дозволяє безпроблемно покращувати, додавати новий та редагувати існуючий функціонал
3. Система має документацію та легка для її інтеграції сторонніми клієнтами

Для зручного користування системою, її наповненню варто встановити наступні критерії

1. Створення та перегляд тестів, категорій користувачами
2. Система авторизації для збору статистики по окремому користувачу
3. Система має працювати навіть при відсутності попередніх відповідей та статистики по користувачу

Під час створення та проектування застосунку варто встановити наступні критерії наявності майбутніх складових

1. Застосунок
2. База даних для зберігання даних
3. Система контролю середовища розробки та розгортання на сервері
4. Система налаштувань для роботи застосунку.

На рисунку 3.1 надо схематичне відображення роботи системи

Рисунок 3.1 – схематичне представлення роботи застосунку

Окремо розглянемо систему рекомендування тесту для користувача рисунок 3.2

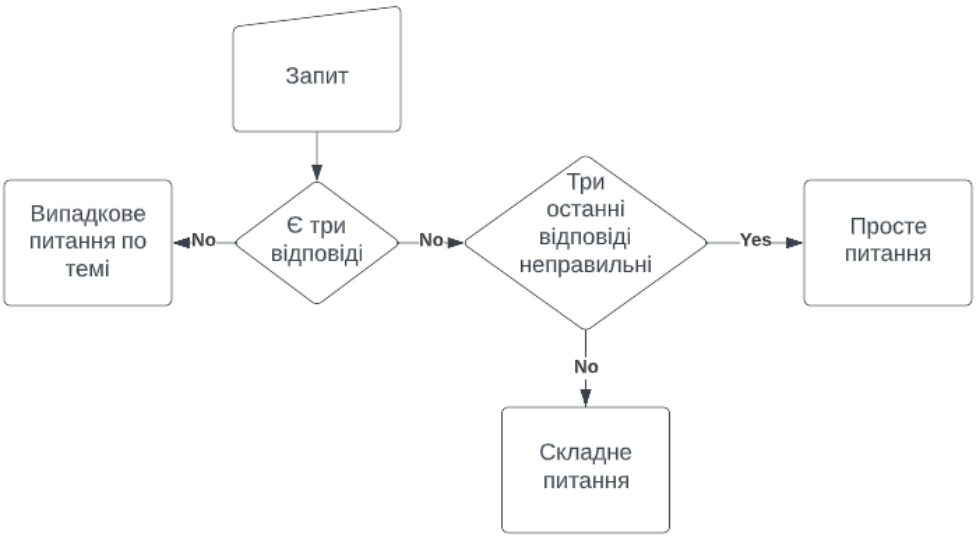

Рисунок 3.2 Система рекомендації тесту

Як ми можемо бачити з малюнку система орієнтується на статистику відповідей користувача та підбирає найбільш релевантне питання. Розглянемо детальніше кодову базу вибору питання:

```
@question_router.get("", response_model=QuestionPublicSchema)
async def question_get_relevant(category_id: Optional[int] = None,
                                user: UserFullSchema = Depends(get_current_user)) ->
QuestionPublicSchema:
    async with Session(expire_on_commit=False) as session:
        async with session.begin():
            token = session_context.set(session)
            try:
                # check user last tries
                filters = [UserQuestionAnswer.user_id == user.id]
                if category_id:
                    filters.append(Question.category_id == category_id)
                query = (
                    select(QuestionAnswer.is_correct)
```

```
                .select_from(UserQuestionAnswer)
                    .join(QuestionAnswer, UserQuestionAnswer.question_answer_id
== QuestionAnswer.id)
                .join(Question, QuestionAnswer.question_id == Question.id)
                .filter(*filters)
                .order_by(UserQuestionAnswer.created_at.desc())
                .limit(3)
            )

            last_tries = (await session.execute(query)).scalars().all()
            if len(last_tries) < 3:
                available_question_query = (
                    select(Question)
                    .order_by(func.random())
                )
                if category_id:
                                                    available_question_query =
available_question_query.filter(Question.category_id == category_id)
                available_question = (
                                                                (await
session.execute(available_question_query)).scalars().one_or_none()
                )
                return QuestionPublicSchema.from_orm(available_question)

            correct_answer = aliased(UserQuestionAnswer)
            un_correct_answer = aliased(UserQuestionAnswer)
                            filters = (UserQuestionAnswer.user_id == user.id,
QuestionAnswer.id.is_not(None))
            if category_id:
```

```
            filters = (*filters, Question.category_id == category_id)
        query = select(Question,
                ((100 * func.count(correct_answer.id)) / (
                                                func.count(correct_answer.id) +
func.count(un_correct_answer.id))).label(
                        "score")) \
            .filter(*filters) \
            .select_from(Question) \
                    .join(QuestionAnswer, QuestionAnswer.question_id ==
Question.id) \
            .join(correct_answer,
                                    and_(correct_answer.question_answer_id ==
QuestionAnswer.id,
                    QuestionAnswer.is_correct == True), isouter=True) \
            .join(un_correct_answer,
                                    and_(un_correct_answer.question_answer_id ==
QuestionAnswer.id,
                    QuestionAnswer.is_correct == False), isouter=True) \
            .group_by(Question.id) \
            .limit(1)

        correct_answer_of_last_tries = sum(
            [answer for answer in last_tries if answer == True])
        if correct_answer_of_last_tries == 0:
            query = query.order_by(desc("score"), func.random())
        else:
            query = query.order_by(asc("score"), func.random())
        question = (
            (await session.execute(query)).scalars().one_or_none()
```

```
                )
                                                question: QuestionPublicSchema =
QuestionPublicSchema.from_orm(question)
                answers = await QuestionAnswer.get_list(question_id=question.id)
                                                                question.answers =
[QuestionAnswerPublicSchema.from_orm(answer) for answer in answers]
            finally:
                session_context.reset(token)
        return question
```

