МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Веб технології»)
на тему:

«Бібліотека для взаємодії з реляційними базами даних»

Виконав студент групи ВТ-21-1
ШУМСЬКИЙ Олександр Вячеславович

Керівник роботи:
САВІЦЬКИЙ Роман Святославович

Рецензент:
ФАНТ Микола Олександрович

Житомир – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»
Зав. кафедри інженерії
програмного забезпечення
Тетяна Вакалюк
«14» лютого 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ШУМСЬКИЙ Олександр Вячеславович**
Керівник роботи: **САВІЦЬКИЙ Роман Святославович**
Тема роботи: **«Бібліотека для взаємодії з реляційними базами даних»**,
затверджена Наказом закладу вищої освіти від **«14» лютого 2025 р., №61/с**
Вихідні дані для роботи: **Об'єктом дослідження є використання об'єктно-реляційного підходу при роботі з базою даних. Предметом дослідження є використання технологій розробки для спрощення роботи з реляційними базами даних.**
Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	**Консультант**	Завдання видав	Завдання прийняв
1	Савіцький Р. С.	27.02.2024р.	27.02.2024р.
2	Савіцький Р. С.	27.02.2024р.	27.02.2024р.
3	Савіцький Р. С.	27.02.2024р.	27.02.2024р.

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з бібліотеки для роботи з базами даних та пояснювальної записки. Пояснювальна записка до випускної роботи містить 61 сторінок, 53 ілюстрацій та 11 таблиць.

У роботі поставлені основні завдання на розробку ORM системи на Golang. Система включає модулі для операцій з базою даних, адміністративної панелі, логування та аутентифікації Також було проведено детальний порівняльний аналіз аналогічних систем. Було обрано підхід монорепозиторію з клієнт-серверною архітектурою. Наведені основні сценарії використання системи, логіка міжмодульної взаємодії. Клієнтська частина розроблена за допомогою бібліотеки HTMX а серверна частина за допомогою фреймворку Fiber. Для консольного клієнту використана бібліотека Cobra.

Ключові слова: ORM, БІБЛІОТЕКА, БАЗА ДАНИХ, GOLANG, SQL, СУТНІСТЬ, CRUD

ABSTRACT

The bachelor's final qualification work consists of an ORM Golang library and an explanatory note. The explanatory note to the final work contains 61 pages, 53 illustrations and 11 tables.

The work sets the main tasks for the development of an ORM system in Golang. The system includes modules for database operations, an administrative panel, logging and authentication. A detailed comparative analysis of similar systems was also conducted. A monorepository approach with a client-server architecture was chosen. The main scenarios for using the system and the logic of intermodule interaction are presented. The client part is developed using the HTMX library, and the server part is developed using the Fiber framework. The Cobra library is used for the console client.

Keywords: ORM, LIBRARY, DATABASE, GOLANG, SQL, ENTITY, CRUD

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення

БД – База даних

ORM – Object-Relational Mapping

JWT – JSON Web Token

JS – JavaScript

TS – TypeScript

CRUD – Create Read Update Delete

ВСТУП

Актуальність теми пов’язана з активним розвитком програмування і потребою до гнучких інструментів для створення сучасних програмних продуктів. Одним із ключових напрямів є використання ORM (Object-Relational Mapping) систем, що дозволяють спрощено та структуровано взаємодіяти з базами даних. ORM дає змогу програмістам працювати з даними через об'єктно-орієнтований підхід, що полегшує розробку та підтримку програм.

Використання ORM системи дозволяє уникнути рутинної роботи з SQL-запитами, підвищити продуктивність та зменшити кількість помилок під час маніпуляції даними.

Мета – дослідження використання ORM систем у Golang для спрощення роботи з базами даних та підвищення ефективності розробки.

Завдання:

аналіз теоретичних засад ORM систем та їх можливостей;

дослідження особливостей ORM у Golang;

розробка моделі програмного продукту з використанням ORM;

створення прикладного додатку для демонстрації роботи ORM;

Предметом дослідження специфічні аспекти проєктування, реалізації та використання ORM систем, зокрема в середовищі мови програмування Golang, їхні можливості, переваги та обмеження.

Об’єктом дослідження технології об'єктно-реляційного відображення (ORM), які забезпечують взаємодію між об'єктно-орієнтованими програмами та реляційними базами даних, а також підходи до їх реалізації у програмному забезпеченні.

За темою випускної кваліфікаційної роботи бакалавра було опубліковано тези:

Савіцький Р. С. ВІДМІННОСТІ РЕЛЯЦІЙНИХ БАЗ ДАНИХ. Тези доповідей X Міжнародна науково-технічна конференція Інформаційно-комп’ютерні

технології: інновації, проблеми, рішення, 01-02 грудня 2022 року. Житомир: «Житомирська політехніка», 2025. С.164-165;

Савіцький Р. С. ACID-ВЛАСТИВОСТІ ТА РІВНІ ІЗОЛЯЦІЇ ТРАНЗАКЦІЙ. Тези доповідей X Міжнародна науково-технічна конференція Інформаційно-комп'ютерні технології: інновації, проблеми, рішення, 01-02 грудня 2022 року. Житомир: «Житомирська політехніка», 2025. С.164-165;

РОЗДІЛ 1.

1.1. Визначення та постановка задачі

Основна мета роботи є розробка ORM системи на Golang, задля підвищення ефективності роботи з базами даних розробниками програмного забезпечення, збільшення надійності написаного коду а також легкого адміністрування. ORM система має складатися з деяких ключових програмних модулів, а саме:

Модуль ядра має основний функціонал системи, який може використовуватись у всіх інших.

Модуль адаптерів має реалізації адаптерів для реляційних баз даних, готових для використання розробниками. Всі взаємодії з проектом відрізняються за рахунок адаптерів. Розробники не зобов'язані використовувати адаптери, адже вони мають можливість написати його самостійно якщо ні одне з готових рішень їм не підходить.

Модуль ORM служить проміжною ланкою між розробником та базою даних, має в собі функціонал абстракції над базою даних, дозволяє робити отримання та зміну даних залежно від потреб розробника, працює за допомогою вказаного адаптеру.

Модуль адміністративної панелі дозволяє проводити роботу з базою даних у веб-інтерфейсі, він служить для адміністраторів баз даних і дає змогу швидко маніпулювати даними.

Модуль консольного додатку дає змогу розробникам працювати з кодогенерацією а також міграціями через зручний консольний інтерфейс.

Основними етапами розробки платформи є:

Написання перевикористовувальної логіки запитів до бази даних;

Розробка готових імплементацій адаптерів для використання розробниками;

Розробка адміністративної панелі;

Розробка консольного інтерфейсу;

Тестування додатку;

Результатом реалізації поставленого завдання є бібліотека ORM системи, яка включає в себе основну логіку роботи з базами даних, готові адаптери для популярних реляційних баз даних, адміністративну панель та консольний інтерфейс.

1.2. Аналіз аналогів програмного продукту

Основна мета ORM системи полягає в збільшенні продуктивності роботи розробників для виконання операцій з базою даних. Система підвищує надійність системи за допомогою чіткої схеми бази даних а також захисту від SQL-ін'єкцій. Також система має зручний інтуїтивний інтерфейс, який дасть змогу редагувати дані додатків нетехнічним людям. Відносно цих критерій розглянемо аналоги системи для дослідження ринку в порівняльній таблиці 1.1. Розглянемо аналоги детальніше:

Таблиця 1.1

Перелік аналогів системи

N	Назва	Підтримувальні БД	Особливості	Плюси	Мінуси
1.	2.	3.	4.	5.	6.
1.	GORM	MySQL, PostgreSQL, SQLite, SQL Server, TiDB, Clickhouse	Найпопулярніша ORM для Golang, активна розробка, авто-міграції, хуки	Простий у використанні, велика спільнота, гнучкість	Повільний у порівнянні з "чистим" SQL, складний для кастомізації
2.	Ent	MySQL, PostgreSQL, SQLite	Генерація схем, типобезпечність, граф-орієнтований API	Висока продуктивність, потужний генератор коду, хороший DX	Високий поріг входу, потребує попередньої генерації коду

Продовження таблиці 1.1

1.	*2.*	*3.*	*4.*	*5.*	*6.*
3.	XORM	MySQL, PostgreSQL, CockroachDB, SQLite, MS SQL, Oracle, TiDB	Простий API, кешування, автоматичні міграції	Простий синтаксис, ефективне кешування, хороша підтримка	Мало різноманітних можливостей
4.	SQLBoiler	MySQL, PostgreSQL, SQLite, MS SQL, CockroachDB	Генерація ORM-коду з SQL-схем, строгий типобезпечний API	Висока продуктивність, прозорість на основі чистого SQL	Не підтримує динамічні запити, складніший у налаштуванні
5.	Beego ORM	MySQL, PostgreSQL, SQLite	Вбудований в Beego framework, підтримка зв’язків	Легко інтегрується з Beego, простота використання	Мала гнучкість, неактивний розвиток

