КВАЛІФІКАЦІЙНА РОБОТА

Група ІПЗс-20
Вівчаренко А.В.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних і прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Вівчаренко Андрій Васильович

УДК 004.4

Розробка, налагодження та оптимізація роботи простого інтерпретатора зображень для незрячих

Спеціальність 121 - «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття освітнього ступеню бакалавр

Нормоконтроль

_____Стисло О.В.

(підпис, дата, розшифрування підпису)

Студент

_____Вівчаренко А.В.

(підпис, дата, розшифрування підпису)

Допускається до захисту

Завідувач кафедри

_____к.т.н., доц. Ващишак С.П.

(підпис, дата, розшифрування підпису)

Керівник роботи

_____ к.т.н., доц. Демчина М.М.

(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО «УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА»

Факультет суспільних і прикладних наук

Кафедра інформаційних технології

Освітній ступінь: «бакалавр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

«__»_____ 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Вівчаренко Андрій Васильович

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Розробка, налагодження та оптимізація роботи простого інтерпретатора зображень для незрячих

Керівник роботи: Демчина Микола Миколайович, к.т.н., доцент

Затверджена наказом вищого навчального закладу від «12» березня 2024 року № 19/1

2. Термін подання студентом роботи: 05.06.2024

3. Вихідні дані роботи: Серверна аплікація, мобільний застосунок

4. Зміст кваліфікаційної роботи (перелік питань які потрібно розробити):

1. Аналіз існуючих рішень для інтерпретації зображень

2. Вибір технологій для розробки

3. Розробка серверної аплікації та демонстративного мобільного застосунку

5. Дата видачі завдання: 14.03.2024

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Розділ	Консультант (прізвище, ініціали та посада)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Огляд існуючого програмного забезпечення для інтерпретації зображень	25.03.2024	Виконано
2	Вибір інструментів для розробки програмного забезпечення для інтерпретації зображень	08.04.2024	Виконано
3	Проектування серверної архітектури та мобільного застосунку	22.04.2024	Виконано
4	Розробка серверної аплікації та мобільного застосунку	06.05.2024	Виконано
5	Оформлення пояснювальної записки	20.05.2024	Виконано
6	Оформлення графічного матеріале та підготовка до захисту роботи	03.06.2024	Виконано

Студент _____ Вівчаренко А.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Демчина М.М.
(підпис) (прізвище та ініціали)

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень):

Сторінка	Опис граф. матеріалу	Сторінка	Опис граф. матеріалу
13	Приклад роботи розпізнавання тексту в мобільному додатку SeeingAI	39	Принцип роботи BLoC паттерну
14	Приклад роботи фукнції читання тексту в мобільному застосунку Envision AI	41	Принцип роботи Flutter Cubits
16	Робота функції опису сцени в додатку Envision AI	42	Архітектура Flutter додатку.
24	Паттерн Controller-Service-Repository	48	Створений Environment в Postman
27	Приклад зв'язку таблиць у реляційній базі даних	51	Галерея зображень в мобільному застосунку

АНОТАЦІЯ

У даній кваліфікаційній роботі розглядається проектування та розробка програмного забезпечення для інтерпретації зображень з метою полегшення доступу незрячих користувачів до візуальної інформації.

У роботі розглянуті можливі шляхи реалізації інтерпретації зображень для вирішення проблеми доступу незрячих користувачів до візуальної інформації.

Спроектовано та розроблено серверну частину програмного забезпечення, використовуючи мову програмування Python, фреймворк Tornado, базу даних PostgreSQL та об'єктно-реляційне відображення Peewee. Розглянуто принципи роботи dependency injection, взаємодію між сервісами та репозиторіями, а також проектування структури бази даних.

Також спроектовано та розроблено мобільний додаток на основі фреймворку Flutter. Описано роботу з BLoC паттерном, використання Cubits, а також проектування інтерфейсу користувача.

КЛЮЧОВІ СЛОВА: ІНТЕРПРЕТАЦІЯ ЗОБЖРЕНЬ, PYTHON, TORNADO, POSTGRESQL, PEEWEE, ANDROID, IOS, FLUTTER, BLOC, CUBITS, RETROFIT

SUMMARY

This qualification work examines the design and development of image interpretation software to facilitate access to visual information for blind users.

The paper considers possible ways of implementing image interpretation to solve the problem of access to visual information for blind users.

The server side of the software was designed and developed using Python programming language, Tornado framework, PostgreSQL database, and Peewee as object-relational mapping. Considered the principles of dependency injection, interaction between services and repositories, and database structure design.

Implemented a mobile application with Flutter framework. Described the work with the BLoC pattern, the use of Cubits, and the design of the user interface.

KEYWORDS: IMAGE INTERPRETATION, PYTHON, TORNADO, POSTGRESQL, PEEWEE, ANDROID, IOS, FLUTTER, BLOC, CUBITS, RETROFIT

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API	–	Application Program Interface
ООП	–	Об’єктно орієнтоване програмування
ОС	–	Операційна система
VL	–	Vision-Language
ACID	–	Atomicity, Consistency, Isolation, Durability
RDBMS	–	Relational database management system
HTTP	–	Hypertext Transfer Protocol
WS	–	Web Socket
REST	–	representational state transfer
ORM	–	Object-relation mapping
TTS	–	text-to-speech
SQL	–	Structured query language
BLoC	–	Business Logic Component
DI	–	dependency injection

ВСТУП

Актуальність теми. У сучасному світі розвиток технологій на крок наближає людство до нових можливостей, зокрема в сфері інформаційної доступності. Проте, не зважаючи на значні досягнення, існують певні сфери, де інновації ще потребують дослідження та вдосконалення. Однією з таких сфер є інтерпретація зображень. Актуальність даної теми полягає в необхідності забезпечення доступності інформації для людей з обмеженими можливостями зору. Зараз існують певні рішення, проте їхні можливості ще далекі від повного задоволення потреб цільової аудиторії. Тому це дослідження має велике значення для розвитку соціальної інклюзії та підвищення якості життя людей з обмеженими можливостями.

Мета роботи. розробка, налагодження та оптимізація простого інтерпретатора зображень для незрячих осіб.

Об’єкт дослідження. Основні принципи роботи та процес інтерпретації зображень за допомогою Vision Language Model.

Предмет дослідження. Програмне забезпечення інтерпретатора зображень, його характеристики, та способи інтеграції в різні системи.

Завдання роботи . Відповідно до тему в роботі покладені такі задачі як:

– аналіз проблем інтерпретації зображень, та огляд існуючих рішень;

– з'ясування основних вимог до інтерпретатора зображень та визначення оптимальних технологій для реалізації;

– розробка програмного забезпечення, інтеграція інтерпретатора зображень та його налагодження;

– оптимізація роботи системи з метою підвищення швидкодії та ефективності інтерпретації зображень.

Методи роботи. Для вирішення поставленого завдання були використані: мови програмування Python, Dart, Typescript, фреймворк Flutter.

Результати роботи. Після виконання кваліфікаційної роботи було створено мобільний додаток з інтегрованою системою для інтерпретації зображень, яка в свою чергу виконується на спеціальному виділеному сервері що інтерпретуватиме отримані зображення та повернення результатів. Перспективи проекту включають можливість розширення функціоналу у майбутньому.

Структура роботи. Розділи – 3. Загальний обсяг основної частини – 44 сторінок. Список використаних джерел – 20.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМ ІНТЕРПРЕТАЦІЇ ЗОБРАЖЕНЬ, ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд існуючого програмного забезпечення для інтерпретації зображень які використовують незрячі

В сучасному світі існує ряд програмних засобів, спрямованих на полегшення повсякденного життя людей з обмеженими можливостями зору. Серед таких програм можна виділити Seeing AI та Envision AI, які надають користувачам зі зниженим зором широкий спектр можливостей щодо інтерпретації зображень та оточуючого середовища.

Seeing AI є популярним додатком від Microsoft, спрямованим на надання різноманітних сервісів для користувачів з візуальними обмеженнями. Основні функції Seeing AI включають:

– розпізнавання тексту: додаток може зчитувати текст зі зображень, що дозволяє незрячим людям отримувати інформацію з фотографій, наприклад, з паперових документів, етикеток чи меню. На (рис 1.1.1) зображено приклад роботи розпізнавання тексту;

– розпізнавання обличчя: Seeing AI може розпізнавати та описувати обличчя людей, які знаходяться на фотографіях, що допомагає незрячим користувачам зрозуміти, хто перебуває поруч з ними;

– опис сцени: додаток може надавати опис сцени на зображеннях, включаючи інформацію про об'єкти, їхнє розташування та взаємодію;

– інші функції: Seeing AI також має ряд інших корисних функцій, таких як розпізнавання кольорів, інтерпретація звукових сигналів та інше.

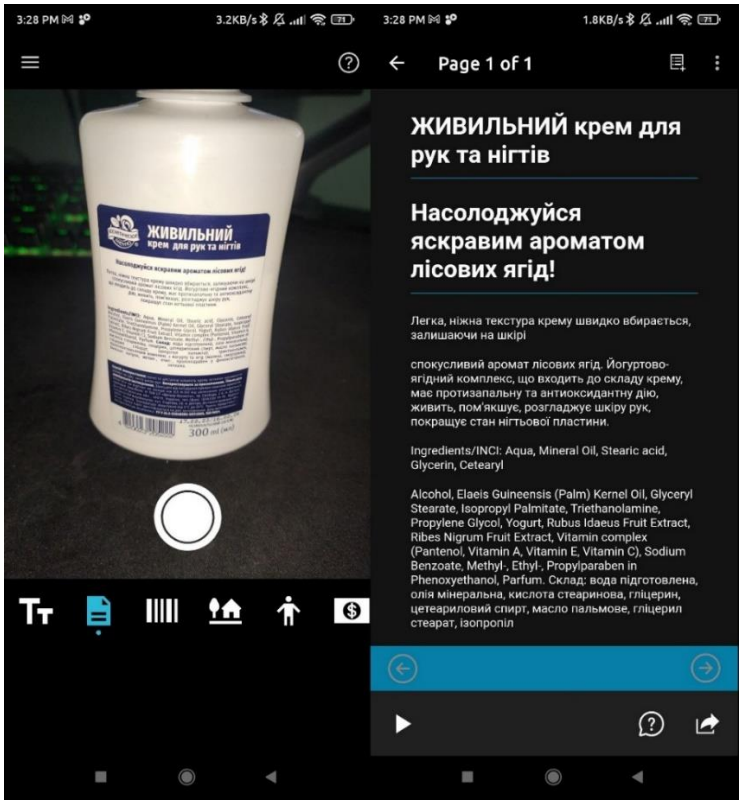


Рисунок 1.1 – Приклад роботи розпізнавання тексту в мобільному додатку SeeingAI

Envision AI [7] є іншим додатком, спеціально розробленим для надання послуг незрячим та людям з візуальними обмеженнями. Деякі ключові функції Envision AI включають:

– додаток може розпізнавати різноманітні об'єкти на зображеннях, а також забезпечує можливість озвучувати назви цих об'єкти за допомогою синтезу мовлення у різних мовах, що забезпечує більшу доступність та зручність для користувачів з вадами зору;

– Envision AI також може розпізнавати текст на зображеннях та навіть рукописний текст, демонстрація цього функціоналу та результати його роботи наведені на рисунку 1.2. це дозволяє користувачам зручно взаємодіяти з текстовою інформацією;

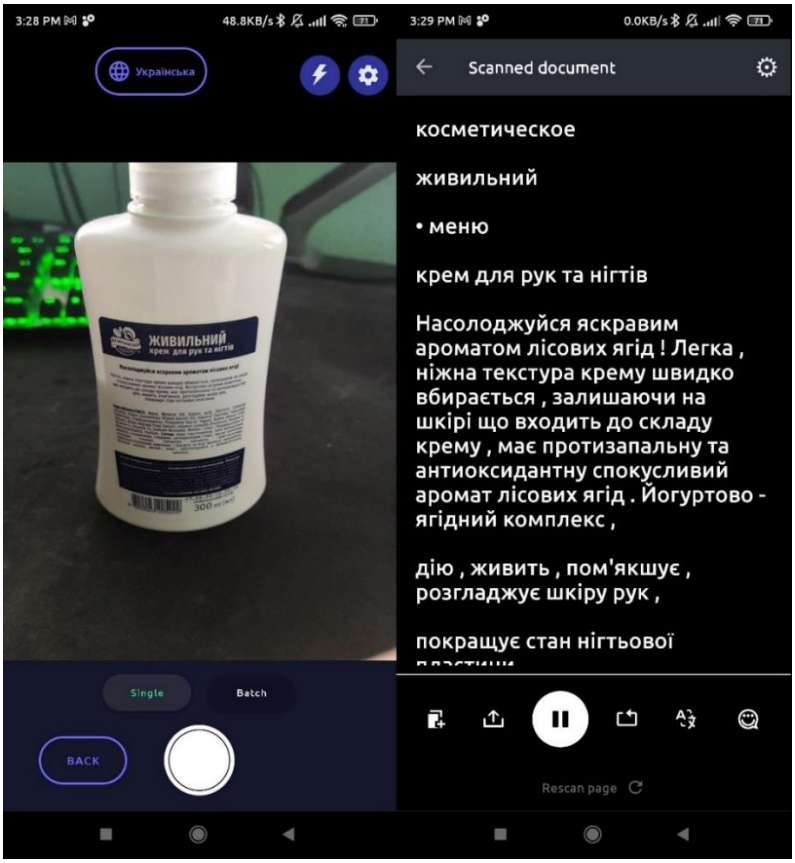


Рисунок 1.2 – Приклад роботи фукнції читання тексту в мобільному застосунку Envision AI

– опис сцени та озвучення інформації: подібно до Seeing AI, Envision AI здатний надавати опис сцени на зображеннях та озвучувати інформацію про навколишнє середовище.

Ці програми представляють значний прогрес у розвитку технологій допомоги для людей з візуальними обмеженнями, але вони також вказують на необхідність постійного вдосконалення та розширення можливостей інтерпретації зображень.

1.2 Аналіз проблеми інтерпретації зображень

Незважаючи на значні досягнення у розробці програмного забезпечення, існують деякі обмеження та недоліки, які потребують уваги. Наприклад, хоча Seeing

AI та Envision AI надають широкий функціонал, деякі користувачі можуть виявити, що деякі об'єкти чи текст розпізнаються недостатньо точно. Крім того, існує необхідність у вдосконаленні алгоритмів для роботи зі складними сценами, такими як зображення з великою кількістю об'єктів або з різноманітним освітленням.

До того ж, існують сфери, де потреби незрячих користувачів можуть не бути повністю задоволені за допомогою існуючого програмного забезпечення. Наприклад, деякі специфічні завдання, такі як інтерпретація зображень в іграх або відображення інформації з веб-сайтів, можуть потребувати спеціалізованих рішень. Також існує проблема інтерпретації в реальному часі, оскільки сучасні технології ще не дозволяють забезпечити миттєвий аналіз інформації. Наразі користувачам доводиться щоразу робити фотографію, а потім чекати на результати обробки зображення, що займає певний час і може викликати незручності. Це пов'язано з тим, що процес обробки вимагає значних обчислювальних ресурсів і складних алгоритмів, які не завжди можуть бути виконані миттєво.