Як ми можемо бачити є три основні розгалуження

1. Користувач має менше трьох відповідей по темі - пропонуємо випадкове питання. Це базова умова для нових користувачів статистику відповідей який ми не встигли зібрати
2. Користувач має три неправильні відповіді - пропонуємо просте випадкове для нього питання. Ця умова допомагає тривати фокус уваги користувача на тестуванні та підтримує його інтерес до проходження тестів а відповідно позитивно впливає на якість засвоєння знань.
3. Користувач має менше трьох останніх відповідей неправильними - пропонуємо випадково складніше питання.

   Окрему увагу варто приділити складності питання для визначення цього параметра ми беремо всі відповіді користувача по окремому питанню та рахуємо відносне значення правильних відповідей. Потім по цій ознаці ми зможемо сортувати питання за складністю для окремого користувача

3.2 Огляд створеного продукту

В результаті роботи над системою була реалізована система реєстрації на аутентифікація користувача використовуючи підхід JWT (відкритий стандарт

токенів доступу, який базується на форматі JSON) токену. Також додана можливість перегляду поточного користувача рисунок 3.3

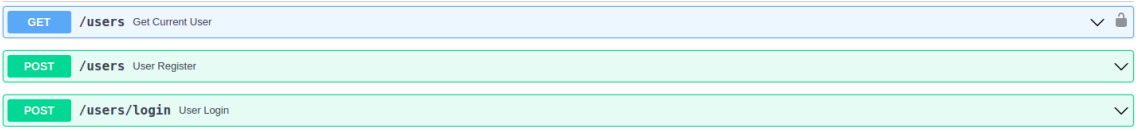

Рисунок 3.3

Методи додавання та авторизації використовують HTTP метод POST, метод отримання інформації - GET згідно специфікації REST

JWT Токен авторизації подається в вигляді зашифрованного унікального ідентифікатора користувача та має подібний вигляд

{ "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6Mn0.p6TL8xJD4O0D-w78PfhCkxc1gqtqRN3FJeKzlvJpRrU"

}

Також було створено методи на додавання, перегляд по унікальному ідентифікатору і отримання релевантного запитання Рисунок 3.4

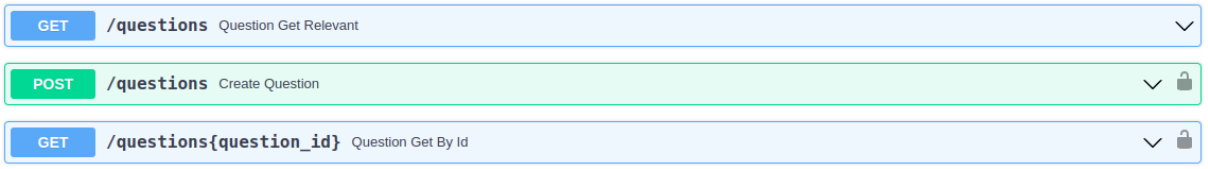

Рисунок 3.4

Відповідь сервенго застосунку на запит релевантного тесту виглядає наступним чином Рисунок 3.5

Response body


```
{
  "category_id": 1,
  "title": "string",
  "description": "string",
  "id": 3,
  "answers": [
    {
      "id": 7,
      "title": "truetrue",
      "question_id": 3
    },
    {
      "id": 8,
      "title": "falsefalse",
      "question_id": 3
    }
  ]
}
```


Рисунок 3.5

Також було створено функціонал створення та перегляду категорій тестів

Рисунок 3.6

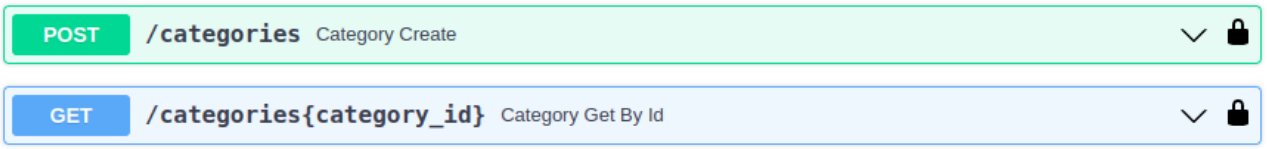

Рисунок 3.6

Виходячи з усього нам потрібно було окремо тривати варіанти відповідей до тесту та відповіді користувачів, отже було створено даний функціонал