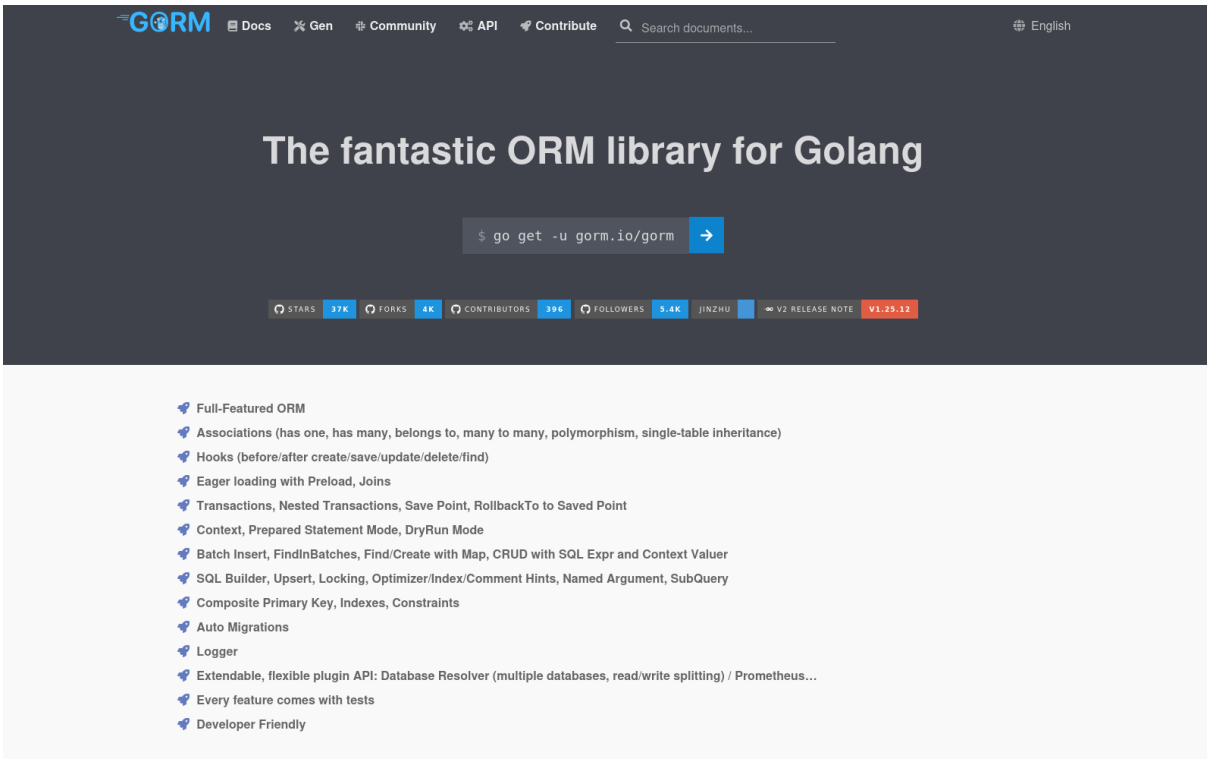

Рис. 1.1. Сайт бібліотеки «GORM»

GORM – це найпопулярніша ORM система [3]. Вона відома своєю зручністю та обширним функціоналом. Серед різноманітних баз даних вона підтримує MySQL, PostgreSQL, SQLite, SQL Server, TiDB та Clickhouse (див. рис.1.1).

Переваги:

Інтуїтивна робота за допомогою будівельника запитів;

Автоматичні міграції, які дають змогу для синхронізації структури БД з моделями Go;

Присутні розширені функції: хуки, "м'яке" видалення (soft delete), транзакції;

Детальна документація та найбільша спільнота серед всіх ORM систем;

Недоліки:

Продуктивність може стати ключовим мінусом у виборі цієї ORM системи для високонавантажених додатків;

Не повна підтримка складних запитів: Для деяких випадків доводиться вдаватися до використання чистого SQL;

Проблеми з пошуком проблем: Помилки можуть бути недостатньо інформативними;

Дана ORM система є хорошим вибором для простих веб-додатків, де швидкість розробки є пріоритетом.

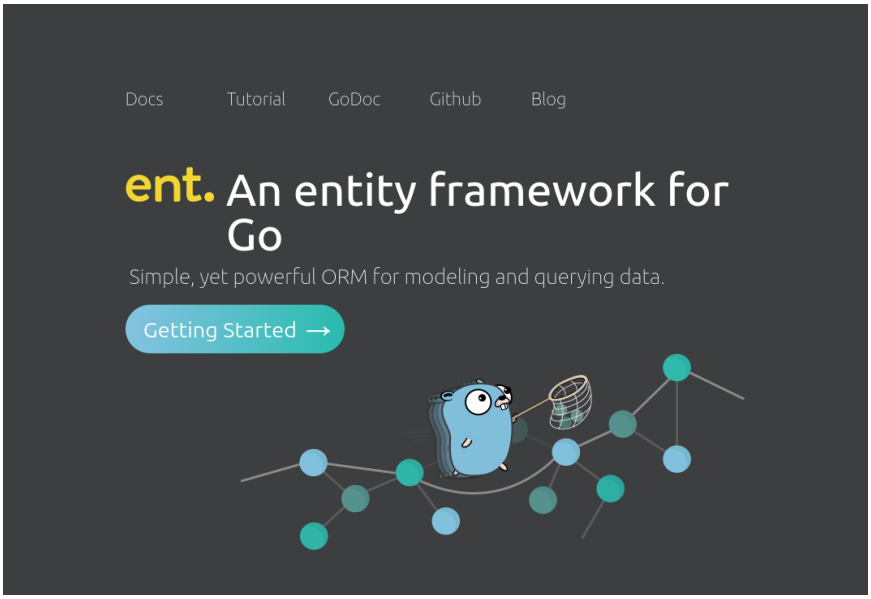

Рис. 1.2. Сайт бібліотеки «Ent»

Ent – це ORM від Facebook з кодогенерацією, орієнтована на типобезпеку та високу продуктивність (див. рис.1.2) [4].

Переваги:

Генерує типобезпечний код на основі схем, що запобігає помилкам під час виконання;

Висока продуктивність;

Наявні можливості для створення складних запитів;

Підтримка міграцій і плагінів для розширення функціоналу;

Недоліки:

Складність налаштування, вимагає попередньої генерації коду;

Мала популярність. Мало прикладів і менш активна спільнота порівняно з іншими ORM;

Дана ORM система є хорошим вибором для проектів з критичними вимогами до безпеки та продуктивності

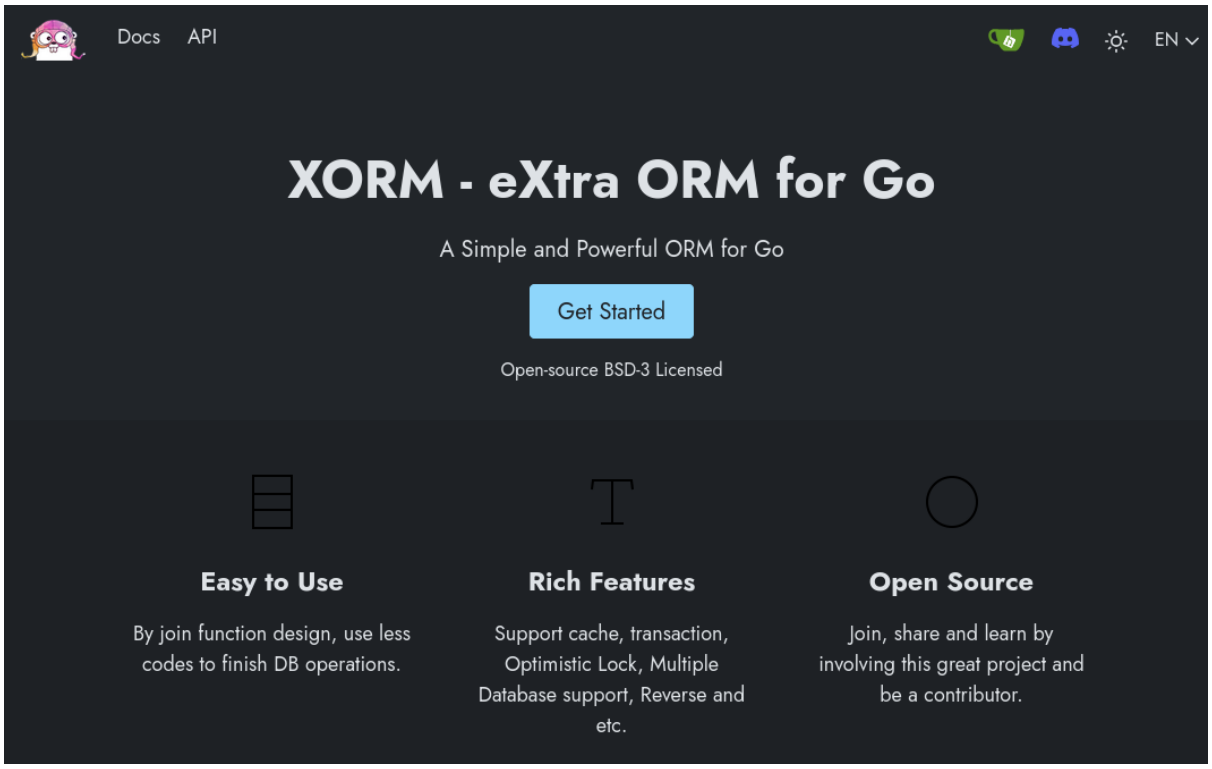

Рис. 1.3. Сайт бібліотеки «XORM»

XORM – це легка, але потужна ORM система з пріоритетом продуктивності (див. рис.1.3). Підтримує широкий спектр баз даних, включаючи Oracle та TiDB.

