

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

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

«26» 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: **розробка бібліотеки для взаємодії з різними реляційними базами даних на мові програмування Golang та підтримкою вбудованої адміністративної панелі.**

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Савіцький Р.С.	17.02.2025р.	17.02.2025р.
2	Савіцький Р.С.	18.02.2025р.	18.02.2025р.
3	Савіцький Р.С.	20.02.2025р.	20.02.2025р.

РЕФЕРАТ

Пояснювальна записка до випускної кваліфікаційної роботи бакалавра складається зі вступу, трьох розділів, висновку та списку використаних джерел.

Текстова частина викладена на 57 сторінках друкованого тексту. Список використаних джерел містить 30 найменувань і займає 3 сторінки. В роботі наведено 53 рисунків та 13 таблиць. Загальний обсяг роботи – 72 сторінки.

У роботі поставлені основні завдання на розробку бібліотеки для взаємодії з реляційними базами даних на мові програмування Golang. Система включає модулі будівельника запитів, ядра системи, готових адаптерів баз даних та адміністративної панелі. Також було проведено детальний порівняльний аналіз аналогічних систем. Було обрано підхід модульного моноліту, яка працює в комбінації з чистою архітектурою. Наведені основні сценарії використання системи, детальна логіка роботи сценаріїв, які охоплюють міжмодульну взаємодію. Клієнтська частина адміністративної панелі розроблена за допомогою бібліотек HTMX, Templ та DaisyUI а серверна частина за допомогою фреймворку Fiber.

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

					ІПЗ.КР.Б.-121-25-ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розроб.		Шумський О.В.			Бібліотека для взаємодії з реляційними базами даних	Літ.	Арк.	Аркушів
							3	72
Керівник		Савіцький Р.С.				ФІКТ Гр. ВТ-21-1		
Н. контр.		Тичина О.П.						
Зав. каф.		Вакалюк Т.А.						

ABSTRACT

The explanatory note to the bachelor's final qualification work consists of an introduction, three sections, a conclusion and a list of sources used.

The text part is set out on 57 pages of printed text. The list of sources used contains 30 items and takes up 3 pages. The work contains 53 figures and 13 tables. The total volume of the work is 72 pages.

The work sets the main tasks for the development of a library for interaction with relational databases in the Golang programming language. The system includes modules for the query builder, the system core, ready-made database adapters and the administrative panel. A detailed comparative analysis of similar systems was also conducted. The modular monolith approach was chosen, which works in combination with a clean architecture. The main scenarios for using the system are presented, as well as the detailed logic of the scenarios that cover intermodule interaction. The client part of the administrative panel is developed using the HTMX, Templ and DaisyUI libraries, and the server part is developed using the Fiber framework.

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

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	6
ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ, ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ	9
1.1 Визначення та постановка задачі.....	9
1.2 Аналіз існуючого програмного забезпечення	10
1.3 Визначення архітектури ПЗ	17
Висновки до першого розділу.....	20
РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	21
2.1 Аналіз вимог до ПЗ.....	21
2.2 Визначення варіантів використання системи	22
2.3 Проектування структури та визначення основних алгоритмів.....	24
2.4 Визначення архітектури.....	34
Висновки до другого розділу	36
РОЗДІЛ 3. ДЕМОНСТРАЦІЯ РОБОТИ ТА ТЕСТУВАННЯ СИСТЕМИ	37
3.1 Встановлення залежностей та налаштування системи.....	37
3.2 Використання будівельника запитів	42
3.3 Використання адміністративної панелі	46
3.4 Інструкція з використання програми.....	50
Висновки до третього розділу	52
ВИСНОВКИ.....	53
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	55
ДОДАТКИ.....	58
Додаток А.	59

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення

БД – База даних

ORM – Object-Relational Mapping

JS – JavaScript

CRUD – Create Read Update Delete

GO – Golang

		Шумський О.В.			ПЗ.КР.Б.-121-25-ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

ВСТУП

Бібліотека для взаємодії з реляційними базами даних орієнтована на підвищення ефективності розробників та покращення безпеки за рахунок роботи з базами даних за допомогою об'єктно-орієнтованого підходу та можливості зручно використовувати адміністративну панель.

Актуальність теми зумовлена зростанням популярності мови програмування Golang та необхідністю в інструментах для покращення ефективності та безпеки при взаємодії з базами даних. Така бібліотека сприятиме покращенню розробницького досвіду при програмуванні проєктів малої та середньої складності. Також бібліотека надає можливість зручного перегляду та редагування даних для нетехнічних людей за допомогою адміністративної панелі.

Зокрема, БД відіграють важливу роль у програмуванні, адже мають велику швидкість, високу безпеку та популярність серед великих компаній [1]. Кожна реляційна БД має свої переваги та недоліки і для кожного додатку найкращою буде, та, яка задовільняє технічні вимоги [2]. Єдиний інтерфейс взаємодії з різними БД є популярним рішенням, адже зменшує складність роботи і кількість проблем при потребі заміни БД в додатку. Розробка бібліотеки, яка спрощує роботу з різними базами даних, відповідає потребам сучасного підходу до розробки додатків.

Крім того, наявні бібліотеки не мають підтримки зручних інструментів, які орієнтовані на спрощення життя для нетехнічних людей і для них є великою складністю за допомогою сторонніх інструментів переглядати та редагувати бази даних через перевантажений інтерфейс інструментів. Створення комплексного рішення з вбудованою підтримкою мінімалістичної панелі з базовим функціоналом допоможе у вирішенні проблеми.

Метою роботи є проєктування та реалізація бібліотеки для взаємодії з реляційними базами даних, що забезпечить простий спосіб будування складних запитів з урахуванням специфіки SQL-синтаксису різних БД, бездоганної типізації та інтеграції з мовою програмування Golang [3]

Завданнями випускної кваліфікаційної роботи бакалавра є:

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

- постановка основних задач;
- аналіз аналогів бібліотеки;
- визначення потрібної архітектури;
- вибір та обґрунтування засобів для реалізації системи;
- визначення варіанті використання бібліотеки;
- розробка необхідного функціоналу та реалізація алгоритмів для створення запитів;
- розробка UI/UX дизайну для частини адміністративної панелі бібліотеки;
- інтеграційне тестування розробленої бібліотеки.

Об’єкт дослідження: інформаційні технології для будування запитів, які допоможуть взаємодіяти з різними базами даних за допомогою спільного інтерфейсу взаємодії. Досліджується можливість використання адміністративної панелі як інструменту для підвищення ефективності розробки та способу взаємодії з БД для нетехнічних людей.

Предмет дослідження: процес розробки бібліотеки для роботи з різними базами даних.

За темою випускної кваліфікаційної роботи бакалавра було опубліковано тези на тему «Відмінності реляційних баз даних» [1], «ACID-властивості та рівні ізоляції транзакцій» [2].

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ, ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1 Визначення та постановка задачі

Розробити бібліотеку для підвищення ефективності роботи з базами даних, збільшити надійність написаного коду, а також легкість адміністрування. Бібліотека має складатися з деяких ключових програмних частин, а саме:

- будівельник запитів, який 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;
- базовий адаптер, який містить в собі готовий функціонал, який відповідає стандарту SQL. Він є основою для написання своїх імплементацій адаптерів;
- реалізації адаптерів для реляційних баз даних, які є готовими для використання розробниками. Логіка генерації SQL коду виконується шляхом адаптерів. Розробникам не обов'язково використовувати готові рішення, адже вони мають можливість написати самостійно;
- адміністративна панель дозволяє проводити роботу з базою даних у вебінтерфейс, яка знадобиться для адміністраторів баз даних і дає змогу швидко маніпулювати даними. Також панель буде корисною при локальній розробці додатка.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- провести аналіз предметної області та наявних рішень у сфері ORM систем;
- обґрунтувати вибір архітектури для розробки бібліотеки;
- визначити та обґрунтувати вибір інструментів та технологій для розв'язування типових проблем;
- розробити функціональні та нефункціональні вимоги до бібліотеки, а також визначити варіанти використання;
- спроектувати структуру класів системи та в деталях визначити як саме будуть відбуватись основні сценарії використання бібліотеки;

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

- розробка базового SQL адаптера;
- розробка готових імплементацій адаптерів для використання розробниками;
- написання логіки обробки даних;
- створення дизайну адміністративної панелі;
- розробка адміністративної панелі;
- мануальне і інтеграційне тестування.

Результатом реалізації поставленого завдання є бібліотека, яка містить основну логіку роботи з базами даних, готові адаптери для реляційних баз даних та адміністративну панель.