Рисунок 3.7

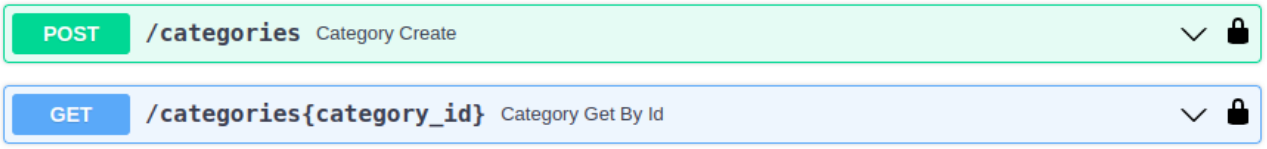

Рисунок 3.7

Окремо варто зазначити що в документацію в ході створення застосунку було додано всі можливі структури данних для спілкування між клієнтом та сервером

Рисунок 3.8

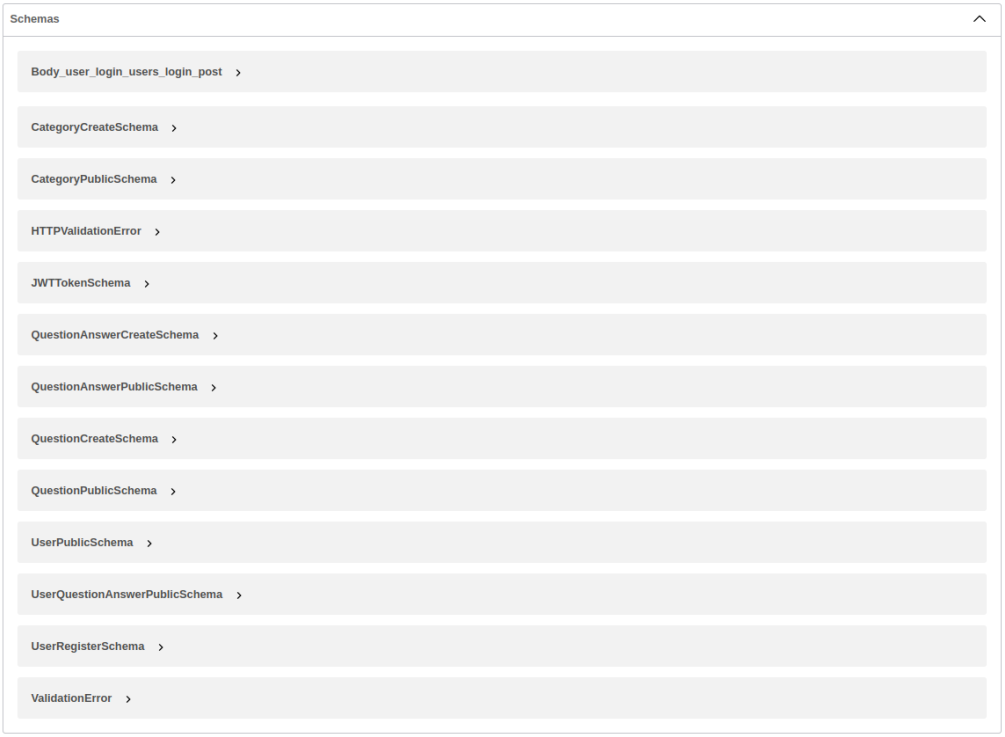

Рисунок 3.8

Опис структури даних для використання несе додатково й інформацію про валідацію кожного поля Рисунок 3.9

Рисунок 3.9

ВИСНОВКИ

Після аналізу проблеми було вирішено що майбутній продукт є актуальним, може знайти свого користувача.

Було вирішено створити власну систему для створення, акумулювання та рекомендації навчальних тестів що і було зроблено

В ході роботи було обрано актуальні та надійні технології які допомогли створити продукт без незапланованих проблем реалізації.

