Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Інститут інформатики та радіоелектроніки,
Факультет комп'ютерних наук і технологій
(повне найменування інституту, назва факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалавр
(ступінь вищої освіти)

на тему РОЗРОБКА ТЕЛЕГРАМ БОТА ДЛЯ ЗБЕРЕЖЕННЯ НОТАТОК
TELEGRAM BOT DEVELOPMENT FOR SAVING NOTES

Виконав: студент(ка) 4 курсу, групи КНТ-117
Спеціальності 121 Інженерія програмного забезпечення
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Інженерія програмного забезпечення

Дьячков В. В.
(прізвище та ініціали)
Керівник Федорончак Т. В.
(прізвище та ініціали)
Рецензент Казурова А. Є.
(прізвище та ініціали)

2021

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Інститут, факультет ІІПЕ, ФКНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
_____С.О. Субботін
“____” _____20___року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

Дьячкова Віталія Володимировича
(прізвище, ім’я, по батькові)

1. Тема проєкту (роботи) Розробка Телеграм бота для збереження нотаток. Telegram Bot Development for Saving Notes
керівник проєкту (роботи) Федорончак Тетяна Василівна, к.т.н., доцент ,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від “30” березня 2021 року № 103
2. Строк подання студентом проєкту (роботи) 21 травня 2021 року
3. Вихідні дані до проєкту (роботи) інформація про сучасні підходи до розробки, перелік існуючих систем керування базами даних, мов програмування, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) аналіз предметної області, розробка архітектури програми, розробка програми, експлуатація, тестування та експериментальне дослідження програми

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	Федорончак Т.В., доцент		
Нормоконтроль	Дейнега Л.Ю., ст. викладач		

7. Дата видачі завдання “ 28 ” лютого 2021 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області	2-3 тижні	Розділ 1
3	Розробка архітектури програми	4 тиждень	Розділ 2
4	Розробка програми	5-6 тижні	Розділ 3
5	Тестування та експериментальне дослідження програми	7 тиждень	Розділ 4
6	Оформлення пояснювальної записки та документів до неї. Нормоконтроль та рецензування	8 тиждень	Додатки
7	Захист роботи	9 тиждень	

Студент(ка) _____ Дьячков В.В
(підпис) (прізвище та ініціали)

Керівник проєкту (роботи) _____ Федорончак Т.В.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 84 с., 10 табл., 36 рис., 3 дод., 22 джерела.

TELEGRAM, BOT, NOTES, PYTHON, ASYNCIO, AIOGRAM, POSTGRESQL, REDIS, DOMAIN-DRIVEN DESIGN, HEXAGONAL ARCHITECTURE, TEST-DRIVEN DEVELOPMENT, PYTEST, DEPENDENCY INJECTION, DEPENDENCY INVERSION PRINCIPLE, BACHELOR

Об’єкт дослідження: процес створення нотаток.

Предмет дослідження: чат-бот для управління (перегляд, створення, редагування, видалення) короткими текстовими нотатками.

Мета роботи: розробка чат-боту для управління нотатками.

Матеріали, методи та технічні засоби: об’єктно-орієнтоване програмування, асинхронне програмування, предметно-орієнтоване проектування, розробка через тестування, методології екстремального програмування, гексагональна архітектура, Telegram Bot API, мануальне тестування, бібліотеки aiogram, pytest та інші, персональний комп’ютер під керуванням процесору Intel Core i5-6500 та управлінням операційної системи Manjaro Linux 21.0.4, IDE PyCharm Professional 21.04

Результати: готовий до роботи чат-бот, який надає користувачам месенджера можливість для управління короткими текстовими нотатками.

Висновки: результати роботи повністю відповідають поставленій меті та визначеному технічному завданню до проекту, вдало реалізована архітектура програми надає можливість легко визначити для неї новий інтерфейс користувача.

Галузь використання: приватний чат з будь-яким користувачем месенджера Telegram.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 84 pages, 10 tables, 36 figures, 3 appendixes, 22 sources.

TELEGRAM, BOT, NOTES, PYTHON, ASYNCIO, AIOGRAM, POSTGRESQL, REDIS, DOMAIN-DRIVEN DESIGN, HEXAGONAL ARCHITECTURE, TEST-DRIVEN DEVELOPMENT, PYTEST, DEPENDENCY INJECTION, DEPENDENCY INVERSION PRINCIPLE, BACHELOR

The object of research is the process of saving personal notes.

The subject of research is the chat-bot for managing (creating, reading, updating, deleting) short text notes.

The purpose of this work is to develop chat-bot for managing notes.

List of materials, methods and technical means: object-oriented programming, asynchronous programming, domain-driven design, test-driven development, hexagonal architecture, Telegram Bot API, manual QA, programming libraries aiogram, pytest, personal computer with Intel Core i5-6500 CPU and OS Manjaro Linux 21.0.4, IDE PyCharm Professional 21.0.4.

The results: ready to deploy, working chat-bot that allows messenger users to manage short text notes.

The conclusion: the results of work fully comply to the purpose of this work and the technical task. A well-developed architecture allows it to easily define the new possible use cases for this system

The application area is the private chat with any Telegram messenger user.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface;

CRUD – Create, Read, Update, Delete;

DDD – Domain-Driven Design;

DI – Dependency Injection;

DIP – Dependency Inversion Principle;

IoC – Inversion of Control;

ORM – Object-Relational Mapping;

SQL – Structured Query Language;

ОС – Операційна Система;

СКБД – Система Керування Базами Даних;

ВСТУП

Велике розмаїття програмного забезпечення, застосунків для персональних комп’ютерів, смартфонів або планшетів призвели до того, що люди надають перевагу універсальнім інструментам, які можуть робити все і одразу. Сьогоднішній користувач буде дуже прискіпливо ставитись до встановлених ним застосунків (особливо, якщо мова йде про мобільні пристрої з малим об’ємом вбудованої пам’яті).

Разом з тим, як розробники програмного забезпечення намагаються надати своєму застосунку як можна більше корисних функцій, користувацька спільнота намагається модифікувати зручні для них інструменти, щоб підлаштувати їх під свої потреби. Багато програмного забезпечення, що розробляється у наш час має систему плагінів або відкритий API (Application Programming Interface) для розширення можливостей програми, її інтеграції з іншими додатками.

Однією з найпопулярніших таких функцій є можливість додавання чат-ботів у різноманітні мессенджери, такі як Telegram, Viber, WhatsApp, Facebook Messenger та ін. Вони дозволяють, наприклад, автоматизувати комунікацію людей з якимись сервісами (наприклад, банками, онлайн-магазинами тощо), а також перенести виконання якихось рутинних справ у звичний для себе інструмент, яким користуєшся кожен день.

Такими рутинними справами може бути ведення домашнього бюджету, або ж, наприклад, створення приміток. Короткі текстові нотатки, які будуть нагадувати про щось на майбутнє. Повідомлення, які можна буде швидко переглянути, уточнити інформацію, під час онлайн-спілкування з кимось зі своїх знайомих.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Зазвичай, програми для створення нотаток окрім безпосереднього збереження текстових даних також надають користувачу ряд додаткових можливостей, таких як розширений пошук, можливість збереження мультимедіа (зображень, відео, аудіо, «офісних» форматів, таких як .docx або .xlxs, можливість поширення своїх нотаток іншим, а також їх оформлення за допомогою інструментів Rich Text), але створення чат-боту ставить розробника у певні рамки:

- обмежений набір інструментів відображення: як що від час створення звичних застосунків у руках розробника є всі можливості розмітки (так, для прикладу, у веб-застосунках використовується мова розмітки HTML), то мессенджери зазвичай надають можливість лише створення інтерактивних кнопок і простої розмітки тексту;
- обмежений набір інструментів керування: як зазначалось раніше, користувачу у більшості випадків будуть доступні лише прості кнопки і текстові команди, ніяких перемикачів, слайдерів тощо.

Крім того, під час підготовки до проектування чат-боту слід брати до уваги вимоги кінцевого користувача. Потенціальний клієнт буде використовувати чат-бота саме через те, що він виконує тільки одну чи декілька поставлених перед ним задач (в нашому випадку – тільки управління нотатками) і те, що їм звично користуватись в рамках улюбленого месенджера чи соціальної мережі. Якщо користувачу потрібен «просунутий» інструмент, або він не може для себе знайти зручного бота – користувач надає перевагу традиційним застосункам, які не мають тих обмежень, що мають чат-боти.

1.2 Огляд месенджерів, що надають можливість написання чат-ботів

Як вже було зазначено у вступі до дипломної роботи, більшість сучасних месенджерів надають можливості розробниками для написання чат ботів, найчастіше за допомогою відкритого Web API. Наповнення месенджера ботами підвищує його привабливість для кінцевого користувача, наповнює його новими смислами.

Під час аналізу доступних на ринку пропозицій були розглянуті наступні продукти:

- WhatsApp;
- Viber;
- Telegram;
- Facebook Messenger.

До переліку вище не потрапили також популярні у світі:

- Discord (через те, що це застосунок переважно для голосового спілкування; боти у Discord найчастіше використовуються для управління Discord-серверами та каналами);
- ВКонтакті (через те, що застосунок заблокований на території України);
- WeChat (через те, що орієнтований виключно на китайський ринок).

З-поміж усього різноманіття представлених продуктів перевагу було надано месенджеру Telegram. До його основних переваг можна віднести:

- достатню популярність [1]: з початку 2021 року Telegram займає перше місце за кількістю завантажень серед месенджерів у менеджерах застосунків Play Market та App Store;
- велику інфраструктуру: Telegram надає найбільшу кількість можливостей розробнику порівняно з іншими месенджерами, окрім того він має гарно оформлену, повну документацію до свого Web API [3], а також

багато клієнтських бібліотек з відкритим програмним кодом на більшості мовах програмування загального застосунку. Багато онлайн-сервісів для хостингу застосунків надають окремі пропозиції для розгортання саме Telegram-ботів у «хмарі»;

- широку спільноту розробників: як вже зазначалось, існує як багато клієнтських бібліотек для написання самих ботів, так і прикладів реалізації, статей, відео-уроків тощо.

Не дивлячись на те, що WhatsApp за останніми дослідженнями все ще залишається найпопулярнішим месенджером в світі [2], його популярність починає зменшуватись (через оновлення політики конфіденційності у застосунку). Крім того, до 2018 року WhatsApp не мав можливості створення ботів і інфраструктура для їх написання все ще залишається не такою розвиненою як у конкурентів.

Viber та Facebook Messenger не мають тих можливостей розмітки, що надає Telegram. Крім того, аудиторія Telegram, за статистикою, є молодшою і краще знайома з інформаційними технологіями і сучасними застосунками. Також, Telegram частіше використовується для «незвичних» чат-ботів, котрі не виконують функції консультанта в онлайн-магазині, а автоматизують процеси реального життя.

1.3 Аналіз застосунків–аналогів

Перш ніж приступати до формалізації технічного завдання і проектування системи необхідно переглянути наявні на ринку пропозиції. Так, на стадії аналізу було розглянуто два популярних Telegram-бота, що призначені для збереження нотаток:

- «Мои заметки» (@ExNoBot) [4];
- «Дела» (@Killthemall_bot) [5].

Обидва варіанти є першими, що видаються за пошуковим запитом до Google. Під час пошуку також розглядались інші варіанти (@Notepad-bot, @dotobot), але всі вони виявились вже не працюючими.

Порівняння вищезазначених ботів наведено у табл. 1.1.

Таблиця 1.1 – Порівняння Telegram-ботів для створення нотаток

Критерій порівняння	**@ExNoBot**	**@Killthemall_bot**
Можливість створення нотаток	+	+
Можливість перегляду збережених нотаток	+	+
Можливість пошуку за нотатками	-	-
Можливість створення відкладених нотаток (нагадувань)	+	-
Можливість редагування нотаток	+	+
Можливість видалення нотаток	+	+
Можливість перегляду списку нотаток	-	-

1.4 Результати проведеного аналізу

За результатами аналізу предметної області було визначено, що при створенні чат-боту головним фактором успіху є створення простого для користування застосунку, що буде гарно виконувати одну свою функцію.

Під час огляду сервісів, що надають змогу написання чат-ботів оптимальним варіантом було обрано месенджер Telegram через його популярність і гарно розвинену інфраструктуру.

Подивившись на наявні в Telegram боти для збереження нотаток, а також провівши їх порівняльну характеристику (таблиця 1.1), стало видно, що готовим на ринку рішенням бракує можливостей зручного перегляду одразу всіх приміток, а також можливості пошуку за ними.

Враховуючи усе вищезазначене, було створено технічне завдання до проекту, наведене в Додатку А.

2 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

2.1 Вибір засобів реалізації

2.1.1 Вибір мови програмування

Враховуючи вимоги до програмного забезпечення, визначені на попередньому етапі, обираючи мову програмування треба звернути увагу на наступні фактори:

- можливість відправлення HTTP-запитів (для взаємодії з Telegram Bot API);
- наявність клієнтських бібліотек для взаємодії з Telegram Bot API (тобто, «обгортки», що буде керувати відправкою запитів до Telegram);
- можливість відправлення запитів до бази даних (тобто наявність бібліотеки-драйверу для роботи з нею; це необхідно для управління даними про користувацькі нотатки);
- наявність навчальних статей, уроків з реалізації Telegram-ботів на розглянутій мові програмування;
- персональні вподобання розробника.

Беручи до уваги усі вищезазначені фактори, було обрано мову програмування Python, оскільки ця мова програмування найперше рекомендується для написання Telegram-ботів, має безліч готових клієнтських бібліотек для взаємодії з Telegram API, має велику спільноту розробників, що постійно поліпшують наявні клієнтські бібліотеки та пишуть навчальні статі, приклади їх використання.

Python також є мовою програмування, що рекомендується для обробки даних великих потоків даних. Завдяки своєму простому синтаксису, а також повної, насиченої стандартної бібліотеки, Python добре підходить для швидкої розробки програмного забезпечення.

Крім усього вищезазначеного, Python диктує стандарт Database API (DB API) [6], який реалізують усі «рушії» для роботи з базами даних (яких

для Python написано безліч), а отже, обираючи цю мову програмування ми залишаємо за собою свободу вибору СКБД.

2.1.2 Вибір клієнтської бібліотеки

Як вже було зазначено раніше, на Python написано безліч клієнтських бібліотек для роботи з Telegram Bot API. Вони виступають «обгорткою» над інтерфейсом Telegram, надаючи розробнику набір функцій і класів для роботи з ним.

Порівняльний аналіз найпопулярніших клієнтських бібліотек наведено у табл. 2.1.

Таблиця 2.1 – Порівняльний аналіз клієнтських бібліотек для роботи з Telegram Bot API на Python

Назва бібліотеки	**Переваги**	**Недоліки**
pyTelegramBotAPI [7]	Проста і дуже зручна для використання реалізація, що надає мінімально можливий набір інструментів для роботи з Telegram Bot API.	Найповільніша за часом обробки запиту з розглянутих бібліотек
	Є найпопулярнішою з розглянутих, має велику спільноту розробників	Надає можливість лише для базової взаємодії з користувачем, бракує функціональності порівняно з аналогами

Продовження таблиці 2.1

Назва бібліотеки	**Переваги**	**Недоліки**
Pyrogram [8]	Використовує MTProto API (власний стандарт, розроблений Telegram), а не Telegram Bot API, що позитивно впливає на швидкодію програми	MTProto API реалізує не всі можливості Telegram Bot API
aiogram [9]	Є асинхронною бібліотекою, що робить її найшвидшою серед розглянутих	Оскільки можливість написання асинхронного коду у Python з'явилася відносно недавно, не всі бібліотеки підтримують такий стиль написання, використання синхронних рішень при цьому вбиває увесь сенс використання aiogram
	Надає можливості роботи з кінцевим автоматом, зверненням до Telegram Bot API як за допомогою webhook так і з використанням polling	
	Є найновішою з розглянутих бібліотек, а отже має найменшу кількість застарілого (Legacy) коду, використовує найновіші функції мови програмування Python	

Використовуючи результати порівняння у табл. 2.1 було обрано бібліотеку «aiogram».