1.2 Аналіз існуючого програмного забезпечення

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

Таблиця 1.1

Перелік аналогів системи

Функціонал	GORM	Prisma Client Go	Ent	SQLC	Власний додаток
Простота використання	+	-	-	+	+
Наявність адміністративної панелі	-	+	-	-	+
Доступ до адміністративної панелі ззовні	-	-	-	-	+
Стилізація адміністративної панелі	-	±	-	-	+

Швидке виконання запитів	-	-	-	+	+
Висока масштабованість	+	+	+	-	-
Вбудовані міграції	+	+	+	-	-
Schema-first підхід	-	+	+	-	-
Підтримка GraphQL	-	+	+	-	-
Вибіркові колонки	+	±	+	+	+
Безпечність типів	-	+	+	+	-
Групування операцій	±	±	+	+	+

GORM – це найпопулярніша бібліотека для роботи з реляційними базами даних [4], що відома своєю зручністю та великим функціоналом. Серед різноманітних баз даних вона підтримує MySQL, PostgreSQL, SQLite, Microsoft SQL Server, TiDB та Clickhouse. Система самостійно покриває майже всі можливі сценарії роботи з базами даних такі як, автоматичні міграції, щоб синхронізувати моделі в програмному коді (рис. 1.1) з таблицями в БД [5], підтримка шардування БД, хуки (функції, які виконуються під час життєвого циклу запитів), можливість додавати свої серіалізатори, перевикористання частин запитів, вбудована підтримка підказок для оптимізаторів баз даних та підтримка сторонніх сервісів.

```

type User struct {
    gorm.Model
    Name      string
    CompanyRefer int
    Company    Company `gorm:"foreignKey:CompanyRefer"`
    // use CompanyRefer as foreign key
}

type Company struct {
    ID    int
    Name  string
}

```

Рис. 1.1 – Опис моделей у бібліотеці «GORM»

Система має зручний спосіб побудови запитів (рис. 1.2), який є захищеним від SQL-ін'єкцій, а також гнучким, адже користувач сам може писати всередині частини SQL коду, незалежно від наявних імплементованих методів. Слід зазначити що наявність великої кількості способів використання є недоліком, адже потребує великої кількості часу для ознайомлення з документацією і може додати в проєкт непотрібну складність.

```
// Get first matched record
db.Where("name = ?", "jinzhu").First(&user)
// SELECT * FROM users WHERE name = 'jinzhu' ORDER BY id LIMIT 1;

// Get all matched records
db.Where("name <> ?", "jinzhu").Find(&users)
// SELECT * FROM users WHERE name <> 'jinzhu';

// IN
db.Where("name IN ?", []string{"jinzhu", "jinzhu 2"}).Find(&users)
// SELECT * FROM users WHERE name IN ('jinzhu','jinzhu 2');

// LIKE
db.Where("name LIKE ?", "%jin%").Find(&users)
// SELECT * FROM users WHERE name LIKE '%jin%';

// AND
db.Where("name = ? AND age >= ?", "jinzhu", "22").Find(&users)
// SELECT * FROM users WHERE name = 'jinzhu' AND age >= 22;

// Time
db.Where("updated_at > ?", lastWeek).Find(&users)
// SELECT * FROM users WHERE updated_at > '2000-01-01 00:00:00';

// BETWEEN
db.Where("created_at BETWEEN ? AND ?", lastWeek, today).Find(&users)
// SELECT * FROM users WHERE created_at BETWEEN '2000-01-01 00:00:00' AND '2000-01-01 00:00:00';
```

Рис. 1.2 – Будування запитів в бібліотеці «GORM»

Також це не завжди хороший вибір у великих проєктах, адже низька швидкодія може стати причиною перенесення частини запитів на користь написання свого чистого SQL коду, що порушує саму концепцію системи та створює неконсистентність кодової бази.

Дана ORM система є хорошим вибором для проєкту середньої та високої складності, де система є основним інструментом для роботи з базою даних, а не допоміжним і швидкість розробки та простота підтримки має більший пріоритет, ніж швидкість виконання запитів.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Prisma Client Go – це адаптація популярної бібліотеки на JavaScript, яка має дуже схожий інтерфейс взаємодії з БД [6]. Особливістю бібліотеки є опис моделей у власному форматі файлів (рис. 1.3), де знаходяться всі моделі системи та конфігурації підключення. Цей спосіб робить можливим побудову бази даних підходами Database First, Code First та Model First, що робить систему гнучким інструментом і може використовуватись як основним інструментом, так і доповненням, яке не втручається в схему бази даних. Даний формат підтримується більшістю редакторів коду, але має великі проблеми з поділом схеми бази даних на окремі файли. При великій кількості таблиць в схемі бази даних суттєво погіршує розробницький досвід. Концепція кодогенерації через консольні команди може викликати проблеми з типами, адже проводити генерацію потрібно кожного разу для роботи додатка.

```
datasource db {
  // could be postgresql or mysql
  provider = "sqlite"
  url      = "file:dev.db"
}

generator db {
  provider = "go run github.com/steebchen/prisma-client-go"
}

model Post {
  id          String @default(cuid()) @id
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
  title       String
  published    Boolean
  desc        String?
}
```

Рис. 1.3 – Опис таблиць в бібліотеці «Prisma Client Go»

Аналогічно до адаптації на JavaScript, Prisma Client Go має всі можливості консольного інтерфейсу, такі як, зручне відкриття адміністративної панелі (рис. 1.4) та роботи з міграціями. Система має простий інтерфейс взаємодії з базою даних, але в цей час для роботи вимагає наявності runtime-бібліотеку, яка має погану швидкодію, великий розмір та проблеми з запуском в різних середовищах, особливо це викликає проблеми в serverless-середовищах, де час “холодного” старту та чіткий максимальний допустимий розмір є ключовими факторами. Також слід зазначити що активна підтримка адаптації для Golang завершилась і система буде отримувати тільки критичні оновлення без нового функціоналу.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Дана система може бути хорошим вибором в малих нішевих проєктах, де вже наявна база даних або у вимогах присутня проста адміністративна панель для зміни даних.

The screenshot shows the Prisma Studio web interface in a browser window. The URL is `prisma.studio/preview?project=chinook`. The interface has a top navigation bar with tabs for Customer, Album, Playlist, Artist, Genre, and Invoice. Below this is a toolbar with buttons for Filters, Fields, All, Showing (100 of 412), and Add record. The main area displays a table with the following columns: BillingAddress A?, BillingCity A?, BillingCountry A?, BillingPostalCode A?, BillingState A?, and Customer. The table contains 20 rows of data, including addresses from Stuttgart, Oslo, Brussels, Edmonton, Boston, Frankfurt, Berlin, Paris, Bordeaux, Dublin, London, and Mountain View.

BillingAddress A?	BillingCity A?	BillingCountry A?	BillingPostalCode A?	BillingState A?	Customer
Theodor-Heuss-Straße	Stuttgart	Germany	70174	null	2
Ullevålsveien 14	Oslo	Norway	0171	null	4
Grétrystraat 63	Brussels	Belgium	1000	null	8
8210 111 ST NW	Edmonton	Canada	T6G 2C7	AB	14
69 Salem Street	Boston	USA	2113	MA	23
Berger Straße 10	Frankfurt	Germany	60316	null	37
Barbarossastraße 19	Berlin	Germany	10779	null	38
8, Rue Hanovre	Paris	France	75002	null	40
9, Place Louis Barthou	Bordeaux	France	33000	null	42
3 Chatham Street	Dublin	Ireland	null	Dublin	46
202 Hoxton Street	London	United Kingdom	N1 5LH	null	52
Theodor-Heuss-Straße	Stuttgart	Germany	70174	null	2
1600 Amphitheatre Par	Mountain View	USA	94043-1351	CA	16
1 Microsoft Way	Redmond	USA	98052-8300	WA	17
1 Infinite Loop	Cupertino	USA	95014	CA	19
801 W 4th Street	Reno	USA	89503	NV	21
319 N. Frances Street	Madison	USA	53703	WI	25

Рис. 1.4 – Адміністративна панель в бібліотеці «Prisma Client Go»

Ent – це фреймворк від компанії Meta, який дозволяє зручно взаємодіяти з БД у вигляді об’єктів та структур, використовуючи статично типізований інтерфейс [7]. В основі фреймворку застосовані концепції кодогенерації на основі Golang функцій (рис. 1.5) та взаємодії з БД як з графом, де вершини це таблиці, а ребра це зв’язки між ними. Кодогенерація збільшує дискомфорт при розробці, але необхідна, щоб покращити швидкодію [8] шляхом перенесень обчислень для генерації SQL-коду на сторону комп’ютерів розробників замість використання вбудованого пакета “reflect” і обчислення при кожному запиті в базу даних. Будівельник запитів є потужним інструментом, який підтримує запити високої складності, а також хорошу інтеграцію з GraphQL [9], gRPC, OpenAI та з різними плагінами.

```
func (Todo) Fields() []ent.Field {
    return []ent.Field{
        field.Text("text").
            NotEmpty(),
        field.Time("created_at").
            Default(time.Now).
            Immutable(),
        field.Enum("status").
            NamedValues(
                "InProgress", "IN_PROGRESS",
                "Completed", "COMPLETED",
            ).
            Default("IN_PROGRESS"),
        field.Int("priority").
            Default(0),
    }
}
```

After adding these fields, we need to run the code-generation as before:

```
go generate ./ent
```

Рис. 1.5 – Опис таблиць у фреймворку «Ent»

Слід зазначити, що фреймворк має дуже високу складність і надмірну кількість операцій при виконанні простих запитів в БД. Крім того, механізм міграцій має недоліки при застосуванні в складних або чутливих схемах. Також можуть виникнути незручності при роботі з виконанням власних SQL-запитів (рис. 1.6), коли потрібний функціонал занадто специфічний для бази даних і не підтримується у фреймворку.

Даний фреймворк є дуже хорошим вибором для великих проєктів, де ключовими пріоритетами є підтримка складних запитів, безпека та швидкість виконання запитів.