В ході створення продукту було використано наступні технології

- мова програмування Python;
- фреймворк FastAPI
- База даних PostgreSQL

Розроблена система повністю відповідає вимогам що було поставлені в ході проектування та аналізу

СПИСОК ЛІТЕРАТУРИ

1. Blending [Електронний ресурс] – Режим доступу: https://homepages.inf.ed.ac.uk/rbf/HIPR2/blend.htm
2. YOLO: You Only Look Once [Електронний ресурс] – Режим доступу: https://laptrinhx.com/yolo-you-only-look-once-2548734518/
3. Semantic Segmentation with Deep Learning [Електронний ресурс] – Режим доступу: https://towardsdatascience.com/semantic-segmentation-with-deep-learning-a-guide-and-code-e52fc8958823
4. FastAPI documentation [Електронний ресурс] – Режим доступу: https://fastapi.tiangolo.com/
5. pydantic documentation [Електронний ресурс] – Режим доступу: https://docs.pydantic.dev/
6. PostgreSQL documentation [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/
7. REST documentation [Електронний ресурс] – Режим доступу: https://idratherbewriting.com/learnapidoc/
8. GraphQL documentation [Електронний ресурс] – Режим доступу: https://graphql.org/learn/
9. gtpc documentation - [Електронний ресурс] – Режим доступу: https://grpc.io/docs/