Нижче наведено ключові аспекти аналізу цієї проблеми:

– об'єктивна інтерпретація зображень: однією з основних проблем є необхідність забезпечення користувачам доступу до об'єктивної та точної інформації, що міститься на зображеннях. Це включає в себе розпізнавання елементів, які можуть бути важливими для розуміння контексту;

– робота зі складними сценами**:** Інтерпретація складних сцен, таких як зображення з багатьма об'єктами або з різним освітленням, є ще однією важливою проблемою. На рисунку 1.3 зображено приклад роботи розпізнавання сцени в додатку Envision AI. Ми бачимо що результатом є дуже узагальнений опис, у цьому випадку користувач не знає що саме зображено на моніторі комп’ютера. У таких випадках необхідно розробляти алгоритми, які здатні адаптуватися до різноманітних умов та ефективно інтерпретувати зображення;

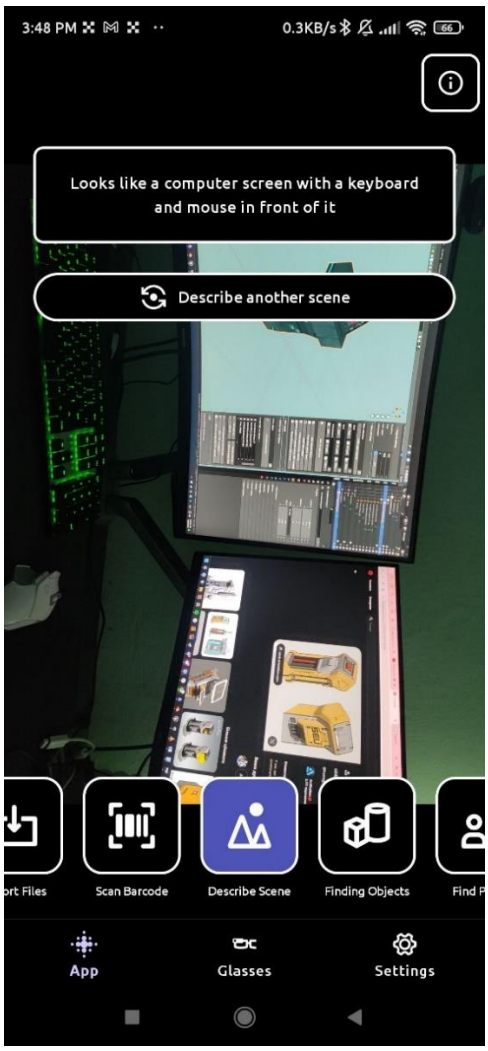


Рисунок 1.3 – Робота функції опису сцени в додатку Envision AI

– швидкість та ефективність: для забезпечення зручного та ефективного використання програмних засобів незрячими користувачами, необхідно також звернути увагу на швидкість та ефективність інтерпретації зображень. Це стосується як часу, необхідного для розпізнавання об'єктів, так і швидкості реакції програми на нові вхідні дані;

– аудіо інтерпретація: незрячим користувачам також потрібна можливість отримувати інформацію з зображень за допомогою аудіо інтерпретації. Це означає, що програмне забезпечення повинно мати можливість озвучувати інформацію, отриману зі зображень, за допомогою синтезу мовлення.

Аналіз цих проблем дозволяє виявити ключові аспекти, які потрібно врахувати при розробці програмного забезпечення для інтерпретації зображень для незрячих користувачів, а також визначити напрямки подальших досліджень та розвитку в цій області.

Таким чином, дослідження та розробка нових програмних засобів для інтерпретації зображень для незрячих користувачів залишається актуальною та перспективною галуззю. Великий потенціал для поліпшення якості життя цієї категорії користувачів може бути реалізований через впровадження нових технологій та алгоритмів, що дозволять надавати точніші та швидші інтерпретації зображень, а також інтерпретацію відео, тобто в реальному часі. Також розширювати й інші можливості їхнього застосування в різних сферах життя.

1.3 Методи інтерпретації зображень та VL моделі

Наразі більшість інтерпретаторів зображень використовують переважно класичні методи комп'ютерного зору, такі як алгоритми виявлення контурів. Однак ці методи мають свої обмеження і недоліки:

– обмежена точність: класичні алгоритми розпізнавання об'єктів можуть недостатньо точно ідентифікувати об'єкти на зображенні, особливо в умовах складних сцен або за наявності перешкод;

– обмеженість функціональності: традиційні методи не завжди здатні адекватно розуміти складні сцени або виконувати завдання, пов'язані з аналізом контексту та мовним описом зображень.

У зв'язку з цим, зараз спостерігається перехід до використання Vision-Language (VL) моделей, які базуються на глибокому навчанні та забезпечують більш точну та ефективну інтерпретацію зображень.

Vision-Language (VL) модель - це тип штучної нейронної мережі, спеціально розроблений для розуміння зображень та мови в контексті один одного. Вони

представляють собою комбінацію компонентів обробки зображень і тексту, які працюють разом для вирішення задач, пов'язаних з аналізом візуальної інформації.

Основні характеристики VL моделей:

– обробка зображень: VL моделі вміють аналізувати зображення, виявляти об'єкти, розпізнавати взаємодії та властивості об'єктів на зображенні;

– розуміння мови: вони також здатні аналізувати та розуміти мовний контекст, включаючи розпізнавання слів, речень, тематику та семантику;

– взаємодія зображення та мови: основна особливість VL моделей полягає в їхній здатності взаємодіяти з візуальною та мовною інформацією одночасно. Вони можуть генерувати мовний опис зображень або виконувати завдання, пов'язані з візуально-мовним зрозумінням;

– глибоке навчання: більшість VL моделей базуються на глибокому навчанні, що дозволяє їм ефективно вивчати складні залежності між зображеннями та мовою та здійснювати адекватні прогнози на їх основі;

– застосування в реальних задачах: вони знаходять широке застосування в різних областях, таких як комп'ютерне зору, автоматизована обробка тексту, віртуальна асистентів, медична діагностика, та багато іншого;

Загалом, VL моделі представляють собою потужний інструмент для аналізу зображень та мови в контексті один одного, що дозволяє вирішувати складні завдання розуміння та інтерпретації візуально-мовної інформації.

Використання таких моделей дозволяє краще розуміти контекст зображення та генерувати більш адекватний текстовий опис для незрячих користувачів, що значно покращує їхню взаємодію з візуальним контентом. Такий підхід є більш гнучким, швидким та ефективним у порівнянні з традиційними методами інтерпретації зображень, оскільки він здатний враховувати більшу кількість факторів і деталей, а також адаптуватися до різних типів зображень і контекстів.

1.4 Постановка задачі для вирішення існуючих проблем

Враховуючи складність і актуальність проблем інтерпретації зображень, необхідно сформулювати конкретні завдання для їх вирішення. Нижче наведено ключові задачі, які мають бути вирішені для покращення програмного забезпечення для інтерпретації зображень:

– адаптація до різних сцен: розробка алгоритмів, які здатні ефективно працювати з різноманітними сценами та умовами зйомки. Це включає в себе адаптацію до різного освітлення, різних кутів огляду, наявності різних об'єктів, та розпізнавання цих на зображенні;

– швидкість та ефективність: розробка методів, що забезпечать швидку та ефективну інтерпретацію зображень, зокрема оптимізація алгоритмів та використання швидкісних технологій обробки даних;

– локалізація: розробка механізмів для локалізації програмного забезпечення на різні мови. Це включає в себе переклад текстів і повідомлень програми на мови користувачів, з урахуванням локальних вимог та стандартів;

– реалізація аудіо інтерпретації: розробка функціоналу, що дозволяє незрячим користувачам отримувати інформацію з зображень через аудіоінтерпретацію. А саме озвучення тексту, опис об'єктів, сцен на зображеннях.

Ці завдання стануть основою для подальшої розробки програмного забезпечення, яке має за мету поліпшення доступності та якості життя незрячих користувачів у сфері інтерпретації зображень.

Висновки до розділу 1

У цьому розділі було проведено огляд існуючого програмного забезпечення для інтерпретації зображень, зосереджуючись на додатках Seeing AI та Envision AI.

Були визначені переваги і обмеження цих програм, а також звернуто увагу на їхній актуальність у контексті покращення їх ефективності для користувачів.

Далі був проведений аналіз проблем інтерпретації зображень, виявлені труднощі та виклики, з якими стикаються користувачі при використанні програмного забезпечення цього типу. Це включає в себе складність розпізнавання об'єктів на зображеннях, адаптацію до різних сцен та умов зйомки, а також швидкість та ефективність роботи програм.

Було проаналізовано проблеми, пов'язані з класичними методами інтерпретації зображень. Ми встановили, що такі методи мають обмеження у точності та функціональності.

У контексті вирішення цих проблем ми описали перехід до використання Vision-Language (VL) моделей, які є потужним інструментом для розуміння зображень та мови в контексті один одного.

Отже, використання VL моделей в програмному забезпеченні для інтерпретації зображень для незрячих користувачів є обіцяним напрямом розвитку, що може призвести до покращення точності, швидкості та функціональності системи, а також спростити її реалізацію та підтримку.

Було поставлено задачі для вирішення цих проблем. Вони включають розробку ефективних алгоритмів розпізнавання об'єктів, адаптацію до різних сцен, покращення швидкості та ефективності програмного забезпечення, розробку доступних інтерфейсів та реалізацію мовної інтерпретації. Крім того, розширення мовної підтримки стане важливим кроком у забезпеченні доступності програм для користувачів з різних країн та культур.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ІНТЕРПРЕТАЦІЇ ЗОБРАНЬ

2.1 Визначення можливих шляхів реалізації інтерпретації зображень для вирішення поставлених завдань

Для вирішення завдань, пов'язаних з інтерпретацією зображень для незрячих користувачів, використовуватиметься open-source Vision-Language (VL) Model під назвою DeepSeek [11], спеціально призначений для реальних застосувань розуміння зображень та мови. Цей підхід дозволить ефективно розпізнавати об'єкти на зображеннях та надавати адекватні описи для незрячих користувачів.

Основними компонентами реалізації будуть:

– мова програмування Python: реалізація серверної частини програми буде здійснюватися мовою програмування Python, яка є широко використовуваною у сфері штучного інтелекту та обробки даних;

– бібліотека Tornado [18] для створення REST API: використання бібліотеки Tornado дозволить легко створити REST API для взаємодії з моделлю VL та іншими компонентами програми. Що забезпечить зручний інтерфейс для взаємодії з системою для розробників;

– база даних Postgresql: потужна та надійна реляційна база даних з відкритим вихідним кодом, яка забезпечує широкі можливості для зберігання та операцій з даними. Вона відома своєю високою продуктивністю, підтримкою ACID принципів та розширеними можливостями для роботи з різними типами даних;

– модель Vision-Language (VL): використання готової моделі VL для аналізу та розуміння зображень у контексті мовного опису. Ця модель дозволить здійснювати аналіз зображень та генерувати відповідні текстові описи;

– модель перекладу MBart-50 [8] від Facebook: ця модель забезпечить можливість перекладати згенерований текстовий опис зображення на різні мови;

– фреймворк Flutter буде використано для створення мобільного додатку, який буде забезпечувати користувачам зручну взаємодію з системою інтерпретації зображень. Flutter є відмінним інструментом для швидкої та ефективної розробки переносних застосунків для різних платформ, таких як Android та iOS.

Використання Flutter дозволить забезпечити:

– кросплатформеність: один і той же код може бути використаний для розробки додатків як для Android, так і для iOS, що значно спрощує розробку та підтримку додатків на різних платформах;

– швидкість розробки**:** Flutter надає широкий набір готових компонентів і інструментів, що дозволяє швидко розробляти функціонал додатків та зосереджуватися на важливих аспектах взаємодії з користувачем;

– можливість розширення та адаптації**:** Flutter дозволяє швидко впроваджувати новий функціонал і змінювати існуючі елементи, що робить його ідеальним вибором для розвитку додатків, що постійно еволюціонують;

– зазначені компоненти дозволять ефективно вирішувати поставлені завдання з інтерпретації зображень для незрячих користувачів, забезпечуючи швидку та якісну обробку зображень та їхнє описання.

2.2 Python та його використання у розробці серверної частини

Python [16] - це високорівнева мова програмування, яка здобула популярність завдяки своїй простоті, зручності та широкому спектру застосувань. У цьому розділі буде розглянуто принципи роботи Python, його сфери використання та обґрунтуємо вибір цієї мови для розробки серверної частини проекту.

Принципи роботи: Python є інтерпретованою мовою програмування, що означає, що код виконується рядок за рядком в реальному часі. Вона відома своєю

простотою синтаксису, що робить її дуже зрозумілою та дружньою для початківців. Python також має велику стандартну бібліотеку, яка містить багато корисних модулів та інструментів для різних завдань.

Сфери використання**:** Python знаходить застосування в різних галузях індустрії, включаючи веб-розробку, наукові дослідження, штучний інтелект, обробку даних, аналітику, автоматизацію та багато іншого. Вона широко використовується в компаніях та організаціях будь-якого розміру, що робить її популярною мовою в програмній індустрії.

Обґрунтування вибору: Python було обрано для розробки серверної частини проекту з кількох причин. По-перше, Python є однією з найбільш популярних мов програмування, що забезпечує велику кількість розробників та велику спільноту, яка може надати підтримку та розв'язати будь-які проблеми. По-друге, Python має багато бібліотек та інструментів для роботи з штучним інтелектом, обробки даних та інших сфер, що створює сприятливі умови для взаємодії з готовими тренованими моделями, такими як ті, що доступні у відкритому доступі на Hugging Face. Це спрощує розробку та інтеграцію існуючих рішень у серверну частину проекту, що збільшує швидкість та ефективність розробки.

2.3 Архітектура серверної частини

Архітектура серверної частини проекту буде побудовано з використанням паттерну Controller-Service-Repository (рис 2.1) що забезпечує чітке розділення функціональності та підтримки принципів SOLID. У даному підході кожен рівень відповідає за виконання певних завдань, що сприяє зручності розробки, тестуванню та підтримці коду.

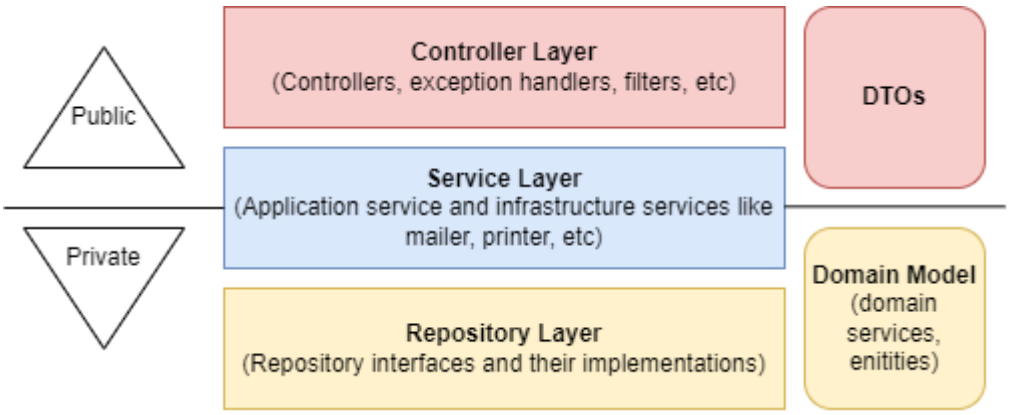


Рисунок 2.1 – Паттерн Controller-Service-Repository

– контролери (Controllers): контролери відповідають за обробку запитів, що надходять на серверну частину. Вони взаємодіють з вхідними даними, обробляють їх та викликають відповідні методи сервісів для обробки логіки бізнес-процесів. Контролери також відповідають за валідацію вхідних даних та обробку різноманітних помилок які можуть виникнути під час роботи сервісів;

– сервіси (Services): сервіси містять бізнес-логіку додатку та виконують операції з даними. Вони отримують запити від контролерів, обробляють їх та здійснюють необхідні дії з даними, такі як звернення до репозиторіїв, обробка зображень та інші операції. Сервіси також відповідають за валідацію та перевірку даних перед збереженням або відправленням на клієнтську частину;

– репозиторії (Repositories): репозиторії відповідають за доступ до даних, таких як база даних або зовнішні сервіси. Вони забезпечують ізоляцію логіки доступу до даних від сервісів та контролерів, що дозволяє змінювати джерела даних без внесення змін у вищі рівні. Репозиторії також можуть включати логіку кешування та оптимізації доступу до даних.

Така архітектура дозволить забезпечити модульність, гнучкість та розширюваність серверної частини проекту, що сприятиме ефективній розробці, тестуванню та підтримці системи інтерпретації зображень.

2.4 Бібліотека Tornado

Tornado – це високопродуктивний веб-сервер та фреймворк для створення мережевих додатків на мові програмування Python. У цьому розділі буде розглянуто основні характеристики та переваги бібліотеки Tornado, а також її використання у розробці серверної частини проекту.

Основні характеристики:

– асинхронність**:** однією з основних особливостей Tornado є його асинхронний підхід. Це дозволяє обробляти багато запитів одночасно, що робить його ідеальним вибором для високонавантажених мережевих додатків;

– вбудований веб-сервер**:** Tornado має вбудований веб-сервер, що спрощує розгортання та налаштування додатку. Це дозволяє швидко створювати та запускати веб-сервери без додаткових конфігурацій;

– реалізація протоколів WebSocket та HTTP**:** Tornado підтримує протоколи WebSocket та HTTP, що дозволяє створювати інтерактивні веб-додатки з реальним часом обміну даними між клієнтом та сервером;

Інтеграція з іншими бібліотеками: Tornado легко інтегрується з іншими бібліотеками та фреймворками Python, що дозволяє розширювати його можливості та використовувати з іншими інструментами.