SQLC – це бібліотека, яка має нестандартний спосіб роботи з базою даних, оснований на генерації Golang коду базуючись на готовій схемі бази даних та SQL-кодів запитів [10]. Вся генерація відбувається через консольний інтерфейс і може повідомляти про проблеми з SQL-запитами ще на стадії генерації (рис. 1.7). Згенерований код має якісну типізацію і високу продуктивність [11], адже він повністю контролюється розробником і має вбудоване покращення захисту від SQL-ін'єкцій. Такий підхід є ідеальним для людей, які хочуть контролювати все самостійно без великої кількості прихованих абстракцій.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

schema.sql :

```
CREATE TABLE records (  
  id SERIAL PRIMARY KEY,  
  counter INT NOT NULL  
);
```

query.sql

```
-- name: GetRecord :one  
SELECT * FROM records  
WHERE id = $1;  
  
-- name: UpdateRecord :exec  
UPDATE records SET counter = $2  
WHERE id = $1;
```

Рис. 1.7 – Опис SQL-запитів у бібліотеці «SQLC»

З мінусів можна зазначити те, що бібліотека не може бути використана як повноцінний інструмент, адже вона не контролює схему БД і це може викликати проблеми при зміні схеми без урахування зміни SQL-запитів.

Дана бібліотека є пріоритетним рішенням для заміни чистих SQL-запитів на типізовані та безпечні варіації (рис. 1.8), а також для проєктів де важливий максимальний контроль та швидкодія запитів.

```
func bumpCounter(ctx context.Context, db *sql.DB, queries *tutorial.Queries, id int32) error {  
    tx, err := db.Begin()  
    if err != nil {  
        return err  
    }  
    defer tx.Rollback()  
    qtx := queries.WithTx(tx)  
    r, err := qtx.GetRecord(ctx, id)  
    if err != nil {  
        return err  
    }  
    if err := qtx.UpdateRecord(ctx, tutorial.UpdateRecordParams{  
        ID:      r.ID,  
        Counter: r.Counter + 1,  
    }); err != nil {  
        return err  
    }  
    return tx.Commit()  
}
```

Рис. 1.8 – Робота з запитамі у бібліотеці «SQLC»

Аналіз аналогічних платформ допоміг визначити сильні та слабкі сторони наявних 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

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

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

Для розробки ORM системи було обрано підхід модульного моноліту (рис. 1.9), де різні частини можуть імпортувати публічний функціонал інших модулів. Такий підхід дає змогу розділити функціонал на логічні частини для покращення зручності використання бібліотеки та спростити процес розробки. Прикладом цього є різні адаптери баз даних. Користувач може обрати один або декілька адаптерів. Підхід створює високу зв'язність між функціоналом всередині компонентів одного модуля і зменшує зв'язність між різними модулями, для того, щоб зміни в одному модулі не викликали проблем в інших модулях. Також цей підхід зменшує дублювання коду до мінімуму, оскільки спільний код можна переносити в окремі допоміжні модулі та використовувати в різних місцях. Системою передбачена велика варіативність клієнтів. Вебклієнт повинен бути сумісним з різноманітними веббраузерами, такими як Google Chrome, Firefox та Safari та працювати на різних пристроях та операційних системах, забезпечуючи надійний і зручний доступ до функціоналу.

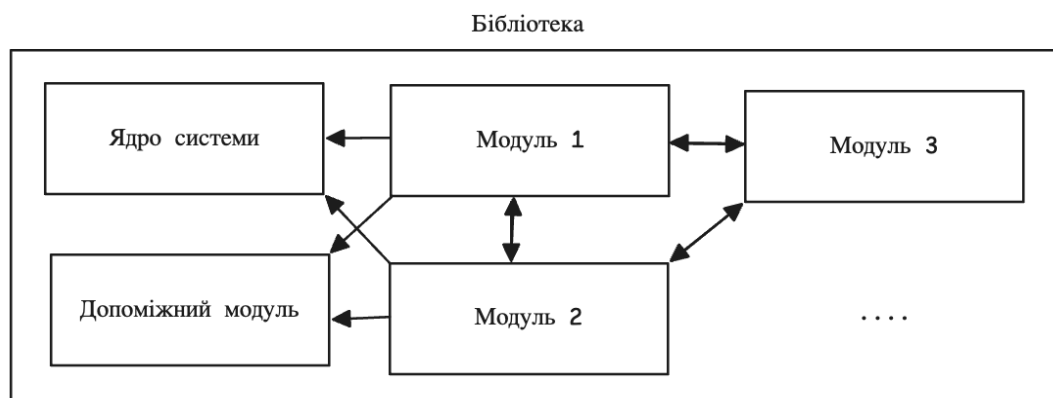


Рис. 1.9 – Схема роботи модульного моноліту

Інтерфейс користувача повинен бути зручним та інтуїтивно зрозумілим, з урахуванням найкращих UI/UX практик. Під час розробки важливо забезпечити

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

адаптивність інтерфейсу при використанні на пристроях з різними розмірами екрана.

При розробці було використано декілька інструментів для ефективної та зручної роботи. В якості основних інтегрованих середовищ розробки використовувалися Webstorm та Goland. Середовища розробки допомагають в складних проєктах завдяки індексації всієї кодової бази та можуть виявити критичні помилки ще до запуску коду.

Для тестування REST API було обрано Postman. Він має весь необхідний функціонал для тестування серверної частини, а також має можливості для автоматизації команд, які необхідні для коректної роботи системи. Він має зручний графічний інтерфейс, який прискорює тестування.

Після аналізу таких бібліотек для клієнтської частини (табл. 1.2), як React [12], Alpine.js та Templ [13] з HTMX [14], була обрана комбінація HTMX та Templ завдяки підтримці серверного рендерингу, що дозволяє не використовувати додатковий вебсервер. Для стилізації обрана бібліотека DaisyUI [15]

Функціоналу Templ достатньо для відмальовування компонентів інтерфейсу адміністративної панелі, а HTMX допоможе зі спрощенням роботи з реактивністю.

Таблиця 1.2

Порівняння бібліотек для створення вебінтерфейсів

Критерій	Templ з HTMX	React	Alpine.js
Складність	Низька	Висока	Низька
Підхід рендерингу	Серверний (HTML over the wire)	Клієнтський (Virtual DOM)	Клієнтський (нативний DOM)
Інтерактивність	через AJAX/HTML	через компоненти	через директиви
Зв'язок із сервером	Вбудований (через атрибути)	Потрібен додатковий код (fetch)	Потрібен додатковий код (fetch)

Продовження таблиці 1.2

Використання з Golang	Ідеально для серверного рендерингу	Потрібен окремий API (REST/GraphQL)	Потрібен окремий API (REST/GraphQL)
Розмір бібліотеки	~10 KB	~40 KB (React) + ~5 KB (ReactDOM)	~10 KB
Крива навчання	Низька	Середня	Низька

Для серверної частини адміністративної панелі були проаналізовані найпопулярніші фреймворки на Golang (табл. 1.3), такі як Fiber [16], Chi [17] та Echo [18]. Для потреб проєкт було обрано Fiber. Він має чудову продуктивність та вбудовану підтримку різноманітних технологій.

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

Таблиця 1.3

Порівняння фреймворків для Golang

Критерій	Fiber	Chi	Echo
Продуктивність	Дуже висока (побудований на fasthttp)	Висока (базується на net/http)	Висока (базується на net/http)
Підтримка WebSockets	Так	Ні	Так
Підтримка шаблонів	Так	Ні	Так
Готові рішення	Багато вбудованих функцій	Мінімалістичний, потребує додаткових бібліотек	Багато вбудованих функцій

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Вбудована Підтримка JWT	Так	Ні	Так
Документація	Детальна та зрозуміла	Хороша, але менш структурована	Хороша, але менш детальна
Підтримка контексту	Власна реалізація	Через стандартний context.Context	Через стандартний context.Context

Висновки до першого розділу

У першому розділі було проведено аналіз наявних аналогів, обґрунтовано вибір інструментів, сформульовані основні завдання та визначено архітектуру програмного забезпечення. Детальний аналіз аналогів дав можливість виявити всі необхідні функціональні та нефункціональні вимоги до системи. Для реалізації бібліотеки було обрано підхід модульного моноліту задля підтримки масштабованості та зручності розробки. Для клієнтської частини адміністративної панелі обрані HTMX та Templ, для зручної розробки простих вебінтерфейсів. Для серверної частини адміністративної панелі обраний фреймворк Fiber, який покриває вимоги обробки запитів.

Загалом, обрана архітектура та технології забезпечують високу надійність, можливість розширення функціоналу бібліотеки, швидку розробку та високу продуктивність системи.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Аналіз вимог до ПЗ

Розроблюване програмне забезпечення орієнтоване на розробників і адміністраторів баз даних. Мета – спростити роботу з базами даних, пришвидшити написання коду та забезпечити базовий рівень безпеки. Система дозволить взаємодіяти з даними через графічний інтерфейс, мінімізуючи необхідність ручного написання SQL-запитів.

На етапі проектування передбачається підтримка роботи з найпоширенішими реляційними базами даних, зокрема PostgreSQL [19], MySQL [20], SQLite [21] та MS SQL [22]. Кожен драйвер бази даних функціонуватиме незалежно, що спрощує підключення нових джерел даних без впливу на вже підключені.

Передбачається створення графічного інтерфейсу, через який адміністратори зможуть виконувати типові операції з базами даних – переглядати записи, редагувати їх та формувати запити без необхідності писати SQL-код вручну. Таким чином, знижується поріг входу для користувачів, які не мають глибоких знань у сфері програмування або не володіють SQL на належному рівні.