ДОДАТКИ

Додаток А. Вихідний код продукту

```
db_base.base_model
import contextvars
import json
from typing import Any, Dict, Iterator, Optional, Tuple, Union
```

```
from sqlalchemy import (
  Column,
  DateTime,
  Integer,
  MetaData,
  func,
  inspect,
  select,
)
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine
from sqlalchemy.orm import declarative_base, sessionmaker


class SessionProxy:
  def __init__(self, contextvar):
    self._contextvar = contextvar

  def __getattr__(self, attr):
    context = self._contextvar.get()
    assert context
    return getattr(context, attr)


convention = {
  "ix": "ix_%(column_0_label)s",
  "uq": "uq_%(table_name)s_%(column_0_name)s",
  "ck": "ck_%(table_name)s_%(constraint_name)s",
  "fk": "fk_%(table_name)s_%(column_0_name)s",
  "pk": "pk_%(table_name)s",
```

```
}

metadata = MetaData(naming_convention=convention)
Model = declarative_base(metadata=metadata)
Session = sessionmaker(class_=AsyncSession)

session_context = contextvars.ContextVar("session")
current_session = SessionProxy(session_context)



async def db_disconnect():
  await Session.kw["bind"].dispose()



async def db_connect(database_uri, echo: bool = False):
  Session.configure(
    bind=create_async_engine(
      database_uri,
      echo=echo,
      json_serializer=json.dumps,
      json_deserializer=json.loads,
    )
  )



# Bases
class BaseSQLAlchemyModel(Model):
  __abstract__ = True
  id = Column(Integer, primary_key=True, autoincrement=True)
```

```
    def to_dict(self) -> Dict[str, Any]:
        return {c.key: getattr(self, c.key) for c in inspect(self).mapper.column_attrs}

    @classmethod
        async def _get(cls, iter_pks: Union[Any, Iterator[Any]] = None, *args,
**kwargs):
        if iter_pks is not None:
            if not isinstance(iter_pks, (tuple, list)):
                iter_pks = (iter_pks,)

            where = {
                pk.name: cond for pk, cond in zip(cls.__table__.primary_key, iter_pks)
            }
            kwargs.update(where)

        query = select(cls).filter_by(**kwargs).filter(*args).order_by(cls.id)
        result = await current_session.execute(query)
        return result

    @classmethod
    async def get(cls, iter_pks: Union[Any, Iterator[Any]], *args, **kwargs):
        result = await cls._get(iter_pks, *args, **kwargs)
        return result.scalars().one_or_none()

    @classmethod
    async def get_list(cls, *args, **kwargs):
        result = await cls._get(None, *args, **kwargs)
        return result.scalars().all()
```

```
    @classmethod
    async def create(cls, **kwargs) -> Model:
        obj = cls(**kwargs)

        current_session.add(obj)
        await current_session.flush()
        await current_session.refresh(obj)

        return obj

    async def update(self, **kwargs) -> Model:
        for key, value in kwargs.items():
            setattr(self, key, value)
        await current_session.flush()
        await current_session.refresh(self)
        return self

    async def delete(self) -> bool:
        await current_session.delete(self)
        return True

    @classmethod
    async def exists(cls, iter_pks: Union[tuple, list] = None, **kwargs) -> bool:
        """
        Check if instance exists.
        Args:
            iter_pks: primary keys in the current table (table may contains several
pks)
```

```
        Returns:
            True if instance exists, else false.
        """
        if iter_pks is not None:
            if not isinstance(iter_pks, (tuple, list)):
                iter_pks = [iter_pks]
            where = {
                pk.name: cond for pk, cond in zip(cls.__table__.primary_key, iter_pks)
            }
            kwargs.update(where)

        query = select([1]).select_from(cls).filter_by(**kwargs).exists().select()
        return bool((await current_session.execute(query)).scalars().one())

    @classmethod
    async def get_or_create(
        cls, defaults: Optional[dict] = None, **kwargs
    ) -> Tuple[Model, bool]:
        if obj := await cls.get(None, **kwargs):
            return obj, False

        defaults = defaults or {}
        return await cls.create(**kwargs, **defaults), True


class CreatedMixin(Model):
    __abstract__ = True
```

```
    created_at = Column(DateTime(timezone=True), server_default=func.now())


class UpdatedMixin(Model):
    __abstract__ = True

    updated_at = Column(DateTime(timezone=True), onupdate=func.now())
db_base.decorators
import functools

from .base_model import Session, session_context


def session_decorator(
        db_session=Session, expire_on_commit: bool = False, add_param: bool = False
):
    """
    Decorator with parameter for async session usage.
    """

    def decorated_get_session(func):
        @functools.wraps(func)
        async def wrapper(*args, **kwargs):
                async with db_session(expire_on_commit=expire_on_commit) as session:
                    async with session.begin():
                        token = session_context.set(session)
                        try:
```

```
                    if add_param:
                        args = list(args)
                        args.insert(0, session)
                    return await func(*args, **kwargs)
                finally:
                    session_context.reset(token)

        return wrapper

    return decorated_get_session
questions.handlers
from typing import Optional

from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy import select, func, alias, and_, desc, asc
from sqlalchemy.orm import aliased
from starlette import status

from app.db_base.base_model import Session, session_context
from app.db_base.decorators import session_decorator
from app.questions.models import Question, Category, QuestionAnswer,