Переваги використання Tornado:

– висока продуктивність: завдяки асинхронному підходу, Tornado забезпечує високу продуктивність та швидкодіючий веб-сервер для обробки багатьох запитів одночасно;

– простота використання: Tornado має простий та зрозумілий синтаксис, що робить його доступним для розробників будь-якого рівня кваліфікації;

– надійність: бібліотека Tornado вже довгий час використовується в реальних проектах та довела свою надійність та стабільність;

– широкі можливості: Tornado підтримує багато функціональних можливостей, включаючи роботу з WebSocket, HTTP протоколами, аутентифікацію, авторизацію та багато іншого.

Бібліотека Tornado була обрана для розробки серверної частини проекту завдяки її асинхронній природі та високій продуктивності. Вона ідеально підходить для створення REST API [20], яке буде взаємодіяти з клієнтською частиною за допомогою HTTP протоколу. Використання Tornado дозволить забезпечити ефективну обробку запитів, надійність та швидкодіючий веб-сервер. Крім того, його простий синтаксис та можливість легкої інтеграції з іншими бібліотеками Python робить його ідеальним вибором для проекту. У наступному розділі буде розглянуто детальніше архітектуру серверної частини, в якій буде використовуватися бібліотека Tornado.

2.5 База даних PostgreSQL

У цьому розділі буде обґрунтовано вибір PostgreSQL в якості бази даних для даного проекту, а також його переваги та відповідність вимогам проекту.

Обґрунтування вибору:

– надійність та стабільність: PostgreSQL відомий своєю надійністю та стабільністю. Він є однією з найбільш надійних та потужних відкритих баз даних, що гарантує безпеку та цілісність даних;

– підтримка ACID: PostgreSQL підтримує принципи ACID (Atomicity, Consistency, Isolation, Durability), що забезпечує цілісність даних та надійність операцій з ними;

– розширені можливості**:** PostgreSQL має широкі можливості для роботи з реляційними та нереляційними даними, що дозволяє ефективно використовувати його в різних типах проектів;

– підтримка JSON: PostgreSQL підтримує роботу з JSON даними, що дозволяє зручно зберігати та обробляти структуровані дані;

– широкий вибір розширень**:** PostgreSQL має велику кількість розширень та доповнень, які розширюють його можливості та дозволяють пристосувати його під конкретні потреби проекту.

PostgreSQL - реляційна база даних (RDBMS) - це тип бази даних, яка організована за допомогою реляційної моделі даних. У реляційній моделі дані представлені у вигляді таблиць, де кожен рядок таблиці представляє окремий запис, а кожний стовпчик - атрибут цього запису. Зв'язки між таблицями встановлюються за допомогою ключів. На рисунку 2.2 зображено приклад зв’язку між двома таблицями.

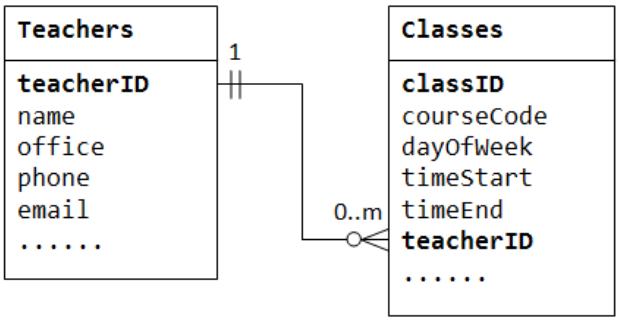


Рисунок 2.2 – Приклад зв’язку таблиць у реляційній базі даних

Реляційні бази даних використовують структуровані дані, що дозволяє виконувати широкий спектр операцій, таких як вибірка, вставка, оновлення та видалення даних, забезпечуючи при цьому цілісність інформації. Ця модель також дозволяє встановлювати взаємозв'язки між даними, що робить її особливо корисною для організації складних структур даних.

Ця модель дозволяє ефективно організовувати та управляти даними шляхом використання стандартизованих мов запитів, таких як SQL (Structured Query Language). Вона забезпечує цілісність даних, дозволяє здійснювати складні

операції, такі як об'єднання, сортування та фільтрація, і є широко використовуваною для зберігання структурованих даних в різних сферах, включаючи бізнес, науку, освіту та інші галузі.

Реляційні бази даних широко використовуються у багатьох галузях, включаючи бізнес, науку, освіту, фінанси та інші. Вони є надійним та ефективним інструментом для зберігання та обробки великих обсягів даних, що робить їх невід'ємною частиною сучасного інформаційного середовища.

Вибір PostgreSQL відповідає вимогам проекту, оскільки він забезпечує надійність, швидкість та масштабованість, що необхідно для ефективного зберігання та обробки даних, що використовуються у серверній частині проекту.

2.6 Об'єктно-реляційне відображення (ORM) і використання Peewee

Об'єктно-реляційне відображення (ORM) - це технологія, яка дозволяє взаємодіяти з реляційною базою даних за допомогою об'єктів програмного забезпечення. Замість безпосереднього використання SQL-запитів, розробник може використовувати методи та класи для взаємодії з базою даних.

ORM [19] дозволяє зменшити кількість коду, потрібного для доступу до даних, а також забезпечує більшу гнучкість і зручність в роботі з базою даних, перетворюючи дані з бази даних в об'єкти програмного коду. Він приховує складність написання SQL-запитів і внутрішню структуру бази даних, що спрощує розробку та підтримку програмного забезпечення, дозволяючи розробникам зосередитися на логіці програми, а не на технічних деталях взаємодії з базою даних. Це значно прискорює процес розробки, знижує ймовірність помилок, пов'язаних з ручним написанням SQL-коду, та покращує читабельність і підтримуваність коду. На рисунку 2.3 зображено принцип роботи ORM, який демонструє, як дані з таблиць бази даних перетворюються в об'єкти і як ці об'єкти взаємодіють з програмним кодом.

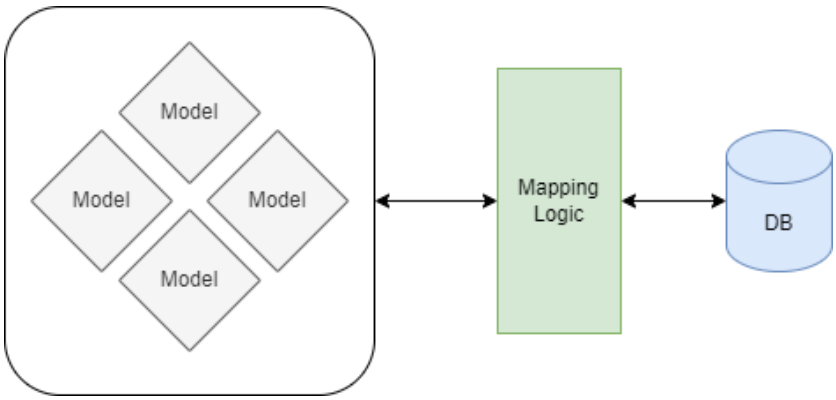


Рисунок 2.3 – Прицнип роботи ORM

У проекті буде використано Peewee ORM [14] для забезпечення зручного та ефективного доступу до даних у базі даних PostgreSQL [15]. Peewee - це легковагий ORM для мови програмування Python, який надає простий та зрозумілий API для роботи з базою даних. Він дозволяє описувати моделі даних у вигляді класів Python та використовувати ці класи для виконання різних операцій з базою даних, таких як створення, читання, оновлення та видалення записів. Peewee також підтримує міграції бази даних, що робить процес розробки та підтримки програмного забезпечення більш простим та зручним.

2.7 Синтез мовлення. TTS

TTS (Text-to-Speech) - це технологія, що дозволяє перетворювати текст на аудіофайли, що містять мовлення. Ця технологія використовується для створення різноманітних аудіо-змісту, такого як аудіокниги, аудіореклама, голосові помічники та інші аудіо-ресурси.

Основні особливості TTS включають синтез голосу на основі введеного тексту, можливість вибору різних голосів та мов, регулювання швидкості та інтонації мовлення, а також підтримку різних мовних акцентів та варіацій.

Модель TTS для української мови: для української мови буде використовуватись модель espnet2, яка дозволяє здійснювати синтез мовлення на основі тексту. Ця модель надає високу якість синтезу мовлення та реалістичний звуковий результат, відтворюючи натуральні розмовні ритми та інтонації.

Модель TTS для інших мов: для інших мов, за винятком української, використовується модель XTTS_v2 [5]. Ця модель є частиною TTS-системи, яка дозволяє синтезувати мовлення. Основні особливості моделі включають підтримку 17 мов, можливість клонування голосу за допомогою короткого 6-секундного аудіофрагмента, а також емоційний та стильовий трансфер за допомогою клонування голосу. Модель має 24-кілогерцеву частоту дискретизації та надає стабільність, покращену вимову та якість аудіо в порівнянні з попередньою версією.

Загалом синтез мовлення буде доступний для наступних мов:

– Англійська;
– Українська;
– Іспанська;
– Французька;
– Німецька;
– Японська;
– Турецька;
– Китайська (спрощена);
– Португальська.

2.8 Проектування архітектури серверної частини

У цьому розділі буде розглянуто детальну архітектуру серверної частини проекту, яка базується на паттерні Controller-Service-Repository. Кожен компонент якого взаємодіє між собою, забезпечуючи чітке розмежування відповідальностей та сприяючи модульності і підтримуваності коду:

– контролер для зображень: відповідає за обробку запитів, пов'язаних із зображеннями та взаємодіє з сервісом зображень для доступу до них;

– контролер DeepSeek веб-сокетів: обробляє веб-сокет запити для взаємодії з DeepSeek сервісом та координує передачу даних клієнту;

– контроллер перекладу: відповідає за обробку запитів, що стосуються перекладу тексту та взаємодіє з сервісом перекладу для перекладу тексту, а також для доступу до перекладів з бази даних;

– контролер TTS: обробляє запити, пов'язані з синтезом молення та взаємодіє з сервісом Text-to-Speech для створення аудіо файлів з текстових даних;

– сервіс зображень: зберігає зображення у пам'яті сервера. Надає унікальні ідентифікатори для кожного зображення. Використовується іншими сервісами для доступу до зображень;

– DeepSeek сервіс (з використанням веб-сокетів): відповідає за взаємодію з DeepSeek VL моделлю для генерації описів зображень. Використовує веб-сокети для передачі запитів та отримання результатів у реальному часі;

– сервіс перекладу: перекладає описи зображень на різні мови за допомогою моделі MBart-50 від Facebook, забезпечуючи високоякісний і точний переклад текстів для користувачів з різних країн та мовних груп;

– сервіс Text-to-Speech (TTS): конвертує текстові описи зображень у аудіофайли за допомогою відповідних TTS моделей, забезпечуючи користувачам можливість прослуховування текстової інформації замість її читання.

На рисунку 2.4 зображено діаграму архітектури серверної частини, яка ілюструє роботу та процес взаємодії між описаними контролерами, сервісами та репозиторіями в серверній аплікації, наочно показуючи, як дані передаються та обробляються на різних рівнях системи. Ця діаграма допомагає краще зрозуміти структуру програмного забезпечення, демонструючи логічні зв'язки та потоки інформації, що забезпечують ефективне функціонування серверної частини.

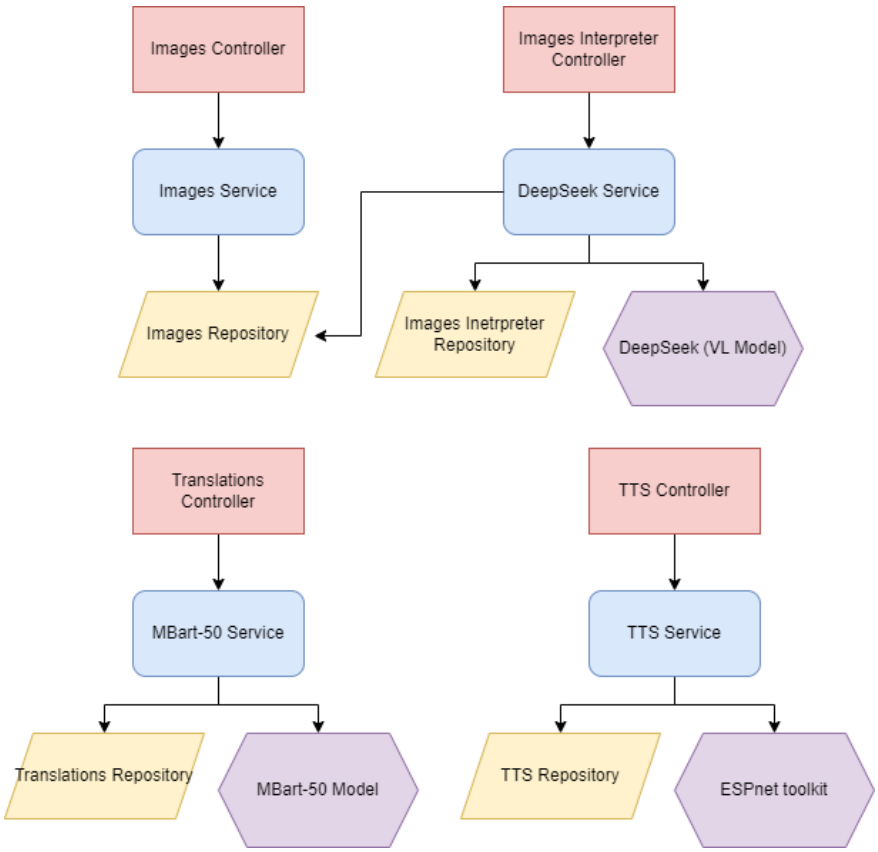


Рисунок 2.4 – Діаграма архітектури серверної частини

Кожен контролер взаємодіє зі своїм відповідним сервісом для виконання конкретних функцій, що дозволяє забезпечити ефективну та структуровану обробку даних у серверній частині застосунку. Такий підхід сприяє чіткій організації коду, робить його більш зрозумілим та легким для підтримки, а також дозволяє розробникам швидше знаходити і виправляти помилки. Крім того, цей методологічний підхід підвищує модульність і масштабованість застосунку.

2.9 Принцип роботи dependency injection

Dependency Injection (DI) – це шаблон проектування програмного забезпечення, який полягає у передачі залежностей класів ззовні. Замість того, щоб клас створював свої залежності власноруч, вони передаються йому ззовні. Це

забезпечує більшу гнучкість, роблячи код більш масштабованим, придатним до тестування та легше підтримуваним.

Застосування dependency injector [6]: у проекті буде використано dependency injector для керування залежностями та їх передачі у різні компоненти. В якості dependency injector для Python ми обрали бібліотеку dependency-injector, яка надає зручний і простий спосіб передачі залежностей у класи та компоненти.

Ця бібліотека дозволяє описати залежності у вигляді контейнера, який потім можна використовувати для створення об'єктів та їх передачі у різні частини програми. Вона підтримує різні способи передачі залежностей, такі як конструктор, властивості або методи.

Завдяки використанню dependency injector код стає більш зручним для тестування та модифікації, а також зменшується зв'язаність між класами, що робить його більш гнучким та легше розширюваним.

Знадобиться тільки один контейнер, який відповідає за керування залежностями та створення необхідних об'єктів. Нижче наведений опис залежностей та сервісів, які потрібно описати у цьому контейнері:

Репозиторії (Repositories):

– images_repository: фабрика для створення репозиторію для роботи з сховищем зображень;

– images_interpreter_repository: фабрика для створення репозиторію для роботи з базою даних, де будуть зберігатись усі результати інтерпретації зображень;

– translations_repository: фабрика для створення репозиторію для роботи з базою даних, де будуть зберігатись усі результати перекладів;

– tts_repository: фабрика для створення репозиторію для роботи з сховищем аудіо файлів створених після перетворення тексту на мовлення.

Сервіси (Services):

– images_service: фабрика для створення сервісу зображень з використанням репозиторію зображень та інструментів зображень;

– deepseek: сервіс що використовує модель DeepSeek VL для аналізу зображень в реальному часі;

– mbart50: сервіс для використання моделі MBart-50 для перекладу;

– tts_service: сервіс для перетворення тексту на мовлення.

2.10 Взаємодія між сервісами та репозиторіями

У цьому розділі буде детально проаналізовано взаємодію між різними сервісами та репозиторіями в контексті проекту. Для зручності, розділ розглядається за окремими сценаріями використання.