Переваги:

Правильно згенеровані запити в поєднанні з вбудованим кешуванням допомагають у розробці високонавантажених систем;

Добре генерує SQL з урахуванням специфік СУБД;

Є вбудована можливість використовувати автоматичні міграції БД;

Розширена підтримка транзакцій і розподілених систем;

Недоліки:

Потрібно багато різних кроків для комплексних запитів;

Обмежена документація. Часто можуть виникнути різні проблеми, які вимагають;

Дана ORM система є чудовим інструментом для додатків з великими обсягами даних або високим навантаженням.

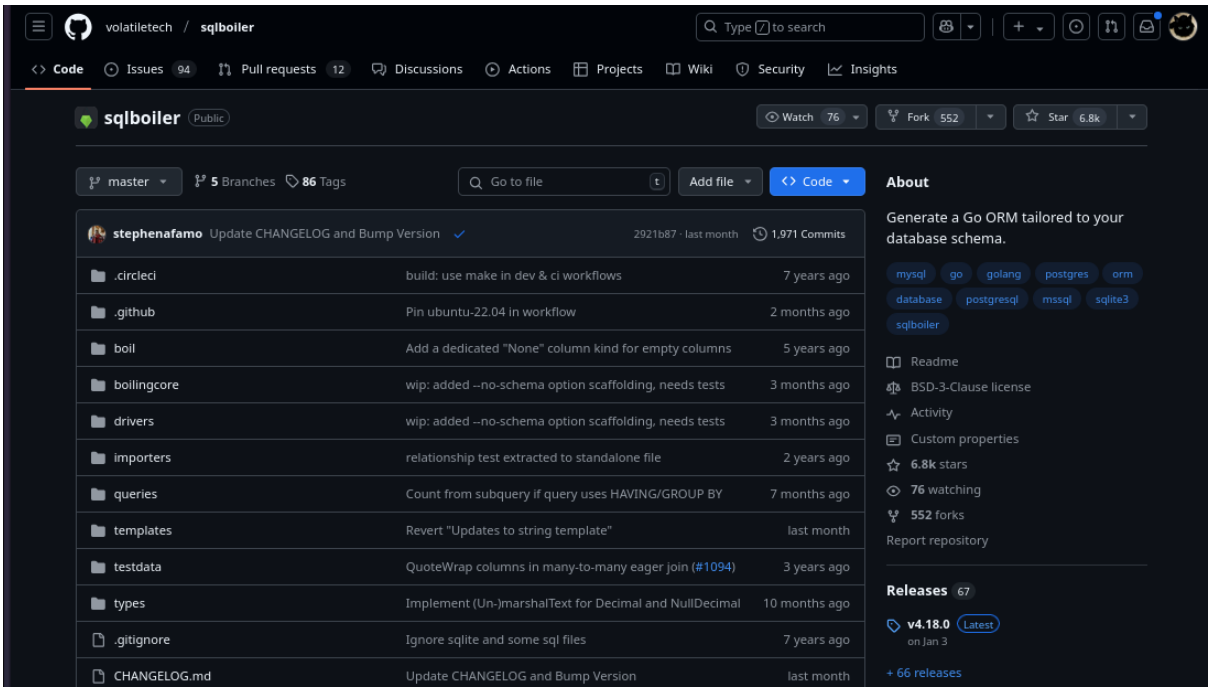

Рис. 1.4. Сторінка бібліотеки «sqlboiler» на сайті «github.com»

SQLBoiler – це ORM система, яка відрізняється від інших підходом «code-first»: вона генерує моделі з існуючої схеми бази даних, створюючи повністю типізований API [5]. Завдяки цьому підходу, SQLBoiler забезпечує високу швидкодію(див. рис.1.4).

Переваги:

Більшість помилок виявляється ще на етапі компіляції;

Висока продуктивність;

Генерація моделей безпосередньо з існуючої схеми мінімізує розбіжності між кодом і структурою бази даних та підвищує надійність системи;

Недоліки:

Необхідність повторного запуску коду при кожній зміні схеми бази даних. Це може спричиняти незручності при розробці додатків;

Мала гнучкість. Підхід з статичний кодом, може набагато менш адаптивним для змін схеми бази даних;

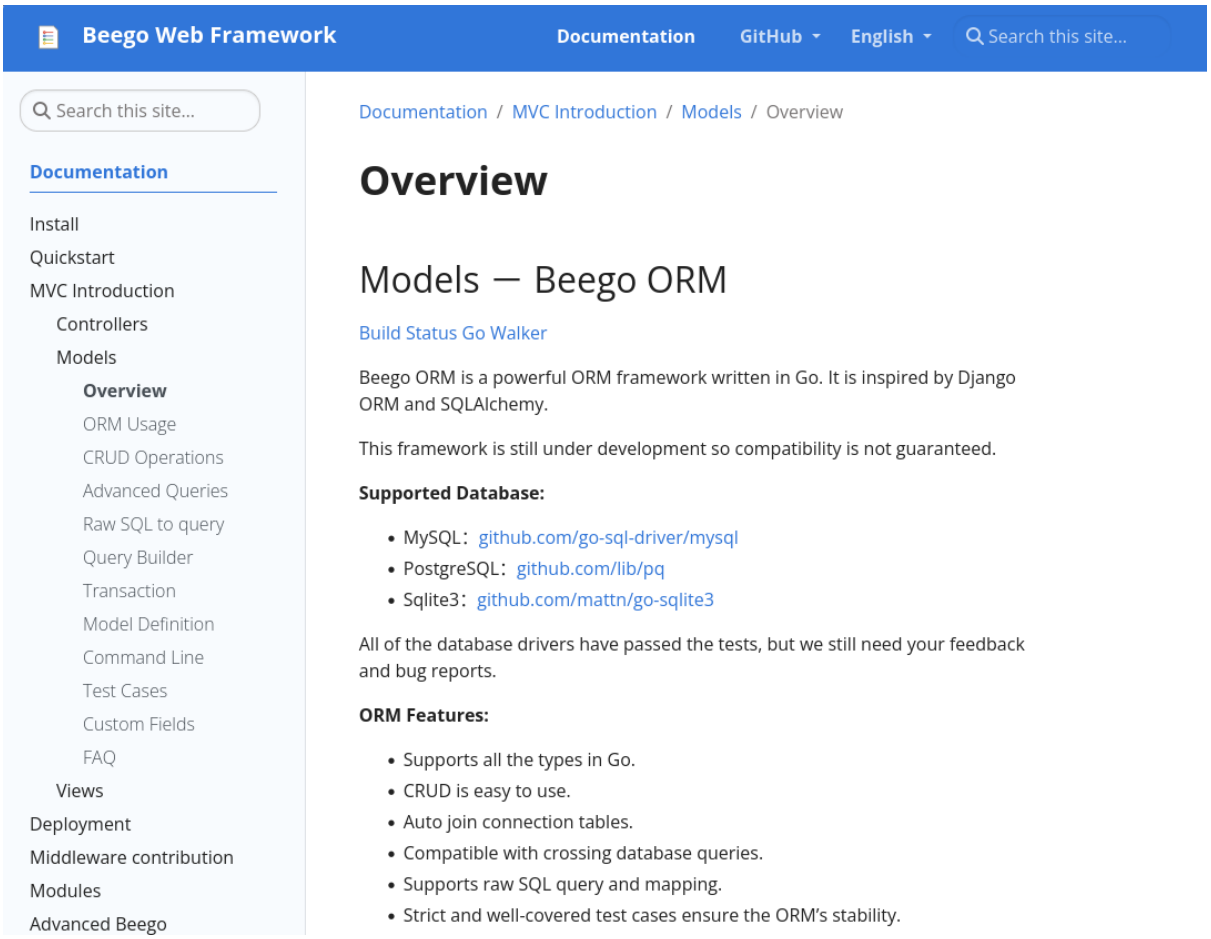

Рис. 1.5. Сайт бібліотеки «Beego»

Beego ORM це частина фреймворку Beego [6], орієнтована на простоту та інтеграцію з іншими модулями Beego (див. рис.1.5). Також вона є агностиком і може бути застосована в будь якому оточенні незалежно від фреймворку.

Переваги:

Потрібно мінімальне налаштування для базових операцій;

Інтеграція з Beego;

Підтримка транзакцій і базових міграцій;

Недоліки:

Обмежений функціонал. Не підходить для складних запитів і нестандартних СУБД;

Мала спільнота;

Beego ORM це хороший вибір для проектів на фреймворці Beego або дуже малих проектів

Аналіз аналогічних платформ допоміг визначити сильні та слабкі сторони існуючих рішень. Як можна побачити з аналогів, всі наведені ORM системи мають підтримку MySQL, PostgreSQL та SQLite але на жаль вони майже не мають в собі вбудованих інструментів, які могли б спростити процес розробки. Таким чином, аналіз аналогів допоміг остаточно визначити необхідний набір функціоналу та розставити пріоритетність функціоналу у розробці системи.