2.1.3 Вибір сховища даних

У якості сховища даних доцільно використовувати одну з реляційних СКБД (систему керування базами даних). По-перше, таким чином не треба буде «вигадувати» свій власний формат зберігання даних. По-друге, СКБД вже буде реалізовувати основні операції для роботи з даними. По-третє, на Python існує багато двигунів для роботи з реляційними базами даних, що реалізують стандарт DB API. Також, модель даних у реляційних СКБД легко знаходить своє відображення у об’єктно-орієнтованому програмуванні.

Для використання було обрано СКБД PostgreSQL. Вона має свій власний, насичений SQL-діалект, є дуже швидка сама по собі, а також має дуже високопродуктивні двигуни для роботи з Python. Одним з таких є асинхронний двигун «asyncpg», який і буде використано під час розробки програмного забезпечення. Саме ця СКБД найчастіше використовується під час розробки Python-проектом і є чимось на кшталт стандарту у цій сфері.

Для збереження стану користувача, даних про його поточну сесію буде використано сховище типу «ключ-значення» Redis, яке також є «стандартом» для виконання саме цієї задачі.

2.1.4 Вибір інструментів тестування

Не дивлячись на те, що Python має вбудований пакет для модульного тестування «unittest» (що реалізує стандарти тестових пакетів xUnit) [10], більшою популярністю користується стороння бібліотека pytest [11]. До її переваг можна віднести:

- високий рівень модульності: до pytest можливо писати свої власні плагіни, розширювати їх за допомогою вже написаних спільнотою, розбивати всі свою тестові модулі на довільну структуру. Одним з таких плагінів, який буде використано дозволяє легко тестувати асинхронний код;

- більш детальний опис помилок при провалі тесту (рівень деталізації опису можна регулювати через параметри командного рядка);
- використання «нативних» інструментів мови програмування Python, а саме ключового слова «assert»;
- зворотня сумісність з тестами, написаними за допомогою модуля unittest.
- слідування стандарту написання Python-коду PEP8 [12];
- можливість параметризації тестів;
- зручний інструмент для написання «фікстур» - вхідних даних для тестів.

2.1.5 Вибір допоміжних інструментів розробки

Для спрощення процесу розробки було прийнято рішення про використання системи контролю версій Git, що дозволить легко переносити проект з однієї машини на іншу, легко відстежувати зміни в проекті, контролювати додавання нових ітерацій проекту тощо.

Використання Git також надасть можливість в подальшому використовувати інструмент «pre-commit» [13], що автоматизує процес контролю якості коду, завдяки запуску лінтерів та тестів при кожному додаванні нового коду до репозиторію.

Задля автоматизації процесу розгортання проекту буде використано програмне забезпечення Docker як засіб контейнеризації. Це програмне забезпечення дозволить легко запускати проект на будь-якому пристрої, що підтримує Docker.

Для контролю якості коду будуть використані наступні інструменти:

- «flake8»: лінтер для мови програмування Python, що перевіряє відповідність коду стандартам програмування PEP8, а також підтримує різноманітні плагіни;

- «mypy»: лінтер для мови програмування Python, що перевіряє правильність використання анотацій типів;
- «isort»: форматор коду Python, що розставляє імпорти сторонніх модулей у визначеному порядку;
- стандартній набір «pre-commit-hooks»: форматери коду, лінтери форматів TOML, YAML і т.і.

Для полегшення процесу управління залежностями проекту буде використано інструмент Poetry. Poetry є обгорткою над стандартним менеджером пакетів мови програмування Python – pip, але дозволяє розділяти залежності на production та development, автоматизує процес створення віртуального середовища, жорстко визначає використані версії бібліотек і надає ще велику кількість переваг, порівняно зі стандартним інструментом.

2.2 Розробка архітектури програми

Під час проектування системи було прийнято рішення про використання предметно-орієнтованого підходу (англ. «Domain-Driven Design», DDD) [14]. Він дозволяє вдало розділити пакети, класи і функції проектованого програмного забезпечення по окремим модулям, кожен з яких відноситься до певного фрагменту предметної області.

Під час аналізу предметної області було виділено наступні її сутності:

- клієнт, тобто користувач Telegram, автор нотаток;
- нотатки, короткі текстові повідомлення, що необхідно зберігати.

Відповідно до визначених сутностей предметної області будуть побудовані пакети «accounts» та «notes» майбутнього програмного застосунку, кожен з яких буде складатись з наступних шарів:

- domain – шар предметної області, що визначає сутності предметної області, також «модель» системи, критичні бізнес правила, інтерфейси для їх реалізації;

- application – шар, де визначаються варіанти використання і сервіси, що оперують сутностями предметної області;
- infrastructure – шар, де визначається взаємодія пакету із зовнішнім світом, описується його API.

Графічне зображення описаної архітектури зображено на рис. 2.1.

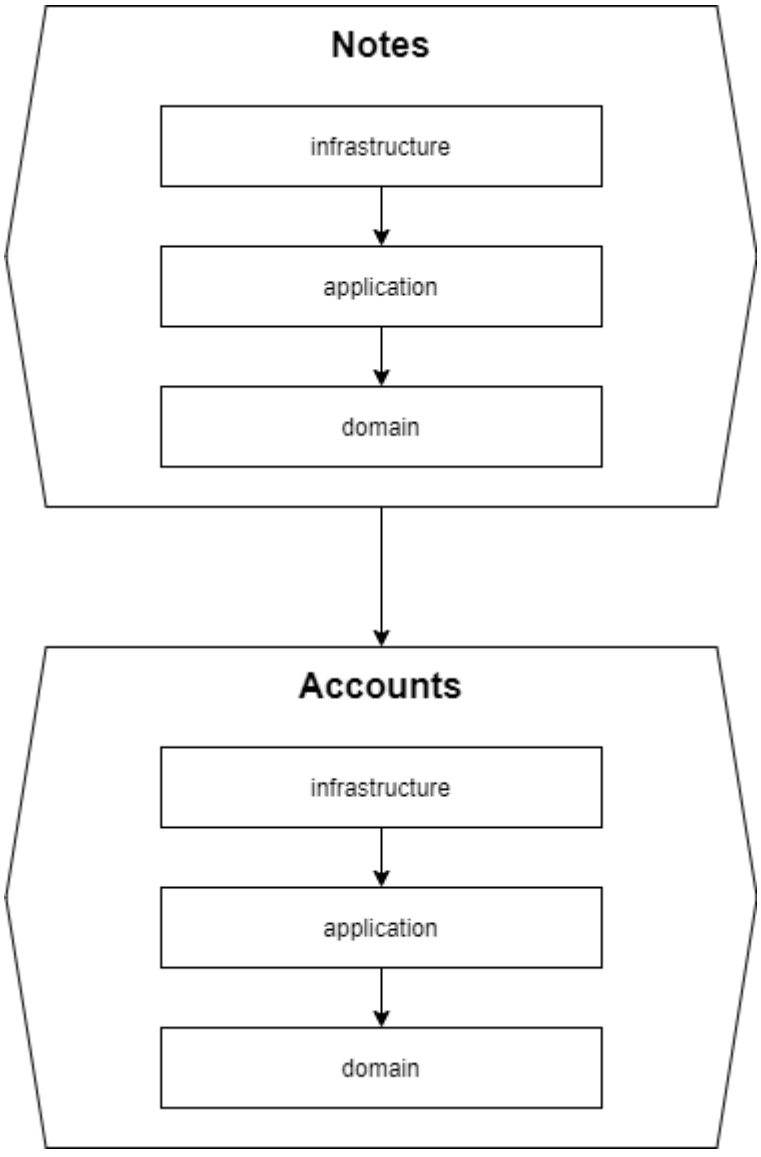


Рисунок 2.1 – Графічне подання розроблених модулів

Як видно з рис. 2.1, напрямок залежностей іде зверху донизу. Жоден нижчий шар не повинен посилатися на шари, шо знаходяться вище.

Модулі програми також виконані у формі шестикутників. Це зроблено, щоб показати, що для їх проектування будуть також

використовуватись принципи гексагональної архітектури [15], також відомої як архітектури портів та адаптерів. Такий підхід дозволить, наприклад, додаючи нові модулі у шарі «infrastructure» знаходити все нові можливості використання для застосунку. Те, що сьогодні було Telegram-ботом завдяки незначним змінам завтра зможе стати чат-ботом для Viber, WhatsApp або навіть повноцінним веб-застосунком. Формат вхідних та вихідних даних не має визначного значення при такому підході і може бути завжди зміненим. Це, в більшості, досягається завдяки відмежуванню «infrastructure» від «application».

3 РОЗРОБКА ПРОГРАМИ

3.1 Розробка схеми бази даних

Оскільки предметна область не є достатньо складною і було виділено всього дві її сутності проектування бази даних не ставить перед розробником нерішучої задачі.

Розроблена схема бази даних наведена на рис. 3.1.

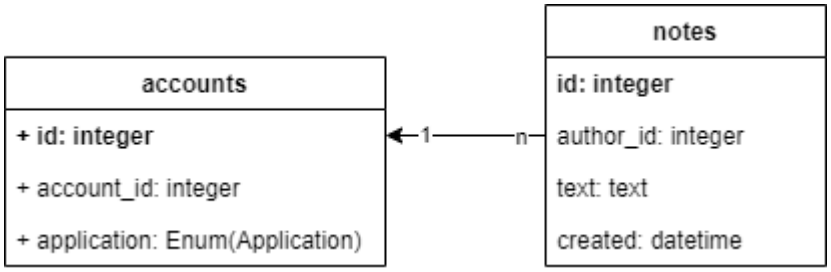


Рисунок 3.1 – Схема розробленої бази даних

Відношення «accounts» містить в собі наступну інформацію:

- «id»: штучно створений унікальний ідентифікатор користувача в нашій системі;
- «account_id»: унікальний ідентифікатор користувача у сторонньому сервісі (наприклад, Telegram);
- «application»: вказує на те, звідки було взято дані цього користувача. Як видно, поле «application» має тип «Enum(Application)», що містить перелік усіх сторонніх сервісів, з яких бере дані наш застосунок. На даний момент цей перелік складається лише з Telegram, але подібна модель даних надасть можливість у майбутньому пристосувати застосунок для нових потреб (наприклад, зробити з нього ще й Viber-бота).

Крім усього вищезазначеного, на відношення «accounts» також накладається обмеження унікальності на пару полей «account_id» та «application»: жоден користувач, зареєстрований у сторонньому сервісі не може мати два чи більше відображень у нашій базі даних.

Відношення «notes» зберігає такі дані:

- «id»: штучно створений унікальний ідентифікатор для кожної примітки;
- «author_id»: посилання на автора примітки – запис з таблиці «accounts»; слід також звернути увагу на те, що «author_id» посилається саме на внутрішній унікальний ідентифікатор користувача «accounts.id», а не на отриманий ідентифікатор зі стороннього сервісу «accounts.account_id»;
- «text»: повний текст примітки (збережений разом з розміткою, якщо така є);
- «created»: дата та час створення примітки, ми зберігаємо ці дані задля того, що відображати найновіші нотатки користувача в першу чергу.

3.2 Налаштування взаємодії з базою даних

Як вже зазначалось у пункті 2.1.3, для взаємодії з базою даних зручніше використовувати ORM, оскільки це дає наступні переваги:

- використання «нативних» інструментів мови програмування Python, взаємодія з базою даних через звичні функції/класи замість використання спеціалізованих запитів (наприклад, мовою SQL (Structured Query Language));
- відокремлення взаємодії з базою даних через високорівневий інтерфейс: це надає змогу змінити використовувану СКБД не вносячи жодних змін в код;
- ORM бере на себе відповідальність за управління підключенням, веденням сесій, відкриття та закриття транзакцій тощо, що дозволяє розробнику зосередитись на безпосередньому вирішенні задачі, а не методів її досягнення.

Для використання під час розробки було обрано ORM «SQLAlchemy», що є найпопулярнішою у спільноті Python-програмістів. На її вибір також вплинули наступні фактори:

- наявність інтегрованого з нею інструменту для створення «міграцій» бази даних – «alembic», що дозволяє покроково вносити зміни до моделі даних і не боятися при цьому втрати певної частини інформації;
- високий рівень підтримки PostgreSQL завдяки своєму модулю «dialects» - так, наприклад, SQLAlchemy дозволяє використовувати специфічні для PostgreSQL типи даних JSONField та ArrayField, можливості повнотекстового пошуку, спеціальних оптимізованих індексів тощо;
- підтримка асинхронних запитів до бази даних, інтеграція з різними асинхронними двигунами для зв’язку з базами даних, що дозволяє оптимізувати робочий час виконання програми.

3.3 Розробка компонентів системи

3.3.1 Структура кореневої папки проекту

Була реалізована спроектована на попередньому етапі архітектура проекту. Структура кореневої папки проекту наведена на рис. 3.2.

На стадії проектування було виділено два основних «гексагони» - «accounts» та «notes». Для того, щоб не засмічувати кореневу папку проекта деталями реалізації обидва пакети були поміщені в інший пакет з назвою «src» (скор. від англ. «source» - джерело). Разом з ними було також розміщено пакет «common», що зберігає спільний для гексагонів код: ініціалізує підключення до всіх необхідних сервісів: бази даних, Telegram-бота тощо, визначає абстрактні базові класи.

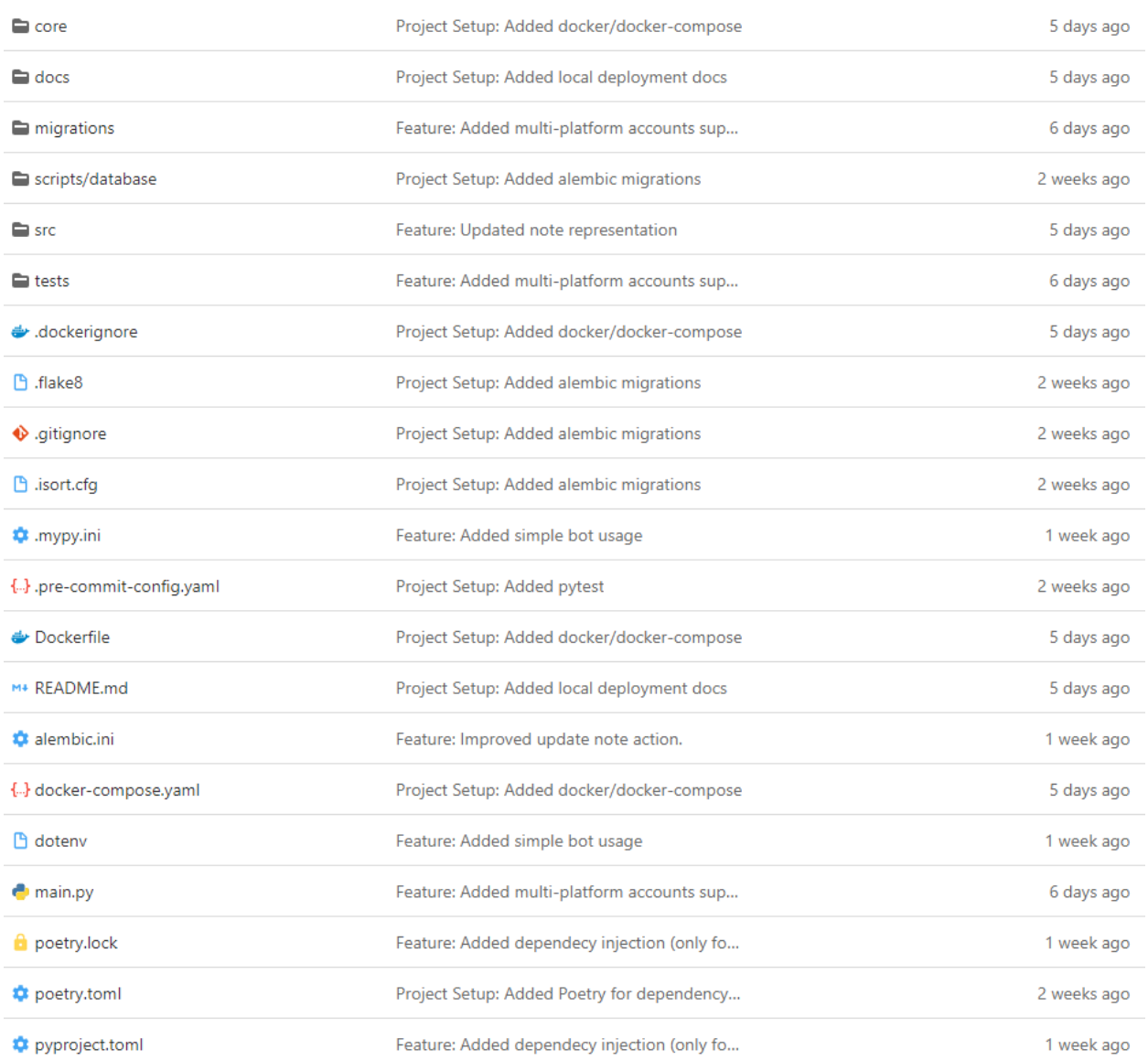


Рисунок 3.2 – Структура кореневої папки проекту

Весь код, що стосується конфігурації проекту (наприклад, витягнення змінних середовища) було поміщено у пакет «core», що має субмодуль «config».