UserQuestionAnswer
from app.questions.schemas import QuestionCreateSchema,
QuestionPublicSchema, CategoryPublicSchema, \
                                  CategoryCreateSchema, QuestionAnswerPublicSchema,
QuestionAnswerCreateSchema, UserQuestionAnswerPublicSchema
from app.users.deps import get_current_user
from app.users.schemas import UserFullSchema
```

```
category_router = APIRouter(prefix="/categories")
question_router = APIRouter(prefix="/questions")
question_answer_router = APIRouter(prefix="/questions_answers")


@category_router.post("", response_model=CategoryPublicSchema)
@session_decorator()
async def category_create(category: CategoryCreateSchema,
                          user: UserFullSchema = Depends(get_current_user)) ->
CategoryPublicSchema:
    obj = await Category.create(**dict(category))
    return CategoryPublicSchema.from_orm(obj)


@category_router.get("{category_id}",
response_model=CategoryPublicSchema)
@session_decorator()
async def category_get_by_id(category_id: int,
                             user: UserFullSchema = Depends(get_current_user)) ->
CategoryPublicSchema:
    obj = await Category.get(category_id)
    return CategoryPublicSchema.from_orm(obj)


@question_router.post("", response_model=QuestionPublicSchema)
@session_decorator()
async def create_question(question: QuestionCreateSchema, user:
UserFullSchema = Depends(get_current_user)):
    question = await Question.create(**dict(question))
```

```
        return QuestionPublicSchema.from_orm(question)


    @question_router.get("{question_id}",
response_model=QuestionPublicSchema)
    @session_decorator()
    async def question_get_by_id(question_id: int,
                                 user: UserFullSchema = Depends(get_current_user)) ->
QuestionPublicSchema:
        obj = await Question.get(question_id)
        question = QuestionPublicSchema.from_orm(obj)
        answers = await QuestionAnswer.get_list(question_id=question.id)
        if answers:
            question.answers = [QuestionAnswerPublicSchema.from_orm(obj) for obj
in answers]
        return question


    @question_router.get("", response_model=QuestionPublicSchema)
    async def question_get_relevant(category_id: Optional[int] = None,
                                    user: UserFullSchema = Depends(get_current_user)) ->
QuestionPublicSchema:
        async with Session(expire_on_commit=False) as session:
            async with session.begin():
                token = session_context.set(session)
                try:
                    # check user last tries
                    filters = [UserQuestionAnswer.user_id == user.id]
                    if category_id:
```

```
            filters.append(Question.category_id == category_id)
        query = (
            select(QuestionAnswer.is_correct)
            .select_from(UserQuestionAnswer)
                .join(QuestionAnswer, UserQuestionAnswer.question_answer_id
== QuestionAnswer.id)
            .join(Question, QuestionAnswer.question_id == Question.id)
            .filter(*filters)
            .order_by(UserQuestionAnswer.created_at.desc())
            .limit(3)
        )

        last_tries = (await session.execute(query)).scalars().all()
        if len(last_tries) < 3:
            available_question_query = (
                select(Question)
                .order_by(func.random())
            )
            if category_id:
                                                    available_question_query =
available_question_query.filter(Question.category_id == category_id)
            available_question = (
                                                                (await
session.execute(available_question_query)).scalars().one_or_none()
            )
            return QuestionPublicSchema.from_orm(available_question)

        correct_answer = aliased(UserQuestionAnswer)
        un_correct_answer = aliased(UserQuestionAnswer)
```

```
                                filters = (UserQuestionAnswer.user_id == user.id,
QuestionAnswer.id.is_not(None))
            if category_id:
                filters = (*filters, Question.category_id == category_id)
            query = select(Question,
                           ((100 * func.count(correct_answer.id)) / (
                                                   func.count(correct_answer.id) +
func.count(un_correct_answer.id))).label(
                                "score")) \
                .filter(*filters) \
                .select_from(Question) \
                               .join(QuestionAnswer, QuestionAnswer.question_id ==
Question.id) \
                .join(correct_answer,
                                         and_(correct_answer.question_answer_id ==
QuestionAnswer.id,
                           QuestionAnswer.is_correct == True), isouter=True) \
                .join(un_correct_answer,
                                          and_(un_correct_answer.question_answer_id ==
QuestionAnswer.id,
                           QuestionAnswer.is_correct == False), isouter=True) \
                .group_by(Question.id) \
                .limit(1)

            correct_answer_of_last_tries = sum(
                [answer for answer in last_tries if answer == True])
            if correct_answer_of_last_tries == 0:
                query = query.order_by(desc("score"), func.random())
            else:
```

```
                query = query.order_by(asc("score"), func.random())
            question = (
                (await session.execute(query)).scalars().one_or_none()
            )
                                                question: QuestionPublicSchema =
QuestionPublicSchema.from_orm(question)
            answers = await QuestionAnswer.get_list(question_id=question.id)