Сервіс DeepSeek відповідає за аналіз зображень та генерацію описів. При використанні цього сервісу виникає наступна послідовність дій:

1. Клієнтська частина відправляє запит на сервер з зображенням.
2. Контролер приймає цей запит та взаємодіє з сервісом DeepSeek.
3. DeepSeek використовує дані зображення для генерації опису.
4. Опис повертається до контролера, який надсилає його клієнту.

Взаємодія з сервісом перекладу MBart-50: сервіс MBart-50 відповідає за переклад згенерованих описів на різні мови. Процес взаємодії з цим сервісом виглядає наступним чином:

1. Опис, отриманий від DeepSeek, передається до сервісу перекладу.
2. MBart-50 здійснює переклад опису на вибрану мову.
3. Перекладений опис повертається до контролера та надсилається клієнту для відображення на екрані мобільного пристрою користувача.

Сервіс TTS відповідає за перетворення текстового опису на аудіо файл. Процес взаємодії з цим сервісом описується наступним чином**.**

1. Опис, який був перекладений, передається до сервісу TTS.
2. TTS генерує аудіо файл на основі отриманого тексту.
3. Аудіо файл повертається до контролера та надсилається клієнту.

Взаємодія з сервісом збереження зображень: сервіс збереження зображень використовується для зберігання зображень на сервері. Це необхідно для подальшої взаємодії з ними в майбутньому. Процес взаємодії включає наступні кроки:

1. Зображення, яке надходить від клієнта, зберігається в репозиторії зображень під унікальним ідентифікатором.

2. Ідентифікатор зображення повертається до контролера та використовується для подальших взаємодій з іншими сервісами.

Ці сценарії взаємодії демонструють, як окремі сервіси та репозиторії взаємодіють між собою для забезпечення функціональності системи, відображаючи потоки даних та логіку їх обробки. Вони показують, як запити проходять через різні компоненти системи, як дані зберігаються, обробляються та передаються далі, забезпечуючи узгодженість і цілісність роботи всіх частин проекту. Ця демонстрація надає розробникам чітке уявлення про те, як компоненти інтегруються та співпрацюють для досягнення загальних цілей системи.

2.11 Проектування структури бази даних

У цьому розділі буде розглянуто проектування структури бази даних для проекту. Добре спроектована база даних є важливою складовою успішного функціонування системи. Для проекту вибрано PostgreSQL як систему керування базами даних через його надійність, потужність та можливості розширення.

Таблиця Image для збереження інформації про зображення, які використовуються в системі:

– id (Primary Key): унікальний ідентифікатор зображення;

– path: шлях до зображення у сховищі (наприклад, GCP Bucket Storage).

Таблиця Description для збереження описів, згенерованих сервісом DeepSeek:

– id (Primary Key): унікальний ідентифікатор опису;

– image_id (Foreign Key): посилання на зображення;

– description_text**:** текстовий опис.

Таблиця Translation для збереження перекладів описів на різні мови:

– id (Primary Key): унікальний ідентифікатор перекладу;

– description_id (Foreign Key): посилання на оригінальний опис;

– src_language: код мови оригіналу;

– target_language: код мови перекладу;

– translated_text: перекладений текст опису.

Таблиця TTS_Audio для збереження аудіо файлів, що створені сервісом TTS:

– id (Primary Key): унікальний ідентифікатор аудіо файлу;

– text: текст, який був використаний для створення аудіо файлу;

– path: шлях до аудіо файлу у сховищі (наприклад, GCP Bucket Storage).

Зв'язок між таблицею Image та Description: один до багатьох. Одне зображення може мати багато описів, але кожен опис відноситься тільки до одного зображення в таблиці Image.

Зв'язок між таблицею Description та Translation: один до багатьох. Один опис може мати багато перекладів, але кожен переклад належить тільки до одного опису.

Зв'язок між таблицею Translation та TTS_Audio: один до одного. Кожен переклад може мати лише один аудіо файл TTS, і кожен аудіо файл може належити лише одному перекладу.

Це базова структура бази даних, яка враховує вимоги проекту у збереженні та обробці зображень, описів та аудіофайлів TTS, забезпечуючи ефективну організацію та доступ до необхідної інформації. Діаграма на рисунку 2.5 наглядно ілюструє кожну таблицю та зв’язки між ними, демонструючи, як дані взаємодіють і об'єднуються в єдину систему для забезпечення цілісності та зручності використання бази даних.

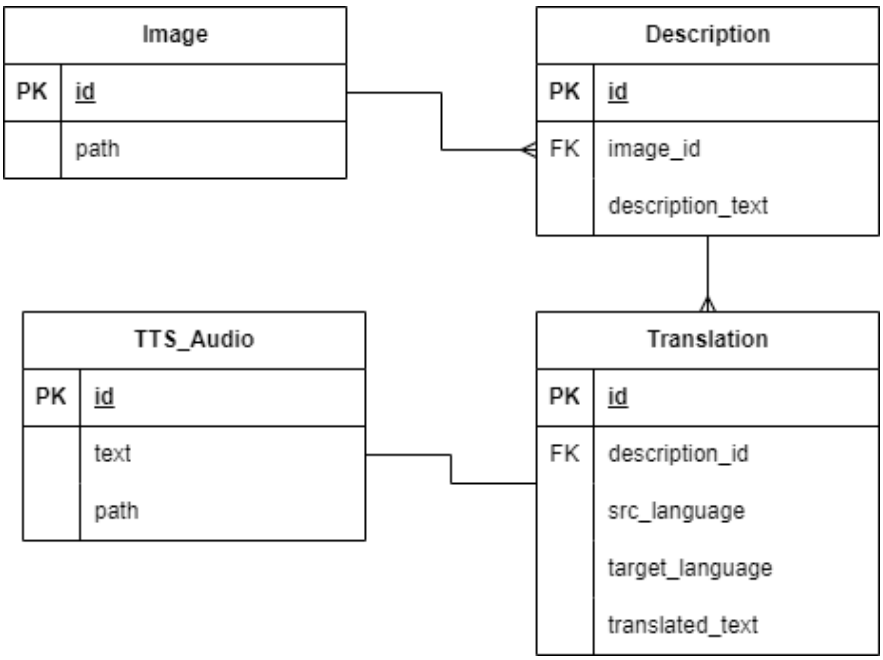


Рисунок 2.5 – Діаграма структури бази даних

2.12 Flutter та його використання у розробці мобільного додатку

Flutter - це фреймворк з відкритим кодом для розробки мобільних, веб- та десктоп-додатків, розроблений Google. Основною особливістю Flutter є те, що він використовує одну кодову базу для розробки додатків для різних платформ, таких як Android та iOS. Ось деякі переваги використання Flutter у розробці мобільних додатків для проекту:

– кросплатформенність: Flutter дозволяє розробникам створювати додатки для різних платформ, використовуючи один код. Це значно спрощує розробку та підтримку додатків для різних операційних систем;

– швидкість розробки: багатофункціональний набір інструментів Flutter, разом із його потужними віджетами і багатою бібліотекою компонентів, дозволяє швидко створювати додатки з красивим інтерфейсом та багатим функціоналом. Крім того, гарна документація і активна спільнота розробників допомагають

швидко знаходити відповіді на питання та вирішувати проблеми, що виникають під час розробки, що додатково прискорює процес створення застосунків.

– гнучкість та кастомізація**:** Flutter надає розробникам велику гнучкість у створенні інтерфейсів користувача, що дозволяє створювати унікальний, інтуїтивно зрозумілий дизайн для кожного додатку;

– широкий вибір готових компонентів**:** Flutter має велику бібліотеку готових компонентів та пакетів, які значно спрощують розробку та дозволяють швидко додавати новий функціонал.

Для даного проекту використання Flutter є ідеальним варіантом, оскільки ми плануємо створити кросплатформений мобільний додаток, який буде доступний для користувачів як на Android, так і на iOS. Flutter дозволить швидко розробити додаток з красивим та інтуїтивно зрозумілим інтерфейсом, який забезпечить зручну взаємодію з системою інтерпретації зображень.

2.13 Принцип роботи BLoC паттерну

BLoC (Business Logic Component) - це архітектурний паттерн, який дозволяє розділити бізнес-логіку від інтерфейсу користувача. Основною ідеєю BLoC є використання трьох компонентів: події (Events), стани (States) та блоки (BLoCs):

– події (Events): це дії або події, які відбуваються в додатку, такі як натискання кнопок, отримання даних з мережі тощо. Події відправляються до відповідного блоку для обробки;

– стани (States): це стани або умови, в яких може знаходитись додаток в залежності від подій та зовнішніх впливів. Інтерфейс користувача динамічно оновлюється відповідно до змін станів;

– блоки (BLoCs): це компоненти, які відповідають за обробку подій та зміну станів додатку. BLoCs містять бізнес-логіку, вони приймають події, обробляють їх та генерують нові стани.

Принцип роботи BLoC полягає в тому, що весь цикл події-стан-блок відбувається асинхронно. Коли відбувається подія, вона відправляється до відповідного блоку, який аналізує її та змінює стан додатку відповідно до бізнес-логіки. Потім новий стан передається відповідному віджету інтерфейсу користувача, який автоматично оновлюється з урахуванням нового стану. На (рис 2.6) зображено прицнип взаємої компонентів у BLoC паттерні.

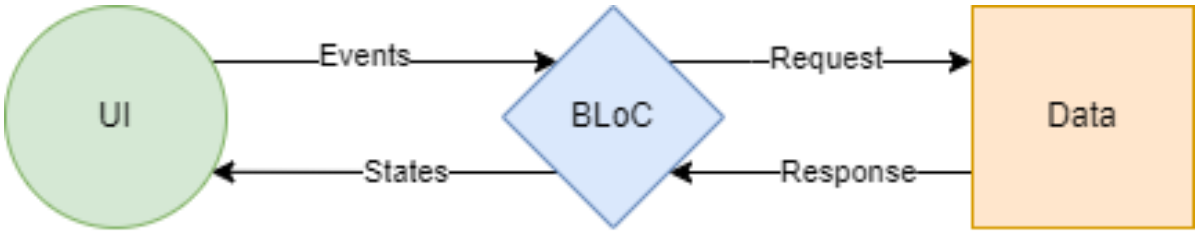


Рисунок 2.6 – Принцип роботи BLoC паттерну

BLoC паттерн дозволяє розробникам створювати ефективні та масштабовані мобільні додатки з чіткою, визначеною бізнес-логікою та розділеним інтерфейсом користувача, що дозволяє полегшити тестування, розширення та підтримку додатків у майбутньому.

2.14 Retrofit, BuildRunner та генерація коду у Flutter

У цьому розділі буде розглянуто використання бібліотек Retrofit та BuildRunner для автоматизації роботи з мережевими запитами в Flutter.

Retrofit [17] - це бібліотека для створення HTTP-клієнтів у мові програмування Dart. Вона надає простий та зручний спосіб виконання мережевих запитів до сервера та обробки їх відповідей. Для цього використовується

аннотаційна модель, яка дозволяє описувати різні аспекти запитів, такі як URL-адреса, метод, заголовки тощо, безпосередньо в інтерфейсі клієнта.

BuildRunner [3] - це інструмент, який дозволяє виконувати різні завдання побудови в проекті Flutter, такі як генерація коду, збірка ресурсів, підтримка кодогенерації тощо. Він інтегрований з Dart SDK та дозволяє автоматизувати рутинні операції побудови проекту.

Завдяки BuildRunner та різним плагінам, таким як retrofit_generator, можна генерувати код для реалізації мережевих запитів на основі інтерфейсів Retrofit. Це зменшити кількість ручних операцій при роботі з мережевими запитами.

У проекті Flutter використання Retrofit та BuildRunner. Шляхом описування інтерфейсів з методами запитів та їх аннотаційною конфігурацією, розробник може автоматично генерувати код для виконання цих запитів без необхідності написання багатократно однотипного коду.

2.15 Принцип роботи Cubits в бібліотеці flutter_bloc

Flutter Cubit [2] - це інша реалізація паттерну управління станом, яка є альтернативою до BLoC. Основна відмінність між Cubit і BLoC полягає у спрощені. Тоді як BLoC має більш складну структуру з використанням подій, станів та блоків, Cubit пропонує простіший підхід до управління станом. Розглянемо основні відмінності Cubits від BLoCs:

– спрощений синтаксис**:** Cubit надає простіший синтаксис порівняно з BLoC, що робить його легшим для розуміння та використання;

– відсутність подій: у Cubit відсутня концепція подій, які використовуються в BLoC. Замість цього, ви працюєте безпосередньо з методами та об'єктами, що представляють стан додатку;

– простота конфігурації**:** Ініціалізація та конфігурація Cubit є простішою порівняно з BLoC. Це дозволяє розробникам уникнути шаблонний код що в свою чергу робить код більш зрозумілим.

Застосування Flutter Cubit у проекті допоможе спростити управління станом, зменшити кількість необхідного коду та покращити продуктивність розробки, особливо у випадках, коли вам не потрібна повна складність паттерну BLoC.

Порівнюючи принцип роботи Cubits (рис. 2.7) та BLoC можна помітити, що працюючи з Cubit зникає необхідність у створенні Events, що в свою чергу в значній мірі зменшує кількість шаблонного коду.

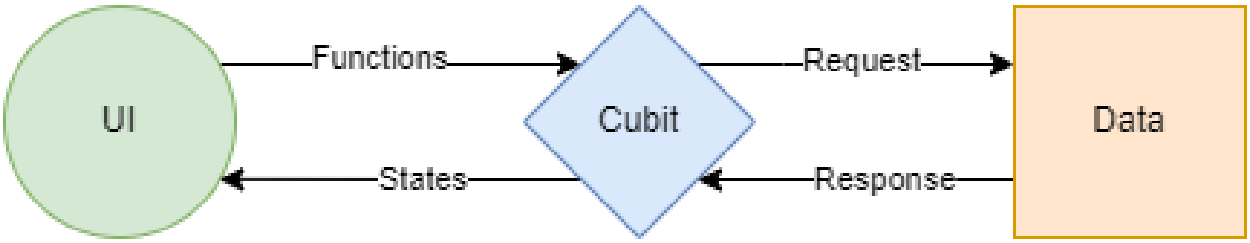


Рисунок 2.7 – Принцип роботи Flutter Cubits

2.16 Архітектура мобільного додатку

Архітектура мобільного додатку базуватиметься на принципах чистої архітектури, яка дозволяє розділити додаток на логічні компоненти та забезпечити його масштабованість та легкість тестування. Нижче наведено її основні складові:

Presentation Layer (Шар представлення):

– екрани (Screens) і Віджети (Widgets): у цьому шарі розміщені усі екрани та віджети, які відображають інформацію для користувача та забезпечують його взаємодію з додатком;

– блоки (BLoCs) або Кубіти (Cubits): використовуються для управління станом додатку та взаємодії з бекендом.

Domain Layer (Доменний шар):

– моделі (Models): представляють собою бізнес-об'єкти, які використовуються в додатку;

– репозиторії (Repositories): відповідають за взаємодію з даними та виконання бізнес-логіки. Наприклад виконання HTTP запитів до серверу.

Data Layer (Шар даних):

– джерела даних (Data Sources): забезпечують доступ до зовнішніх даних (наприклад, бази даних, API);

– мапери (Mappers): використовуються для перетворення даних з одного формату в інший. Наприклад з отриманого JSON формату перетворити його у спеціальний об’єкт Data класу, в якому також може бути можливість визначити додаткові методи для роботи з цими даними.

На рисунку 2.8 зображено взаємодію між різними шарами та їх компонентами. Така архітектура мобільного додатку забезпечує чітке розділення відповідальностей між компонентами, що в свою чергу сприяє покращенню підтримуваності, масштабованості та стабільності додатку.