1.3. Визначення архітектури ПЗ

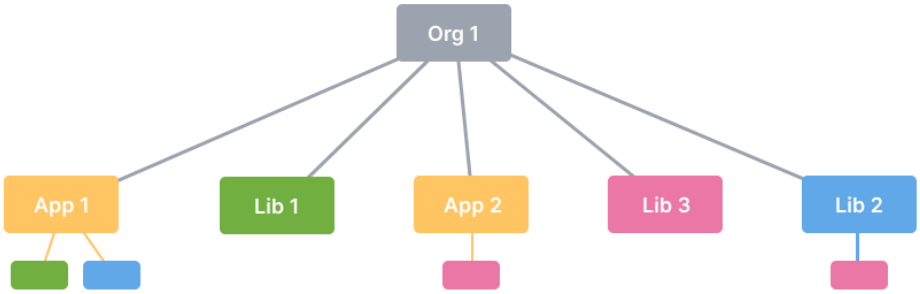

Рис. 1.6 Схема роботи монорепозиторія

Для розробки ORM системи було обрано підхід модульного монорепозиторія (див. рис.1.6) де різні частини можуть спілкуватись одна з одною за допомогою різних протоколів зв'язку. Такий підхід дає змогу користувачу дуже гнучко використовувати систему і підключати тільки ті частини які йому дійсно потрібні. Прикладом цього є різні адаптери баз даних, користувач може обрати один або декілька адаптерів і на основі цього використовувати систему. Також це дає змогу заміняти готові частини системи на свої власні імплементуючи взаємодію з інтерфейсами інших частин системи. Також цей підхід зводить дублювання коду до мінімуму, так як спільний код можна переносити в окремі бібліотеки і використовувати в різних місцях.

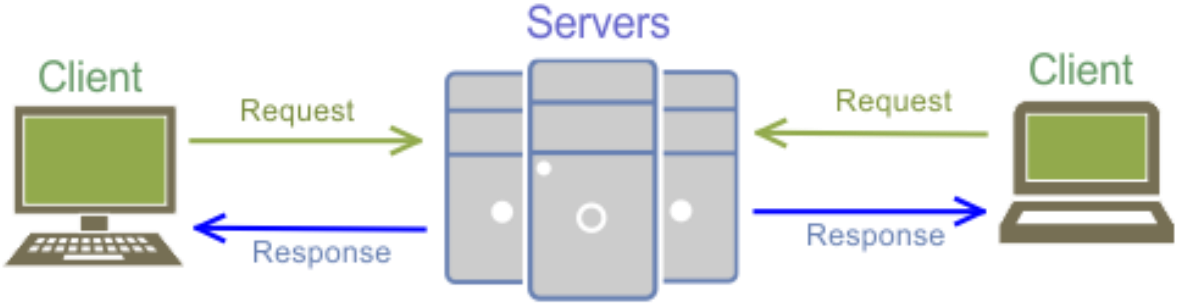

Рис. 1.7 Клієнт-серверна архітектура

В деяких випадках використовується клієнт-серверна архітектура (див. рис.1.7), де одні модулі є клієнтами а інші серверами.

Серверне представлення слугує для реалізації бізнес логіки системи. Всі важливі дані зберігаються тільки на сервері. В системі передбачені різні серверні частини для подальшої взаємодії з клієнтськими.

Клієнтське представлення слугує для відображення даних в зручному для користувача форматі. В системи передбачена варіативність різних клієнтів. Веб-клієнт повинен бути сумісним з різними веб-браузерами, таких як Mozilla FireFox та Google Chrome, і працювати на різних платформах, забезпечуючи зручний та надійний доступ до основних функцій. Інтерфейс користувача має бути інтуїтивним та зручним, з використанням сучасного дизайну. При розробці важливо забезпечити адаптивність для зручного використання на пристроях з різними розмірами екрану. Також важлива можливість використання сенсорних пристроїв для зручного використання. Консольний клієнт повинен мати вбудовані підказки для простого користування.

1.4. Обґрунтування вибору інструментальних засобів

При розробці було використано декілька інструментів для ефективної та зручної роботи. В якості основних інтегрованих середовищ розробки використовувалися Webstorm та Goland разом з необхідними плагінами для використаних бібліотек. Ці середовища розробки дуже добре себе показують в складних проектах завдяки індексації всього проекту і можуть виявити критичні помилки ще до запуску коду.

Для тестування REST API було обрано Postman. Він має в собі весь необхідний функціонал для тестування серверної частини а також має купу можливостей для автоматизації команд, які необхідні для коректної роботи системи. Він має зручний графічний інтерфейс який прискорює тестування системи і робить пошук проблем більш зручним.

Таблиця 1.2

Порівняння бібліотек для створення веб-інтерфейсів

N	Критерій	HTMX	React	Alpine.js
1.	*2.*	*3.*	*4.*	*5.*
1.	Складність	Низька	Висока	Низька
2.	Підхід рендерингу	Серверний (HTML over the wire)	Клієнтський (Virtual DOM)	Клієнтський (нативний DOM)
3.	Інтерактивність	через AJAX/HTML	через компоненти	через директиви
4.	Зв'язок із сервером	Вбудований (через атрибути)	Потрібен додатковий код (fetch)	Потрібен додатковий код (fetch)
5.	Використання з Golang	Ідеально для серверного рендерингу	Потрібен окремий API (REST/GraphQL)	Потрібен окремий API (REST/GraphQL)
6.	Розмір бібліотеки	~10 KB	~40 KB (React) + ~5 KB (ReactDOM)	~10 KB
7.	Крива навчання	Дуже низька	Середня	Низька

Проаналізувавши бібліотеки для простих інтерактивних веб-інтерфейсів для клієнтської частини була обрана бібліотека HTMX [5] завдяки своїй простоті і підтримці серверного рендеру, яка не буде вимагати ще одного веб-сервера. Функціоналу HTMX цілком вистачить для відмальовування інтерактивного інтерфейсу адміністративної панелі.

Для серверної частини були проаналізовані найпопулярніші фреймворки на Golang. Для потреб проекту було обрано Fiber. Він має чудову продуктивність та вбудовану підтримку різноманітних технологій. Для консольної частини обрана бібліотека Cobra [6]

Таблиця 1.3

Порівняння фреймворків для Golang

N	Критерій	Fiber	Chi	Echo
1.	*2.*	*3.*	*4.*	*5.*
1.	Продуктивність	Дуже висока (побудований на fasthttp)	Висока (базується на net/http)	Висока (базується на net/http)
2.	Підтримка WebSockets	Так	Ні	Так
3.	Підтримка шаблонів	Так	Ні	Так
4.	Готові рішення	Багато вбудованих функцій	Мінімалістичний, потребує додаткових бібліотек	Багато вбудованих функцій
5.	Вбудована Підтримка JWT	Так	Ні	Так
6.	Документація	Детальна та зрозуміла	Хороша, але менш структурована	Хороша, але менш детальна
7.	Підтримка контексту	Власна реалізація	Через стандартний context.Context	Через стандартний context.Context

Вибір інструментів був обумовлений їх відповідністю поставленим вимогам. Використання сучасних технологій дозволяє покращити ефективність розробки системи.

Висновки до першого розділу

У першому розділі було проведено детальний аналіз існуючих аналогів, обґрунтовано вибір інструментальних засобів, сформульовано основні завдання та визначено архітектуру програмного забезпечення. Детальний аналіз аналогів дозволив визначити необхідні функціональні вимоги до системи. Для реалізації проекту було обрано підхід модульного монорепозиторія з клієнт-серверною

архітектурою, яка забезпечує масштабованість і гнучкість. Клієнтська частина створена на основі HTMX, що дозволяє зробити зручний користувацький веб-інтерфейс. Серверна частина створена на фреймворці Fiber, що забезпечує ефективну обробку запитів. При виборі технологічного стеку враховувалися потреби системи.

Загалом, описана архітектура та обрані технології забезпечують високу продуктивність, надійність, пришвидшення швидкості розробки додатків і можливість легкого розширення функціоналу системи.

РОЗДІЛ 2.

2.1. Технічне завдання на розробку системи

Найменування програмного засобу

Повне найменування програмної системи: «Бібліотека для взаємодії з реляційними базами даних» (надалі «програма»). Коротка назва програмної системи – «ORM».

Призначення розробки та область застосування: Програмна система «ORM» призначена для розробників програмного забезпечення, вона спрощує та робить безпечнішою роботу з базою даних.

Мета проекту: проект створюється для спрощення роботи розробників з базами даних у Golang, забезпечення автоматизації процесів взаємодії з базами даних та підвищення продуктивності команди.

Представлення проєкту: проект буде реалізовано як бібліотека для мови програмування Golang із графічним інтерфейсом для управління базами даних.

Вимоги адміністраторів:

- Підтримка популярних SQL баз даних (PostgreSQL, MySQL, SQLite, MS SQL);
- Зручний графічний інтерфейс для перегляду, редагування та аналізу даних у реальному часі;
- Можливість генерації запитів через GUI без написання коду;

Вимоги розробників:

Можливість виконувати складні запити;

Підтримка сортування, фільтрування, обмеження кількості записів а також

Характеристика об’єкта комп’ютеризації:

Користувач бібліотеки та GUI-інструменту зможе швидко та ефективно працювати з базами даних, створювати та керувати моделями бази даних, а також отримувати візуальне представлення даних без необхідності писати SQL-запити вручну.

Функціональні вимоги:

- Можливість роботи з моделями: створення, редагування та видалення моделей через зміну конфігурації вручну або за допомогою GUI;
- Автоматичне створення міграцій: система повинна генерувати міграції автоматично на основі змін у схемі моделей;
- Підтримка зв’язків: визначення та підтримка відношень між моделями (один-до-одного, один-до-багатьох, багато-до-одного);
- Графічний інтерфейс: інструмент для перегляду та редагування даних із функціями пошуку та фільтрації;

Нефункціональні вимоги:

Сприйняття:

- Час, необхідний для навчання роботи з ORM, – 1-2 години для звичайних користувачів;
- Інтерфейс має бути зрозумілим і забезпечувати інтерактивні підказки;

Надійність:

- Гарантія збереження даних під час обробки транзакцій навіть у разі збою;
- Система повинна мати захист від SQL-ін’єкцій;

Продуктивність:

- Підтримка мінімум 100 одночасних користувачів GUI;
- Висока швидкість виконання запитів (не більше 500 мс на простий запит);

Масштабованість:

- Підтримка великих баз даних із мільйонами записів без втрати продуктивності;

2.2. Аналіз вимог до програмного продукту

Для забезпечення ефективної розробки та відповідності встановленим стандартам і вимогам, наведено високорівневі вимоги, які повинна задовольняти ORM-система для Golang. Ці вимоги спрямовані на забезпечення зручності використання, високої продуктивності та гнучкості в роботі з базами даних. Глосарій наведено в додатку А відповідно.