                                                        question.answers =
[QuestionAnswerPublicSchema.from_orm(answer) for answer in answers]
        finally:
            session_context.reset(token)
    return question



@question_answer_router.post("",
response_model=QuestionAnswerPublicSchema)
@session_decorator()
async def create_question_answer(question_answer:
QuestionAnswerCreateSchema,
                                 user: UserFullSchema = Depends(get_current_user)) ->
QuestionAnswerPublicSchema:
    question_answer = await QuestionAnswer.create(**dict(question_answer))
    return QuestionAnswerPublicSchema.from_orm(question_answer)



@question_answer_router.get("{question_answer_id}",
response_model=QuestionAnswerPublicSchema)
@session_decorator()
```

```
async def get_question_answer_by_id(question_id: int,
                                    user: UserFullSchema = Depends(get_current_user)) ->
QuestionAnswerPublicSchema:
    obj = await QuestionAnswer.get(question_id)
    return QuestionAnswerPublicSchema.from_orm(obj)



@question_answer_router.post("/answer/",
response_model=UserQuestionAnswerPublicSchema)
@session_decorator()
async def user_answer_question(question_answer_id: int,
                               user: UserFullSchema = Depends(get_current_user)) ->
UserQuestionAnswerPublicSchema:
    obj = await QuestionAnswer.get(question_answer_id)
    if not obj:
        raise  HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Answer not found")

     await  UserQuestionAnswer.create(question_answer_id=question_answer_id,
user_id=user.id)
    return UserQuestionAnswerPublicSchema(question_answer_id=obj.id,
                                          is_correct=obj.is_correct)
questions.models
from sqlalchemy import Column, String, Text, Integer, ForeignKey, Boolean
from sqlalchemy.orm import relationship

from app.db_base.base_model import BaseSQLAlchemyModel, CreatedMixin
from app.users.models import User
```

```
class Category(BaseSQLAlchemyModel):
    __tablename__ = "categories"

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


class Question(BaseSQLAlchemyModel):
    __tablename__ = "questions"

    title = Column(String(255), nullable=False)
    description = Column(Text)

    category_id = Column(Integer, ForeignKey("categories.id"), nullable=False)

    category = relationship("Category")


class QuestionAnswer(BaseSQLAlchemyModel):
    __tablename__ = "questions_answers"

    title = Column(String(255), nullable=False)
    is_correct = Column(Boolean, nullable=False)

    question_id = Column(Integer, ForeignKey("questions.id"), nullable=False)

    question = relationship(Question)
```

```
class UserQuestionAnswer(BaseSQLAlchemyModel, CreatedMixin):
    __tablename__ = "user_questions_answers"

     question_answer_id = Column(Integer, ForeignKey("questions_answers.id"),
nullable=False)
    user_id = Column(Integer, ForeignKey("users.id"), nullable=False)

    question_answer = relationship(QuestionAnswer)
    user = relationship(User)
questions.schemas
from typing import Optional

from pydantic import BaseModel, Field


class CategoryCreateSchema(BaseModel):
    title: str = Field(min_length=5, max_length=50)


class CategoryPublicSchema(CategoryCreateSchema):
    id: int

    class Config:
        orm_mode = True


class QuestionCreateSchema(BaseModel):
    category_id: int
    title: str = Field(min_length=5, max_length=255)
```

```
    description: str = Field(min_length=5)


class QuestionAnswerCreateSchema(BaseModel):
    title: str = Field(min_length=5, max_length=255)
    is_correct: bool
    question_id: int


class QuestionAnswerPublicSchema(BaseModel):
    id: int
    title: str = Field(min_length=5, max_length=255)
    question_id: int

    class Config:
        orm_mode = True


class QuestionPublicSchema(QuestionCreateSchema):
    id: int
    answers: Optional[list[QuestionAnswerPublicSchema]]

    class Config:
        orm_mode = True


class UserQuestionAnswerSchema(BaseModel):
    question_answer_id: int
```

```
class UserQuestionAnswerPublicSchema(BaseModel):
  question_answer_id: int
  is_correct: bool

  class Config:
     orm_mode = True

users.deps
from fastapi import Depends, HTTPException, status
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from jose import jwt
from pydantic import ValidationError

from .models import User
from .schemas import UserFullSchema, JWTTokenDataSchema,
UserPublicSchema
from ..config import config
from ..db_base.decorators import session_decorator

reuseable_oauth = OAuth2PasswordBearer(
  tokenUrl="users/login",
  scheme_name="JWT"
)


@session_decorator()
```

```
    async def get_current_user(token: str = Depends(reuseable_oauth)) ->
UserFullSchema:
        try:
            payload = jwt.decode(
                token, config.jwt_secret, algorithms=["HS256"]
            )
            token_data = JWTTokenDataSchema(**payload)

        except(jwt.JWTError, ValidationError) as e:
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="Could not validate credentials",
                headers={"WWW-Authenticate": "Bearer"},
            )

        user = await User.get(token_data.id)

        if user is None:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail="Could not find user",
            )

        return UserPublicSchema.from_orm(user)

    users.exceptions
    from dataclasses import dataclass
```

```
@dataclass
class ErrorsDetails:
  user_already_exist = "User with same email already exist"
  user_not_found = "User not found"
  invalid_password = "Password invalid"


class UserAlreadyExist(Exception):
  pass


class InvalidPassword(Exception):
  pass


class UserDoesNotExist(Exception):
  pass
```