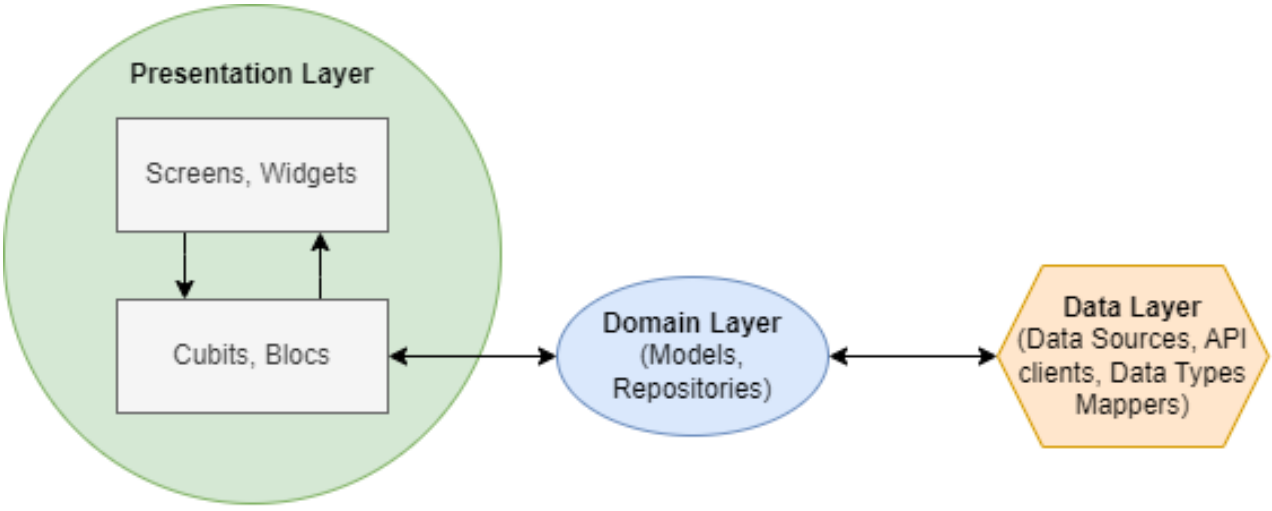


Рисунок 2.8 – Архітектура Flutter додатку.

Висновки до розділу 2

У розділі 2 було розглянуто різні аспекти проектування та розроблення програмного забезпечення для інтерпретації зображень.

Вибір технологій для серверної частини: було проведено аналіз можливих шляхів реалізації інтерпретації зображень та визначено, що для серверної частини проекту найбільш підходять Python, Tornado, PostgreSQL та Peewee ORM.

Архітектура серверної частини: була розроблена архітектура серверної частини, яка включає в себе принцип роботи dependency injection, взаємодію між сервісами та репозиторіями, а також проектування структури бази даних.

Синтез мовлення (TTS): в рамках серверної частини було реалізовано синтез мовлення (TTS) для створення аудіофайлів на основі тексту.

Flutter у розробці мобільного додатку: для розробки мобільного додатку було обрано Flutter. Було детально розглянуто принцип роботи BLoC та Cubits, а також використання Retrofit, BuildRunner та генерацію коду у Flutter для роботи з мережевими запитами, а саме взаємодією із REST API.

Проектування архітектури мобільного додатку: була розроблена архітектура мобільного додатку, яка включає в себе реалізацію BLoC та Cubits для управління станом додатку, а також інтерфейс користувача з використанням різноманітних віджетів та компонентів Flutter.

Розділ 2 надає глибоке розуміння технологій та інструментів, які використовуються для створення програмного забезпечення для інтерпретації зображень, та демонструє їхню ефективність у розробці як серверної, так і мобільної частин проекту.

РОЗДІЛ 3. РОЗРОБКА СЕРВЕРНОЇ АПЛІКАЦІЇ, МОБІЛЬНОГО ЗАСТОСУНКУ ТА ЇХ ТЕСТУВАННЯ

3.1 Розробка сутностей в базі даних за допомогою ORM peewee

Для зберігання даних у базі даних Postgresql використовується ORM peewee. Нижче наведено опис сутностей та їх взаємозв'язків:

– ImageEntity: представляє сутність зображення в базі даних. Містить поля для унікального ідентифікатора (id);

– ImagePromptEntity: представляє сутність результату обробки зображення. Містить поля: image_id - зовнішній ключ, що посилається на ImageEntity, text – текст, результат обробки зображення за допомогою DeepSeek, added_text – так як DeepSeek обробляє зображення в реальному часі, то в цьому полі буде збережено останню зміну яку було додано до основного тексту, done - прапорець, який вказує, чи завершений запит;

– ImagePromptTranslationEntity: представляє сутність перекладу підказки для зображення. Містить поля: prompt_image_id - зовнішній ключ, що посилається на ImagePromptEntity, text - перекладений текст результату обробки зображення, src_lang - мова оригіналу, target_lang - мова перекладу;

– TTSEntity: представляє сутність аудіофайлу, що є результатом синтезу мовлення. Містить поля: translation_id - зовнішній ключ, що посилається на ImagePromptTranslationEntity, tts_output_text - текст, який було синтезовано.

Ці сутності допомагають в організації та зберіганні даних, необхідних для роботи системи з інтерпретації зображень та генерації аудіофайлів. Код з використанням ORM peewee наведено у Додатку А.

3.2 Розробка репозиторіїв

У цьому розділі описано реалізацію репозиторіїв, які використовуються для доступу до даних у базі даних або локальному сховищі.

– BaseLocalStorageRepository: цей клас є базовим для всіх репозиторіїв, які здійснюють доступ до локального сховища. Він містить загальні методи для отримання, створення та видалення елементів з локального сховища;

– ImagesRepository: репозиторій для зображень. Він розширює функціонал базового локального сховища для збереження зображень;

– TTSRepository: репозиторій для аудіофайлів TTS. Він також розширює базовий локальний репозиторій і визначає каталог для збереження аудіофайлів TTS;

– TtsSamplesRepository: репозиторій для зразків аудіофайлів TTS. Використовується для зберігання пробних аудіофайлів, що використовуються для тестування та налагодження TTS;

Створені репозиторії забезпечать необхідних доступ та функціонал для роботи з даними та файлами. Код реалізації репозиторіїв наведено у Додатку Б.

3.3 Розробка сервісів

У цьому розділі описано реалізацію сервісів, які забезпечують основну логіку додатку та взаємодію з даними.

– DeepSeekService: цей сервіс відіграє ключову роль у взаємодії з моделлю інтерпретації зображень. Під час ініціалізації він завантажує модель обробки мови та генерації тексту, а також налаштовує параметри для генерації тексту на основі вхідних зображень. Після ініціалізації він готовий обробляти запити на опис зображень. Сервіс отримує зображення та текстовий запит, обробляє їх через модель та повертає результат у формі текстового опису зображення;

– ImagesToolsService: цей сервіс використовується для масштабування зображень. Він отримує шлях до зображення та використовує бібліотеку PIL для зменшення його розмірів. Після масштабування зображення зберігається знову, а оновлений шлях повертається для подальшого використання;

– ImagesService: цей сервіс відповідає за управління зображеннями у додатку. Він забезпечує функції отримання, збереження та масштабування зображень. Під час отримання зображення сервіс перевіряє наявність файлу у локальному сховищі та повертає його, якщо воно є. У випадку відсутності зображення в сховищі, воно завантажується зовнішнім джерелом та зберігається локально. Після збереження сервіс автоматично масштабує зображення, щоб зменшити його розмір та покращити продуктивність додатку;

– MBart50Service: цей сервіс відповідає за переклад тексту з однієї мови на іншу за допомогою моделі MBart50. Він ініціалізує модель та токенізатор для перекладу тексту. Після ініціалізації сервіс готовий приймати запити на переклад тексту та повертати результат у відповідній мові;

– TTSService: цей сервіс використовується для синтезу мовлення (TTS) на основі тексту. Він ініціалізує два різних TTS двигуни для української та решти мов. Після ініціалізації сервіс приймає текстовий запит на синтез мовлення та повертає аудіофайл із згенерованим мовленням. Код реалізації описаних сервісів наведено у Додатку В.

3.4 Розробка контроллерів

У даному розділі наведено опис контролерів, які використовуються для обробки HTTP-запитів та WebSocket-з'єднань:

– ImageHandler: цей контролер відповідає за обробку запитів, що стосуються зображень. Метод GET призначений для отримання списку зображень

або конкретного зображення за його ідентифікатором. Метод POST використовується для завантаження нових зображень;

– WebSocketHandler: цей контролер відповідає за обробку WebSocket-з'єднань. Він приймає та обробляє повідомлення, пов'язані з обробкою зображень, що надходять через WebSocket;

– TranslatorHandler: цей контролер відповідає за обробку запитів щодо перекладу тексту. Метод GET використовується для отримання перекладів, а метод POST - для створення нових перекладів;

– TTSHandler: цей контролер відповідає за обробку запитів, пов'язаних з синтезом мовлення (TTS). Метод GET використовується для отримання аудіофайлів TTS на основі ідентифікатора перекладу.

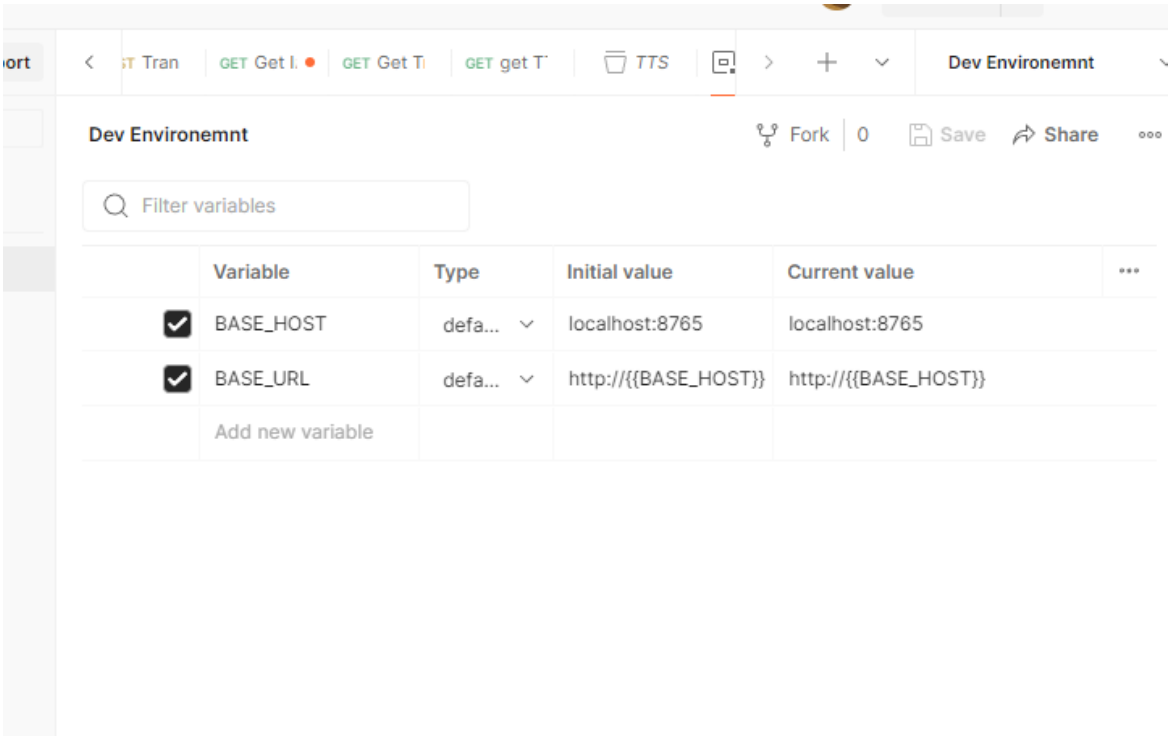
Ці контролери виконують основні завдання обробки запитів, а також інтерфейси взаємодії з іншими компонентами системи для виконання потрібних операцій з даними.

3.5 Тестування серверної частини

У цьому розділі буде розглянуто тестування серверної частини проекту за допомогою інструменту Postman.

Postman - це інструмент для тестування API, який дозволяє відправляти HTTP-запити на сервер та перевіряти його відповіді. Далі буде розглянуто, як використовувати Postman для тестування API.

Для початку було створено Environment, де визначено змінні, що будуть використовуватися у запитах для забезпечення гнучкості та зручності при тестуванні API. На рисунку 3.1 зображено, як виглядає Environment в Postman. Це дозволяє швидко змінювати конфігурацію для різних середовищ, таких як розробка, тестування та робочого продукту, полегшуючи процес налагодження та підтримки.



Рисноук 3.1 – Створений Environment в Postman

Далі будуть наведені створені HTTP запити.

– **GET** /image – отримати список усіх зображень (додаток Ґ рис. 1);

– **GET** /image?id=<id> - Отримати файл зображення за вказаним ідентифікатором в базі даних (додаток Ґ, рис. 2);

– **GET** /image?id=<id>&prompt=true – отримати результат обробки конкретного зображення (додаток Ґ, рис. 3);

– **POST** /image – Завантажити зображення (додаток Ґ, рис. 4);

– **WS** /ws – встановити WebSocket з’єднання;

– **POST** /translator – виконати переклад тексту (додаток Ґ, рис. 5);

– **GET** /translator?image_prompt_id=<id> - Отримати список наявних перекладів для обробленого зображення (додаток Ґ, рис. 6);

– **GET** /tts?translation_id=<id> - Отримати аудіофайл синтезу мовлення (TTS) для перекладу обробленого зображення (додаток Ґ, рис.7).

3.7 Розробка сервісів мобільного застосунку

У цьому розділі буде розглянуто імплементацію сервісів за допомогою бібліотеки Retrofit. Ці сервіси взаємодіють з серверною частиною застосунку та надають доступ до різних функціональностей через відповідні API.

ImagesClient:

– getImages(): метод для отримання списку завантажених зображень;

– getImagePrompt(String id, bool prompt): метод для отримання вказаного зображення з додатковою інформацією.

TranslationsClient:

– createTranslation(TranslationRequestModel body): метод для створення нового перекладу на сервері на основі наданої моделі запиту;

– getAllTranslations(int imagePromptId): метод для отримання списку всіх перекладів для певного зображення на основі його ідентифікатора.

TTSClient:

– getTTS(int translationId): метод для отримання аудіофайлу з сервера, який містить голосовий варіант перекладу тексту за його ідентифікатором.

Кожен з цих сервісів має відповідні методи для виконання різних запитів до сервера та обробки відповідей. Вони допомагають забезпечити ефективну взаємодію між мобільним додатком та серверною частиною. Додаток Д містить код з імплементацією цих сервісів.

3.8 Розробка BLoC (Cubits)

У цьому розділі буде детально розглянуто реалізацію BLoC (Business Logic Component) за допомогою Cubit в Flutter для ефективного керування станом додатку та виконання бізнес-логіки. Ми ознайомимося з основними принципами BLoC, дослідимо переваги використання Cubit у порівнянні з традиційним підходом, а

також розглянемо практичні приклади, які демонструють, як можна інтегрувати Cubit для побудови масштабованих і підтримуваних додатків. Ви дізнаєтесь, як створювати та використовувати Cubit для обробки подій і зміни стану, що забезпечить більш структурований і організований підхід до розробки додатків за допомогою фреймворку Flutter.

Було реалізовано наступні Cubits:

– ImagesGalleryCubit**:** відповідає за керування станом галереї зображень. Він має методи для ініціалізації галереї та завантаження зображень з сервера. Також він відповідає за завантаження нового зображення на сервер;

– WebSocketCubit: використовується для підключення до WebSocket сервера. Він надає можливість встановлення та збереження з'єднання з сервером, а також відправлення та отримання повідомлень;

– ImagePromptCubit: керує відображенням інформації про конкретне зображення, включаючи його опис та переклади на різні мови. Він ініціалізує отримання даних про зображення та його переклади;

– TranslationCubit**:** відповідає за переклад тексту на різні мови. Він ініціює запити на сервер для отримання перекладу та керує їхнім станом;

– TtsCubit: використовується для отримання аудіофайлів з текстового опису. Він ініціює запити на сервер для отримання аудіофайлу.

Кожен Cubit відповідає за конкретний аспект додатку та дозволяє легко розширювати його функціональність. Код реалізації Cubits наведено у Додатку Е.

3.9 Розробка інтерфейсу користувача

У цьому розділі буде розглянуто розробку демонстративного інтерфейсу користувача для мобільного додатку DeepSeek. Окрім того, розглянемо використання бібліотек та інструментів Flutter.

App: кореневий віджет додатку. Він ініціалізує всі необхідні BlocProvider та відображає MaterialApp. У випадку, якщо WebSocket не підключений або відбувається підключення, він відображає кнопку для повторної спроси з’єднання.

HomePage: цей віджет відповідає за головну сторінку додатку. Він включає в себе AppBar та ImagesGalleryView, що показує галерею зображень.

ImagesGalleryView: відповідає за відображення галереї зображень. Він використовує BlocBuilder для слідкування за станом ImagesGalleryCubit та відображенням відповідного вмісту. Крім того, він має можливість завантаження нового зображення з камери або галереї (рис. 3.2).