Аналіз вимог користувачів дозволив визначити основні варіанти використання ORM-системи для Golang. На рис. 1.1 наведено діаграму варіантів використання системи.

У розроблюваній ORM-системі розрізняють двох ключових акторів: розробник та адміністратор бази даних. Короткий опис акторів представлено в таблиці 1.1.

Таблиця 1.1

Виявлення акторів

N	Актор	Короткий опис
1.	2.	3.
1.	Адміністратор	Дивитись всі дані а також може їх редагувати. Має доступ до логів системи.
2.	Розробник	Використовує бібліотеку як допоміжну річ у розробці системи (описує таблиці та звязки між ними, виконує запити до бази даних)

Таблиця 1.2

Виявлення варіантів використання

N	Основний актор	Найменування	Формулювання
1.	2.	3.	4.
1.	Адміністратор / Розробник	Робота з моделями	Цей варіант дає змогу працювати з моделями бази даних.
		Генерація та виконання міграцій	Цей варіант дозволяє працювати з міграціями БД (генерація, перегляд а також відкат якщо виникли проблеми).
		Виконання нестандартних SQL запитів	Цей варіант дозволяє писати свої SQL запити. Результат можна побачити в зручному форматі. Може використовуватись як для налагодження в процесі розробки так і для редагування даних по нестандартним критеріям.
2.	Адміністратор	Перегляд та управління даними через GUI	Цей варіант дозволяє переглядати дані в зручному форматі а також редагувати їх.
		Аутентифікація в GUI	Цей варіант дозволяє аутентифікацію через різних користувачів.
		Робота з журналом подій	Цей варіант дозволяє перегляд а також очистку журналу подій. Ця інформація може знадобитись якщо
3.	Розробник	Робота з автогенерацією коду	Цей варіант дозволяє геренувати код для подальшого використання в системі.

Рис. 1.1. Діаграма варіантів використання системи

2.3. Розробка моделі програмного комплексу на логічному рівні

Діаграми активності, разом із діаграмами варіантів використання та діаграмами станів, належать до діаграм поведінки, оскільки вони описують, що відбувається в системі, що моделюється.

Замовники та розробники часто стикаються з питаннями і завданнями, в яких необхідно досягти згоди, тому важливо спілкуватися чітко та лаконічно, аби забезпечити порозуміння з обох сторін. Діаграми активності допомагають людям з різним рівнем технічної підготовки розуміти однакові процеси і приймати оптимальні рішення під час розробки системи.

Діаграма активності (діаграма діяльності) служить для моделювання послідовностей дій у бізнес-процесах, показуючи, як методи або дії взаємодіють між собою. Ці послідовності можуть включати обробку даних в різних областях або паралельне виконання дій. Діаграми активності схожі на блок-схеми алгоритмів і представлені у вигляді графа, де дії є вузлами, а переходи між ними — зв'язками.

Кожен стан на діаграмі активності відповідає виконанню певної операції, а перехід до наступного стану можливий лише після завершення попередньої операції. Таким чином, діаграма активності є специфічним випадком діаграми станів.

Функції та застосування діаграм активності:

- Демонстрація логіки алгоритмів;
- Ілюстрація бізнес-процесів між користувачами та системою;
- Оптимізація та вдосконалення процесів, роз'яснення складних сценаріїв використання.

На рис. 2.1 зображено процес взаємодії користувача з графічним інтерфейсом (GUI) та сервером при виконанні транзакцій у системі. Користувач обирає кілька операцій у GUI, які потім надсилаються серверу у вигляді транзакції. Сервер ініціює транзакцію, відкриваючи її для подальшого виконання. Після цього починається цикл виконання операцій. Кожна операція виконується по черзі, і процес продовжується, поки є ще операції для виконання.

Коли всі операції виконано, сервер перевіряє, чи були вони успішними. Якщо всі операції пройшли без помилок, транзакція фіксується, і дані оновлюються в базі. В іншому випадку, якщо хоча б одна операція не виконалась успішно, сервер відкачує транзакцію і повідомляє користувача про помилку. Цей процес забезпечує надійність і консистентність даних у системі, гарантуючи, що або всі операції будуть виконані, або жодна з них не буде застосована, якщо виникла помилка в процесі.

Рис. 2.1. Діаграма виконання операцій в GUI

На рис. 2.2 зображена послідовність дій при авторизації адміністратора в системі. Спочатку адміністратор вводить логін та пароль, після чого система перевіряє правильність введених даних. Якщо дані вірні, система генерує JWT токен і надає його користувачу для подальшого доступу. В разі, якщо введені дані неправильні, система виводить повідомлення про помилку.

Після того як адміністратор отримує токен, він використовує його для доступу до графічного інтерфейсу. Система перевіряє дійсність токену. Якщо токен є дійсним, адміністратор отримує доступ до GUI. Якщо токен недійсний, доступ до системи відмовляється. Цей процес забезпечує контроль за безпекою доступу до інтерфейсу.

Рис. 2.2. Діаграма авторизації

На рис. 2.3 зображено процес взаємодії розробника з ORM системою під час роботи з моделями та міграціями. Перш за все, розробник змінює модель. Залежно від того, як саме ці зміни були внесені, система приймає різні варіанти дій. Якщо модель була змінена через графічний інтерфейс користувача (GUI), зміни зберігаються безпосередньо через нього. Якщо ж зміни вносяться вручну, вони також зберігаються, але без використання GUI.

Далі система автоматично створює міграцію, яка відповідає внесеним змінам у моделі. Якщо міграція успішно створена, вона показується розробнику для перевірки. Якщо ж система не змогла створити міграцію, вона виводить повідомлення про помилку. Після цього розробник може застосувати міграцію, і система перевіряє, чи була вона успішно застосована до бази даних. Якщо застосування міграції пройшло успішно, система повідомляє розробника про успіх, а в разі помилки виводить відповідне повідомлення про невдачу.

Процес охоплює важливі етапи взаємодії з моделями та міграціями, забезпечуючи контроль за змінами та можливістю застосування міграцій, що є основою ефективної роботи з базою даних у рамках цієї ORM системи для Golang.

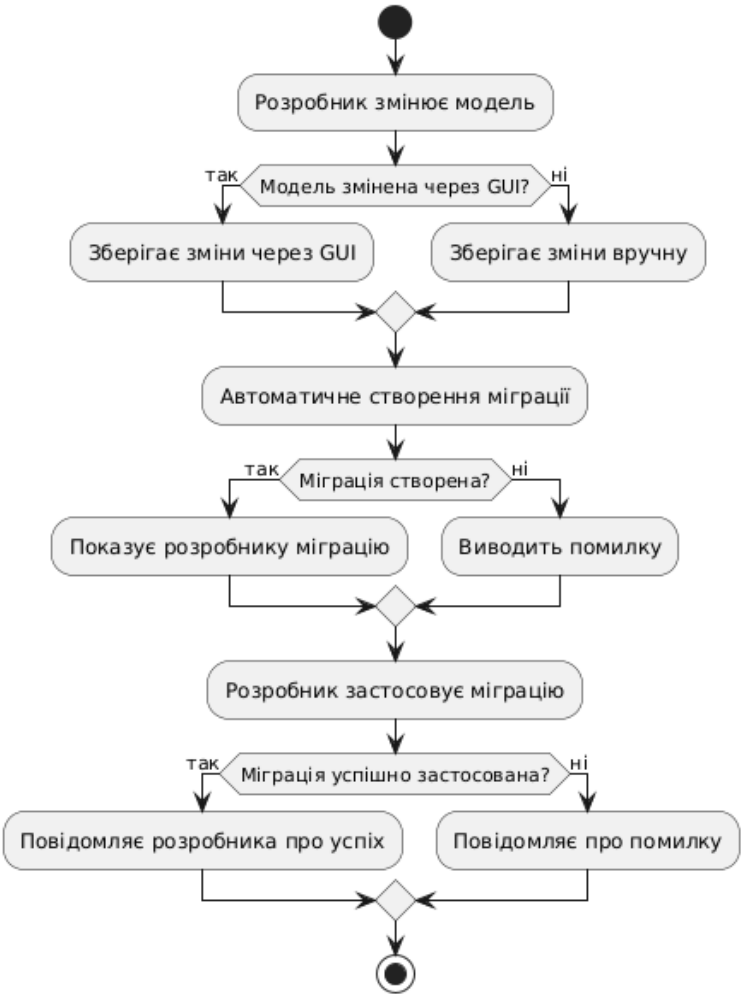

Рис. 2.3. Діаграма створення міграцій

Після визначення дійових осіб та варіантів використання, а також створення ілюстрації поведінки системи і опису послідовності виконання операцій, можна перейти до розробки попередньої об’єктно-орієнтованої моделі системи. Створення діаграми класів розпочнемо з визначення основних сутностей та їхніх зв’язків у системі.