users.handlers

```
from fastapi import APIRouter, HTTPException, Depends
from fastapi.security import OAuth2PasswordRequestForm
from starlette import status

from app.users.schemas import UserRegisterSchema, UserPublicSchema, JWTTokenSchema, UserFullSchema
from app.users.service import UserService
from . import exceptions
from .deps import get_current_user
from ..db_base.decorators import session_decorator
```

```
user_router = APIRouter(prefix="/users")


@user_router.post("", response_model=UserPublicSchema)
@session_decorator()
async def user_register(user: UserRegisterSchema) -> UserPublicSchema:
  try:
    user = await UserService.register(user)
    return user
  except exceptions.UserAlreadyExist:
      raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=exceptions.ErrorsDetails.user_already_exist)


@user_router.post("/login", response_model=JWTTokenSchema)
@session_decorator()
async def user_login(user: OAuth2PasswordRequestForm = Depends()) ->
JWTTokenSchema:
  try:
    return await UserService.login(user)
  except exceptions.UserDoesNotExist:
      raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail=exceptions.ErrorsDetails.user_not_found)
  except exceptions.InvalidPassword:
    raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail=exceptions.ErrorsDetails.invalid_password)
```

```
@user_router.get("", response_model=UserPublicSchema)
@session_decorator()
async def get_current_user(user: UserFullSchema = Depends(get_current_user)):
  return user
users.models
from sqlalchemy import Column, String, Text

from app.db_base.base_model import BaseSQLAlchemyModel


class User(BaseSQLAlchemyModel):
  __tablename__ = "users"

  first_name = Column(String(20), nullable=False)
  last_name = Column(String(20), nullable=False)
  email = Column(String(100), nullable=False)
  password = Column(Text, nullable=False)
users.schemas
from pydantic import BaseModel, Field, EmailStr, validator


class UserBaseSchema(BaseModel):
  first_name: str = Field(min_length=2, max_length=20)
  last_name: str = Field(min_length=2, max_length=20)
  email: EmailStr


class UserRegisterSchema(UserBaseSchema):
```

```
    password: str = Field(min_length=7, max_length=30)
    password_confirmation: str = Field(min_length=7, max_length=30)

    @validator('password_confirmation', pre=False)
    def passwords_match(cls, v, values, **kwargs):
        if 'password' in values and v != values['password']:
            raise ValueError('Passwords do not match')
        return v


class UserPublicSchema(UserBaseSchema):
    id: int

    class Config:
        orm_mode = True


class UserLoginSchema(BaseModel):
    email: EmailStr
    password: str


class UserFullSchema(UserPublicSchema):
    password: str

    class Config:
        orm_mode = True
```

```
class JWTTokenSchema(BaseModel):
  access_token: str


class JWTTokenDataSchema(BaseModel):
  id: int
```

```
users.service
import hashlib
import time

from fastapi.security import OAuth2PasswordRequestForm
from jose import jwt

from app.config import config
from app.users.schemas import UserRegisterSchema, UserPublicSchema,
UserLoginSchema, UserFullSchema, JWTTokenSchema
from . import exceptions
from .models import User


class UserService:

  @staticmethod
  def create_password_hash(password) -> str:
    pass_hash = hashlib.pbkdf2_hmac(
      'sha256',  # The hash digest algorithm for HMAC
      password.encode('utf-8'),  # Convert the password to bytes
      config.salt.encode('utf-8'),  # Provide the salt
```

```
                100000,  # It is recommended to use at least 100,000 iterations of SHA-256
                dklen=128  # Get a 128 byte key
            )
            return str(pass_hash)

        @classmethod
        async def register(cls, user: UserRegisterSchema) -> UserPublicSchema:
            if (await User.exists(None, email=user.email)):
                raise exceptions.UserAlreadyExist

            pass_hash = cls.create_password_hash(user.password)
            user_data = dict(user)
            user_data.pop("password_confirmation")
            user_data["password"] = pass_hash
            user = await User.create(**user_data)
            return UserPublicSchema.from_orm(user)

        @classmethod
        def create_jwt_token(cls, user: UserFullSchema) -> str:
            payload = {"id": user.id}
            return jwt.encode(payload, config.jwt_secret, algorithm="HS256")

        @classmethod
            async def login(cls, data: OAuth2PasswordRequestForm) -> JWTTokenSchema:
            user: UserFullSchema = await User.get(None, email=data.username)
            if not user:
                raise exceptions.UserDoesNotExist
```

```
            if not cls._is_password_correct(user, data.password):
                raise exceptions.InvalidPassword

            token = cls.create_jwt_token(user)
            return JWTTokenSchema(access_token=token)

        @classmethod
        def _is_password_correct(cls, user: UserFullSchema, entered_password: str)
-> bool:
            entered_pass_hash = cls.create_password_hash(entered_password)
            return user.password == entered_pass_hash
    config
    import os
    import pathlib
    from typing import Any

    from pydantic import BaseSettings, PostgresDsn
    from yaml import CLoader as Loader
    from yaml import load


    class Config(BaseSettings):
      database_uri: PostgresDsn
      is_debug: bool
      salt: str
      jwt_secret: str

      class Config:
```

```
env_file_encoding = "utf-8"

@staticmethod
def _yaml_config_settings_source(_: BaseSettings) -> dict[str, Any]:
    """
    A simple settings source that loads variables from a Yaml file
    at the project's root.

    Here we happen to choose to use the `env_file_encoding` from Config
    when reading `config.json`
    """
    root_dir = pathlib.Path(__file__).parent.parent
    config_path = root_dir.joinpath("local.yaml")

    try:
        with open(config_path, "r") as config_file:
            config = load(config_file, Loader=Loader)
    except FileNotFoundError:
        print("Create config file (local.yaml) in root directory")

    os.environ["database_uri"] = str(config.get("database_uri"))
    return config

@classmethod
def customise_sources(
        cls,
        init_settings,
        env_settings,
        file_secret_settings,
```

```
        ):
            return (
                init_settings,
                cls._yaml_config_settings_source,
                env_settings,
                file_secret_settings,
            )


config = Config()

entry
import argparse
import logging
import sys

import uvicorn
from fastapi import FastAPI

from app.config import config
from app.db_base.base_model import db_connect
from app.questions.handlers import question_router, category_router,
question_answer_router
from app.users.handlers import user_router

parser = argparse.ArgumentParser(
  description="API service for questions",
  formatter_class=argparse.RawDescriptionHelpFormatter,
)
```

```
    parser.add_argument(
      "-c",
      "--config",
      dest="config_file",
      help="path to configuration file",
    )
    parser.add_argument("--dev", action="store_true", help="Run bot in polling")
    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("--downgrade", action="store_true", help="Downgrade database")

    args = parser.parse_args()

    logger = logging.getLogger("main")

    if args.migrate:
      from alembic import command
      from alembic.config import Config

      alembic_cfg = Config("alembic.ini")
      command.upgrade(alembic_cfg, "head")
      logger.info("Migrate database")
      sys.exit(0)

    if args.revision:
      from alembic import command
```

```
    from alembic.config import Config

    alembic_cfg = Config("alembic.ini")
    alembic_cfg.set_main_option("sqlalchemy.url", config.database_uri)
    message = input("Comment revision: ")
    command.revision(alembic_cfg, message, autogenerate=True)
    logger.info("Create database migration")
    sys.exit(0)

if args.downgrade:
    from alembic import command
    from alembic.config import Config

    alembic_cfg = Config("alembic.ini")
    revision = input("Downgrade revision (-1 for previous, Enter to skip): ")
    if revision:
        logger.info(f"Downgrade database to revision {revision}")
        command.downgrade(alembic_cfg, revision)
    else:
        logger.info("Downgrade skipped.")
    sys.exit(0)

app: FastAPI = FastAPI()



@app.on_event("startup")
async def on_startup():
    await db_connect(config.database_uri)
```

```
if args.dev:
    app.include_router(user_router)
    app.include_router(question_router)
    app.include_router(category_router)
    app.include_router(question_answer_router)
    uvicorn.run(app, host="127.0.0.0", port=8080)
```