Користувач системи зможе ефективно працювати з моделями, створювати, змінювати та видаляти записи як через GUI, так і за допомогою вбудованого конструктора запитів. Система підтримуватиме базові типи зв'язків між моделями, зокрема один-до-одного, багато-до-одного та один-до-багатьох. Інтерфейс має бути зручним і зрозумілим, з можливістю обрати тему оформлення відповідно до уподобань користувача.

У проєкті передбачено підтримку різних драйверів баз даних, а також можливість розширення функціоналу за рахунок створення користувацьких адаптерів. Для запобігання помилкам і атакам реалізовуватимуться механізми захисту від SQL-ін'єкцій та необережного виконання запитів без умов, що могли б призвести до масових змін у базі.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Очікується, що початкове ознайомлення з ORM займе не більше 1–2 годин. Система повинна бути простою у використанні та надійною, з захистом від критичних помилок користувача. У плані продуктивності планується забезпечити стабільну роботу з мінімум 100 одночасними користувачами графічного інтерфейсу та швидке виконання простих запитів – не довше 500 мілісекунд.

Система проектується з урахуванням можливості масштабування. Передбачається підтримка великих обсягів даних до мільйонів записів без втрати продуктивності. Окрім того, користувачі зможуть створювати адаптери для додаткових СУБД, розширюючи сумісність із новими системами.

2.2 Визначення варіантів використання системи

Аналіз вимог користувачів дозволив визначити основні варіанти використання бібліотеки. На рисунку 2.1 наведено діаграму варіантів використання системи.

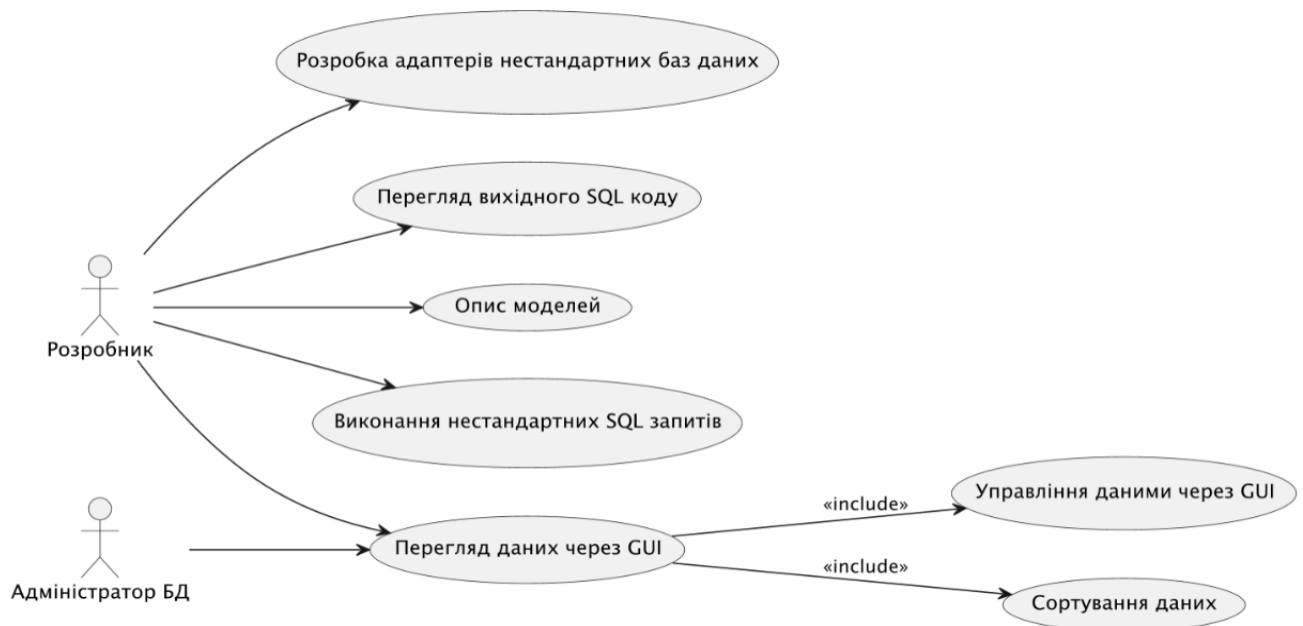


Рис. 2.1 – Діаграма варіантів використання бібліотеки

У ORM-системі розрізняють двох ключових акторів: розробник та адміністратор бази даних. Короткий опис акторів представлено в таблицях 2.1-2.8. Слід зазначити, що адміністратор може тільки переглядати та редагувати дані в таблицях, а також сортувати. Натомість розробник може використовувати бібліотеку в програмному коді задля виконання запитів до бази даних. Також

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

розробник має доступ до всього функціоналу адміністратора при локальній розробці додатка.

Таблиця 2.1

Виявлення акторів

Актор	Короткий опис
Адміністратор	Переглядає і редагує дані
Розробник	Використовує бібліотеку як інструмент у розробці додатка. Описує таблиці, зв'язки між ними та виконує SQL запити

Таблиця 2.2

Опис прецеденту “Перегляд даних через GUI”

Назва	Перегляд даних через GUI
Опис	Цей варіант дозволяє переглядати таблиці та їх дані в зручному форматі.
Виконавці	Адміністратор / Розробник

Таблиця 2.3

Опис прецеденту “Управління даними через GUI”

Назва	Управління даними через GUI
Опис	Варіант дозволяє редагувати дані таблиць.
Виконавці	Адміністратор / Розробник

Таблиця 2.4

Опис прецеденту “Сортування даних”

Назва	Сортування даних
Опис	Варіант дозволяє сортувати дані в таблицях
Виконавці	Адміністратор / Розробник

Таблиця 2.5

Опис прецеденту “Розробка адаптерів нестандартних баз даних”

Назва	Розробка адаптерів нестандартних баз даних
Опис	Варіант дозволяє створити власні адаптери при умові якщо ні один з готових не підходить під поточну базу даних
Виконавці	Розробник

Таблиця 2.6

Опис прецеденту “Перегляд вихідного SQL коду”

Назва	Перегляд вихідного SQL коду
Опис	Цей варіант дозволяє переглядати згенерований SQL код
Виконавці	Розробник

Таблиця 2.7

Опис прецеденту “Опис моделей”

Назва	Опис моделей
Опис	Цей варіант дозволяє описувати структури відносно полів в базі даних
Виконавці	Розробник

Таблиця 2.8

Опис прецеденту “Виконання нестандартних SQL запитів”

Назва	Виконання нестандартних SQL запитів
Опис	Цей варіант дозволяє виконувати нестандартні запити, які не є передбаченими в ORM системі
Виконавці	Розробник

2.3 Проєктування структури та визначення основних алгоритмів

Бібліотека складається з чотирьох головних модулів: “adapters”, “core”, “orm” та “studio”. Модулі “adapters”, “orm” та “studio” майже повністю незалежні між

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

собою та мають залежність від ядра системи (модуль “core”). Виключенням є тільки модуль “studio”, який має залежності від “orm”.

Модуль “core” містить необхідні інтерфейси для роботи всієї системи. Також тут знаходиться сукупність “CoreUtils”, яка містить необхідні допоміжні методи для всіх інших модулів. На рисунку 2.2 можна побачити умовний зв'язок композиції між інтерфейсом “Entity” та структурою Model. При цьому те, що мова програмування Golang не дає можливості вказати те, що всі структури, які імплементують інтерфейс “Entity”, мають містити структуру “Model”, програмне забезпечення повинно розроблятися з урахуванням цього правила. Також існує залежність між інтерфейсом “DbAdapter” та структурою “ORMConfig”. Кожен адаптер, який імплементує даний інтерфейс, повинен мати методи для роботи з “ORMConfig”.

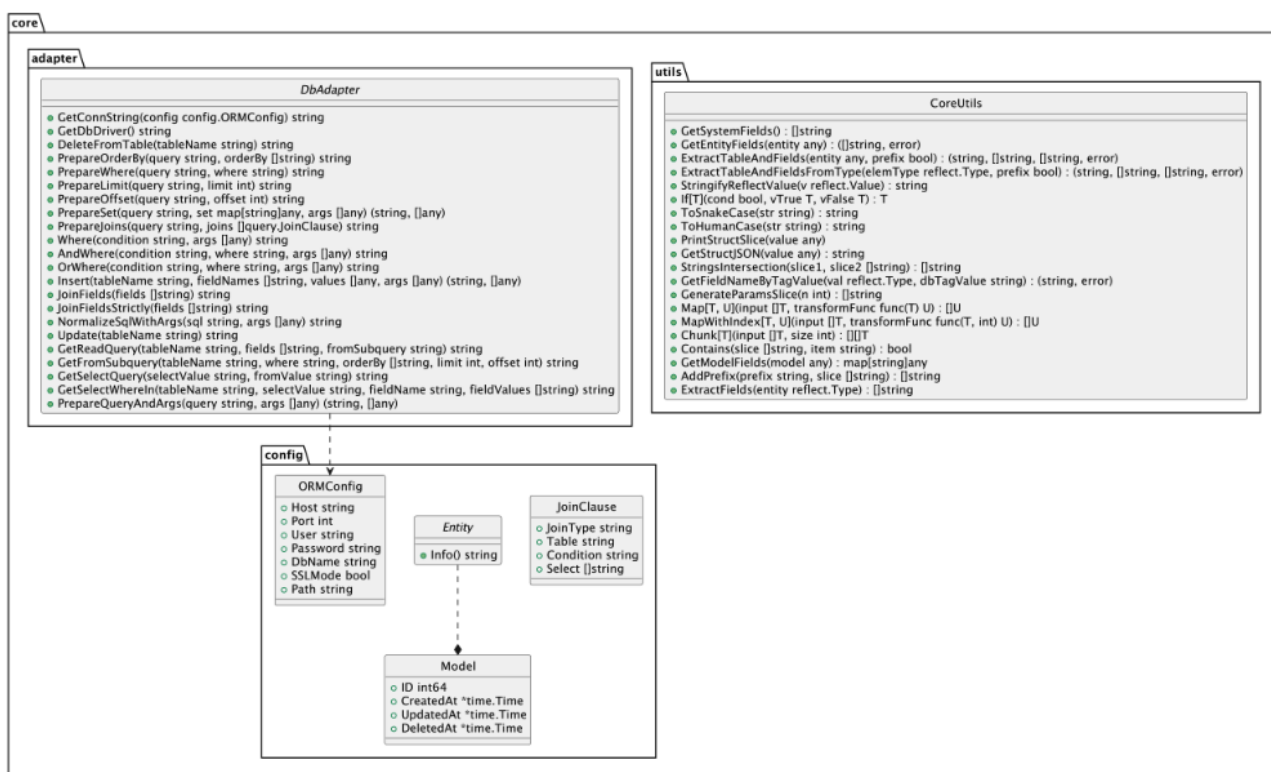


Рис. 2.2 – Діаграма класів модуля “core”