Основні сутності системи:

- DbClient – клієнт бази даних;
- QueryBuilder – конструктор SQL запитів;
- MigrationManager – менеджер міграцій;

- ORMConfig – конфігурація ORM системи;
- Model – модель бази даних;
- User – користувач;
- PostgresAdapter – адаптер для бази даних PostgreSQL;
- SQLiteAdapter – адаптер для бази даних SQLite;
- MySQLAdapter – адаптер для бази даних MySQL;
- Studio – адміністративна панель;
- UI – користувацький інтерфейс адміністративної панелі;
- MigrationView – інтерфейс міграцій адміністративної панелі;
- TableView – інтерфейс таблиць адміністративної панелі;

Між об’єктами «DbClient» і «ORMConfig», «DbClient» і «IDbAdapter», «Studio» і «DbClient», «Studio» і «TableView» , «Studio» і «UI» та «Studio» і «MigrationView» встановлено зв’язок агрегації. Це означає, що видалення об’єкта-контейнера не призводить до видалення об’єктів, які він зберігає.

Агрегація є спеціалізованим видом асоціації, що описує зв’язки «один-до-багатьох» або «багато-до-багатьох», а також відносини «частина-ціле» між кількома об’єктами. У такому зв’язку один об’єкт, зазвичай, виступає контейнером або колекцією інших об’єктів. Важливо, що контейнер не контролює життєвий цикл компонентів, які можуть існувати незалежно від нього.

На діаграмі класів є об’єкти взаємодіють через композицію — більш суворий вид агрегації, коли один об’єкт складається з інших. Особливістю цього зв’язку є те, що компонент може існувати лише як частина контейнера. У разі видалення об’єкта-контейнера необхідно також видалити всі об’єкти, які він містить.

У нашій моделі цей зв’язок використовується між об’єктами «DbClient» і «IEntity», «Model» і «Task» та «Model» і «User». Це підкреслює, що кожен екземпляр IEntity може існувати лише в контексті відповідного DbClient, який виступає його контейнером і повністю контролює його життєвий цикл.

На діаграмі класів зв’язок залежності використовується для позначення тимчасового відношення між об’єктами, коли один об’єкт використовує

функціональність іншого, але не володіє ним і не контролює його життєвий цикл. Цей зв’язок вказує на те, що об’єкт залежить від іншого для виконання певної операції.

У нашій моделі зв’язок залежності застосовується між наступними об’єктами: «TableView» і «DbClient», «DbClient» і «QueryBuilder» та «DbClient» і «MigrationManager»

У кожному з цих випадків DbClient виступає джерелом функціональності, яку тимчасово використовують відповідні об’єкти, не впливаючи на його життєвий цикл.

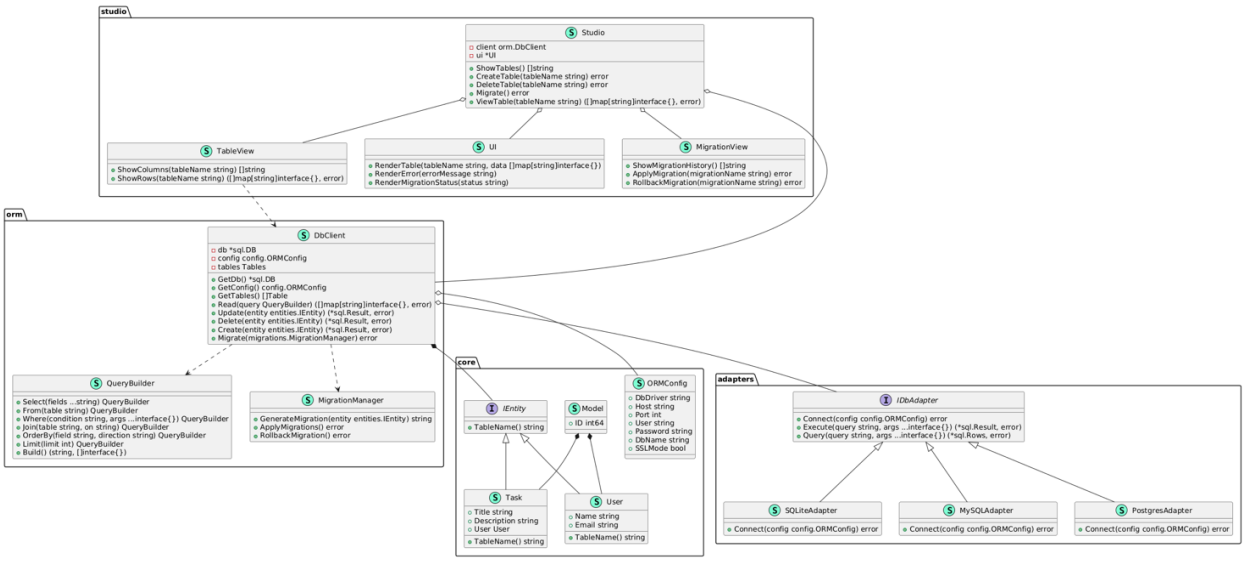

Рис. 2.4. Діаграма класів

До створеної діаграми класів було застосовано (рис. 2.4.) чотири патерни Builder, Adapter, Factory Method та Dependency Injection.

Builder – це паттерн, який використовується для створення складних об'єктів шляхом поетапної побудови, розділяючи процес генерації об'єкта від його структури, що дозволяє використовувати той самий процес побудови для створення різних варіацій об'єктів. У випадку класу QueryBuilder цей підхід застосовується для побудови SQL-запитів. Основна ідея полягає в тому, що кожен метод класу QueryBuilder повертає сам об'єкт QueryBuilder, завдяки чому забезпечується можливість виклику методів у ланцюжку (method chaining). Наприклад, виклики

builder.Select().Where().OrderBy() виконуються як одна послідовність, що дозволяє інтуїтивно та гнучко налаштовувати складні запити. Ця концепція є ключовою для розробників, які прагнуть мінімізувати дублювання коду, уникнути потенційних помилок при написанні запитів вручну та забезпечити більшу читабельність коду. Такий підхід також спрощує додавання умов, фільтрів, сортування чи об'єднань, оскільки кожна з цих дій додається як окремий етап до побудови кінцевого SQL-запиту. Як результат, QueryBuilder виступає універсальним інструментом, що дозволяє адаптувати запит до різноманітних потреб користувача, зберігаючи при цьому чітку й організовану структуру.

Adapter – це паттерн проєктування, що забезпечує сумісність між структурами з різними інтерфейсами, дозволяючи їм взаємодіяти без потреби змінювати їх код. У цьому випадку реалізація виконана через структури PostgresAdapter, SQLiteAdapter та MySQLAdapter, кожен з яких призначений для роботи з відповідною базою даних: PostgreSQL, SQLite або MySQL. Основою цього підходу є загальний інтерфейс IDbAdapter, який визначає стандартний набір методів — такі як Connect, Execute та Query. Цей інтерфейс слугує контрактом, який зобов’язуються виконувати всі адаптери, незалежно від типу бази даних, з якою вони працюють. Наприклад, метод Connect забезпечує підключення до бази даних, Execute відповідає за виконання SQL-команд, а Query використовується для отримання результатів запитів. Кожен із адаптерів реалізує ці методи, враховуючи специфіку роботи конкретної бази даних. Завдяки такій абстракції додаток може взаємодіяти з будь-якою підтримуваною базою даних через уніфікований інтерфейс, що дозволяє змінювати базу даних або додавати підтримку нових систем без необхідності переписувати код в інших частинах системи. Це значно підвищує гнучкість, полегшує підтримку та розширення програмного забезпечення, а також забезпечує чітке розділення відповідальностей.

Factory Method — це паттерн, що дозволяє створювати об'єкти без зазначення їх конкретних класів, делегуючи процес створення підкласифікованим методам. У даному прикладі реалізація виконана через клас DbClient та інтерфейс IDbAdapter.

DbClient виступає в ролі основного клієнта, який взаємодіє з базою даних через абстрактний інтерфейс IDbAdapter, не залежачи від специфіки конкретних реалізацій. Конкретний адаптер, наприклад PostgresAdapter, SQLiteAdapter або MySQLAdapter, створюється за допомогою фабричного методу залежно від налаштувань конфігурації чи вхідних параметрів. Такий підхід забезпечує інкапсуляцію логіки створення об'єктів, дозволяє легко розширювати функціонал шляхом додавання нових адаптерів без зміни існуючого коду клієнта, а також спрощує тестування, адже клієнт завжди працює з уніфікованим інтерфейсом. Таким чином, DbClient лише запитує необхідний адаптер через фабричний метод, що дозволяє динамічно обирати реалізацію залежно від контексту використання, забезпечуючи гнучкість і дотримання принципів SOLID, зокрема принципу відкритості/закритості та інверсії залежностей.

Dependency Injection (ін’єкція залежностей) — це важливий паттерн проєктування, який дозволяє відокремити створення залежностей від їх використання, підвищуючи гнучкість і тестованість системи. У цьому випадку він реалізований через класи Studio, UI та DbClient, де кожен клас отримує свої залежності через параметри конструктора замість того, щоб створювати їх самостійно. Наприклад, клас Studio залежить від DbClient, але DbClient передається до Studio під час створення, що забезпечує можливість використовувати різні реалізації DbClient без змін у класі Studio. Аналогічно, у класі UI залежність також передається зовні. Такий підхід дозволяє зменшити зв'язність між класами, роблячи систему більш модульною та зрозумілою. Крім того, це значно полегшує тестування, оскільки можна легко замінювати реальні залежності на тестові при перевірці функціоналу кожного класу окремо. Таким чином, Dependency Injection сприяє кращій підтримуваності та розширюваності коду, забезпечуючи можливість вносити зміни в поведінку програми через зміну переданих залежностей без необхідності редагування основного класу.