Всю документацію стосовно проекту було розміщено у директорії «docs».

Задля автоматизації процесу розробки і розгортання проекту було написано декілька bash-скриптів, які було розміщено у директорії «scripts».

Міграції бази даних, а також налаштування інструменту «alembic» було розміщено у директорії «migrations» та файлі «alembic.ini» відповідно.

Автоматичні модульні тести було розміщено у пакеті «tests».

Також, зразок заповнення змінних середовища було залишено у файлі «dotenv».

3.3.2 Структура розроблених гексагонів

Кожен з виділених гексагонів реалізує структуру, зображену на рис. 3.3.

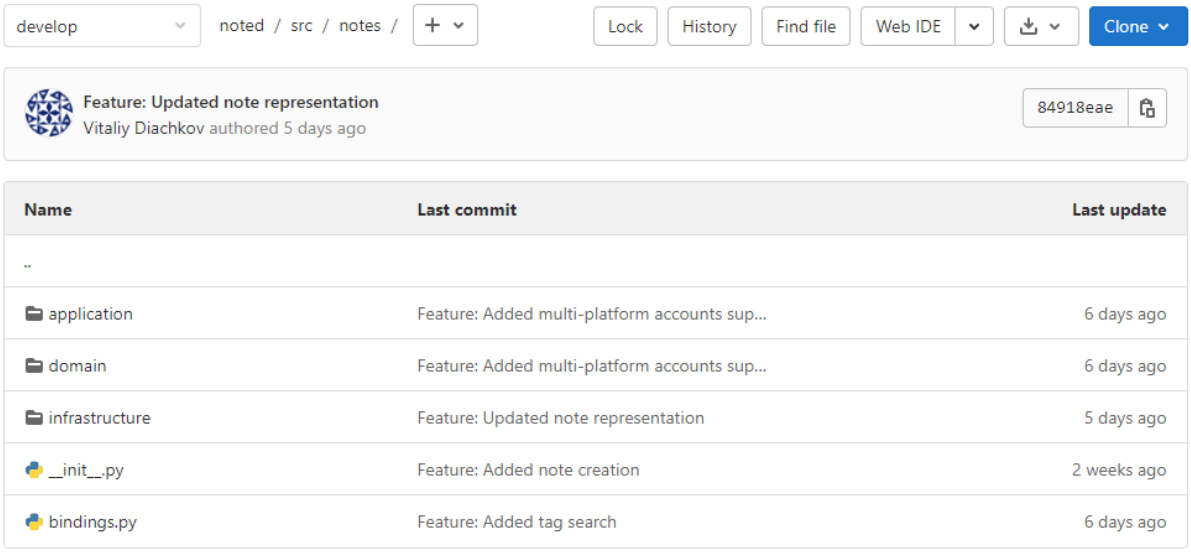


Рисунок 3.3 – Структура розроблених гексагонів

Три внутрішніх пакети зберігають код, що відноситься відповідно до «domain», «application» та «infrastructure» шарів. Також, в корені кожного гексагону знаходяться додаткові файли конфігурації, якщо такі потрібні. Так, наприклад, на рис. 3.2 зображено файл «bindings.py», що зберігає налаштування інструменту впровадження залежностей (англ. «Dependency Injection», DI) [16].

Впровадження залежностей — шаблон проєктування програмного забезпечення, що передбачає надання зовнішньої залежності програмному компоненту, використовуючи «інверсію управління» (англ. «Inversion of control», IoC) для розв'язання (отримання) залежностей [17].

Впровадження — це передача залежності (тобто, сервісу) залежному об'єкту (тобто, клієнту). Передавати залежності клієнту замість дозволити клієнту створити сервіс є фундаментальною вимогою до цього шаблону проектування.

Типовий domain-пакет реалізує структуру, наведену на рис. 3.4. Тут описується класи, що є відображеннями сутностей предметної області, реалізуються класи-фабрики для створення об’єктів, а також визначаються інтерфейси для класів-сховищ (сукупності). Усі зазначені класи та інтерфейси є втіленням основних будівельних блоків DDD.

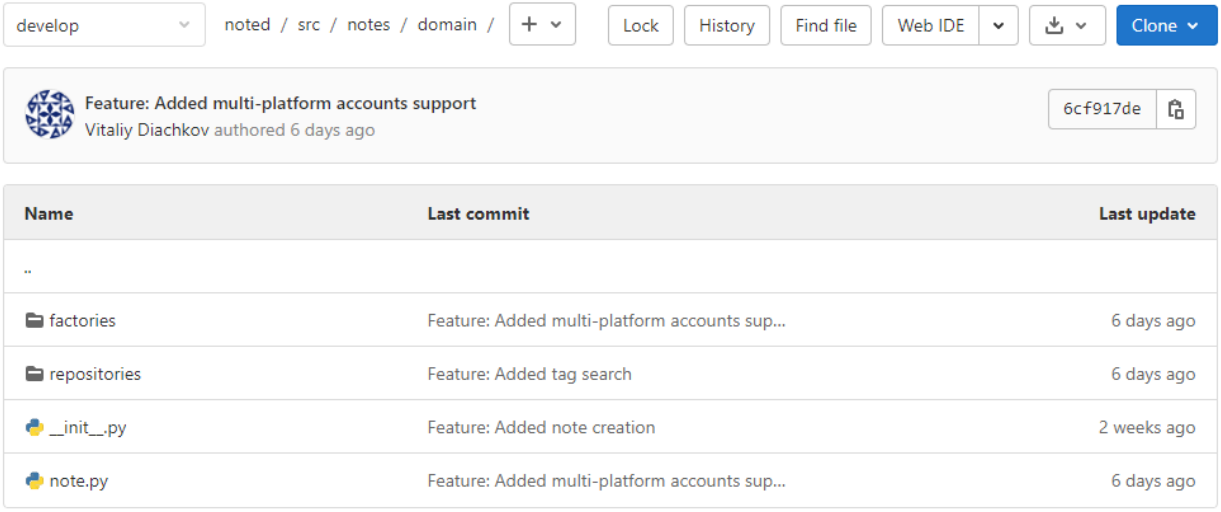


Рисунок 3.4 – Структура типового domain-пакету

Сутність являє собою категорію індивідуальних об'єктів, які залишаються незмінними на різних етапах програми, для яких атрибути не грають великого значення, а послідовність та ідентичність, які поширюється в житті усієї системи називаються сутностями [18]. Класи-сутності називаються на її честь: Account, Note.

Фабрики відповідають за створення об'єктів предметної області. Делегування процесу створення об’єктів іншому класу дає можливість підміняти реалізацію в майбутньому [18]. Класи-фабрики називаються як

сутності, що вони продукують, але з припискою «Factory»: AccountFactory, NoteFactory.

Сукупності представляють колекції об'єктів, які пов'язані між собою завдяки головній сутності (Root Entity), інакше відомій як «Aggregate Root». Коренева сутність колекції об'єктів гарантує узгодженість змін, що вносяться до сукупності, забороняючи зовнішнім об'єктам посилатися на членів колекції. Сукупності являють собою інтерфейси для реалізації сховищ [18]. Вони називаються як сутності предметної області, але в множині: Accounts, Notes.

Application-пакети слідують структурі, наведеній на рис. 3.5.

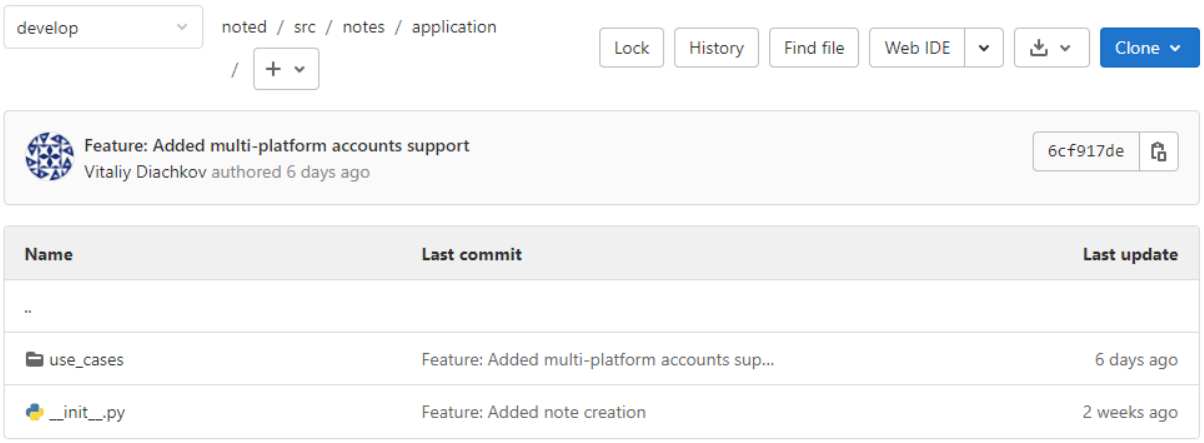


Рисунок 3.5 – Структура application-пакету

В цьому шарі визначаються так названі сценарії використання (англ. «use cases») або в термінології DDD сервіси – класи, що описують конкретні варіанти використання, є відображенням дій, що виконуються над сутностями предметної області. Їх типовим прикладом можуть бути CRUD (англ. «Created, Read, Update, Delete») операції. Сценарії використання описуються у якості інтерфейсів, що названі на честь операції, що вони виконують: CreateAccount, UpdateNote тощо. Виокремлення інтерфейсу сценаріїв використання дозволяє у майбутньому підміняти їх реалізацію.

Кожен сценарій використання приймає у якості параметру запит на виконання дії. Класи-запити є представленням шаблону «Об’єкт значення» (англ. «Value Object») – об'єкт, який містить атрибути, але не має концептуальної ідентичності. Він повинен розглядатися як незмінний об'єкт [18]. Класи-запити закінчуються на «Request»: CreateAccountRequest, UpdateNoteRequest.

У результаті свого виконання сценарії використання повертають відповіді – ще одні приклади об’єктів значення. Вони названі на честь результату, що отримано після виконання дії: CreatedAccount, UpdatedNote.

Зразок описаних сценаріїв використання наведено на рис. 3.6. Структура пакету-сценарію використання наведено на рис. 3.7.

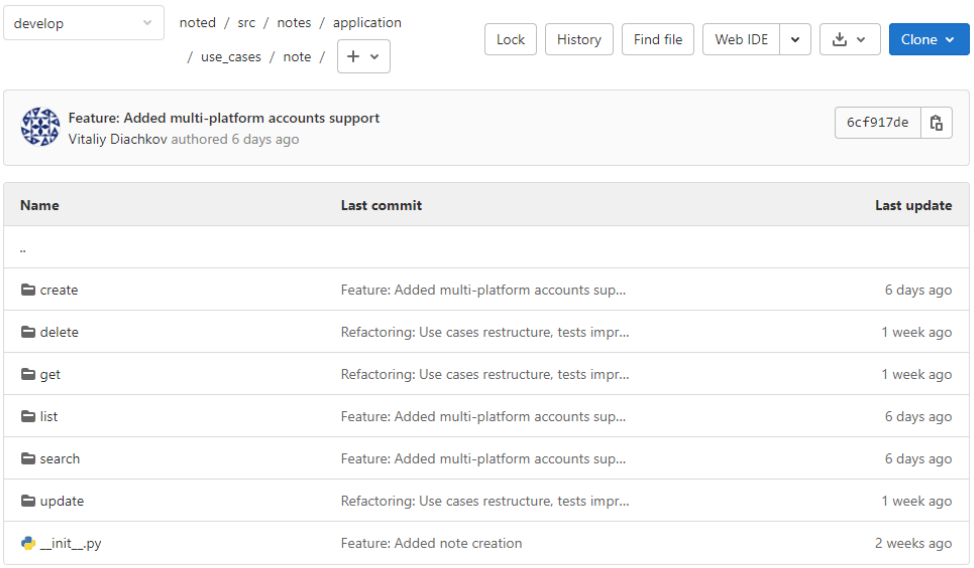


Рисунок 3.6 – Пакети-сценарії використання

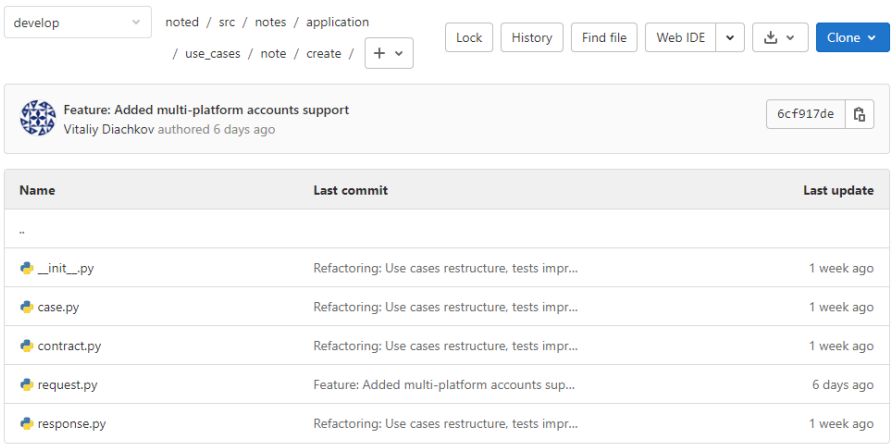


Рисунок 3.7 – Структура пакету-сценарію використання

У загальному випадку infrastructure-пакет виглядає так, як показано на рис. 3.8. Всередині такого пакету визначається вся взаємодія гексагону з зовнішнім світом, або у нашому випадку – з базою даних та месенджером.

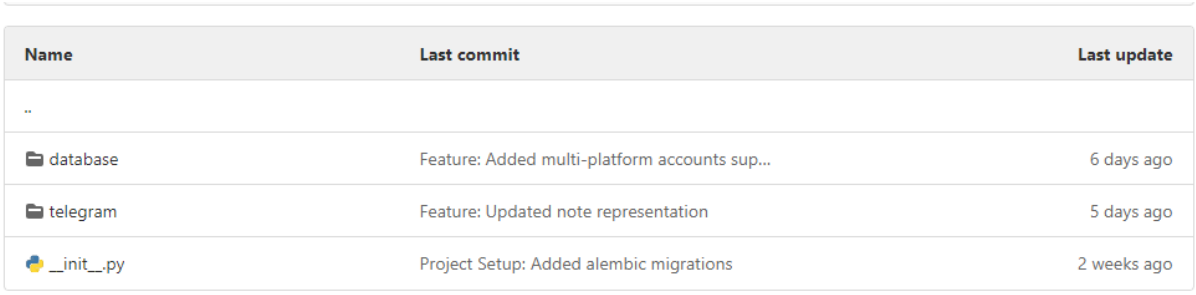


Рисунок 3.8 – Структура infrastructure-пакету

Всередині пакету «database» реалізуються інтерфейси сукупностей з domain-шару, реалізуються так звані сховища – класи, яким делегуються отримання та збереження об’єктів до зовнішнього сховища (бази даних) [18].