Рисунок 3.2 – Галерея зображень в мобільному застосунку

ImagePage**:** відображає деталі конкретного зображення, включаючи опис та переклад (рис. 3.3). Він також має можливість відтворення аудіофайлу з текстового опису за допомогою віджету TTSPlayer. В свою чергу сам віджет використовує

плагін – audioplayes [1] для відтворення аудіо файлів. Цей плагін дозволяє відтворювати різні формати аудіо файлів на різних операційних система, а саме: android, ios, linux, macos, windows.

Рисунок 3.3 – Сторінка з описом зображення перекладеним українською.

TTSPlayer: відповідає за відтворення аудіофайлів і забезпечує користувачів можливістю слухати текст у голосовому форматі. Він використовує AudioPlayer для управління аудіострімом, що дозволяє виконувати функції відтворення, паузи та зупинки аудіо. Крім того, TTSPlayer інтегрує FlutterSlider для зручної перемотування аудіо, що дозволяє користувачам легко навігувати по аудіофайлу і знаходити потрібні фрагменти. Детальніше про структуру та функціональність цього компоненту можна ознайомитися на рисунку 3.4, який ілюструє його роботу та взаємодію з іншими частинами системи.

Рисунок 3.4 – Сторінка з описом зображення англійською.

3.10 Тестування мобільного застосунку

Тестування мобільного застосунку є важливою складовою процесу розробки, спрямованою на забезпечення якості програмного забезпечення та задоволення вимог користувачів. Для повноцінного тестування мобільного застосунку необхідно виконати ряд тест-кейсів, що охоплюють різні аспекти його функціональності та взаємодії з користувачем.

Тестування функціональності**.** Далі буде розглянуто тест-кейси для перевірки основної функціональності мобільного застосунку:

– тестування перегляду зображень**:** перевірка коректності відображення зображень у галереї, можливості прокрутки списку зображень та їх перегляду;

– тестування функції завантаження зображень: перевірка можливості завантаження зображень з різних джерел (камера, галерея) та коректності їх відображення в застосунку;

– тестування синтезу мовлення**:** перевірка правильності синтезу мовлення для опису зображень та можливості відтворення аудіоопису.

Тестування взаємодії з користувачем:

– тестування навігації: перевірка коректності роботи всіх кнопок, посилань та інших елементів навігації застосунку;

– тестування взаємодії з елементами інтерфейсу: перевірка відповідності реакції застосунку на різні дії користувача (натискання кнопок, скролінг тощо);

– тестування обробки помилок**:** перевірка коректності повідомлень про помилки, виникнення яких може статися під час використання застосунку.

Тестування сумісності:

– тестування на різних пристроях: перевірка коректності роботи застосунку на різних пристроях з різними версіями операційної системи;

– тестування на різних версіях операційної системи: перевірка сумісності застосунку з різними версіями операційних систем Android та iOS.

Ці тест-кейси допоможуть забезпечити високу якість та надійність мобільного застосунку, забезпечуючи задоволення вимог користувачів та оптимальний досвід його використання.

Висновки до розділу 3

У розділі 3 було виконано розробку серверної аплікації, мобільного застосунку та проведенні тестування обох компонентів.

Розробка серверної частини: були розроблені сутності в базі даних, репозиторії, сервіси та контролери для забезпечення функціональності серверної аплікації. Використання ORM peewee дозволило зручно та ефективно взаємодіяти з базою даних PostgreSQL.

Розробка мобільного застосунку: були розроблені сервіси та використано BLoC (Cubits) для управління станом додатку. Інтерфейс користувача був розроблений з використанням різноманітних віджетів та компонентів Flutter.

Тестування мобільного застосунку: були проведені тести для перевірки функціональності та стабільності мобільного застосунку. Це дозволило переконатися, що додаток працює коректно на різних пристроях та під час різних сценаріїв використання.

Результатом роботи у цьому розділі є розроблені та протестовані серверна аплікація та мобільний застосунок, які відповідають поставленим вимогам.

ВИСНОВКИ

У цій роботі було проведено комплексне дослідження проблем інтерпретації зображень користувачів та розроблено програмне забезпечення для подолання цих проблем. Під час роботи над проектом було виконано наступне:

Огляд існуючого ПЗ для інтерпретації зображень для незрячих: виконаний огляд існуючого програмного забезпечення показав, що існують певні рішення, але вони не завжди ефективні або доступні.

Аналіз проблеми інтерпретації зображень: проведений аналіз проблеми дозволив визначити ключові труднощі, з якими зіштовхуються незрячі користувачі при взаємодії з зображеннями.

Постановка задачі: на основі проведеного аналізу були сформульовані завдання для розробки програмного забезпечення, спрямованого на покращення інтерпретації зображень. Створення демонстративного інтерфейсу для користувачів та забезпечення обробки зображень.

Використання технологій: в розробці серверної частини були використані Python, Tornado, PostgreSQL та Peewee ORM для забезпечення ефективної та безперебійної роботи системи.

Розробка: було розроблено мобільний додаток на платформі Flutter, що дозволяє користувачам отримувати інформацію з зображень за допомогою голосового синтезу.

Тестування серверної частини і мобільного застосунку: обидва компоненти програмного забезпечення були піддані інтенсивному тестуванню для перевірки їх функціональності та стабільності.

Результати дослідження та розробки можуть бути корисними для подальших досліджень у цій області та розробки подібних систем, або вдосконалення вже існуючих рішень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Audioplayers. Flutter. URL: https://pub.dev/packages/audioplayers (дата звернення: 27.04.2024)

2. Bloc state management library. Bloc. URL: https://bloclibrary.dev/ (дата звернення: 25.04.2024).

3. Build_runner. Dart package. Dart packages. URL: https://pub.dev/packages/build_runner (дата звернення: 24.03.2024).

4. Dependency injection – Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Dependency_injection (дата звернення: 27.03.2024).

5. Coqui/XTTS-v2 · hugging face. Hugging Face – The AI community building the future. URL: https://huggingface.co/coqui/XTTS-v2 (дата звернення: 22.03.2024).

6. Dependency injector. Python Dependency Injector 4.41.0. URL: https://python-dependency-injector.ets-labs.org/ (дата звернення: 25.03.2024).

7. Envision - perceive possibility. Envision - Perceive Possibility. URL: https://www.letsenvision.com/ (дата звернення: 15.03.2024).

8. Facebook/mbart-large-50 · hugging face. Hugging Face – The AI community building the future. URL: https://huggingface.co/facebook/mbart-large-50 (дата звернення: 24.03.2024).

9. Flutter documentation. Flutter Docs. URL: https://docs.flutter.dev/ (дата звернення: 27.04.2024).

10. Image_picker. Flutter package. Dart packages. URL: https://pub.dev/packages/image_picker (дата звернення: 25.04.2024).

11. Lu H. DeepSeek-VL: Towards Real-World Vision-Language Understanding. ArXiv.org. 2024. URL: https://arxiv.org/abs/2403.05525 (дата звернення: 15.03.2024).

12. MBart and MBart-50. Hugging Face – The AI community building the future. URL: https://huggingface.co/docs/transformers/en/model_doc/mbart (дата звернення: 17.03.2024).

13. MVC design pattern – geeksforgeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/mvc-design-pattern/ (дата звернення: 28.04.2024).

14. Peewee – peewee 3.17.3 documentation. peewee – peewee 3.17.3 documentation. URL: https://docs.peewee-orm.com/en/latest/ (дата звернення: 28.04.2024).

15. PostgreSQL: documentation. PostgreSQL: The world's most advanced open source database. URL: https://www.postgresql.org/docs/ (дата звернення: 25.04.2024).

16. Python documentation. 3.11.8. URL: https://docs.python.org/3.11/ (дата звернення: 17.04.2024).

17. Retrofit. Dart Packages. URL: https://pub.dev/packages/retrofit (дата звернення: 27.04.2024).

18. Tornado Web Server – Tornado 6.4 documentation. Tornado Web Server – Tornado 6.4 documentation. URL: https://www.tornadoweb.org/en/stable/ (дата звернення: 13.04.2024).

19. What is an ORM (object relational mapper)?. Prisma's Data Guide. URL: https://www.prisma.io/dataguide/types/relational/what-is-an-orm (дата звернення: 12.04.2024).

20. What is a REST API? | IBM. IBM in Deutschland, Österreich und der Schweiz. URL: https://www.ibm.com/topics/rest-apis (дата звернення: 12.04.2024).

ДОДАТКИ

Додаток А

Сутності бази даних описані за допомогою ORM peewee

```
db = pw.PostgresqlDatabase('postgres', user='postgres',
password='postgres', host='localhost', port=5432)
db.connect()

class BaseEntity(pw.Model):
    class Meta:
        database = db

class ImageEntity(BaseEntity):
    def to_model(self) -> ImageModel:
        return ImageModel(
            id=self.id,
        )
class ImagePromptEntity(BaseEntity):
    page_url = CharField(max_length=512)
    image_id = ForeignKeyField(ImageEntity)
    text = CharField(max_length=16384)
    added_text = CharField(max_length=1024)
    done = BooleanField()

class ImagePromptTranslationEntity(BaseEntity):
    prompt_image_id = ForeignKeyField(ImagePromptEntity,
backref='translations')
    text = CharField(max_length=16384)
    src_lang = CharField()
    target_lang = CharField()

class TTSEntity(BaseEntity):
    translation_id = ForeignKeyField(ImagePromptTranslationEntity,
backref='tts')
    tts_output_text = CharField(max_length=32678, null=False)

db.create_tables([
    ImageEntity,
    ImagePromptEntity,
    ImagePromptTranslationEntity,
    TTSEntity,
])
```

Додаток Б

Код реалізації репозиторіїв серверної частини

```
class BaseLocalStorageRepository:

    def __init__(self):
        self.directory = 'storage/temp'
        pass

    def get_path(self, item_id):
        files = os.listdir(self.directory)
        for file in files:
            if item_id in file:
                return f"{self.directory}/{file}"

    def get_new_path(self, item_uuid, item_type):
        return f"{self.directory}/{item_uuid}.{item_type}"

    def get_item(self, item_id):
        # read image from disk
        path = self.get_path(item_id)
        if not path:
            return None
        with open(path, "rb") as f:
            return f.read()

    def get_items(self) -> list[str]:
        # read images from disk
        files = []
        for file in os.listdir(self.directory):
            files.append(file.replace(".jpg", ""))

        return files

    def create(self, new_id: int, data, item_type: str) -> str:
        # save image to disk
        new_uid = str(new_id)
        with open(self.get_new_path(new_uid, item_type), "wb") as f:
            f.write(data)

        return new_uid

    def delete(self, item_id):
        # delete image from disk
        os.remove(self.get_path(item_id))
```

```
        return True

class ImagesRepository(BaseLocalStorageRepository):

    def __init__(self):
        super().__init__()
        self.directory = 'storage/images'
        pass

class TTSRepository(BaseLocalStorageRepository):

    def __init__(self):
        super().__init__()
        self.directory = 'storage/tts'

class TtsSamplesRepository(BaseLocalStorageRepository):

    def __init__(self):
        super().__init__()
        self.directory = 'storage/tts_samples'
```

Додаток В

Код реалізації контролерів серверної частини

```
class WebSocketHandler(tornado.websocket.WebSocketHandler):

    def check_origin(self, origin: str) -> bool:
        return True

    def set_default_headers(self) -> None:
        self.set_header("Access-Control-Allow-Origin", "*")

    def open(self):
        print("WebSocket opened")

    @inject
    def on_message(self, message,
                   images_service: ImagesService =
Provide[MainContainer.images_service],
                   images_repository: ImagesRepository =
Provide[MainContainer.images_repository],
                   deepseek_service: DeepSeekService =
Provide[MainContainer.deepseek],
                   ):

        request = SocketRequest.model_validate_json(message)

        if request.type == SocketRequestType.prompt_image:
            request =
SocketPromptImageRequest.model_validate_json(message)

            images_data = [ImageModel(id=request.image_id)]
            image_id = request.image_id
            # for image_id in request.image_id:
            #
images_data.append(ImageModel(uuid=request.image_id))

            response = SocketPromptImageInitResponse(
                image_id=image_id,
                page_url=request.page_url
            )
            try:
                self.write_message(response.model_dump())

                answer = deepseek_service.get(
                    uuid=request.page_url,
                    images_data=images_data,
```

```
                    images_repository=images_repository
                )

                text = ""

                image_prompt_entity = ImagePromptEntity(
                    image_id=image_id,
                    page_url=request.page_url,
                    text=text,
                    added_text="",
                    done=False
                )
                image_prompt_entity.save()

                for c in answer:
                    text += c
                    response = SocketPromptImageProgressResponse(
                        id=image_prompt_entity.id,
                        image_id=image_id,
                        text=text,
                        added_text=c,
                        done=False,
                        page_url=request.page_url
                    )
                    # update prompt entity and save to db
                    ImagePromptEntity.update(text=text, added_text=c,
done=False).where(
                        ImagePromptEntity.image_id ==
image_id).execute()
                    self.write_message(response.model_dump())

                response = SocketPromptImageProgressResponse(
                    id=image_prompt_entity.id,
                    image_id=image_id,
                    text=text,
                    added_text="",
                    done=True,
                    page_url=request.page_url
                )

                ImagePromptTranslationEntity.create(
                    prompt_image_id=image_prompt_entity.id,
                    text=text,
                    src_lang='en_XX',
                    target_lang="en_XX"
                )

                ImagePromptEntity.update(text=text, added_text="",
done=True).where(
```

```
                    ImagePromptEntity.image_id == image_id).execute()

deepseek_service.update_deepseek_response(uuid=request.page_url,
answer=text)
                self.write_message(response.model_dump())

            except Exception as e:
                print(e)
                pass

    def on_close(self):
        print("WebSocket closed")

class ImageHandler(tornado.web.RequestHandler):

    @inject
    def get(self, images_service: ImagesService =
Provide[MainContainer.images_service]):

        prompt = self.get_argument("prompt", None)
        image_id = self.get_argument('id', None)

        if prompt and image_id:
            prompt_entity = (ImagePromptEntity.select()
            .where(
                (ImagePromptEntity.image_id == image_id)
            ))

            if prompt_entity.exists():
                prompt_entity = prompt_entity.get()
                prompt_model = prompt_entity.to_model()
                self.write(prompt_model.model_dump())
                return
            else:
                self.write(ErrorResponse(
                    error=404,
                    message="Prompt image not found"
                ).model_dump())
                return

        if image_id is None:
            print("Requested all images")
            images = images_service.get_images()

            # set the response header
            self.set_header('Content-Type', 'application/json')

            # dump list
            self.write({"images": [image.model_dump() for image in
```

```
images]})

        else:
            print(f"Requested image with ID: {image_id}")

            image = images_service.get_image(image_id)

            # set the response header
            self.set_header('Content-Type', 'image/png')
            self.write(image)

    @inject
    def post(self, images_service: ImagesService =
Provide[MainContainer.images_service]):
        print("Received image")

        images = []

        for field_name, files in self.request.files.items():
            for info in files:
                filename, content_type = info['filename'],
info['content_type']

                body = info['body']
                extension = filename.split(".")[1]

                new_image_id = images_service.post_image(body,
extension)
                images.append(ImageModel(id=new_image_id))
                print(f"Received {filename} of type {content_type}
with {len(body)} bytes \n UUID: {new_image_id}")

        self.write(ImagesList(images=images).model_dump())


class TranslatorHandler(tornado.web.RequestHandler):

    @inject
    def get(self):
        image_prompt_id =
self.request.arguments.get("image_prompt_id", None)

        if image_prompt_id is None:
            self.write(ErrorResponse(
                error=400,
                message="Missing image_prompt_id"
            ).model_dump())
            return
```

```
        image_prompt_id = int(image_prompt_id[0])

        image_prompt_translations =
(ImagePromptTranslationEntity.select().where(
            (ImagePromptTranslationEntity.prompt_image_id ==
image_prompt_id)
        ))
        image_prompt_translations = image_prompt_translations.dicts()

        data = []

        for translation in image_prompt_translations:
            print(translation)
            data.append(TranslatorTranslateResponse(**translation))

        response = TranslatorGetResponse(
            translations=data
        )

        self.write(response.model_dump())

    @inject
    def post(self, mbart50_service: MBart50Service =
Provide[MainContainer.mbart50],
             ):
        request =
TranslatorTranslateRequest.model_validate_json(self.request.body)

        image_prompt = (ImagePromptEntity.select().where(
            (ImagePromptEntity.id == request.prompt_image_id)
        ))

        if not image_prompt.exists():
            self.write(ErrorResponse(
                error=404,
                message="Prompt image not found"
            ).model_dump())
            return
        image_prompt = image_prompt.get()

        existing_translation = (
            ImagePromptTranslationEntity.select()
            .where(
                (ImagePromptTranslationEntity.prompt_image_id ==
request.prompt_image_id),
                (ImagePromptTranslationEntity.src_lang ==
request.src_lang),
                (ImagePromptTranslationEntity.target_lang ==
request.target_lang)
```

```
            )
        )

        if existing_translation.exists():

self.write(existing_translation.get().to_model().model_dump())
            return

        translated_text = mbart50_service.translate(
            text=image_prompt.text,
            src_lang=request.src_lang,
            target_lang=request.target_lang
        )

        translation_entity = ImagePromptTranslationEntity.create(
            prompt_image_id=image_prompt.id,
            text=str(translated_text[0]),
            src_lang=request.src_lang,
            target_lang=request.target_lang
        )
        translation_entity.save()

        response = TranslatorTranslateResponse(
            id=translation_entity.id,
            prompt_image_id=request.prompt_image_id,
            text=str(translated_text[0]),
            src_lang=request.src_lang,
            target_lang=request.target_lang
        )

        self.write(response.model_dump())

class TTSHandler(tornado.web.RequestHandler):

    @inject
    def get(self,
            tts_service: TTSService =
Provide[MainContainer.tts_service],
            ):

        translation_id = self.get_argument("translation_id", None)

        if translation_id is None:
            self.write(ErrorResponse(
                error=400,
                message="Missing translation_id"
            ).model_dump())
            return
```

```
        translation_id = int(translation_id)

        translation = (ImagePromptTranslationEntity.select().where(
            (ImagePromptTranslationEntity.id == translation_id)))
        if not translation.exists():
            self.write(ErrorResponse(
                error=404,
                message="Translation not found"
            ).model_dump())
            return

        translation = translation.get()
        language =
tts_mbart50_code_to_tts_code.get(translation.target_lang)
        if language not in tts_supported_languages:
            self.write(ErrorResponse(
                error=400,
                message="Translation language not supported"
            ).model_dump())
            return

        tts_file = tts_service.create_tts(
            text=translation.text,
            language=language,
            translation_id=translation_id
        )

        if tts_file:
            self.set_header('Content-Type', 'audio/wav')
            self.write(tts_file)
            return

        self.write(ErrorResponse(
            error=500,
            message="Failed to create TTS"
        ).model_dump())

app = tornado.web.Application([
    (r'/ws', WebSocketHandler),
    (r'/image', ImageHandler),
    (r'/translator', TranslatorHandler),  (r'/tts', TTSHandler),
])
```