Діаграми послідовності та взаємодії компонентів є ключовими інструментами для візуалізації та аналізу роботи системи. Вони дозволяють

зрозуміти, як система реагує на події, як її компоненти співпрацюють для виконання завдань та як між собою взаємодіють окремі елементи системи.

На прикладі діаграми на рисунку 2.5, демонструється послідовність дій під час роботи адміністратора з системою управління базами даних. Зокрема, показано, як адміністратор ініціює запит через інтерфейс адміністративної панелі, як обробляються запити до бази даних та виконуються міграції.

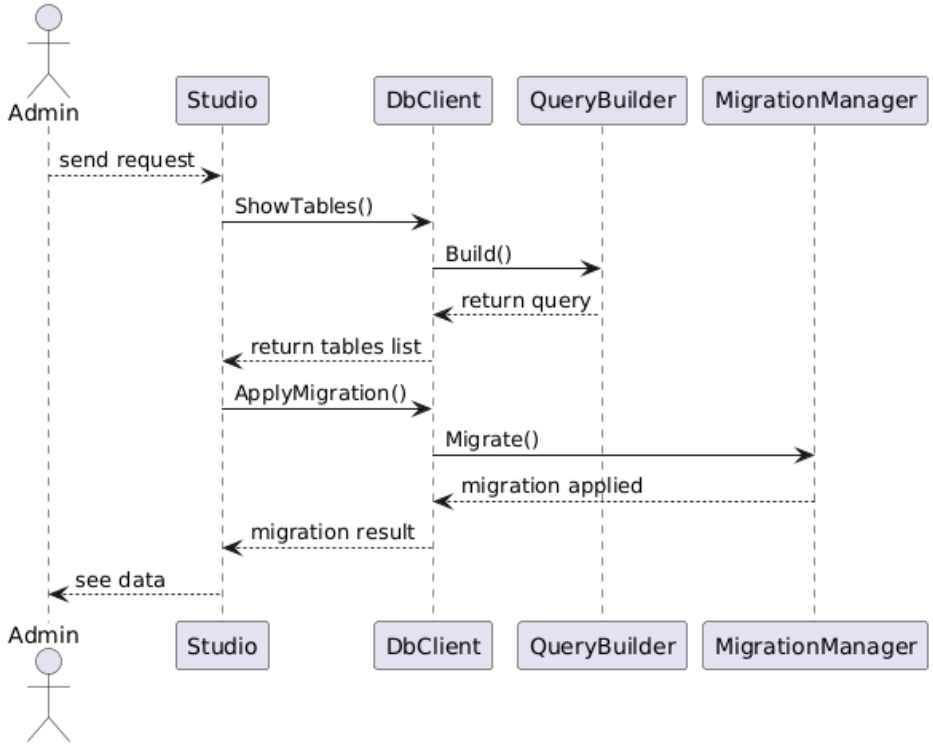

Рис. 2.5. Діаграма послідовності використання адміністративної панелі

Розуміння точної роботи кожного компонента системи є важливим для будь-якого програміста. Діаграми компонентів є ефективним засобом, який допомагає уявити архітектуру та взаємодію частин програмного забезпечення.

Ці діаграми дозволяють іншим розробникам краще зрозуміти структуру певної системи. Вони можуть описувати програмні рішення, реалізовані в будь-якій мові або стилі програмування. Основна мета таких діаграм – показати зв’язки між компонентами системи. «Компоненти» можуть означати різні елементи: модулі класів, файли, бібліотеки, виконувані файли, пакети або цілі підсистеми, що взаємодіють між собою.

На діаграмі компонентів, представленій вище, показана архітектура системи з використанням ORM (Object-Relational Mapping). Наприклад, компонент «Frontend» взаємодіє з «Admin Backend» через REST API (HTTPS). «Admin Backend», у свою чергу, використовує базу даних для зберігання інформації через протокол TCP/IP, а також записує логи у NoSQL базу даних (Elasticsearch). Компонент «App Backend» містить дві частини: веб-додаток та CLI, які спільно взаємодіють з основною бізнес-логікою і базою даних.

Кожен компонент має свою специфічну роль у системі. Наприклад, компонент «Authentication Service» реалізує обробку OAuth2, забезпечуючи автентифікацію користувачів. Інші компоненти, такі як бази даних, забезпечують збереження даних у SQL чи NoSQL форматах.

Діаграми компонентів (рис. 2.6.) допомагають чітко побачити зв’язки між частинами системи, зрозуміти її структуру і взаємодію між компонентами. Для полегшення розуміння на діаграмі можуть бути додані нотатки для кожного компонента, що пояснюють його функції та зв’язки з іншими елементами.

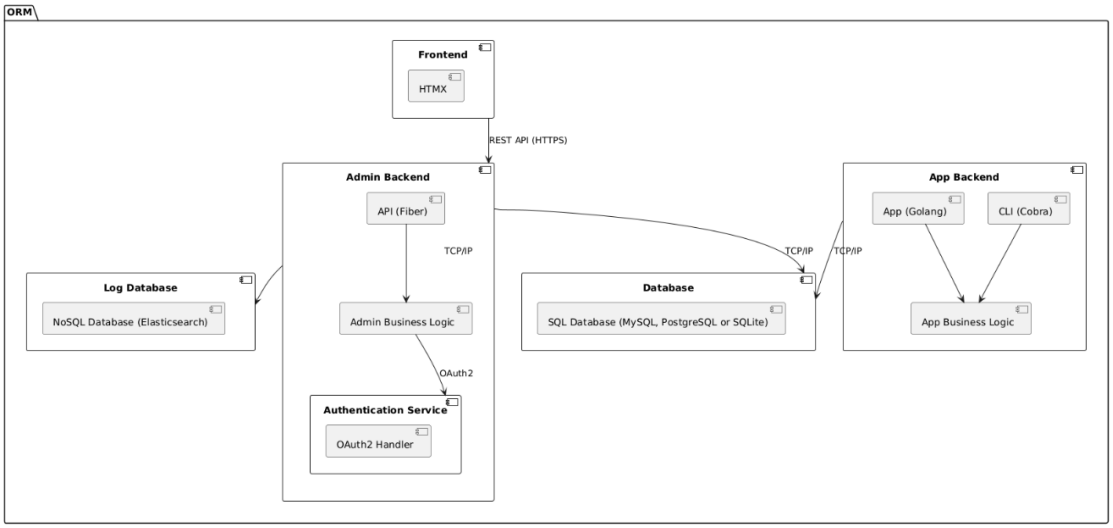

Рис. 2.6. Діаграма компонентів

2.4. Фізична модель

Під час проєктування системи були застосовані основні принципи чистої архітектури, що забезпечили поділ програмного забезпечення на окремі рівні. Ключовим принципом, який визначає роботу цієї архітектури, є правило

залежностей: залежності у вихідному коді повинні спрямовуватися виключно всередину (рис. 3.1).

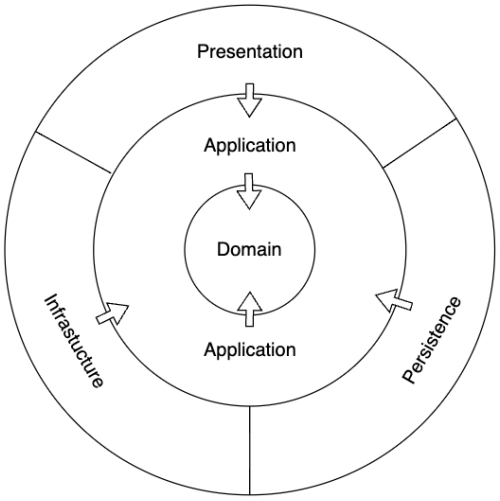

Рис. 3.1. Схема шарів

Усі елементи, що знаходяться у внутрішньому колі, повинні бути повністю ізольовані від зовнішніх. Внутрішнє коло не може мати жодних знань про зовнішні шари системи. Наприклад, імена функцій, класів, змінних чи інших іменованих елементів, оголошених у зовнішніх колах, не повинні використовуватися в коді, розташованому у внутрішніх колах.

Це стосується також форматів даних: формати, визначені у зовнішніх колах (особливо ті, що генеруються фреймворками), не повинні застосовуватися у внутрішніх. Ніякий елемент зовнішнього кола не повинен впливати на логіку чи структуру внутрішніх кіл.

Головні компоненти системи структуровані у вигляді кількох шарів (рис. 3.2), чітко розділених та інкапсульованих, з односторонніми залежностями між ними:

- Domain: Це ядро системи, фундамент, навколо якого будується весь проєкт. Domain є ізольованим від зовнішніх залежностей і містить лише сутності, перерахування, константи та, за необхідності, спеціальні винятки.
- Application: У поєднанні з Domain, Application формує ядро рішення, яке здатне функціонувати незалежно від зовнішніх рівнів. Цей рівень відповідає за створення сутностей, виконання бізнес-правил та повернення результатів для API. Application залежить лише від Domain.