Нижче наведено приклад визначення абстрактного класу-сукупності з domain-шару:

```
class Notes(ABC):
    """Notes repository abstraction."""

    @staticmethod
    @abstractmethod
    async def search_for_author(
        author_id: int,
```

```
        query: str,
        from_: Optional[int] = None,
        to: Optional[int] = None
    ) -> list[Note]:
        """Search author notes with given query."""
```

Також приклад реалізації класу-сукупності у вигляді класу-сховища:

```
class NoteRepository(Notes):

    @classmethod
    async def search_for_author(
        cls,
        author_id: int,
        query: str,
        from_: Optional[int] = None,
        to: Optional[int] = None
    ) -> list[Note]:
        async with async_session() as session:
            db_notes = await session.run_sync(
                cls._search_notes,
                author_id=author_id,
                query=query,
                from_=from_,
                to=to
            )
            return [
                cls._convert_note_model_to_note(db_note)
                for db_note in db_notes
            ]

    @classmethod
    def _search_notes(
        cls,
        session: Session,
        author_id: int,
        query: str,
        from_: Optional[int] = None,
        to: Optional[int] = None,
    ) -> list[NoteModel]:
        sqlquery = session.query(NoteModel).options(
            joinedload(NoteModel.author)
        ).filter(
            NoteModel.author_id == author_id,
            NoteModel.text.contains(query)
        )
        return cls._paginate(sqlquery, from_, to)

    @classmethod
    def _convert_note_model_to_note(cls, model: NoteModel) -> Note:
        return Note(
            id=model.id,
            text=model.text,
            created=model.created,
            author=Account(
                id=model.author.id,
                account_id=model.author.account_id,
                application=model.author.application
            )
        )
```

NoteModel у прикладі вище – клас-репрезентація відношення з бази даних, нащадок BaseModel з SQLAlchemy. Як видно з наведеного раніше фрагменту коду, реалізація сховище приймає і повертає об’єкти типу Note – класу-сутності, визначеному у domain-шарі. Це зроблено для того, аби «нижчі» шари не знали нічого про деталі підключення до бази даних і взагалі існування якогось класу-представлення NoteModel.

У пакеті «telegram» усередині inftrastructure-пакету визначаються оброблювачі вхідних повідомлень, фільтри для цих оброблювачів, клавіатури для відповідей тощо.

Окрему увагу необхідно приділити дотриманню принципу інверсії залежностей (англ. «Dependency Inversion Principle», DIP) [19]. Розглянемо типову реалізацію сценарію використання у application-шарі:

```
from injector import inject

from src.notes.application.use_cases.note.create.contract import \
    CreateNote as CreateNoteContract
from src.notes.application.use_cases.note.create.request import (
    CreateNoteRequest,
)
from src.notes.application.use_cases.note.create.response import CreatedNote
from src.notes.domain.factories.note import NoteFactory
from src.notes.domain.repositories.note import Notes


class CreateNote(CreateNoteContract):
    @inject  # type: ignore
    def __init__(self, factory: NoteFactory, notes: Notes):
        self._factory = factory
        self._notes = notes

    async def execute(self, request: CreateNoteRequest) -> CreatedNote:
        note = self._factory.build(**request.to_dict())
        return CreatedNote(await self._notes.save(note))
```

Як видно з фрагменту коду вище, клас CreateNote залежить від класу-сукупності Notes з domain-шара. Клас Notes є абстрактним і не може бути використаним тут, але його реалізація знаходиться в шарі «inftrastructure» (клас NotesRepository, розглянутий раніше). В цьому фрагменті коду наш сценарій використання нічого не знає про шари, що знаходяться вище, а

оскільки ланцюг викликів йде від шару infrastructure до domain, то реалізація абстрактного класу Notes просто впроваджується за допомогою DI-інструменту «injector», фрагмент конфігурації якого наведена нижче:

```
from injector import Binder, Module

from src.notes.infrastructure.database.repositories.note import
NoteRepository


class NotesModule(Module):
    def configure(self, binder: Binder) -> None:
        binder.bind(Notes, to=NoteRepository)
```

Якби клас CreateNote напряму посилався на клас NotesRepository, то application-шару було б відомо про inftrastructure-шар, що порушувало б DIP.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПРЕМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Автоматичне тестування

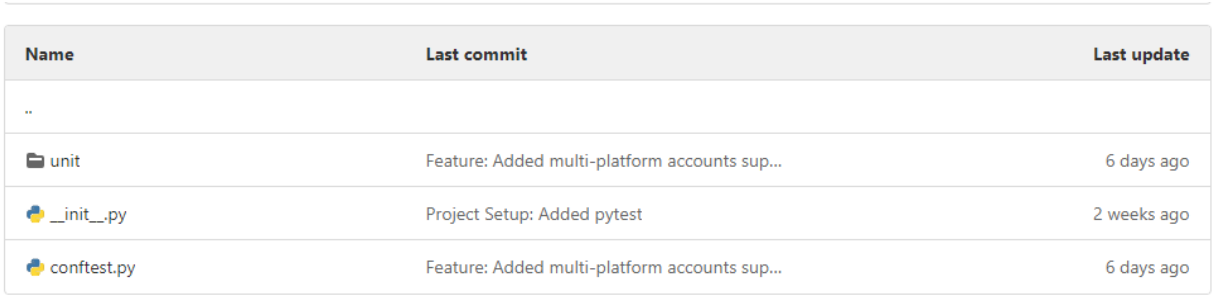
Для автоматичного тестування кодова база була покрита модульними тестами. Модульне тестування (англ. «Unit testing») — це метод тестування програмного забезпечення, який полягає в окремому тестуванні кожного модуля коду програми. Модулем називають найменшу частину програми, яка може бути протестованою. Зазвичай unit-тести застосовують для того, щоб упевнитися, що код відповідає вимогам архітектури та має очікувану поведінку [20].

Під час розробки даного програмного забезпечення використовувався підхід розробки через тестування (англ. «Test-Driven Development», TDD) [21], що характеризується короткими ітераціями розробки, що починаються з попереднього написання тестів, які визначають необхідні покращення або нові функції. Кожна ітерація має на меті розробити код, який пройде ці тести. Нарешті, програміст або група вдосконалюють код для погодження змін. Один із ключових моментів TDD полягає у тому, що підготовка тестів перед написанням самого коду пришвидшує процес внесення змін.

Цикл розробки з використанням підходу TDD:

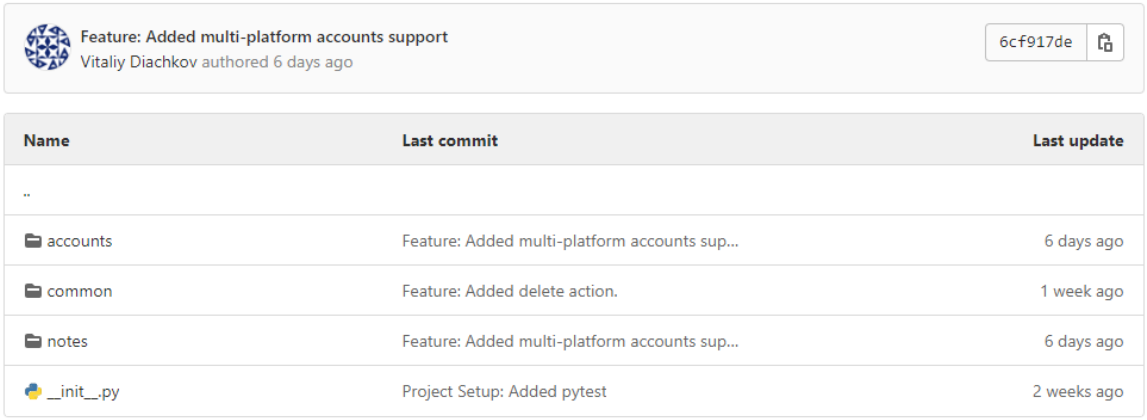
- додати тест;
- запустити усі тести, і подивитись, чи старі пройшли, а недавно додані провалились;
- написати код;
- запустити усі тести і подивитись, чи вони пройшли;
- удосконалити код;
- запустити усі тести і подивитись, чи вони пройшли;
- додати зміни до спільної кодової бази.

Використовуючи вищезазначений порядок дій була оформлена основна кодова база автоматичних тестів для розробленого застосунку. Структура написаних тестів зображена на рис. 4.1 – 4.3.



Name	Last commit	Last update
..		
unit	Feature: Added multi-platform accounts sup...	6 days ago
__init__.py	Project Setup: Added pytest	2 weeks ago
conftest.py	Feature: Added multi-platform accounts sup...	6 days ago

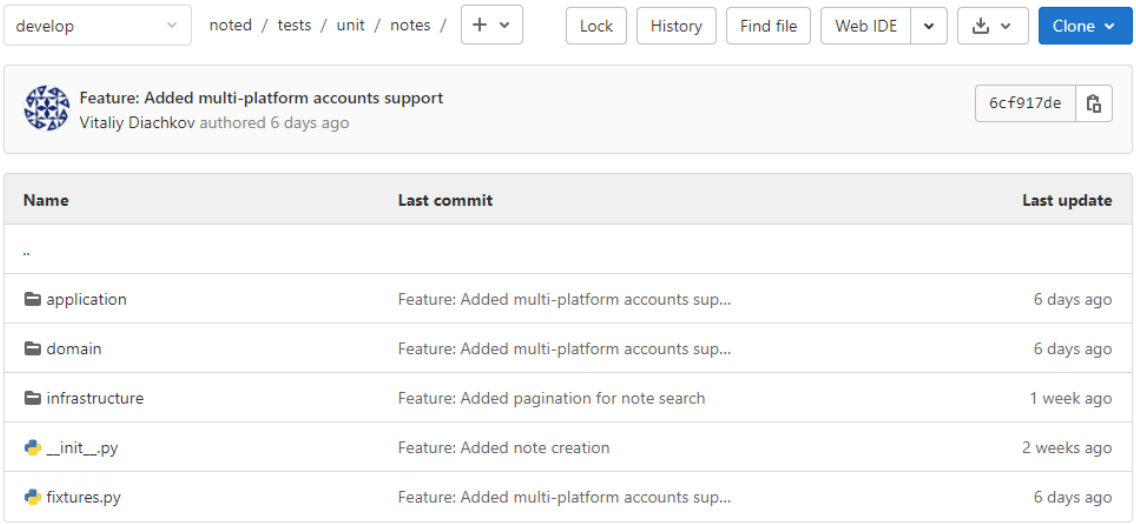
Рисунок 4.1 – Структура пакету «tests»



Feature: Added multi-platform accounts support
Vitaliy Diachkov authored 6 days ago
6cf917de

Name	Last commit	Last update
..		
accounts	Feature: Added multi-platform accounts sup...	6 days ago
common	Feature: Added delete action.	1 week ago
notes	Feature: Added multi-platform accounts sup...	6 days ago
__init__.py	Project Setup: Added pytest	2 weeks ago

Рисунок 4.2 – Структура пакету «tests.unit»



develop | noted / tests / unit / notes / | Lock | History | Find file | Web IDE | Clone

Feature: Added multi-platform accounts support
Vitaliy Diachkov authored 6 days ago
6cf917de

Name	Last commit	Last update
..		
application	Feature: Added multi-platform accounts sup...	6 days ago
domain	Feature: Added multi-platform accounts sup...	6 days ago
infrastructure	Feature: Added pagination for note search	1 week ago
__init__.py	Feature: Added note creation	2 weeks ago
fixtures.py	Feature: Added multi-platform accounts sup...	6 days ago

Рисунок 4.3 – Структура пакету «tests.unit.notes»

Як видно з наведених рисунків, структура unit-тестів повністю повторює структуру пакетів в основному пакеті «src».

Для контролю коректності написання тестів та повноти покриття ними кодової бази було використано спеціалізований інструмент «coverage» для Python та плагін «pytest-cov», що показує результат покриття коду тестами у 100%. Звісно, повного покриття тестами у досягти не вдалося, а подібний результат показано тільки через те, що в тестових модулях не були імпортовані певні модулі первинного коду з infrastructure-шару, але «серце» програмного забезпечення – application та domain шари, були повністю покриті автоматичними тестами.

Консольний вивід запуску тестів наведено нижче:

```
(.venv) vitaliy@pc:~/coding/stud/diploma/noted:(develop)$ pytest --cov
==================== test session starts ====================
platform linux -- Python 3.9.4, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /home/vitaliy/coding/stud/diploma/noted, configfile: pyproject.toml
plugins: cov-2.11.1, asyncio-0.15.1, mock-3.6.0
collected 34 items

tests/unit/accounts/application/use_cases/account/create/test_case.py ..
tests/unit/common/entities/test_base_entity.py .
tests/unit/notes/application/use_cases/note/create/test_case.py .
tests/unit/notes/application/use_cases/note/delete/test_case.py ..
tests/unit/notes/application/use_cases/note/get/test_case.py ..
tests/unit/notes/application/use_cases/note/list/test_case.py .
tests/unit/notes/application/use_cases/note/search/test_case.py ..
tests/unit/notes/application/use_cases/note/update/test_case.py ..
tests/unit/notes/domain/factories/test_note.py .
tests/unit/notes/infrastructure/telegram/test_filters.py ..................

----------- coverage: platform linux, python 3.9.4-final-0 -----------
Name                                                                    Stmts   Miss
Cover
-------------------------------------------------------------------------------
-------
src/__init__.py                                                             0      0
100%
src/accounts/__init__.py                                                    0      0
100%
src/accounts/application/__init__.py                                        0      0
100%
src/accounts/application/use_cases/__init__.py                              0      0
100%
src/accounts/application/use_cases/accounts/__init__.py                     0      0
100%
src/accounts/application/use_cases/accounts/create/__init__.py              0      0
100%
src/accounts/application/use_cases/accounts/create/case.py                 17      0
100%
```

```
src/accounts/application/use_cases/accounts/create/contract.py        7       0
100%
src/accounts/application/use_cases/accounts/create/request.py         8       0
100%
src/accounts/application/use_cases/accounts/create/response.py        6       0
100%
src/accounts/domain/__init__.py                                       0       0
100%
src/accounts/domain/account.py                                        9       0
100%
src/accounts/domain/application.py                                    3       0
100%
src/accounts/domain/repositories/__init__.py                          0       0
100%
src/accounts/domain/repositories/accounts.py                         12       0
100%
src/common/__init__.py                                                0       0
100%
src/common/entities/__init__.py                                       0       0
100%
src/common/entities/base.py                                           6       0
100%
src/notes/__init__.py                                                 0       0
100%
src/notes/application/__init__.py                                     0       0
100%
src/notes/application/use_cases/__init__.py                           0       0
100%
src/notes/application/use_cases/note/__init__.py                      0       0
100%
src/notes/application/use_cases/note/create/__init__.py               0       0
100%
src/notes/application/use_cases/note/create/case.py                  14       0
100%
src/notes/application/use_cases/note/create/contract.py               6       0
100%
src/notes/application/use_cases/note/create/request.py                8       0
100%
src/notes/application/use_cases/note/create/response.py               6       0
100%
src/notes/application/use_cases/note/delete/__init__.py               0       0
100%
src/notes/application/use_cases/note/delete/case.py                  16       0
100%
src/notes/application/use_cases/note/delete/contract.py               7       0
100%
src/notes/application/use_cases/note/delete/request.py                5       0
100%
src/notes/application/use_cases/note/delete/response.py               5       0
100%
src/notes/application/use_cases/note/get/__init__.py                  0       0
100%
src/notes/application/use_cases/note/get/case.py                     14       0
100%
src/notes/application/use_cases/note/get/contract.py                  7       0
100%
src/notes/application/use_cases/note/get/request.py                   5       0
100%
src/notes/application/use_cases/note/get/response.py                  6       0
100%
src/notes/application/use_cases/note/list/__init__.py                 0       0
100%
src/notes/application/use_cases/note/list/case.py                    12       0
100%
```

```
src/notes/application/use_cases/note/list/contract.py                6      0
100%
src/notes/application/use_cases/note/list/request.py                 8      0
100%
src/notes/application/use_cases/note/list/response.py                6      0
100%
src/notes/application/use_cases/note/search/__init__.py              0      0
100%
src/notes/application/use_cases/note/search/case.py                 14      0
100%
src/notes/application/use_cases/note/search/contract.py              7      0
100%
src/notes/application/use_cases/note/search/request.py               9      0
100%
src/notes/application/use_cases/note/search/response.py              6      0
100%
src/notes/application/use_cases/note/update/__init__.py              0      0
100%
src/notes/application/use_cases/note/update/case.py                 15      0
100%
src/notes/application/use_cases/note/update/contract.py              7      0
100%
src/notes/application/use_cases/note/update/request.py               8      0
100%
src/notes/application/use_cases/note/update/response.py              6      0
100%
src/notes/domain/__init__.py                                         0      0
100%
src/notes/domain/factories/__init__.py                               0      0
100%
src/notes/domain/factories/note.py                                   7      0
100%
src/notes/domain/note.py                                            11      0
100%
src/notes/domain/repositories/__init__.py                            0      0
100%
src/notes/domain/repositories/note.py                               23      0
100%
src/notes/infrastructure/__init__.py                                 0      0
100%
src/notes/infrastructure/telegram/__init__.py                        0      0
100%
src/notes/infrastructure/telegram/filters.py                        16      0
100%
-------------------------------------------------------------------------------
-------
TOTAL                                                              328      0
100%


==================== 34 passed in 0.76s ====================
```