Модуль “adapters” (рис. 2.3) представляє сукупність готових адаптерів, які умовно наслідуються від абстракції “AdapterBase”.

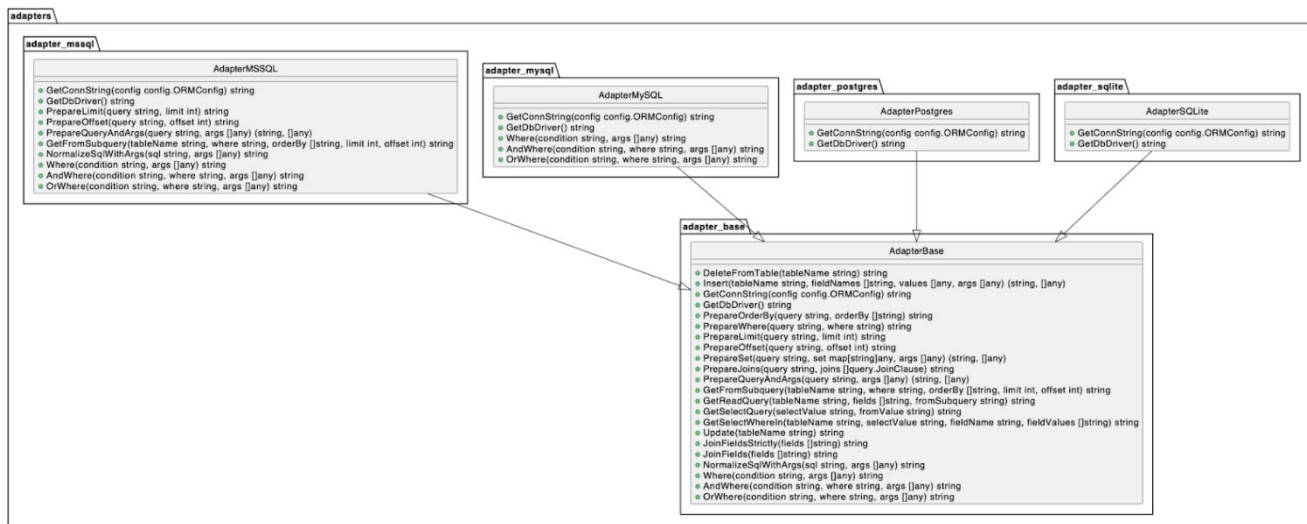


Рис. 2.3 – Діаграма класів модуля “adapters”

Насправді з урахуванням специфіки мови програмування, наслідування та поліморфізм не може бути реалізований в класичному вигляді, тому “AdapterBase” – це умовна сукупність з усіма базовими функціями для генерації SQL-коду, а кожен з адаптерів містить всі методи для імплементації інтерфейсу “DbAdapter” з модуля “core”, частина з яких береться з сукупності “AdapterBase”

Модуль “orm” реалізовує будівельник запитів, який необхідний для будування запитів. На рисунку 2.4 можна побачити зв’язок між “QueryBuilder” та “DbClient” для того, щоб у методах існував доступ до поточного адаптера, який міститься у клієнті БД. Також можна побачити зв’язок “QueryBuilder” до себе. При будуванні запиту використовується концепція методового ланцюгування, щоб при використанні деяких методів можна було отримати той же видозмінений об’єкт структури “QueryBuilder”.

		Шумський О.В.							Арк.
		Савицький Р.С.							
Змн.	Арк.	№ докум.	Підпис	Дата					

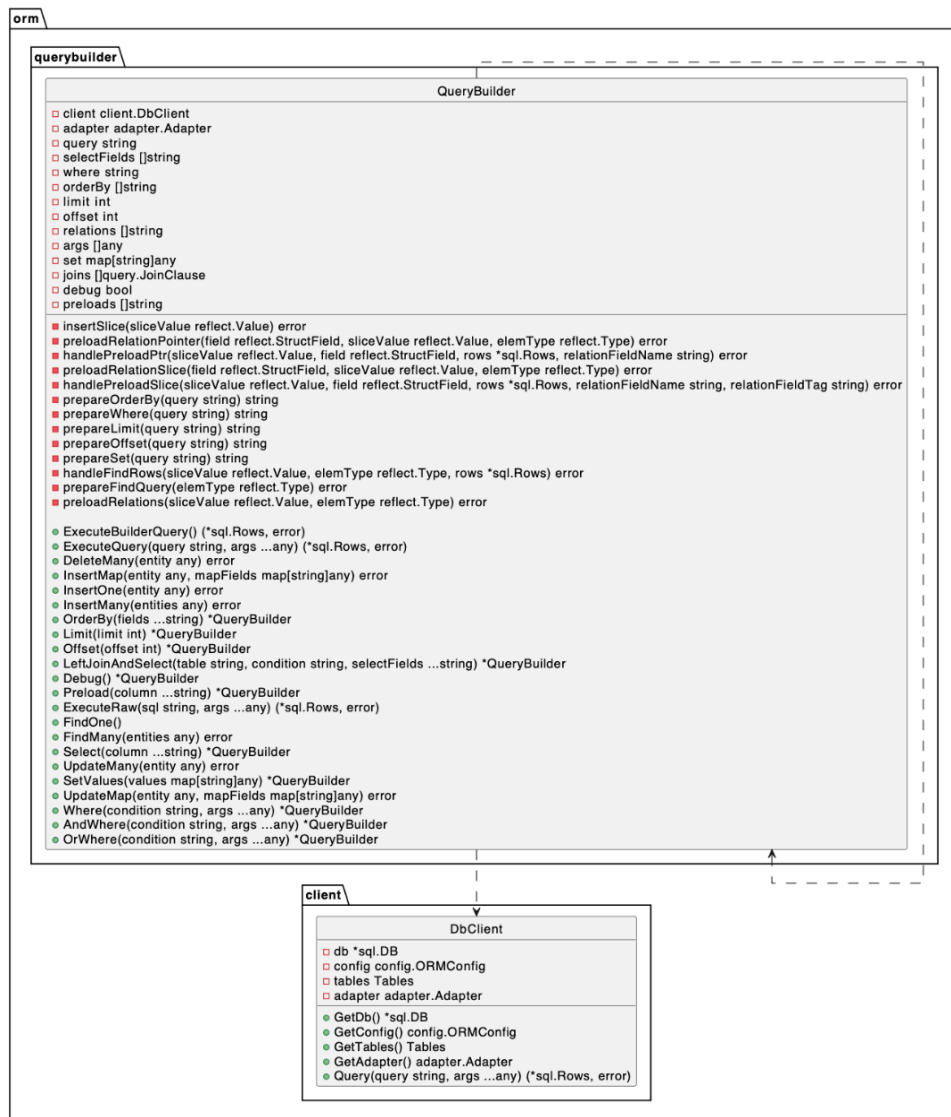


Рис. 2.4 – Діаграма класів модуля “orm”

Модуль “studio” відповідає за адміністративну панель для БД. На рисунку 2.5 можна побачити багато сутностей, які насправді є TempI компонентами, які за допомогою кодогенерації переводяться у Golang функції. Розподіл на пакети потрібен для групування компонентів по області використання та можливості поділу перевикористовувальних частин від конкретних. Пакет “layout” містить компоненти для формування структури сторінок і розділ інтерфейсу на логічні блоки. Пакет “recordForm” містить компоненти готових форм з усіма потрібними полями. Пакет “components” містить готові компоненти, які потрібні для взаємодії користувача з адміністративною панеллю, наприклад “AddRecord” потрібен для відкриття модального вікна з полями для додавання запису в таблицю БД. Пакет “icons” містить компоненти зі всіма необхідними іконками для інтерфейсу. Іконки

широко використовуються у багатьох різних компонентах, тому для покращення візуалізації стрілки не показані на діаграмі класів. Пакет “settings” містить сторінку налаштувань та її дочірні компоненти.