– Infrastructure: Цей рівень забезпечує інтеграцію із зовнішніми системами, такими як надсилання електронних листів або виклики сторонніх API. Він реалізує інтерфейси Application і залежить виключно від нього.
– Persistence: Цей рівень також відповідає за зв’язок із зовнішніми системами, але спеціалізується на роботі з базами даних. Він залежить від Application та Domain і реалізує відповідну логіку доступу до даних.
– Presentation: Цей рівень служить інтерфейсом для взаємодії з зовнішнім світом. Він дозволяє клієнтам надсилати запити та отримувати результати у зручному форматі, наприклад через REST API, консольні програми чи графічний інтерфейс. Presentation залежить від Application і може взаємодіяти з Persistence. У цьому шарі відсутня бізнес-логіка, його завдання – перетворювати запити користувачів на форму, зрозумілу для Application, і повертати результати. Також він відповідальний за генерацію сторінок за допомогою HTML, CSS та JavaScript а також консольних утиліт.

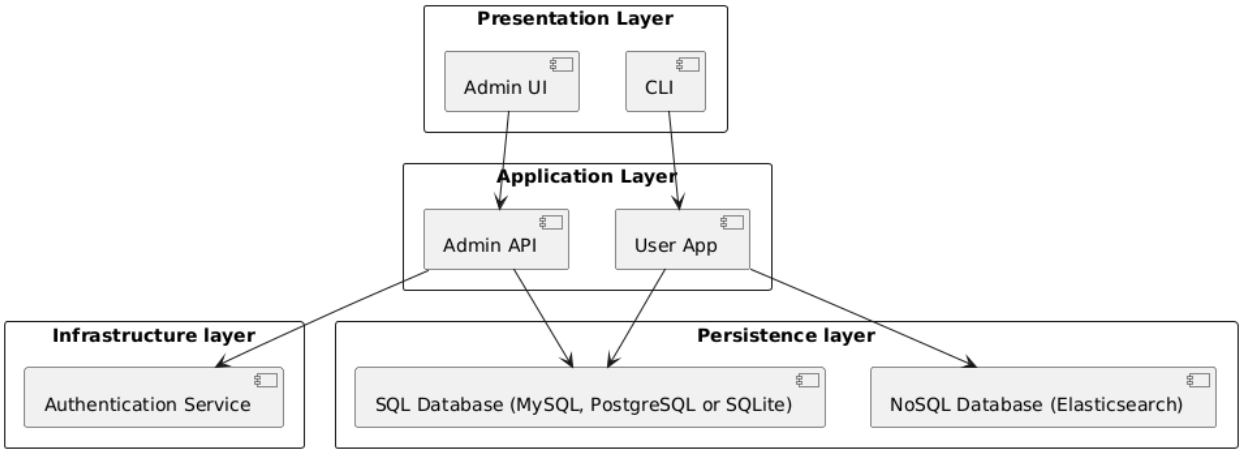

Рис. 3.2. Діаграма шарів системи.

Діаграма розгортання використовується для аналізу апаратної складової системи. Вона дозволяє оцінити необхідну апаратну конфігурацію для виконання окремих процесів системи та описати їх взаємодію як між собою, так і з іншими апаратними пристроями.

Вузол – це фізичний компонент системи, що володіє певними обчислювальними ресурсами. Прикладами вузлів можуть бути сервери, комп’ютери, пристрої для зберігання даних або хмарні платформи.

На діаграмі розгортання, окрім вузлів, відображаються також зв’язки між ними. Ці зв’язки представляють фізичні з’єднання між вузлами, а також залежності між вузлами та компонентами, які можуть бути додатково зображені на діаграмі. Фізичні з’єднання є різновидом асоціацій і позначаються лініями без стрілок. Наявність такої лінії вказує на необхідність створення фізичного каналу для обміну даними між відповідними вузлами. Тип з’єднання може бути деталізований за допомогою приміток, значень або обмежень.

Діаграми розгортання можуть бути представлені окремо або разом із діаграмами компонентів. Об’єднання цих діаграм доцільне для демонстрації того, які компоненти виконуються на конкретних вузлах.

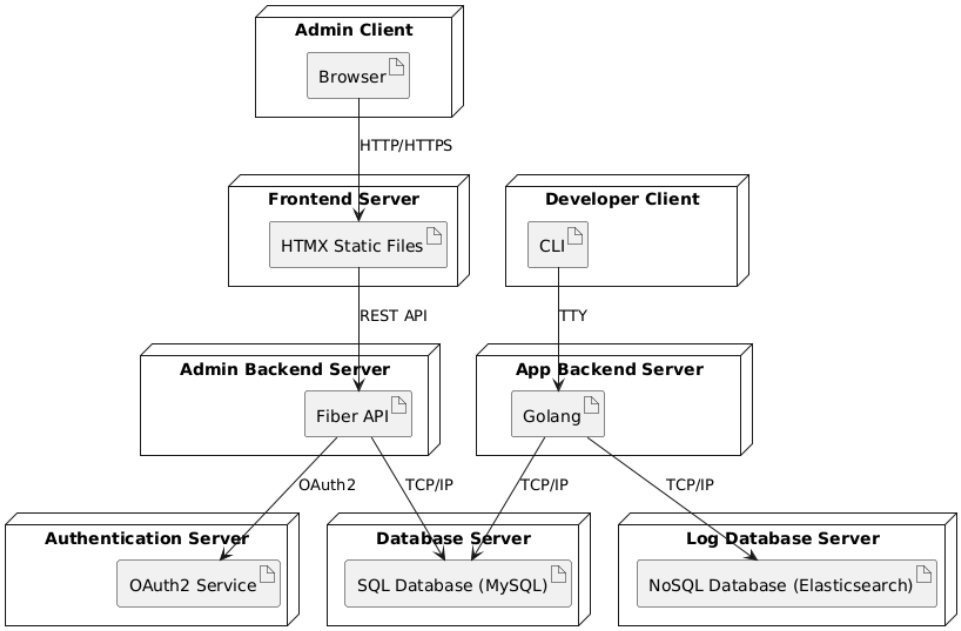

Рис. 3.3. Діаграма розгортання

Система складається з восьми основних вузлів (рис. 3.3):

- Admin Client: Браузери адміністраторів БД;
- Developer Client: Консольний рядок розробника;
- Frontend Server: Сервер для роздачі статичних файлів HTMX-застосунку.
- App Backend Server: Користувацька програма (будь-який програмний код написаний на Golang);
- Admin Backend Server: Сервер для виконання бізнес-логіки, обробки API-запитів (Fiber);

- Database Server: Сервер для зберігання даних (MySQL, PostgreSQL або SQLite база даних);
- Log Database Server: Сервер для зберігання логів (Elasticsearch база даних);
- Authentication Service: Сервіс автентифікації через OAuth2;

Frontend Server, Admin Backend Server, App Backend Server, Authenication Server, Database Server та Log Database Server можуть бути розгорнуті на різних віртуальних машинах в хмарі, що дозволяє масштабувати систему в залежності від попиту і забезпечити доступ до неї з будь-якого місця. Frontend server взаємодіє з компонентом Admin Backend Server що містить логіку роботи з адміністративною панелю системи, для забезпечення доступу до функціональності. Крім того, він може взаємодіяти з базою даних і сервісом авторизації. Компонент App Backend Server є будь яким користувацьми програмним кодом який в свою чергу використовує основну базу даних і базу даних для логів системи.

Адміністратори взаємодіють з системою через Admin Client за допомогою веб-браузера та артефакту index.html, який є веб-інтерфейсом, отриманим з вебсервера.

Розробники взаємодіють з системою через Admin Client (так само як і у випадку з адміністраторами) а також Developer Client за допомогою консольної утиліти.

Висновки до другого розділу

На основі детального аналізу вимог до програмного продукту було сформульовано технічне завдання, яке чітко визначає ключові функціональні та нефункціональні вимоги. Цей документ став основою для побудови системи, орієнтованої на задоволення потреб користувачів.

У процесі розробки було визначено два основних актори, для кожного з яких розроблено сценарії використання. Ці сценарії деталізують їхню взаємодію із системою, забезпечуючи розуміння користувацьких потреб і ролей.

Модель логічного рівня наочно відображає динаміку та структуру системи, а також взаємодію між її окремими компонентами. Для її створення було

використано кілька видів діаграм, включно з діаграмами активностей, станів, класів і послідовностей. У межах розробки цієї моделі активно застосовувалися патерни проєктування, що забезпечують гнучкість, масштабованість і відповідність принципам сучасного програмного забезпечення.

Фізична модель системи включає п’ять основних компонентів: вебсервер адміністративної панелі, сервер аутентифікації, сервер бази даних, сервер бази даних логів та користувацький програмний код. Ці компоненти працюють у тісній взаємодії для реалізації функціональних можливостей системи. Розробка фізичної моделі враховувала вимоги до розширюваності, адаптивності та високої продуктивності, що дозволяє системі ефективно відповідати сучасним викликам.

ВИСНОВКИ

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційна документація React [Електронний ресурс] – Режим доступу до ресурсу: https://react.dev/reference/react
2. Офіційна документація JavaScript [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript
3. Офіційна документація Node.js [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/docs/latest/api/
4. Офіційна документація Next.js [Електронний ресурс] – Режим доступу до ресурсу: https://nextjs.org/docs
5. Офіційна документація PostgreSQL [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/
6. Linear [Електронний ресурс] – Режим доступу до ресурсу: https://linear.app/
7. GitHub Projects [Електронний ресурс] – Режим доступу до ресурсу: https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects
8. Jira [Електронний ресурс] – Режим доступу до ресурсу: https://www.atlassian.com/software/jira