Як видно з результатів запуску автоматичних тестів, час їх виконання складає менше однієї секунди. Цього вдалося досягти за рахунок вибору правильної архітектури, впровадження шаблону Dependency Injection, виділення усіх звернень до зовнішніх сховищ (баз даних) через окремий

інтерфейс, а отже провести повне тестування системи можливе без підняття бази даних та запуску самого боту.

4.2 Мануальне тестування

Ще на процесі розробки було визначено основні сценарії використання користувачем системи (які знаходять своє представлення в application-шарі гексагонів).

Для кожного сценарію використання було написано принаймні один позитивний тест-кейс – артефакт, що описує сукупність кроків, конкретних умов та параметрів, необхідних для перевірки реалізації функції, що тестується чи її частини [22].

Усі сформовані тест-кейси описані в табл. 4.1 – 4.8.

Таблиця 4.1 – Тест-кейс «TestCreateAccount»

Назва тест-кейсу	Перевірка початку роботи з ботом
Передумови виконання	
Крок	**Очікуваний результат**
Виконати команду «/start»	Бот відповідає повідомленням з описом своїх можливостей і доступних команд. У базі даних з’являється новий допис с обліковим записом користувача

Таблиця 4.2 – Тест-кейс «TestCreateNote»

Назва тест-кейсу	Перевірка створення нотаток
Передумови виконання	Початий процес взаємодії з ботом командою «/start»

Продовження таблиці 4.2

Крок	**Очікуваний результат**
Відправити повідомлення боту з довільним текстом	Бот відповідає повідомленням «Note successfully created!»
Виконати команду «/notes»	У списку нотаток з’явилася нова з повідомленням яке було відправлено боту на попередньому кроці (із збереженням розмітки тексту)

Таблиця 4.3 – Тест-кейс «TestDeleteNote»

Назва тест-кейсу	Перевірка видалення нотаток
Передумови виконання	Початий процес взаємодії з ботом командою «/start», принаймні одна створена нотатка
Крок	**Очікуваний результат**
Виконати команду «/notes»	Отримано список створених нотаток
Натиснути кнопку «Delete» навпроти однієї з нотаток	Отримано спливаюче повідомлення «Note was successfully deleted!», видалена нотатка зникла зі списку всіх нотаток

Таблиця 4.4 – Тест-кейс «TestGetNote»

Назва тест-кейсу	Перевірка отримання тексту нотатки
Передумови виконання	Початий процес взаємодії з ботом командою «/start», принаймні одна створена нотатка

Продовження таблиці 4.4

Крок	**Очікуваний результат**
Виконати команду «/notes»	Отримано список створених нотаток
Натиснути на кнопку з текстом нотатки	Повідомлення зі списом нотаток змінює свій текст на текст нотатки (із збереженням розмітки)

Таблиця 4.5 – Тест-кейс «TestListNotes»

Назва тест-кейсу	Перевірка отримання списку нотаток
Передумови виконання	Початий процес взаємодії з ботом командою «/start», принаймні одна створена нотатка
Крок	**Очікуваний результат**
Виконати команду «/notes»	Отримано список створених нотаток у вигляді клавіатури, кожен новий рядок якої складається з: кнопки з текстом нотатки, кнопки «Edit», кнопки «Delete»

Таблиця 4.6 – Тест-кейс «TestListNotesPagination»

Назва тест-кейсу	Перевірка отримання списку нотаток
Передумови виконання	Початий процес взаємодії з ботом командою «/start», 15 створених нотаток
Крок	**Очікуваний результат**
Виконати команду «/notes»	Отримано список створених нотаток (відсортованих за датою створення, спочатку найновіші) у вигляді клавіатури, кожен новий рядок якої складається з: кнопки з текстом нотатки, кнопки «Edit», кнопки «Delete». З самого низу клавіатури є кнопка «Next»

Продовження таблиці 4.6

Натиснути кнопку «Next»	Отримано наступну сторінку нотаток. Знизу є кнопки «Previous» та «Next»
Натиснути кнопку «Next» ще раз	Отримано останню сторінку нотаток. Знизу присутня лише кнопка «Previous»
Натиснути кнопку «Previous»	Отримано попередню сторінку нотаток

Таблиця 4.7 – Тест-кейс «TestSearchNotes»

Назва тест-кейсу	Перевірка отримання списку нотаток
Передумови виконання	Початий процес взаємодії з ботом командою «/start», створена нотатка з текстом, що містить «#tag»
Крок	**Очікуваний результат**
Відправити боту повідомлення «#tag»	Отримано список нотаток (подібний тому, що отримується після виконання команди «/notes»), але у результатах присутня лише нотатки, що мають #tag в своєму тексті

Таблиця 4.8 – Тест-кейс «TestUpdateNote»

Назва тест-кейсу	Перевірка створення нотаток
Передумови виконання	Початий процес взаємодії з ботом командою «/start», принаймні одна створена нотатка
Крок	**Очікуваний результат**
Виконати команду «/notes»	Отримано список нотаток
Натиснути кнопку «Edit» навпроти однієї з нотаток	Повідомлення зі списком нотаток змінює свій текст на «Please, enter a new note text»

Продовження таблиці 4.9

Відправити новий текст нотатки боту	Бот відповідає повідомленням «Note was successfully updated!»
Виконати команду «/notes»	Нотатка в отриманому списку має оновлений текст

4.3 Експлуатація

Виконання програми відбувається за рахунок запуску головного скрипта «main.py», що знаходиться у кореневій директорії проекту. В ньому ініціалізуються усі файли конфігурацій, реєструються оброблювачі повідомлень Telegram-бота та запускається процес відстеження подій чат-бота.

Повний процес розгортання проекту, його налаштування і підготовки до виконання наведено у файлах «docs/local_deployment.md» та «docs/server_deployment.md».

Скриншоти роботи програми зображено на рис. 4.4 – 4.6.

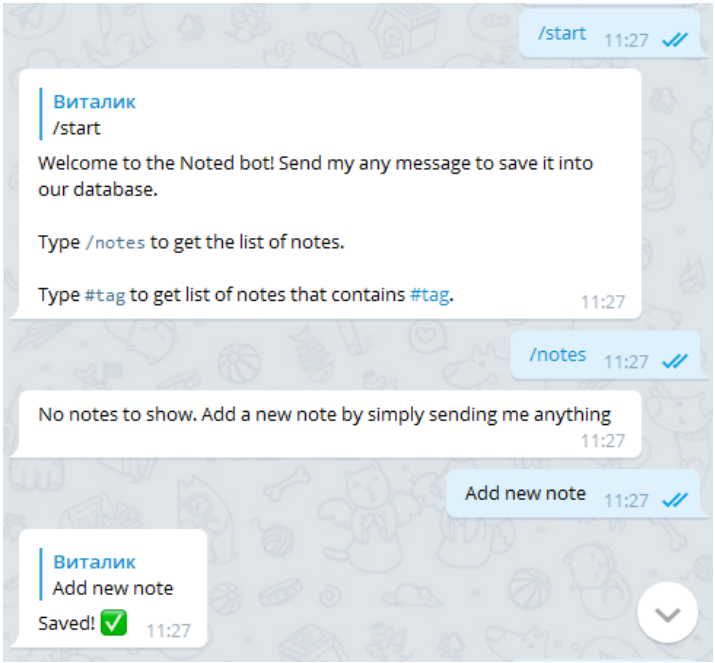


Рисунок 4.4 – Початок роботи з ботом і створення першої нотатки

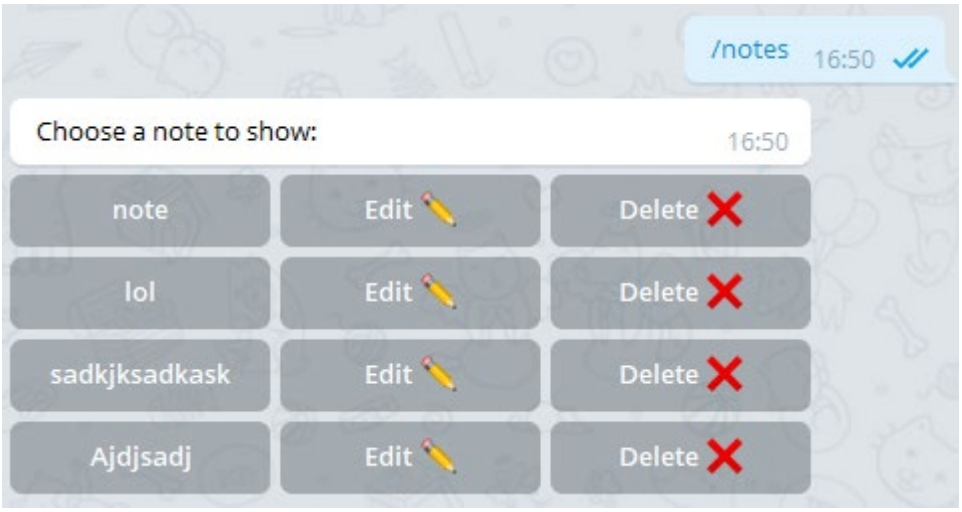


Рисунок 4.5 – Перегляд списку нотаток

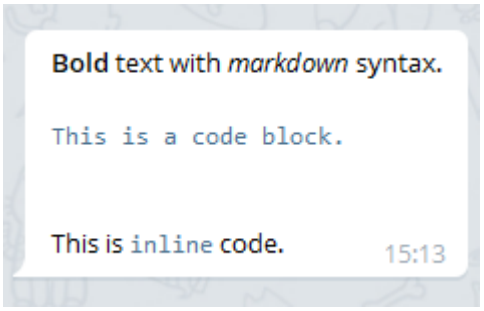


Рисунок 4.6 – Перегляд повного тексту нотатки (із збереженням розмітки)

ВИСНОВКИ

У результаті виконання дипломної роботи було отримано готового до роботи Telegram-бота для управління короткими текстовими нотатками: їх створення, редагування, видалення та перегляд з можливістю пошуку за тегами.

Першим кроком до реалізації даного програмного забезпечення став аналіз предметної області. На цьому етапу було вкотре підтверджено актуальність розробки чат-бота для створення нотаток, обрано Telegram як найзручніший месенджер для досягнення поставленої мети, проведено огляд та порівняльний аналіз наявних аналогів, під час якого були виявлені проблеми, що мають готові рішення. В результаті проходження цього етапу розробки було створено технічне завдання до проекту.

Потім, згідно з поставленими завданнями було проведено етап проектування системи, на якому було обрано основні інструменти реалізації (мову програмування Python, бібліотеку aiogram, СКБД PostgreSQL і Redis та інші), а також високорівневу структуру проекту з дотриманням принципів DDD та гексагональної архітектури.

Третім етапом стала безпосередня реалізація проекту: імплементація визначеної архітектури з заздалегідь обраними інструментами. На цьому етапі були прийняті ще декілька важливих рішень щодо архітектури пакетів нижчих рівнів, що досягло не порушити початкової структури проекту, а також легко перейти до тестування програми.

Заключними етапами стали тестування і введення проекту до експлуатації. Завдяки використанню TDD підходу вдалося легко досягти майже повного покриття кодової бази модульними тестами, а виділення сценаріїв використання на процесі розробки полегшили процес створення та виконання тест-кейсів. Написана в процесі розробки документація щодо виконання програми, а також використання найновіших інструментів автоматизації процесу розгортання (таких як засіб контейнеризації Docker,

інструмент для управління залежностями проекту Poetry та інші), дозволили легко ввести готовий програмний продукт в експлуатацію.

Реалізоване програмне забезпечення відповідає всім вимогам, поставленим в технічному завданні до проекту. Отримана програма є стійкою до змін і передбачає можливості свого подальшого розширення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Другов рассказал, получему Telegram стал самым популярным приложением в мире в январе [Электронный ресурс]. – Режим доступа: https://habr.com/ru/news/t/541512/

2. Most popular messenger apps [Electronic resource]. – Access mode: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/

3. Telegram Bot API [Electronic resource]. – Access mode: https://core.telegram.org/bots/api

4. Telegram: Contact @exnobot [Electronic resource]. – Access mode: https://t.me/exnobot

5. Telegram: Contact @killthemall_bot [Electronic resource]. – Access mode: https://t.me/killthemall_bot

6. PEP 249 -- Python Database API Specification v2.0 [Electronic resource]. – Access mode: https://www.python.org/dev/peps/pep-0249/

7. eternnoir/pyTelegramBotAPI: Python telegram bot api. [Electronic resource]. – Access mode: https://github.com/eternnoir/pyTelegramBotAPI

8. pyrogram/pyrogram: Telegram MTProto API Client Library and Framework in Pure Python for Users and Bots [Electronic resource]. – Access mode: https://github.com/pyrogram/pyrogram

9. aiogram/aiogram: Is a pretty simple and fully asynchronous framework for Telegram Bot API written in Python 3.7 with asyncio and aiohttp. [Electronic resource]. – Access mode: https://github.com/aiogram/aiogram

10. unittest – Unit testing framework [Electronic resource]. – Access mode: https://docs.python.org/3/library/unittest.html

11. pytest: helps you write better programs [Electronic resource]. – Access mode: https://docs.pytest.org/en/6.2.x/

12. PEP 8 -- Style Guide for Python Code [Electronic resource]. – Acess mode: https://www.python.org/dev/peps/pep-0008/

13. pre-commit [Electronic resource]. – Access mode: https://pre-commit.com/

14. Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software [Text] / E. Evans. – Boston : Addison-Wesley Professional, 2003 - 560 p.