Додаток Ґ

Код опису залежностей у dependency injection

```
class MainContainer(containers.DeclarativeContainer):
    config = providers.Configuration()

    # Repositories
    images_repository = providers.Factory(
        ImagesRepository
    )
    translations_repository = providers.Factory(
        TranslationsRepository
    )

    tts_repository = providers.Factory(
        TTSRepository
    )

    tts_samples_repository = providers.Factory(
        TtsSamplesRepository
    )

    # Services
    images_tools_service = providers.Singleton(
        ImagesToolsService
    )

    images_service = providers.Factory(
        ImagesService,
        images_repository=images_repository,
        images_tools_service=images_tools_service
    )

    deepseek = providers.Singleton(
        DeepSeekService,
        images_interpreter_repository=images_interpreter_repository,
        images_repository=images_repository,
        model_path="deepseek-ai/deepseek-vl-1.3b-chat",
    )

    mbart50 = providers.Singleton(
        MBart50Service,
        model_path="facebook/mbart-large-50-many-to-many-mmt",
        translations_repository=translations_repository
    )
```

```
    tts_service = providers.Singleton(
        TTSService,
        tts_repository=tts_repository,
        tts_samples_repository=tts_samples_repository
    )



if __name__ == "__main__":
    container = MainContainer()
    container.init_resources()
    container.wire(modules=[
        sys.modules[__name__],
        sys.modules['controllers.image_controller'],
        sys.modules['controllers.translator_controller'],
        sys.modules['controllers.socket_controller'],
        sys.modules['controllers.tts_controller'],
        sys.modules['services.images.images_service'],
        sys.modules['services.deep_seek.deep_seek_service'],
        sys.modules['services.mbart50.mbart50_service'],
        sys.modules['services.tts.tts_service'],
    ])
```

Додаток Е

Взаємодія з REST API у програмному забезпеченні Postman

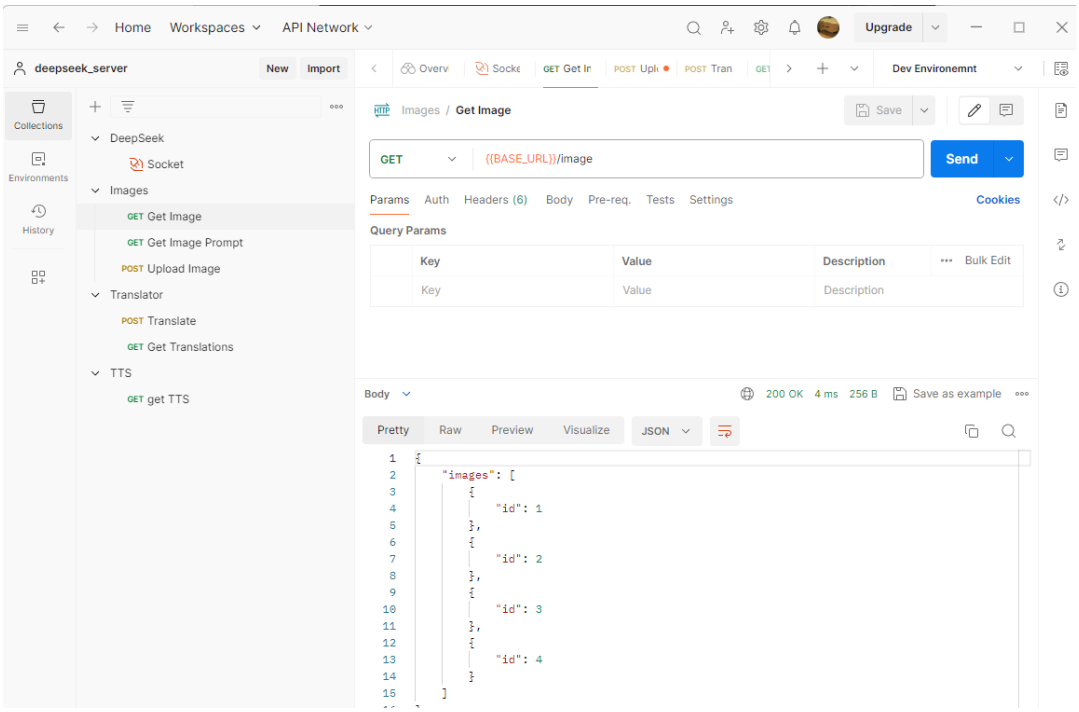


Рисунок 1 – Отримання списку усіх зображень у Postman

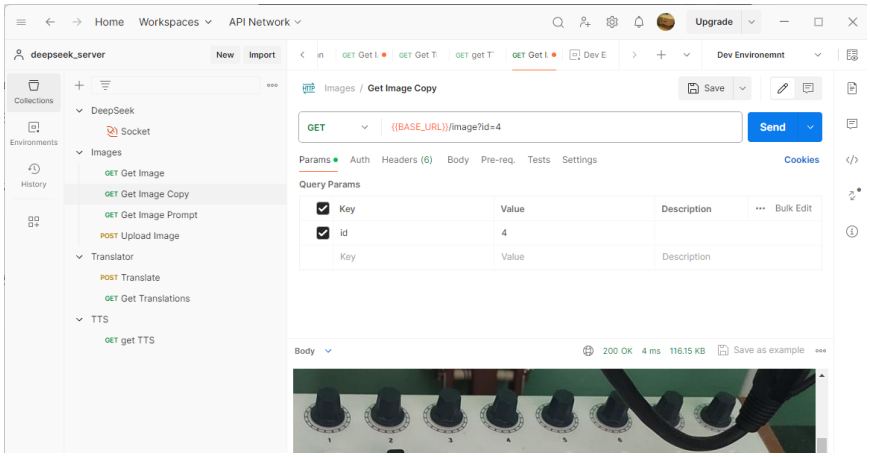


Рисунок 2 – Отримання зображення за ідентифікатором у Postman

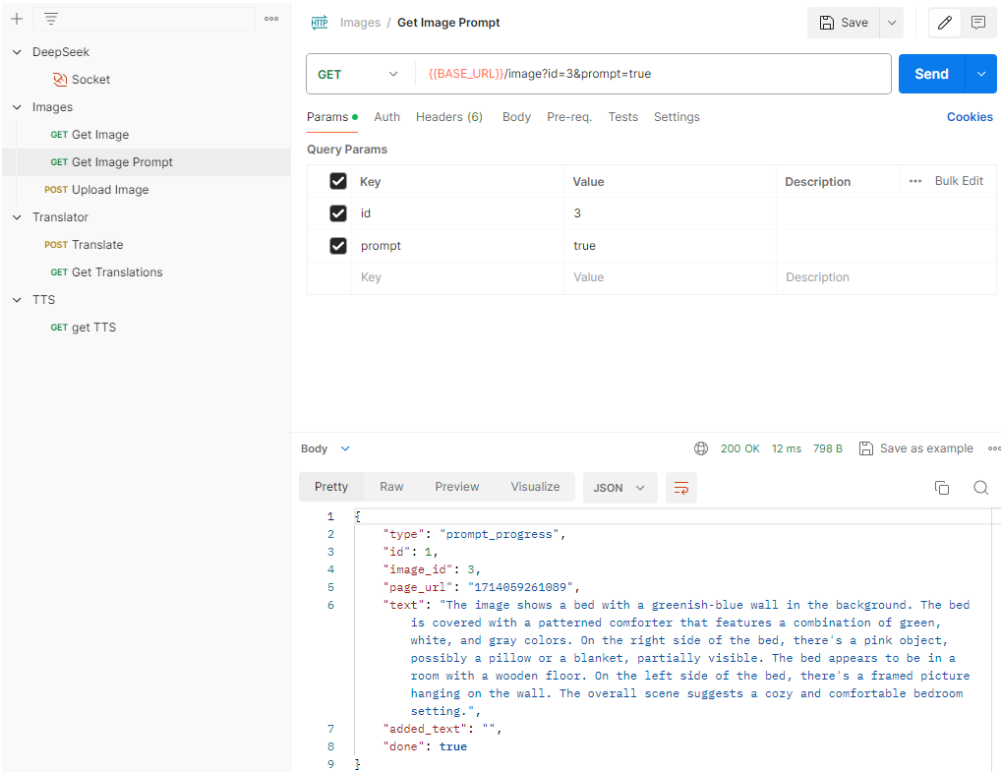


Рисунок 3 – Отримання результату оброблення зображення в Postman

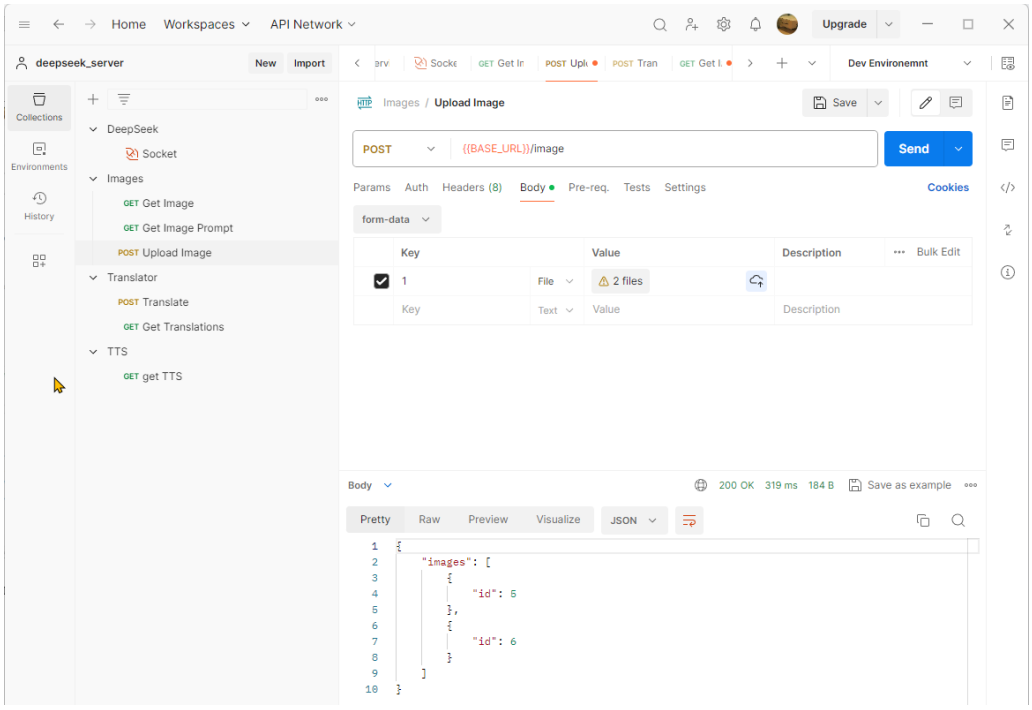


Рисунок 4 – Резьутат завантаження зображень у Postman

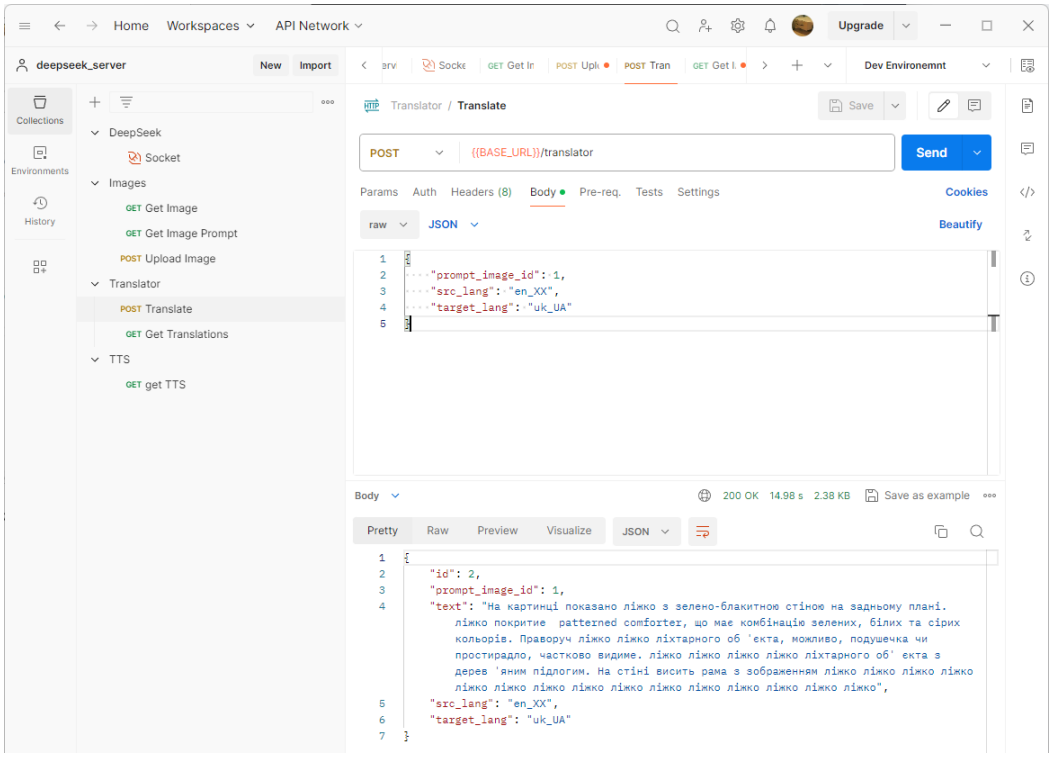


Рисунок 5 – Результат виконання запитну перекладу в Postman

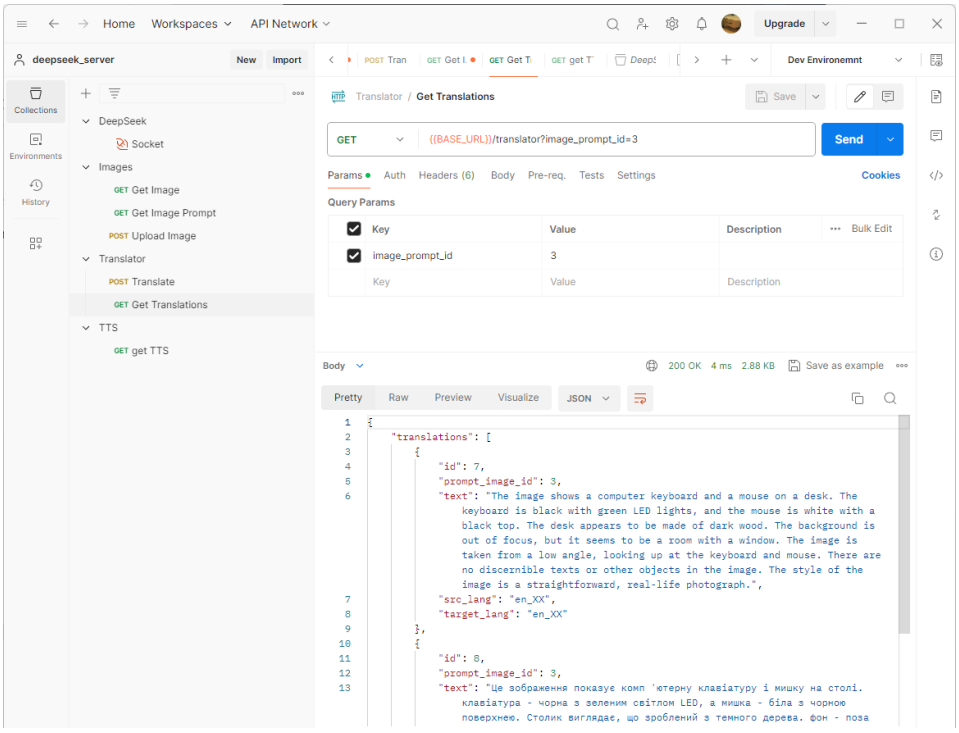


Рисунок 6 – Результат виконання запиту отримання наявних перекладів резльутату оброблення зображення у Postman