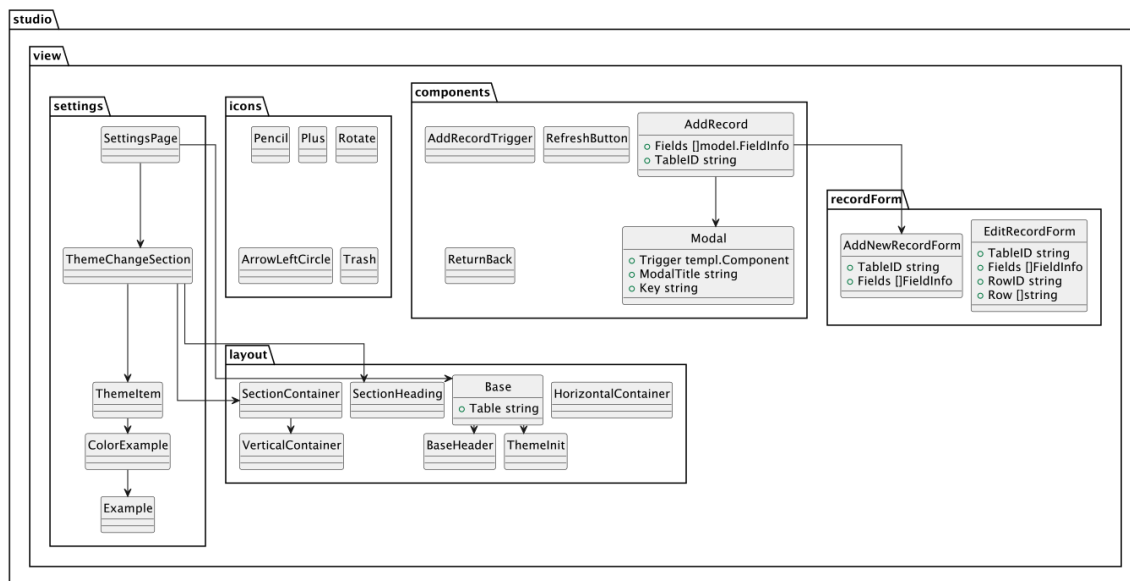


Рис. 2.5 – Діаграма класів модуля “studio”, без частини таблиць

Пакет “tables” знаходиться всередині модуля “studio” (рис. 2.6.), як і пакети на попередній діаграмі, але для зручного перегляду винесений на інше представлення. Пакет містить сторінки для роботи з таблицями БД, а також необхідні дочірні компоненти.

До створеної діаграми класів було застосовано патерни [23] Builder та Adapter.

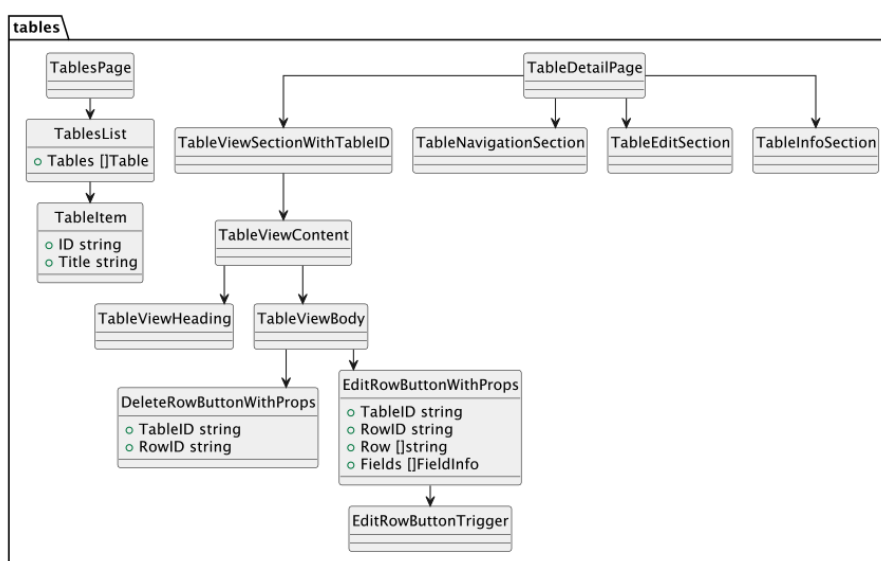


Рис. 2.6 – Діаграма класів пакету таблиць у модулі “studio”

Builder – це патерн (рис. 2.7), який використовується для створення об'єктів за допомогою поетапного будування різних варіацій об'єктів. У випадку структури QueryBuilder цей принцип застосовується для будування SQL-запитів. Ключова ідея полягає в тому, що кожен метод структури QueryBuilder повертає сам об'єкт QueryBuilder і завдяки цьому забезпечується можливість виклику методів у ланцюгу (method chaining). Наприклад, виклики builder.Where().OrderBy().FindMany() виконуються як одна послідовність і дозволяють зручно та гнучко будувати складні запити. Ця концепція є основою в мінімізації дублювання коду і уникання можливих помилок під час написання запитів вручну, а також забезпечення більшої читабельності коду, бо такий підхід спрощує додавання умов, фільтрів, сортування чи об'єднань.

```
var users []entities.User
err := querybuilder.CreateQueryBuilder(client).
    Preload( column...: "orders").
    Preload( column...: "role").
    FindMany(&users)
```

Рис. 2.7 – Приклад використання патерну “Builder”

Adapter – це патерн проєктування, який дає можливість сумісної роботи між структурами та спільним інтерфейсом даючи можливість замінити їх між собою без потреби робити зміни в коді. В цьому випадку реалізація виконана за допомогою структур AdapterPostgres, AdapterSQLite, AdapterMSSQL та AdapterMySQL. Кожен з яких потрібен для роботи з відповідною базою даних: PostgreSQL, SQLite, MS SQL та MySQL. Головною частиною підходу є спільний інтерфейс Adapter, який зазначає стандартний набір методів для генерації SQL-запитів. Інтерфейс виконує роль контракту, який зобов'язує всі адаптери мати необхідні методи, для того, щоб їх можна було використати в місцях, де параметром є інтерфейс Adapter. На рисунку 2.8 зазначено спільний інтерфейс, який імплементує кожен з адаптерів, враховуючи специфіку роботи конкретної бази даних. Завдяки такій абстракції бібліотека може працювати з будь-якою базою даних через універсальний інтерфейс. Це значно покращує гнучкість системи, спрощує підтримку, покращує розширення функціоналу та забезпечує чітке розділення відповідальностей.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

type DbAdapter interface { 4 usages 4 implementations  Shumsky Alexander
    GetConnString(config config.ORMConfig) string 4 implementations
    GetDbDriver() string 4 implementations
    DeleteFromTable(tableName string) string 4 implementations
    PrepareOrderBy(query string, orderBy []string) string 4 implementations
    PrepareWhere(query string, where string) string 4 implementations

```

Рис. 2.8 – Інтерфейс адаптера бази даних

Singleton – це патерн, який дозволяє ініціалізувати екземпляр певного класу один раз і постійно його перевикористовувати. Цей патерн дозволяє пришвидшити роботу застосунку за рахунок ініціалізації підключення до бази даних один раз і взаємодії з ним через клієнт БД замість ініціалізації при кожному запиті. Приклад реалізації можна побачити на рисунку 2.9 в частині відкриття підключення до БД через “sql.Open()” [24].

Factory method – це патерн проєктування, який дозволяє делегувати логіку створення об’єкта іншим класам, тим самим зменшуючи залежність між створенням бізнес-логіки і конкретною реалізацією класу. Приклад можна побачити на рисунку 2.9.

```

func CreateClient(config config.ORMConfig, currentAdapter adapter.DbAdapter) DbClient {
    connStr := currentAdapter.GetConnString(config)
    db, err := sql.Open(currentAdapter.GetDbDriver(), connStr)
    if err != nil {
        log.Fatal(err)
    }

    return DbClient{
        db:      db,
        config:  config,
        adapter: currentAdapter,
    }
}

```

Рис. 2.9 – Приклад використання патернів “Factory method” та “Singleton”

Розглянемо метод UpdateMany (рис. 2.10), який відповідає за масове оновлення даних. Цей метод використовується як завершальний етап побудови запиту через QueryBuilder. Задля надійності системи перед виконанням запиту виконуються різні перевірки. Передана структура, яка описує поля бази даних, повинна відповідати правильному типу даних. Також задля запобігання типових проблем недбалості розробників, існують перевірки нових даних для записів в базі

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

даних, а також вказаної умови. Без умови запит оновить усі записи в таблиці, якщо користувач дійсно хоче оновити всі записи, він зобов'язаний вказати умову, яка буде виконуватись завжди.