15. Гексагональная архитектура [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/267125/

16. Dependency Injection [Electronic resource]. – Acess mode: http://tutorials.jenkov.com/dependency-injection/index.html

17. Впровадження залежностей [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/%D0%92%D0%BF%D1%80%D0%BE%D0%B2%D0%B0%D0%B4%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F_%D0%B7%D0%B0%D0%BB%D0%B5%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B5%D0%B9

18. Предметно-орієнтоване проектування [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D0%BC%D0%B5%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D1%96%D1%94%D0%BD%D1%82%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F

19. SOLID Design Principles Explained: Dependency Inversion Principle with Code Examples [Electronic resource]. – Acess mode: https://stackify.com/dependency-inversion-principle/

20. Модульне тестування [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F

21. Beck, K. Test Driven Development: By Example [Text] / K. Beck – Boston : Addison-Wesley Professional, 2002 – 240 p.

22. Тестовий випадок [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%BE%D0%B2%D0%B8%D0%B9_%D0%B2%D0%B8%D0%BF%D0%B0%D0%B4%D0%BE%D0%BA

Додаток А

Технічне завдання

А.1 Підстави для розробки

Підставою для розробки програмного продукту стало завдання до бакалаврської дипломної роботи, затверджене наказом закладу вищої навчальної освіти від 30 березня 2021 року №103.

А.2 Призначення розробки

Розроблюване програмне забезпечення призначено для автоматизації процесу управління короткими текстовими нотатками у месенджері Telegram.

А.3 Вимоги до програми

А.3.1 Функціональні вимоги

Розроблюване програмне забезпечення має надавати користувачу наступні можливості:

- створення текстових нотаток у Telegram (зі збереженнями розмітки, тобто використання жирного шрифту, курсиву, підкреслення, блоків з наперед визначеною розміткою (у форматі markdown);
- перегляду попередньо створених нотаток;
- пошуку за нотатками;
- редагування попередньо створених нотаток;
- видалення попередньо створених нотаток.

А.3.2 Нефункціональні вимоги

Розроблюване програмне забезпечення повинно відповідати наступним нефункціональним вимогам:

- захищати нотатки користувачів від несанкціонованого доступу (лише користувач, що створив нотатки, має мати доступ до нього);
- мати можливість оброблення багатьох запитів одночасно;
- відповідати користувачеві на його команди менш ніж за 1 секунду;
- повинно бути сумісним і без помилок виконуватись на сімействі операційних систем (ОС) Linux (зокрема, заснованих на дистрибутиві Debian або Arch Linux);
- бути пристосованим до масштабування, перенесення на інші платформи.

А.4 Стадії та етапи розробки

Під час розробки даного програмного забезпечення необхідне дотримання наступних етапів:

– аналіз предметної області, огляд готових рішень проблеми;

– проектування системи, вибір інструментів реалізації, визначення модулів майбутнього програмного продукту;

– розробка програмного продукту повинна проходити в декілька етапів. на кожному з котрих повинна отримуватись нова ітерація програмного продукту. По черзі реалізуючи кожну з функціональних та нефункціональних вимог, розробник повинен проводити контроль якості розроблюваного продукту, доповнюючи базу автоматичних модульних тестів та проводячи мануальне тестування системи;

– після завершення етапу розробки повинен бути проведений етап регресійного тестування. Усі дефекти, що будуть знайдені під час нього повинні бути усунуті на цьому етапі;

– здача проекту в експлуатацію, конфігурація та розгортання його в робочому оточенні.

А.5 Порядок контролю та приймання

Розроблений програмний продукт повинен відповідати усім вимогам, поставленим до нього у пункті А.2 (що має бути перевірено на стадії завершального тестування програми).

Порядок приймання роботи відповідає порядку приймання бакалаврських дипломних робіт.

Додаток Б

Текст програми

Б.1 Код файлу «main.py»

```
from aiogram import Dispatcher
from aiogram.utils import executor
from injector import Injector

from src.accounts.bindings import AccountsModule
from src.accounts.infrastructure.telegram.routing import (
    AccountsHandlerRegistrator,
)
from src.common.telegram.dispatcher import dispatcher
from src.notes.bindings import NotesModule
from src.notes.infrastructure.telegram.routing import NotesHandlerRegistrator


async def startup(dis: Dispatcher):
    injector = Injector([
        NotesModule(),
        AccountsModule(),
    ])
    registrators = [
        NotesHandlerRegistrator(injector=injector),
        AccountsHandlerRegistrator(injector=injector)
    ]

    for registrator in registrators:
        registrator.register(dis)


async def shutdown(dis: Dispatcher):
    await dis.storage.close()
    await dis.storage.wait_closed()


def main():
    executor.start_polling(
        dispatcher=dispatcher,
        on_startup=startup,
        on_shutdown=shutdown
    )


if __name__ == '__main__':
    main()
```

Б.2 Код файлу «core/config/__init__.py»

```
import os  # noqa

from dotenv import load_dotenv


load_dotenv()


# Load specific configurations
from .database import *  # noqa  # type: ignore
from .telegram import *  # noqa  # type: ignore
```

Б.3 Код файлу «core/config/database.py»

```
import os


# Database configuration
DB_NAME = os.getenv('DB_NAME')
DB_USER = os.getenv('DB_USER')
DB_PASS = os.getenv('DB_PASS')
DB_HOST = os.getenv('DB_HOST')
DB_PORT = os.getenv('DB_PORT', '5432')

DB_URL = (
    f'postgresql+asyncpg://{DB_USER}:{DB_PASS}@{DB_HOST}:{DB_PORT}/{DB_NAME}'
)
```

Б.4 Код файлу «core/config/telegram.py»

```
import os


TELEGRAM_BOT_TOKEN = os.getenv('TELEGRAM_BOT_TOKEN')
```

Б.5 Код файлу «src/common/entities/base.py»

```
from dataclasses import asdict, dataclass
from typing import Any


@dataclass
class Entity:
    def to_dict(self) -> dict[str, Any]:
        return asdict(self)
```

Б.6 Код файлу «src/common/database/base.py»

```
from sqlalchemy import Column, Integer
from sqlalchemy.ext.declarative import as_declarative

from .engine import engine


@as_declarative(bind=engine)
class BaseModel:
    id = Column(Integer, primary_key=True)  # noqa: A003
```

Б.7 Код файлу «src/common/database/engine.py»

```
from core.config import DB_URL  # type: ignore
from sqlalchemy.ext.asyncio import create_async_engine


engine = create_async_engine(url=DB_URL)
```

Б.8 Код файлу «src/common/database/session.py»

```
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import sessionmaker

from .engine import engine


async_session = sessionmaker(
    bind=engine,
    expire_on_commit=False,
    class_=AsyncSession
)
```

Б.9 Код файлу «src/common/telegram/bot.py»

```
from aiogram import Bot
from core.config import TELEGRAM_BOT_TOKEN  # type: ignore


bot = Bot(token=TELEGRAM_BOT_TOKEN)
```

Б.10 Код файлу «src/common/telegram/dispatcher.py»

```
from aiogram.contrib.fsm_storage.memory import MemoryStorage
from aiogram.dispatcher import Dispatcher

from .bot import bot


dispatcher = Dispatcher(bot=bot, storage=MemoryStorage())
```

Б.11 Код файлу «src/common/telegram/base/registrator.py»

```
from abc import ABC, abstractmethod

from aiogram import Dispatcher


class HandlerRegistrator(ABC):
    @abstractmethod
    def register(self, dispatcher: Dispatcher) -> None:
        """Register message/error/callback handlers."""
```

Б.12 Код файлу «src/common/telegram/base/handlers/callback_query.py»

```
from abc import ABC, abstractmethod

from aiogram.dispatcher import FSMContext
from aiogram.types import CallbackQuery


class CallbackQueryHandler(ABC):
    @abstractmethod
    async def __call__(
        self,
        callback_query: CallbackQuery,
        state: FSMContext,
    ) -> None:
        """Handle callback query."""
```

Б.13 Код файлу «src/common/telegram/base/handelrs/message.py»

```
from abc import ABC, abstractmethod

from aiogram.dispatcher import FSMContext
from aiogram.types import Message


class MessageHandler(ABC):
    @abstractmethod
    async def __call__(self, message: Message, state: FSMContext):
        """Handle incoming message."""
```

Б.14 Код файлу «src/notes/domain/note.py»

```
from dataclasses import dataclass
from datetime import datetime
from typing import Optional

from src.accounts.domain.account import Account
from src.common.entities.base import Entity


@dataclass
class Note(Entity):
    """
    Note entity representation.
    """

    author: Account
    text: str
    created: datetime
    id: Optional[int] = None  # noqa: A003
```

Б.15 Код файлу «src/notes/domain/factories/note.py»

```
from datetime import datetime

from src.accounts.domain.account import Account
from src.notes.domain.note import Note


class NoteFactory:
    @staticmethod
    def build(author: Account, text: str) -> Note:
        return Note(
            author=author,
            text=text,
            created=datetime.now()
        )
```

Б.16 Код файлу «src/notes/domain/repositories/note.py»

```
from abc import ABC, abstractmethod
from typing import Any, Optional

from src.notes.domain.note import Note


class Notes(ABC):
    """Notes repository abstraction."""

    @staticmethod
    @abstractmethod
    async def search_for_author(
        author_id: int,
        query: str,
        from_: Optional[int] = None,
        to: Optional[int] = None
    ) -> list[Note]:
        """Search author notes with given query."""

    @staticmethod
    @abstractmethod
    async def save(note: Note) -> Note:
        """Save note to a database."""

    @staticmethod
    @abstractmethod
    async def get_by_id(note_id: int) -> Optional[Note]:
        """Return note by its ID if one found else None."""

    @staticmethod
    @abstractmethod
    async def update_by_id(note_id: int, **kwargs: Any) -> None:
        """Update note by id."""

    @staticmethod
    @abstractmethod
    async def delete_by_id(note_id: int) -> None:
        """Delete note by id."""
```

```
    @staticmethod
    async def get_notes_count(author_id: int) -> int:
        """Get notes count for specific author."""

    @staticmethod
    async def get_notes(
        author_id: int,
        from_: Optional[int] = None,
        to: Optional[int] = None
    ) -> list[Note]:
        """Get all notes for specific author."""
```

Б.17 Код файлу «src/notes/application/use_cases/note/create/request.py»

```
from typing import Any

from src.accounts.domain.account import Account


class CreateNoteRequest:
    def __init__(
        self,
        *,
        author: Account,
        text: str
    ):
        self._author = author
        self._text = text

    def to_dict(self) -> dict[str, Any]:
        return {
            'author': self._author,
            'text': self._text
        }
```

Б.18 Код файлу «src/notes/application/use_cases/note/create/response.py»

```
from src.notes.domain.note import Note


class CreatedNote:
    def __init__(self, note: Note):
        self._note = note

    def note(self) -> Note:
        return self._note
```

Б.19 Код файлу «src/notes/application/use_cases/note/create/contract.py»

```
from abc import ABC, abstractmethod

from src.notes.application.use_cases.note.create.request import (
    CreateNoteRequest,
)
from src.notes.application.use_cases.note.create.response import CreatedNote


class CreateNote(ABC):
    """Create note action."""

    @abstractmethod
    async def execute(self, request: CreateNoteRequest) -> CreatedNote:
        """Perform note creation."""
```

Б.20 Код файлу «src/notes/application/use_cases/note/create/case.py»

```
from injector import inject

from src.notes.application.use_cases.note.create.contract import \
    CreateNote as CreateNoteContract
from src.notes.application.use_cases.note.create.request import (
    CreateNoteRequest,
)
from src.notes.application.use_cases.note.create.response import CreatedNote
from src.notes.domain.factories.note import NoteFactory
from src.notes.domain.repositories.note import Notes


class CreateNote(CreateNoteContract):
    @inject  # type: ignore
    def __init__(self, factory: NoteFactory, notes: Notes):
        self._factory = factory
        self._notes = notes

    async def execute(self, request: CreateNoteRequest) -> CreatedNote:
        note = self._factory.build(**request.to_dict())
        return CreatedNote(await self._notes.save(note))
```

Б.21 Код файлу «src/notes/infrastructure/database/models/note.py»

```
from sqlalchemy import Column, DateTime, ForeignKey, Integer, Text
from sqlalchemy.orm import relationship

from src.accounts.infrastructure.database.models.account import Account
from src.common.database.base import BaseModel


class Note(BaseModel):
    __tablename__ = 'notes'
```

```
    author_id = Column(Integer, ForeignKey('accounts.id'), nullable=False)
    author = relationship(Account)
    text = Column(Text, nullable=False)
    created = Column(DateTime, nullable=False)
```

Б. 22 Код файлу «src/notes/infrastructure/database/repositories/note.py»

```
from typing import Any, Optional

from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import Query, Session, joinedload
from sqlalchemy.sql.expression import delete, desc, select, update

from src.accounts.domain.account import Account
from src.common.database.session import async_session
from src.notes.domain.note import Note
from src.notes.domain.repositories.note import Notes
from src.notes.infrastructure.database.models.note import Note as NoteModel


class NoteRepository(Notes):

    @classmethod
    async def search_for_author(
        cls,
        author_id: int,
        query: str,
        from_: Optional[int] = None,
        to: Optional[int] = None
    ) -> list[Note]:
        async with async_session() as session:
            db_notes = await session.run_sync(
                cls._search_notes,
                author_id=author_id,
                query=query,
                from_=from_,
                to=to
            )
            return [
                cls._convert_note_model_to_note(db_note)
                for db_note in db_notes
            ]

    @classmethod
    async def get_notes_count(cls, author_id: int) -> int:
        async with async_session() as session:
            return await session.run_sync(
                cls._get_notes_count,
                author_id=author_id
            )

    @classmethod
    async def get_notes(
        cls,
        author_id: int,
        from_: Optional[int] = None,
        to: Optional[int] = None
    ) -> list[Note]:
```

```
        async with async_session() as session:
            notes = await session.run_sync(
                cls._get_notes,
                author_id=author_id,
                from_=from_,
                to=to
            )
            return [cls._convert_note_model_to_note(nm) for nm in notes]

    @classmethod
    async def update_by_id(cls, note_id: int, **kwargs: Any) -> None:
        async with async_session() as session:
            await cls._update(session, note_id, **kwargs)

    @classmethod
    async def save(cls, note: Note) -> Note:
        async with async_session() as session:
            db_note = await cls._insert(
                session=session,
                note=cls._convert_note_to_note_model(note)
            )
            note.id = db_note.id
            return note

    @classmethod
    async def get_by_id(cls, note_id: int) -> Optional[Note]:
        async with async_session() as session:
            db_note = await cls._get_by_id(session, note_id)

            if db_note is not None:
                return cls._convert_note_model_to_note(db_note)

    @classmethod
    async def delete_by_id(cls, note_id: int) -> None:
        async with async_session() as session:
            await cls._delete_by_id(session, note_id)

    @classmethod
    async def _delete_by_id(cls, session: AsyncSession, note_id: int) ->
None:
        statement = delete(NoteModel).where(NoteModel.id == note_id)
        await session.execute(statement)
        await session.commit()

    @classmethod
    async def _insert(
        cls,
        session: AsyncSession,
        note: NoteModel
    ) -> NoteModel:
        session.add(note)
        await session.commit()
        return note

    @classmethod
    async def _update(
        cls,
        session: AsyncSession,
        note_id: int,
        **kwargs: Any
    ) -> None:
        statement = update(NoteModel).where(
            NoteModel.id == note_id
        ).values(**kwargs)
```

```
        await session.execute(statement)
        await session.commit()

    @classmethod
    async def _get_by_id(
        cls,
        session: AsyncSession,
        note_id: int
    ) -> NoteModel:
        statement = select(NoteModel).options(
            joinedload(NoteModel.author)
        ).where(NoteModel.id == note_id)
        result = await session.execute(statement)
        db_note = result.scalars().first()
        return db_note

    @classmethod
    def _get_notes_count(cls, session: Session, author_id: int) -> int:
        return session.query(NoteModel).filter(
            NoteModel.author_id == author_id
        ).count()

    @classmethod
    def _get_notes(
        cls,
        session: Session,
        author_id: int,
        from_: Optional[int] = None,
        to: Optional[int] = None
    ) -> list[NoteModel]:
        query = session.query(NoteModel).options(
            joinedload(NoteModel.author)
        ).filter(
            NoteModel.author_id == author_id
        ).order_by(
            desc(NoteModel.created)
        )
        return cls._paginate(query, from_, to)

    @classmethod
    def _search_notes(
        cls,
        session: Session,
        author_id: int,
        query: str,
        from_: Optional[int] = None,
        to: Optional[int] = None,
    ) -> list[NoteModel]:
        sqlquery = session.query(NoteModel).options(
            joinedload(NoteModel.author)
        ).filter(
            NoteModel.author_id == author_id,
            NoteModel.text.contains(query)
        )
        return cls._paginate(sqlquery, from_, to)

    @classmethod
    def _paginate(
        cls,
        query: Query,
        from_: Optional[int] = None,
        to: Optional[int] = None
    ) -> list[NoteModel]:
        if from_ is not None:
```

```
            query = query.offset(from_)

        if to is not None:
            query = query.limit(to)

        return query.all()

    @classmethod
    def _convert_note_to_note_model(cls, note: Note) -> NoteModel:
        model = NoteModel(  # type: ignore
            created=note.created,
            author_id=note.author.id,
            text=note.text
        )

        if note.id is not None:
            model.id = note.id

        return model

    @classmethod
    def _convert_note_model_to_note(cls, model: NoteModel) -> Note:
        return Note(
            id=model.id,
            text=model.text,
            created=model.created,
            author=Account(
                id=model.author.id,
                account_id=model.author.account_id,
                application=model.author.application
            )
        )
```