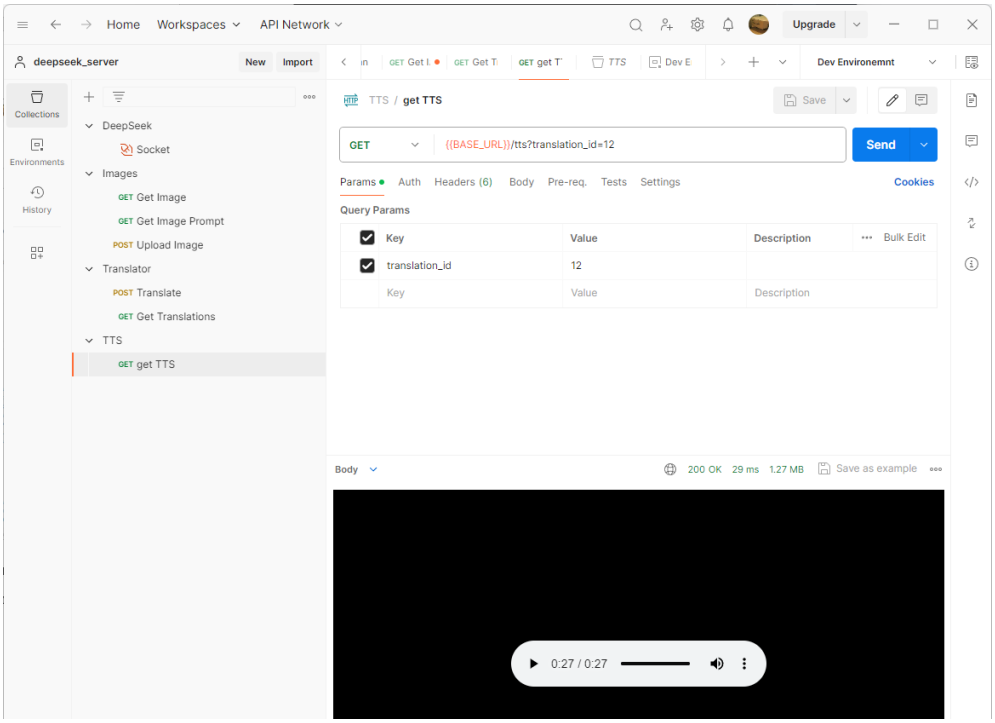


Рисунок 7 – Результат виконання запиту синтезу мовлення (TTS) в Postman

Додаток Ж

Реалізація сервісів мобільного застосунку

```
@RestApi(baseUrl: 'http://$baseUri/image')
abstract class ImagesClient {
  factory ImagesClient(Dio dio, {String baseUrl}) = _ImagesClient;

  @GET('')
  Future<ImagesGalleryResponse> getImages();

  @GET('')
  Future<ImagePromptModel> getImagePrompt({
    @Query('id') required String id,
    @Query('prompt') bool prompt = true,
  });
}


class ImagesService {
  Future<UploadedImagesResponse?> uploadImages(List<File> images)
async {
    FormData formData = FormData.fromMap({
      'files': images.map(
        (e) {
          print(e.path.split('/').last);
          return MultipartFile.fromFileSync(
            e.path,
            filename: e.path.split('/').last,
          );
        },
      ).toList(),
    });

    final res = await Dio().post(
      'http://$baseUri/image',
      data: formData,
    );

    try {
      print(res.statusCode);
      if (res.statusCode == 200) {
        return UploadedImagesResponse.fromJson(res.data);
      } else {
        return null;
      }
    } catch (e) {
```

```
      print(e);
      return null;
    }
  }
}
```

```
@RestApi(baseUrl: 'http://$baseUri/translator')
abstract class TranslationsClient {
  factory TranslationsClient(
    Dio dio, {
    String baseUrl,
  }) = _TranslationsClient;

  @POST('')
  Future<TranslationResponseModel> createTranslation(
      {@Body() required TranslationRequestModel body});

  @GET("")
  Future<TranslationListResponse> getAllTranslations({
    @Query('image_prompt_id') required int imagePromptId,
  });
}

@RestApi(baseUrl: 'http://$baseUri/tts')
abstract class TTSClient {
  factory TTSClient(Dio dio, {String baseUrl}) = _TTSClient;

  @GET('')
  @DioResponseType(ResponseType.bytes)
  Future<HttpResponse<List<int>>> getTTS({
    @Query('translation_id') required int translationId,
  });
}
```

Додаток З

Реалізація Cubits в мобільному застосунку

```
class ImagesGalleryCubit extends Cubit<ImagesGalleryState> {
  ImagesGalleryCubit() : super(ImagesGalleryInitial());

  final ImagesClient _imagesClient = ImagesClient(
    Dio(BaseOptions()),
  );
  final ImagesService _imagesService = ImagesService();

  Future<void> init() async {
    emit(
      ImagesGalleryLoading(),
    );
    try {
      final response = await _imagesClient.getImages();
      final images = response.images.reversed.toList();

      emit(
        ImagesGalleryLoaded(
          images: images,
        ),
      );
    } catch (e) {
      emit(
        ImagesGalleryError(
          message: e.toString(),
        ),
      );
    }
  }

  Future<void> _uploadFile(File file) async {
    await _imagesService.uploadImages([file]).then(
      (res) {
        init();
      },
    ).catchError((e) {});
  }

  Future<void> uploadImage(
    BuildContext context,
    ImageSource source,
  ) async {
    final image = await
```

```
ImagePicker.platform.getImageFromSource(source: source);

    if (image != null) {
      final file = File(image.path);
      await _uploadFile(file);
    }
  }
}


@immutable
abstract class ImagesGalleryState {
  const ImagesGalleryState();
}

class ImagesGalleryInitial extends ImagesGalleryState {}

class ImagesGalleryLoading extends ImagesGalleryState {}

class ImagesGalleryLoaded extends ImagesGalleryState {
  final List<ImageModel> images;

  const ImagesGalleryLoaded({
    required this.images,
  });
}

class ImagesGalleryError extends ImagesGalleryState {
  final String message;

  const ImagesGalleryError({
    required this.message,
  });
}


class WebSocketCubit extends Cubit<WebSocketState> {
  WebSocketCubit() : super(WebSocketInitial());

  static const String _serverAddress = 'ws://$baseUri/ws';

  final ImagesService _imagesService = ImagesService();

  WebSocket? _client;

  Future<void> connectToServer() async {
    emit(WebSocketConnectingToServer());

    try {
```

```
      _client = await WebSocket.connect(_serverAddress);
      _client!.listen((event) {
        print(event);
        final json = jsonDecode(event);
        final res = SocketResponse.fromJson(json);

        switch (res.type) {
          case SocketResponseType.prompt_init:
            emit(WebSocketPromptImage(
                answer: '',
                done: false,
                socketPromptImageInitResponse:
                    SocketPromptImageInitResponse.fromJson(json)));
            break;
          case SocketResponseType.prompt_progress:
            final progress =
SocketPromptImageProgressResponse.fromJson(json);
            if (state is WebSocketPromptImage) {
              emit(
                WebSocketPromptImage(
                  answer: progress.text,
                  done: progress.done,
                  socketPromptImageInitResponse: (state as
WebSocketPromptImage)
                      .socketPromptImageInitResponse,
                ),
              );
            }

            break;
        }
      }, onError: (e) {
        emit(WebSocketError(
            'Failed to connect to server, error: ${e.toString()}'));

        _client?.close();

        Future.delayed(const Duration(seconds: 1), () {
          connectToServer();
        });
      }, onDone: () {
        emit(WebSocketReconnectingToServer());

        Future.delayed(const Duration(seconds: 1), () {
          connectToServer();
        });
      });

      emit(WebSocketConnected());
```

```
    } catch (e) {
      emit(WebSocketError('Failed to connect to server\n
${e.toString()}'));
    }
  }

  Future<void> resetState() async {
    emit(WebSocketConnected());
  }

  Future<void> prompt(int imageId) async {
    if (_client == null) {
      return;
    }

    _client!.add(
      jsonEncode(
        SocketPromptImageRequest(
          imageId: imageId,
          pageUrl: DateTime.now().millisecondsSinceEpoch.toString(),
        ).toJson(),
      ),
    );
  }
}


@immutable
abstract class WebSocketState {
  const WebSocketState();
}

class WebSocketInitial extends WebSocketState {}

class WebSocketConnectingToServer extends WebSocketState {}

class WebSocketReconnectingToServer extends WebSocketState {}

class WebSocketError extends WebSocketState {
  final String message;

  WebSocketError(this.message);
}

class WebSocketConnected extends WebSocketState {}

class WebSocketPromptImage extends WebSocketState {
  final SocketPromptImageInitResponse? socketPromptImageInitResponse;
  final String answer;
```

```
  final bool done;

  const WebSocketPromptImage({
    this.socketPromptImageInitResponse,
    required this.answer,
    required this.done,
  });
}


class ImagePromptCubit extends Cubit<ImagePromptState> {
  ImagePromptCubit({
    required this.translationCubit,
  }) : super(ImagePromptInitial());

  final TranslationCubit translationCubit;

  final ImagesClient _imagesClient = ImagesClient(
    Dio(),
  );
  final TranslationsClient _translationsClient = TranslationsClient(
    Dio(),
  );

  Future<void> init(ImageModel imageModel) async {
    emit(
      ImagePromptLoading(),
    );
    try {
      final prompt = await _imagesClient.getImagePrompt(
        id: imageModel.id.toString(),
      );

      final translations = await
_translationsClient.getAllTranslations(
        imagePromptId: prompt.id,
      );

      emit(
        ImagePromptLoaded(
          image: prompt,
          translations: translations.translations,
        ),
      );

      onSetTranslation(TranslationCode.EN);

      translationCubit.startTranslation(prompt, TranslationCode.EN);
```

```
    } catch (e) {
      emit(const ImagePromptLoaded(
        image: null,
        translations: [],
      ));
    }
  }

  Future<void> onSetTranslation(TranslationCode code) async {
    final currentState = state;

    if (currentState is! ImagePromptLoaded || currentState.image ==
null) {
      return;
    }

    translationCubit.startTranslation(currentState.image!, code);
  }
}


@immutable
abstract class ImagePromptState {
  const ImagePromptState();
}

class ImagePromptInitial extends ImagePromptState {}

class ImagePromptLoading extends ImagePromptState {}

class ImagePromptLoaded extends ImagePromptState {
  final ImagePromptModel? image;
  final List<TranslationResponseModel> translations;

  final TranslationCode translationCode;
  const ImagePromptLoaded({
    required this.image,
    required this.translations,
    this.translationCode = TranslationCode.EN,
  });
}

class ImagePromptError extends ImagePromptState {
  final String message;

  const ImagePromptError({
    required this.message,
  });
}
```

```
class TranslationCubit extends Cubit<TranslationState> {
  TranslationCubit() : super(TranslationInitial());

  final TranslationsClient _translationsClient = TranslationsClient(
    Dio(),
  );

  Future<void> startTranslation(
      ImagePromptModel promptImage, TranslationCode code) async {
    try {
      emit(
        TranslationInProgress(
          translationCode: code,
        ),
      );

      final response = await _translationsClient.createTranslation(
        body: TranslationRequestModel(
            promptImageId: promptImage.id,
            srcLang: TranslationCode.EN,
            targetLang: code),
      );

      emit(
        TranslationDone(
          translation: response,
        ),
      );
    } catch (e) {
      emit(
        TranslationError(
          message: e.toString(),
          translationCode: code,
        ),
      );
    }
  }
}


@immutable
abstract class TranslationState {
  const TranslationState();
}

class TranslationInitial extends TranslationState {}
```

```
class TranslationInProgress extends TranslationState {
  final TranslationCode translationCode;

  const TranslationInProgress({
    required this.translationCode,
  });
}
class TranslationDone extends TranslationState {
  final TranslationResponseModel translation;

  const TranslationDone({
    required this.translation,
  });
}

class TranslationError extends TranslationState {
  final TranslationCode translationCode;
  final String message;

  const TranslationError({
    required this.translationCode,
    required this.message,
  });
}

class TtsCubit extends Cubit<TtsState> {
  TtsCubit({required this.translation}) : super(TtsInitial());

  final TranslationResponseModel translation;

  final TTSClient _ttsClient = TTSClient(
    Dio(),
  );

  Future<void> init() async {
    emit(
      TtsLoading(),
    );
    try {
      final tts = await _ttsClient.getTTS(
        translationId: translation.id,
      );

      if (tts.response.statusCode == 200) {
        emit(
          TtsLoaded(
            tts.response.data,
          ),
        );
```

```
      } else {
        emit(
          TtsError(
            tts.response.statusMessage ?? 'Error',
          ),
        );
      }
    } catch (e) {
      emit(
        TtsError(
          e.toString(),
        ),
      );
    }
  }
}

@immutable
abstract class TtsState {}

class TtsInitial extends TtsState {}

class TtsLoading extends TtsState {}

class TtsLoaded extends TtsState {
  final Uint8List tts;

  TtsLoaded(this.tts);
}

class TtsError extends TtsState {
  final String error;

  TtsError(this.error);
}
```

метадані

Заголовок

Розробка, налагодження та оптимізація роботи простого інтерпретатора зображень для незрячих

Автор

Вівчаренко А. В. Науковий керівник / Експерт

підрозділ

King Danylo University

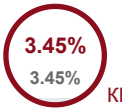
Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

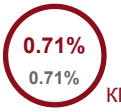
Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли	·	64
Білі знаки	ß	0
Парафрази (SmartMarks)	a	28

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



КП 1



КП 2

25
Довжина фрази для коефіцієнта подібності 2

11368
Кількість слів

96678
Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	http://repository.ukd.edu.ua/bitstream/handle/123456789/392/%D0%9A%D0%A0%20%D0%9F%D0%B0%D1%88%D0%BD%D0%B8%D0%BA%20%D0%90.%20%D0%9F..pdf?sequence=1	45	0.40 %
2	http://repository.ukd.edu.ua/bitstream/handle/123456789/396/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0%20%D0%A1%D1%82%D1%80%D1%96%D0%BB%D0%B5%D1%86%D1%8C%D0%BA%D0%B8%D0%B9.pdf?sequence=1	36	0.32 %
3	http://repository.ukd.edu.ua/bitstream/handle/123456789/381/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0%20%D0%92%D0%BE%D0%BB%D0%BA%20%D0%90.pdf?sequence=1	24	0.21 %