```
// UpdateMany initializes an UPDATE query for the specified entity.
func (qb *QueryBuilder) UpdateMany(entity any) error { 1 usage  Shumsky Alexander
    tableName, _, _, err := utils.ExtractTableAndFields(entity, prefix: true)
    if err != nil { return err }

    if len(qb.where) == 0 {
        return errors.New( text: "UPDATE query requires at least one WHERE clause to prevent accidental update of al
    }

    if len(qb.set) == 0 { return errors.New( text: "no updates provided") }

    query := qb.adapter.Update(tableName)
    query = qb.prepareSet(query)
    query = qb.prepareWhere(query)

    qb.query = query

    _, err = qb.ExecuteBuilderQuery()

    return err
}
```

Рис. 2.10 – Код оновлення записів БД

Для отримання всіх необхідних даних про таблицю реалізована функція ExtractTableAndFields (рис. 2.11). З допомогою вбудованого пакета reflect [25] проводиться перевірка правильності типу переданого параметру та отримання назви таблиці. Після цього отримуються назви системних та користувацьких описаних полів бази даних. Також функція отримає параметр включення режиму префіксів то до всіх назв полів буде додаватись назва таблиці, що спростить роботу з груповими SQL-запитами в кілька таблиць одночасно.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```
// ExtractTableAndFields Extracts table name and field names from an entity.
func ExtractTableAndFields(entity any, prefix bool) (string, []string, []string, error) { 4 usages
    entityType := reflect.TypeOf(entity)
    if entityType.Kind() != reflect.Ptr || entityType.Elem().Kind() != reflect.Struct {
        return "", nil, nil, errors.New( text: "entity must be a pointer to a struct")
    }

    entityType = entityType.Elem()
    tableName := ""
    if tableNameMethod, ok := reflect.New(entityType).Interface().(entities.Entity); ok {
        tableName = tableNameMethod.Info()
    } else {
        return "", nil, nil, errors.New( text: "entity struct must implement Entity interface")
    }

    entityFieldNames := ExtractFields(entityType)
    systemFieldNames := GetSystemFields()

    if !prefix { return tableName, entityFieldNames, systemFieldNames, nil }

    mappedEntityFields := AddPrefix(tableName, entityFieldNames)
    mappedSystemFields := AddPrefix(tableName, systemFieldNames)

    return tableName, mappedEntityFields, mappedSystemFields, nil
}
```

Рис. 2.11 – Код отримання інформації з сутностей БД

Для того, щоб отримати опис полів бази даних зі структури було створено допоміжну функцію ExtractFields (рис. 2.12). Ітеруючись по кожному полю структури, функція шукає значення тегу “db” і “relation”. Тег “db” вказує на назву поля в базі даних для того, щоб система могла провести паралель між полями структури та колонками таблиці в базі даних. Тег “relation” вказує на те що поле структури насправді не існує в базі даних, але його можна передзавантажити з іншої таблиці, наприклад в сутності користувача (“User”) є можливість забрати товари (“[]Product”).

```
func ExtractFields(entity reflect.Type) []string { 4 usages
    var fieldNames []string

    for i := 0; i < entity.NumField(); i++ {
        fieldTags := entity.Field(i).Tag
        dbTag := fieldTags.Get( key: "db")
        relationTag := fieldTags.Get( key: "relation")

        if dbTag != "" && relationTag == "" {
            fieldNames = append(fieldNames, dbTag)
        }
    }

    return fieldNames
}
```

Рис. 2.12 – Код отримання назви полів з сутностей БД

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

На рисунку 2.13 розглянемо послідовність створення клієнту бази даних на прикладі MS SQL. Клієнтська програма викликає функцію CreateClient для того, щоб створити об'єкт структури DbClient. Для створення вказується конфігурація та поточний адаптер БД. В цей час відбувається генерація рядку підключення, а також створення з'єднання з базою даних. Наостанок клієнтська програма отримує клієнт для подальших маніпуляцій.

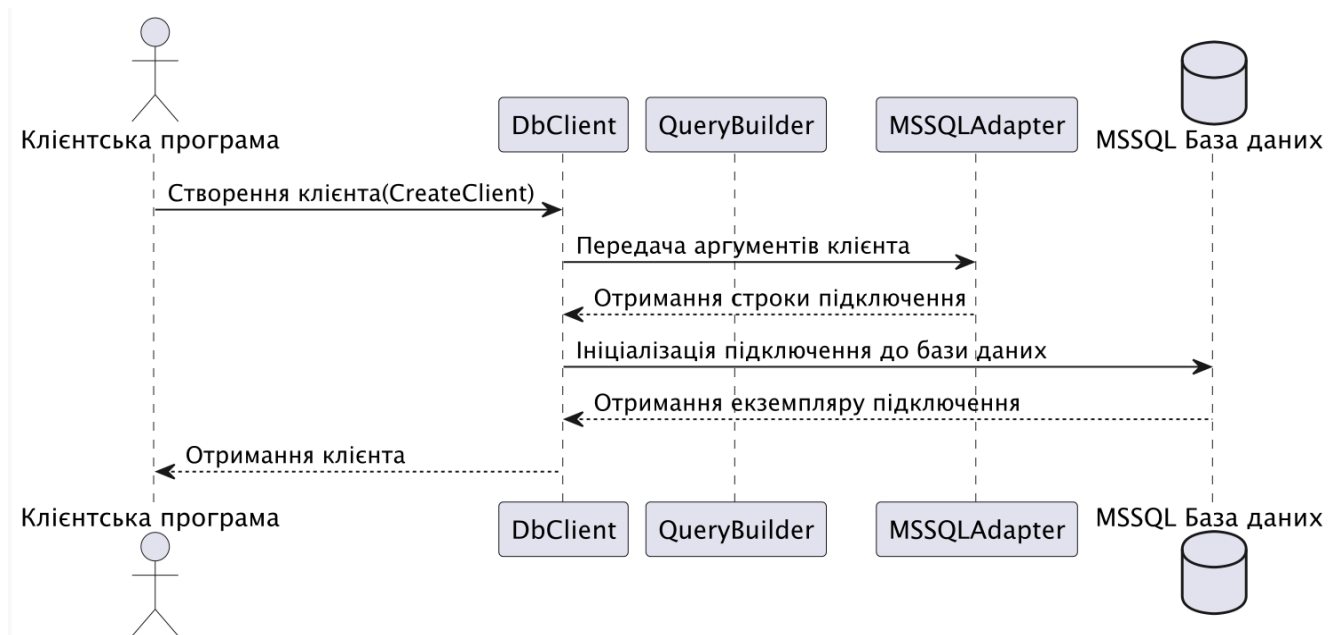


Рис. 2.13 – Діаграма послідовності отримання клієнта бази даних

На рисунку 2.14 зображено послідовність отримання даних з бази даних. Для генерації запиту потрібно створити об'єкт будівельника запитів. За допомогою техніки методового ланцюгування в запит додаються умови, сортування, ліміти та інші модифікації запиту. Фінальним етапом є виклик методу FindMany, який не повертає будівельник, а самостійно генерує SQL-запит для конкретної бази даних та передає на виконання клієнту. Після отримання результатів метод перевіряє чи потрібно завантажувати поля з інших таблиць, якщо так, то відбувається побудова нового запиту і сценарій отримання даних з бази даних повторюється знову для кожної вказаної таблиці. Наприкінці метод FindMany приводить всі дані до конкретного типу (в даному прикладі це []User) і далі клієнт може їх отримати.

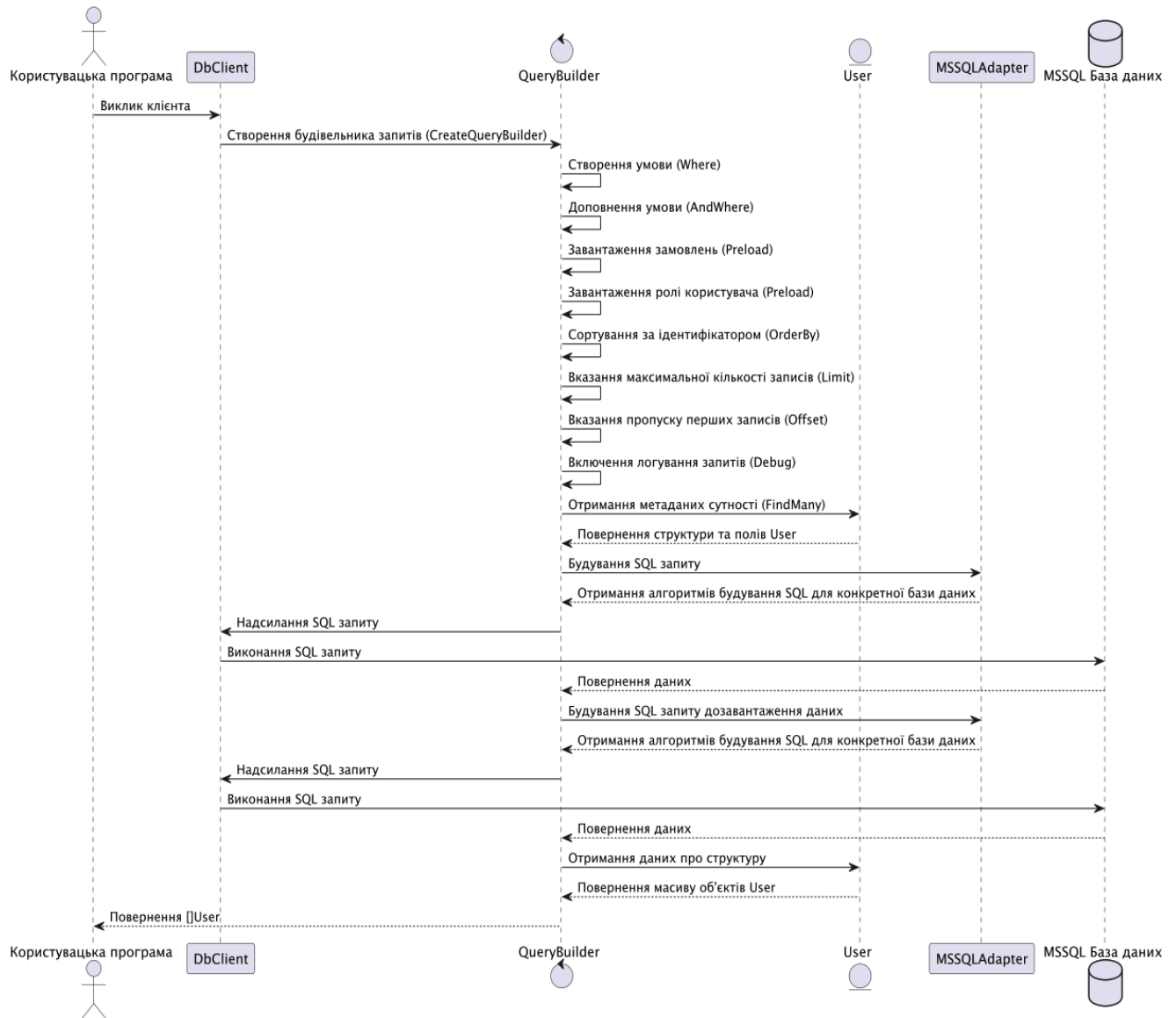


Рис. 2.14 – Діаграма послідовності виконання запити отримання списку даних

2.4 Визначення архітектури

Під час проектування системи були використані принципи чистої архітектури, які забезпечили поділ програми на окремі шари. Основним принципом, який описує роботу архітектури, є правило залежностей. Залежності у вихідному коді зобов'язані спрямовуватися всередину (рис. 2.15).