Б.23 Код файлу «src/notes/infrastructure/telegram/handlers/callback.py»

```
from aiogram.dispatcher import FSMContext
from aiogram.types import CallbackQuery, ParseMode
from injector import inject

from src.accounts.infrastructure.telegram.account_manager import (
    TelegramAccountManager,
)
from src.common.telegram.base.handlers.callback_query import (
    CallbackQueryHandler,
)
from src.notes.application.use_cases.note.delete.contract import DeleteNote
from src.notes.application.use_cases.note.delete.request import (
    DeleteNoteRequest,
)
from src.notes.application.use_cases.note.get.contract import GetNote
from src.notes.application.use_cases.note.get.request import GetNoteRequest
from src.notes.infrastructure.telegram.keyboards import (
    NotesKeyboard,
    SearchResultsKeyboard,
)
from src.notes.infrastructure.telegram.states import EDITING_STATE
```

```
class UpdateNoteCallbackHandler(CallbackQueryHandler):
    async def __call__(
        self,
        callback_query: CallbackQuery,
        state: FSMContext
    ) -> None:
        note_id = int(callback_query.data.removeprefix('edit_note_'))

        await state.set_state(EDITING_STATE)
        await state.set_data({'note_id': note_id})
        await callback_query.answer()
        await callback_query.message.edit_text(
            text='Please, send me a new note content'
        )


class NoteDetailsCallbackHandler(CallbackQueryHandler):
    @inject  # type: ignore
    def __init__(self, case: GetNote):
        self._case = case

    async def __call__(
        self,
        callback_query: CallbackQuery,
        state: FSMContext
    ) -> None:
        note_id = int(callback_query.data.removeprefix('note_'))

        request = GetNoteRequest(note_id=note_id)

        try:
            response = await self._case.execute(request)
        except GetNote.NotFound:
            await callback_query.message.edit_text(
                'Note not found',
                reply_markup=None
            )
            return

        note = response.get_note()

        await callback_query.bot.answer_callback_query(callback_query.id)
        await callback_query.message.edit_text(
            note.text,
            reply_markup=None,
            parse_mode=ParseMode.MARKDOWN_V2
        )


class DeleteNoteCallbackHandler(CallbackQueryHandler):
    @inject  # type: ignore
    def __init__(
        self,
        case: DeleteNote,
        keyboard: NotesKeyboard,
        search_keyboard: SearchResultsKeyboard,
        account_manager: TelegramAccountManager
    ):
        self._case = case
        self._keyboard = keyboard
        self._search_keyboard = search_keyboard
        self._account_manager = account_manager

    async def __call__(
```

```
        self,
        callback_query: CallbackQuery,
        state: FSMContext
    ) -> None:
        note_id = int(callback_query.data.removeprefix('delete_note_'))

        request = DeleteNoteRequest(note_id=note_id)

        try:
            await self._case.execute(request)
        except DeleteNote.NotFound:
            await callback_query.message.edit_text(
                'Note not found',
                reply_markup=None
            )
            return

        data = await state.get_data()
        page = data.get('page', 0)
        account = await self._account_manager.get_account(
            account_id=callback_query.from_user.id
        )

        if 'query' in data:
            keyboard = await self._search_keyboard.build(
                author_id=account.id,
                page=page,
                query=data['query']
            )

            await callback_query.message.edit_text(
                'Search results:',
                reply_markup=keyboard
            )
        else:
            keyboard = await self._keyboard.build(
                author_id=account.id,
                page=page
            )

            await callback_query.message.edit_text(
                'Choose a note to show:',
                reply_markup=keyboard
            )

        await callback_query.answer(
            text='Note was successfully deleted.',
            show_alert=False
        )


class ListNotesCallbackHandler(CallbackQueryHandler):
    @inject  # type: ignore
    def __init__(
        self,
        keyboard: NotesKeyboard,
        account_manager: TelegramAccountManager
    ):
        self._keyboard = keyboard
        self._account_manager = account_manager

    async def __call__(
        self,
        callback_query: CallbackQuery,
```

```
        state: FSMContext
    ) -> None:
        account = await self._account_manager.get_account(
            account_id=callback_query.from_user.id
        )
        page = int(callback_query.data.removeprefix('list_notes_'))

        keyboard = await self._keyboard.build(author_id=account.id,
page=page)
        await state.set_data({'page': page})

        if keyboard is not None:
            await callback_query.message.edit_text(
                text='Choose a note to show:',
                reply_markup=keyboard
            )
        else:
            await callback_query.message.edit_text(
                text=(
                    'No notes to show. Add a new note by simply sending me '
                    'anything'
                )
            )


class SearchNotesCallbackHandler(CallbackQueryHandler):
    @inject  # type: ignore
    def __init__(
        self,
        keyboard: SearchResultsKeyboard,
        account_manager: TelegramAccountManager
    ):
        self._keyboard = keyboard
        self._account_manager = account_manager

    async def __call__(
        self,
        callback_query: CallbackQuery,
        state: FSMContext
    ) -> None:
        account = await self._account_manager.get_account(
            account_id=callback_query.from_user.id
        )
        data = await state.get_data()
        page = int(callback_query.data.removeprefix('search_notes_'))
        query = data.pop('query', '')

        keyboard = await self._keyboard.build(
            author_id=account.id,
            query=query,
            page=page
        )

        await callback_query.message.edit_text(
            text='Search results:',
            reply_markup=keyboard
        )
        await state.set_data({'page': page, 'query': query})
```

Б.24 Код файлу «src/notes/infrastructure/telegram/handlers/message.py»

```
from aiogram.dispatcher import FSMContext
from aiogram.types import Message
from aiogram.utils.emoji import emojize
from injector import inject

from src.accounts.infrastructure.telegram.account_manager import (
    TelegramAccountManager,
)
from src.common.telegram.base.handlers.message import MessageHandler
from src.notes.application.use_cases.note.create.contract import CreateNote
from src.notes.application.use_cases.note.create.request import (
    CreateNoteRequest,
)
from src.notes.application.use_cases.note.update.contract import UpdateNote
from src.notes.application.use_cases.note.update.request import (
    UpdateNoteRequest,
)
from src.notes.infrastructure.telegram.keyboards import (
    NotesKeyboard,
    SearchResultsKeyboard,
)


class ListNotesHandler(MessageHandler):
    @inject  # type: ignore
    def __init__(
        self,
        keyboard: NotesKeyboard,
        account_manager: TelegramAccountManager
    ):
        self._keyboard = keyboard
        self._account_manager = account_manager

    async def __call__(self, message: Message, state: FSMContext) -> None:
        account = await
self._account_manager.get_account(message.from_user.id)

        keyboard = await self._keyboard.build(account.id)
        await state.set_data({'page': 0})

        if keyboard is not None:
            await message.bot.send_message(
                message.from_user.id,
                text='Choose a note to show:',
                reply_markup=keyboard
            )
        else:
            await message.bot.send_message(
                message.from_user.id,
                text=(
                    'No notes to show. Add a new note by simply sending me '
                    'anything'
                )
            )


class EditNoteHandler(MessageHandler):
```

```
    @inject  # type: ignore
    def __init__(self, case: UpdateNote):
        self._case = case

    async def __call__(self, message: Message, state: FSMContext) -> None:
        state_data = await state.get_data()
        note_id = state_data.get('note_id', None)

        if note_id is None:
            await message.reply(
                text=(
                    'Sorry, an error ocurred in our storage. Please, try '
                    'again or contact our developer: @diachkow'
                )
            )
            return

        request = UpdateNoteRequest(note_id=note_id, text=message.md_text)

        try:
            await self._case.execute(request)
        except UpdateNote.NotFound:
            await message.reply('Note is not found in our storage.')
            return

        await state.set_state(None)
        await message.reply('Note successfully updated!')


class CreateNoteMessageHandler(MessageHandler):
    @inject  # type: ignore
    def __init__(
        self,
        case: CreateNote,
        account_manager: TelegramAccountManager
    ):
        self._case = case
        self._account_manager = account_manager

    async def __call__(self, message: Message, state: FSMContext):
        account = await
self._account_manager.get_account(message.from_user.id)
        text = self._get_note_text(message)

        request = CreateNoteRequest(
            author=account,
            text=text
        )

        await self._case.execute(request)
        await message.reply(emojize('Saved! ✅'))

    @staticmethod
    def _get_note_text(message: Message) -> str:
        return message.md_text.replace('```', '```\n')


class SearchNotesMessageHandler(MessageHandler):
    @inject  # type: ignore
    def __init__(
        self,
        keyboard: SearchResultsKeyboard,
        account_manager: TelegramAccountManager
    ):
```

```
        self._keyboard = keyboard
        self._account_manager = account_manager

    async def __call__(self, message: Message, state: FSMContext):
        account = await
self._account_manager.get_account(message.from_user.id)
        query = message.text

        keyboard = await self._keyboard.build(account.id, query)

        await state.set_data({'query': query, 'page': 0})
        await message.bot.send_message(
            message.from_user.id,
            text='Search results:',
            reply_markup=keyboard,
        )
```

Б.25 Код файлу «src/notes/infrastructure/telegram/filters.py»

```
import re

from aiogram.types import CallbackQuery, Message


def is_detail_note_handler(callback: CallbackQuery) -> bool:
    return callback.data.startswith('note_')


def is_delete_note_handler(callback: CallbackQuery) -> bool:
    return callback.data.startswith('delete_note_')


def is_edit_note_handler(callback: CallbackQuery) -> bool:
    return callback.data.startswith('edit_note_')


def is_list_notes_handler(callback: CallbackQuery) -> bool:
    return callback.data.startswith('list_notes_')


def is_search_notes_handler(callback: CallbackQuery) -> bool:
    return callback.data.startswith('search_notes_')


def is_not_command_message(message: Message) -> bool:
    return not message.text.startswith('/')


def is_search_message(message: Message) -> bool:
    return bool(re.match(r'^#\S+$', message.text))
```

Б.26 Код файлу «src/notes/infrastructure/telegram/keyboards.py»

```
from typing import Optional

from aiogram.types import InlineKeyboardButton, InlineKeyboardMarkup
from injector import inject
```

```
from src.notes.application.use_cases.note.list.contract import ListNotes
from src.notes.application.use_cases.note.list.request import
ListNotesRequest
from src.notes.application.use_cases.note.search.contract import SearchNotes
from src.notes.application.use_cases.note.search.request import (
    SearchNotesRequest,
)


class _AppendButtonsMixin:
    PREFIX: str
    PAGE_SIZE: int

    def _append_page_buttons(
        self,
        keyboard: InlineKeyboardMarkup,
        page: int,
        notes_count: int
    ) -> None:
        if page == 0 and notes_count == self.PAGE_SIZE:
            # if this is a first page and next page is required
            next_page_button = self._create_next_page_button(page + 1)
            keyboard.row(next_page_button)
            return
        elif page == 0:
            return

        if notes_count < self.PAGE_SIZE:  # If this is a last page
            prev_page_button = self._create_prev_page_button(page - 1)
            keyboard.row(prev_page_button)
            return

        # This is a mid-length page (not the first and not the last one)
        prev_page_button = self._create_prev_page_button(page - 1)
        next_page_button = self._create_next_page_button(page + 1)
        keyboard.row(prev_page_button, next_page_button)

    def _create_next_page_button(self, next_page: int) ->
InlineKeyboardButton:
        return InlineKeyboardButton(
            'Next ➡',
            callback_data=f'{self.PREFIX}{next_page}',
        )

    def _create_prev_page_button(self, prev_page: int) ->
InlineKeyboardButton:
        return InlineKeyboardButton(
            '⬅ Previous',
            callback_data=f'{self.PREFIX}{prev_page}'
        )


class _PaginationParametersMixin:
    PAGE_SIZE: int

    def _get_pagination_params(self, page: int) -> tuple[int, int]:
        from_ = page * self.PAGE_SIZE
        to = from_ + self.PAGE_SIZE
        return from_, to


class NotesKeyboard(_AppendButtonsMixin, _PaginationParametersMixin):
```

```
    PAGE_SIZE = 7
    PREFIX = 'list_notes'

    @inject  # type: ignore
    def __init__(self, case: ListNotes):
        self._case = case

    async def build(
        self,
        author_id: Optional[int],
        page: int = 0
    ) -> Optional[InlineKeyboardMarkup]:
        from_, to = self._get_pagination_params(page=page)

        request = ListNotesRequest(
            author_id=author_id,
            from_=from_,
            to=to
        )

        response = await self._case.execute(request=request)
        notes = response.get_notes()

        if not notes and page == 0:
            return None

        keyboard = InlineKeyboardMarkup()
        for note in notes:
            button = InlineKeyboardButton(
                note.text,
                callback_data=f'note_{note.id}',
            )
            edit_button = InlineKeyboardButton(
                'Edit ✒',
                callback_data=f'edit_note_{note.id}'
            )
            delete_button = InlineKeyboardButton(
                'Delete ✖',
                callback_data=f'delete_note_{note.id}'
            )
            keyboard.add(button, edit_button, delete_button)

        self._append_page_buttons(keyboard, page, len(notes))
        return keyboard


class SearchResultsKeyboard(_AppendButtonsMixin, _PaginationParametersMixin):
    PAGE_SIZE = 7
    PREFIX = 'search_notes_'

    @inject  # type: ignore
    def __init__(self, case: SearchNotes, page_size: int):
        self._page_size = page_size or self.PAGE_SIZE
        self._case = case

    async def build(
        self,
        author_id: Optional[int],
        query: str,
        page: int = 0
    ) -> Optional[InlineKeyboardMarkup]:
        from_, to = self._get_pagination_params(page)
        request = SearchNotesRequest(
```

```
            author_id=author_id,
            query=query,
            from_=from_,
            to=to
        )

        try:
            response = await self._case.execute(request)
        except SearchNotes.NotFound:
            return None

        notes = response.get_notes()
        keyboard = InlineKeyboardMarkup()

        for note in notes:
            retrieve_button = InlineKeyboardButton(
                note.text,
                callback_data=f'note_{note.id}',
            )
            edit_button = InlineKeyboardButton(
                'Edit ✏',
                callback_data=f'edit_note_{note.id}'
            )
            delete_button = InlineKeyboardButton(
                'Delete ✖',
                callback_data=f'delete_note_{note.id}'
            )
            keyboard.row(retrieve_button, edit_button, delete_button)

        self._append_page_buttons(keyboard, page=page,
notes_count=len(notes))
        return keyboard
```

Б.27 Код файлу «src/notes/infrastructure/telegram/routing.py»

```
from aiogram import Dispatcher
from injector import Injector

from src.common.telegram.base.registrator import HandlerRegistrator
from src.notes.infrastructure.telegram.handlers.callback import (
    DeleteNoteCallbackHandler,
    ListNotesCallbackHandler,
    NoteDetailsCallbackHandler,
    SearchNotesCallbackHandler,
    UpdateNoteCallbackHandler,
)
from src.notes.infrastructure.telegram.handlers.message import (
    CreateNoteMessageHandler,
    EditNoteHandler,
    ListNotesHandler,
    SearchNotesMessageHandler,
)

from .filters import (
    is_delete_note_handler,
    is_detail_note_handler,
    is_edit_note_handler,
    is_list_notes_handler,
    is_not_command_message,
    is_search_message,
```

```
    is_search_notes_handler,
)
from .states import EDITING_STATE


class NotesHandlerRegistrator(HandlerRegistrator):
    def __init__(self, injector: Injector):
        self._injector = injector

    def register(self, dispatcher: Dispatcher) -> None:
        dispatcher.register_message_handler(
            self._injector.get(ListNotesHandler),
            state='*',
            commands=['notes']
        )
        dispatcher.register_message_handler(
            self._injector.get(SearchNotesMessageHandler),
            is_search_message,
            state='*'
        )
        dispatcher.register_message_handler(
            self._injector.get(CreateNoteMessageHandler),
            is_not_command_message,
            state=[None]
        )
        dispatcher.register_message_handler(
            self._injector.get(EditNoteHandler),
            is_not_command_message,
            state=[EDITING_STATE]
        )

        dispatcher.register_callback_query_handler(
            self._injector.get(DeleteNoteCallbackHandler),
            is_delete_note_handler
        )
        dispatcher.register_callback_query_handler(
            self._injector.get(UpdateNoteCallbackHandler),
            is_edit_note_handler,
        )
        dispatcher.register_callback_query_handler(
            self._injector.get(NoteDetailsCallbackHandler),
            is_detail_note_handler
        )
        dispatcher.register_callback_query_handler(
            self._injector.get(ListNotesCallbackHandler),
            is_list_notes_handler,
        )
        dispatcher.register_callback_query_handler(
            self._injector.get(SearchNotesCallbackHandler),
            is_search_notes_handler,
        )
```

Повний код програми можна подивитись у віддаленому Git-репозиторії за посиланням: https://gitlab.com/vitaliy-diachkov/noted/.

Додаток В

Слайди презентації

Національний університет “Запорізька політехніка”

Кафедра програмних засобів

Дипломна кваліфікаційна робота

Розробка Телеграм бота для збереження нотаток

Підготував: ст. гр. КНТ-117 — Дьячков В. В.

Керівник: доцент — Федорончак Т. В.

1

Рисунок В.1 – Слайд презентації №1

Об’єкт дослідження - процес створення нотаток.

Предмет дослідження - чат-бот для управління (перегляд, створення, редагування, видалення) короткими текстовими нотатками.

Мета роботи - розробка чат-боту для управління нотатками.

2

Рисунок В.2 – Слайд презентації №2

Аналіз предметної області

- Звичні застосунки для збереження нотаток намагаються привернути уваги користувача за допомогою додаткової функціональності: продвинутого текстового редактору, можливості збереження медіа, інтеграція з іншими ресурсами
- Чат-бот для створення нотаток розрахований на зовсім іншу аудиторію
- Написання чат-бота ставить розробника в певні рамки

Список розглянутих месенджерів:

- WhatsApp
- Viber
- Facebook Messenger
- Telegram ✔

3

Рисунок В.3 – Слайд презентації №3

Порівняння аналогів

Критерій порівняння	@exnobot	@killthemall_bot
Можливість створення нотаток	+	+
Можливість перегляду збережених нотаток	+	+
Можливість пошуку за нотатками	-	-
Можливість створення відкладених нотаток (нагадувань)	+	-
Можливість редагування нотаток	+	+
Можливість видалення нотаток	+	+
Можливість перегляду списку нотаток	-	-

4

Рисунок В.4 – Слайд презентації №4

Вибір засобів реалізації

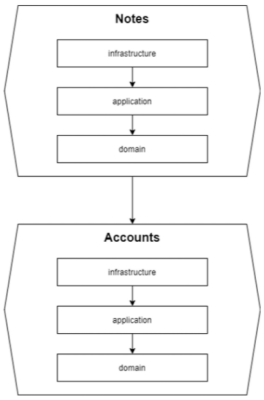
- мова програмування Python;
- клієнтська бібліотека для роботи з Telegram Bot API aiogram;
- інструмент для управління залежностями проекту poetry;
- система управління базами даних PostgreSQL;
- сховище даних користувацької сесії Redis;
- система контролю версій Git;
- засіб контейнеризації Docker.

5

Рисунок В.5 – Слайд презентації №5

Проектування архітектури

При проектуванні архітектури майбутнього застосунку було використано підходи предметно орієнтованого проектування (англ. Domain-Driven Design, DDD) і гексагональної архітектури (архітектури портів та адаптерів).



6

Рисунок В.6 – Слайд презентації №6

Взаємодія з базою даних

Для організації взаємодії з базою даних доцільно використовувати ORM (Object-Relational Mapping), оскільки цей інструмент надає низькорівневий інтерфейс для роботи з базою даних, а тому вибір конкретної СКБД може бути переглянутий на будь якій стадії проекту.

В якості ORM для розроблюваного програмного забезпечення було обратно SQLAlchemy.

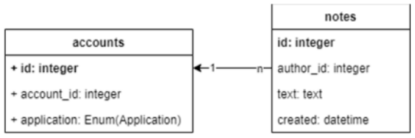


Схема бази даних

7

Рисунок В.7 – Слайд презентації №7

Сутність

Являє собою категорію індивідуальних об'єктів, які залишаються незмінними на різних етапах програми, для яких атрибути не грають великого значення, а послідовність та ідентичність, які поширюється в житті усієї системи називаються сутностями.
Класи-сутності називаються на її честь: Account, Note.

```
@dataclass
class Note(Entity):
    author: Account
    text: str
    created: datetime
    id: Optional[int] = None
```

8

Рисунок В.8 – Слайд презентації №8

Фабрика

Створення об'єктів предметної області повинно бути делеговане до спеціалізованих фабрик. Це дає можливість підміняти реалізацію створення об'єктів.

```
class NoteFactory:
    @staticmethod
    def build(author: Account, text: str) -> Note:
        return Note(
            author=author,
            text=text,
            created=datetime.now()
        )
```

9

Рисунок В.9 – Слайд презентації №9

Сукупність

Колекція об'єктів, які пов'язані між собою завдяки головній сутності (Root Entity), інакше відомій як Aggregate root. Коренева сутність колекції об'єктів гарантує узгодженість змін, що вносяться до сукупності, забороняючи зовнішнім об'єктам посилатися на членів колекції.

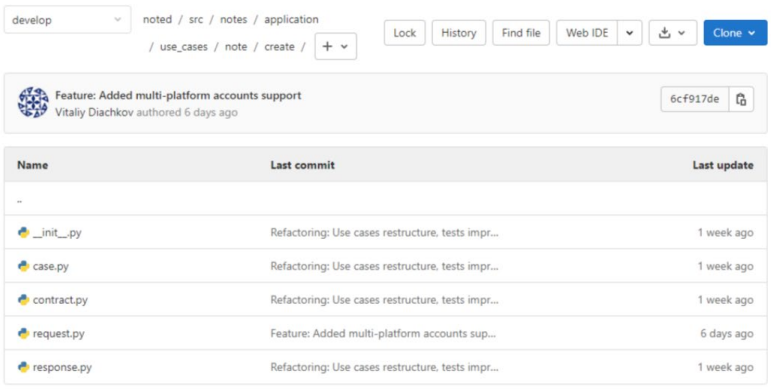
```python
class Notes(ABC):
    @staticmethod
    @abstractmethod
    async def get_by_id(note_id: int) -> Optional[Note]:
        """Return note by its ID if one found else None."""

    @staticmethod
    @abstractmethod
    async def save(note: Note) -> Note:
        """Save note to a database."""

    @staticmethod
    @abstractmethod
    async def update_by_id(note_id: int, **kwargs: Any) -> None:
        """Update note by id."""
```

10

Рисунок В.10 – Слайд презентації №10



Структура окремого пакету use-case

11

Рисунок В.11 – Слайд презентації №11

Об'єкт значення (Value Object)

Об'єкт, який містить атрибути, але не має концептуальної ідентичності. Він повинен розглядатися як незмінний об'єкт.

Приклад: Коли українці обмінюються гривнями, вони зазвичай не розрізняють унікальність купюр, вони стурбовані номінальною вартістю банкноти. У цьому контексті, гривня є об'єктом значення (Value object). Проте, НБУ може бути стурбована кожною унікальною гривнею, в цьому контексті - кожна гривня буде сутністю (Entity).

```python
class CreateNoteRequest:
    def __init__(
        self,
        *,
        author: Account,
        text: str
    ):
        self._author = author
        self._text = text

    def to_dict(self) -> dict[str, Any]:
        return {
            'author': self._author,
            'text': self._text
        }


class CreatedNote:
    def __init__(self, note: Note):
        self._note = note

    def note(self) -> Note:
        return self._note
```

12

Рисунок В.12 – Слайд презентації №12

Контракт сценарію викори

Абстрактний інтерфейс, що є представленням певної дії над сутністю предметної області.

Виокремлення інтерфейсу сценаріїв використання дозволяє у майбутньому підміняти їх реалізацію.

```python
class CreateNote(ABC):
    """Create note action."""

    @abstractmethod
    async def execute(self, request: CreateNoteRequest) -> CreatedNote:
        """Perform note creation."""
```

13

Рисунок В.13 – Слайд презентації №13

Сценарій використання

описують конкретні варіанти використання, є відображенням дій, що виконуються над сутностями предметної області. Їх типовим прикладом можуть бути CRUD (англ. «Created, Read, Update, Delete») операції.

```python
from src.notes.application.use_cases.note.create.contract import \
    CreateNote as CreateNoteContract


class CreateNote(CreateNoteContract):
    @inject  # type: ignore
    def __init__(self, factory: NoteFactory, notes: Notes):
        self._factory = factory
        self._notes = notes

    async def execute(self, request: CreateNoteRequest) -> CreatedNote:
        note = self._factory.build(**request.to_dict())
        return CreatedNote(await self._notes.save(note))
```

14

Рисунок В.14 – Слайд презентації №14

Сховище

Отримання об'єктів предметної області повинно делегуватися в спеціалізовані сховища об'єктів. Це дає можливість підміняти місце збереження об'єктів.

Сховище є реалізацією інтерфейсу, наданого сукупністю в domain-шарі.

```python
class NoteRepository(Notes):

    @classmethod
    async def get_by_id(cls, note_id: int) -> Optional[Note]:
        async with async_session() as session:
            db_note = await cls._get_by_id(session, note_id)

            if db_note is not None:
                return cls._convert_note_model_to_note(db_note)

    @classmethod
    async def _get_by_id(
        cls,
        session: AsyncSession,
        note_id: int
    ) -> NoteModel:
        statement = select(NoteModel).options(
            joinedload(NoteModel.author)
        ).where(NoteModel.id == note_id)
        result = await session.execute(statement)
        db_note = result.scalars().first()
        return db_note

    @classmethod
    def _convert_note_model_to_note(cls, model: NoteModel) -> Note:
        return Note(
            id=model.id,
            text=model.text,
            created=model.created,
            author=Account(
                id=model.author.id,
                account_id=model.author.account_id,
                application=model.author.application
            )
        )
```

15

Рисунок В.15 – Слайд презентації №15

Dependency Inversion Principle & Dependency Injection

```python
class CreateNote(CreateNoteContract):
    @inject  # type: ignore
    def __init__(self, factory: NoteFactory, notes: Notes):
        self._factory = factory
        self._notes = notes

    async def execute(self, request: CreateNoteRequest) -> CreatedNote:
        note = self._factory.build(**request.to_dict())
        return CreatedNote(await self._notes.save(note))
```

16

Рисунок В.16 – Слайд презентації №16

Розробка через тестування

Технологія розробки програмного забезпечення, яка використовує короткі ітерації розробки, що починаються з попереднього написання тестів, які визначають необхідні покращення або нові функції. Кожна ітерація має на меті розробити код, який пройде ці тести. Нарешті, програміст або група вдосконалюють код для погодження змін.

Один із ключових моментів TDD полягає у тому, що підготовка тестів перед написанням самого коду пришвидшує процес внесення змін.

Варто зауважити, що керована тестами розробка є методологією розробки програмного забезпечення, а не його тестування.

Test-Driven Development відноситься до концепції **екстремального програмування**.

17

Рисунок В.17 – Слайд презентації №17

Автоматичне тестування

```
(.venv) vitaliy@pc:~/coding/stud/diploma/noted:(develop)$ pytest
==================== test session starts ====================
platform linux -- Python 3.9.4, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /home/vitaliy/coding/stud/diploma/noted, configfile: pyproject.toml
plugins: cov-2.11.1, asyncio-0.15.1, mock-3.6.0
collected 34 items

tests/unit/accounts/application/use_cases/account/create/test_case.py ..
tests/unit/common/entities/test_base_entity.py .
tests/unit/notes/application/use_cases/note/create/test_case.py .
tests/unit/notes/application/use_cases/note/delete/test_case.py ..
tests/unit/notes/application/use_cases/note/get/test_case.py ..
tests/unit/notes/application/use_cases/note/list/test_case.py .
tests/unit/notes/application/use_cases/note/search/test_case.py ..
tests/unit/notes/application/use_cases/note/update/test_case.py ..
tests/unit/notes/domain/factories/test_note.py .
tests/unit/notes/infrastructure/telegram/test_filters.py .....................

---------- coverage: platform linux, python 3.9.4-final-0 ----------
Name                                                                  Stmts   Miss  Cover
-----------------------------------------------------------------------------------------

...

-----------------------------------------------------------------------------------------
TOTAL                                                                   328      0   100%

============================== 34 passed in 0.76s ==============================
```

18

Рисунок В.18 – Слайд презентації №18

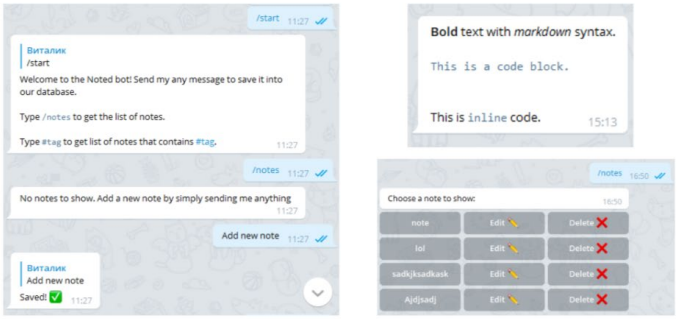
Мануальне тестування

Назва тест-кейсу	Перевірка створення нотаток
Передумови виконання	Початий процес взаємодії з ботом командою «/start»
Крок	**Очікуваний результат**
Відправити повідомлення боту з довільним текстом	Бот відповідає повідомленням «Note successfully created!»
Виконати команду «/notes»	У списку нотаток з'явилася нова з повідомленням яке було відправлено боту на попередньому кроці (із збереженням розмітки тексту)

Приклад тест-кейсу

19

Рисунок В.19 – Слайд презентації №19



Скріншоти роботи програми

20

Рисунок В.20 – Слайд презентації №20

Висновки

У результаті виконання дипломної роботи було отримано готового до роботи Telegram-бота для управління короткими текстовими нотатками: їх створення, редагування, видалення та перегляд з можливістю пошуку за тегами.

Реалізоване програмне забезпечення відповідає всім вимогам, поставленим в технічному завданні до проекту. Отримана програма є стійкою до змін і передбачає можливості свого подальшого розширення.

21

Рисунок В.21 – Слайд презентації №21