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

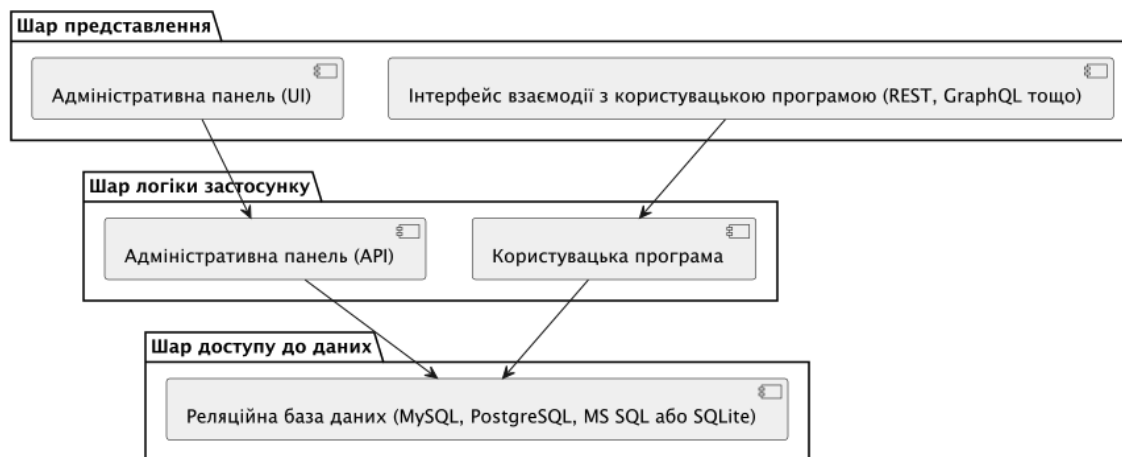


Рис. 2.15 – Діаграма шарів системи

Усі елементи, що знаходяться у внутрішньому колі зобов'язані бути повністю ізольованими від зовнішніх. Внутрішнє коло не повинно мати жодних знань про зовнішні шари програми. Елементи зовнішнього кола не повинні впливати на логіку або структуру внутрішніх кіл.

Головні компоненти системи представлені у вигляді кількох шарів, чітко розділених між собою та інкапсульованих, з урахуванням односторонніх залежностей між ними:

- Domain – це ядро програми. Навколо нього будується весь проєкт. Domain є ізольованим шаром від зовнішніх залежностей і містить лише структури даних, перерахування та константи.
- Application разом з Domain формує ядро рішення, яке здатне функціонувати незалежно від зовнішніх шарів. Цей шар відповідає за виконання бізнес-правил та повернення результатів. Application може залежати лише від Domain.
- Infrastructure забезпечує інтеграцію з зовнішніми системами. Він реалізує інтерфейси Application і залежить виключно від нього.
- Persistence відповідає за зв'язок із зовнішніми системами, але спеціалізується конкретно на роботі з базами даних. Він залежить від нижчих шарів і реалізує відповідну логіку.
- Presentation служить інтерфейсом для взаємодії з зовнішніми сервісами. Він надає можливість клієнтам надсилати запити та отримувати результати у

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

потрібному форматі. Ці дані можуть повертатись у форматі REST API, консольній програмі, графічному інтерфейсі, а також в інших форматах. Presentation залежить від Application і може взаємодіяти з Persistence. Головне завдання – перетворення запитів клієнтів на форму, зрозумілу для Application, і повертати результати. Також він відповідальний за генерацію сторінок за допомогою HTML, CSS та JavaScript, а також консольних утиліт.

Висновки до другого розділу

У другому розділі на основі аналізу вимог до продукту було сформоване технічне завдання, яке зазначає головні функціональні та нефункціональні вимоги. Ці вимоги необхідні для подальшого проектування бібліотеки, враховуючи потреби кінцевого користувача.

Були визначені головні актори, які використовують систему та можливі прецеденти використання системи.

Всю систему поділено на 4 модулі для розмежування логічних частин системи. Для кожного модуля спроектована діаграма класів, яка зазначає головні сутності. Були створені основні алгоритми для роботи з сутностями та клієнтом БД. Для опису етапів виконання базових сценаріїв в взаємодії з бібліотекою були створені діаграми послідовностей, на яких зображено зв'язок між акторами.

Для розробки системи обраний підхід чистої архітектури, який включає поділ програмного коду частини, які мають виключно односторонній зв'язок. Такий підхід зменшує заплутаність кодової бази а також спрощує подальшу підтримку системи.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 3. ДЕМОНСТРАЦІЯ РОБОТИ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Встановлення залежностей та налаштування системи

Для початку роботи з бібліотекою необхідно мати готовий проєкт з мовою програмування Golang і наявне підключення до будь-якої підтриманої бази даних.

На рисунку 3.1 показане встановлення необхідних пакетів, які потрібні для всіх сценаріїв використання системи.

```
Terminal Local x bash (/bin/bash) x + v
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/nonamecat19/go-orm/core@latest
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/nonamecat19/go-orm/orm@latest
```

Рис. 3.1 – Встановлення основних залежностей

Для роботи з БД можна встановити готові рішення адаптерів та драйверів. Для PostgreSQL (рис. 3.2), MySQL (рис. 3.3), Microsoft SQL Server (рис. 3.4) та SQLite (рис. 3.5) потрібно завантажити власні адаптери та драйвери. Якщо використовується інша база даних з самописним адаптером, то цей етап можна пропустити.

```
Terminal Local x bash (/bin/bash) x + v
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/nonamecat19/go-orm/adapter-postgres@latest
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/lib/pq
```

Рис. 3.2 – Встановлення залежностей адаптера PostgreSQL

```
Terminal Local x bash (/bin/bash) x + v
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/nonamecat19/go-orm/adapter-mysql@latest
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/go-sql-driver/mysql
```

Рис. 3.3 – Встановлення залежностей адаптера MySQL

```
Terminal Local x bash (/bin/bash) x + v
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/nonamecat19/go-orm/adapter-mssql@latest
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/denisenkom/go-mssqldb
```

Рис. 3.4 – Встановлення залежностей адаптера Microsoft SQL Server

```
Terminal Local x bash (/bin/bash) x + v
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/nonamecat19/go-orm/adapter-sqlite@latest
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/mattn/go-sqlite3
```

Рис. 3.5 – Встановлення залежностей адаптера SQLite

На рисунку 3.6 показані необхідні залежності для роботи адміністративної панелі. Також слід зазначити що проєкт повинен працювати на фреймворку “Fiber”, якщо застосунок зроблений на інших технологіях, то для коректної роботи необхідно запустити паралельно необхідний роутер на іншому порту в системі.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```
Terminal Local x bash (/bin/bash) x + v
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/nonamecat19/go-orm/studio/latest
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/nonamecat19/go-orm/studio/lib/app/latest
nnc@nnc:~/GolandProjects/shopping-list-test/backend$ go get github.com/nonamecat19/go-orm/studio/internal/view/settings/latest
```

Рис. 3.6 – Встановлення залежностей адміністративної панелі

На рисунку 3.7 можна побачити приклади підключення імпортів в систему.

```
import (
    adaptermssql "github.com/nonamecat19/go-orm/adapter-mssql/lib"
    adaptermysql "github.com/nonamecat19/go-orm/adapter-mysql/lib"
    adapterpostgres "github.com/nonamecat19/go-orm/adapter-postgres/lib"
    adaptersqlite "github.com/nonamecat19/go-orm/adapter-sqlite/lib"
    "github.com/nonamecat19/go-orm/core/lib/config"
    dbclient "github.com/nonamecat19/go-orm/orm/lib/client"
)
```

Рис. 3.7 – Імпортування залежностей адаптерів, конфігурації та клієнта

На рисунку 3.8 можна побачити приклад ініціалізації клієнта PostgreSQL. Для кожної БД необхідні поля для генерації рядку підключення, необхідність кожного параметру можна побачити в таблиці 3.1. Також передбачена можливість одночасного використання великої кількості баз даних, шляхом створення різних клієнтів.

```
func GetPostgresClient() dbclient.DbClient { no usages
    dbConfig := config.ORMConfig{
        Host:      "127.0.0.1",
        Port:      15432,
        User:      "postgres",
        Password:  "root",
        DbName:    "orm",
        SSLMode:   false,
    }

    postgresAdapter := adapterpostgres.AdapterPostgres{}
    return dbclient.CreateClient(dbConfig, postgresAdapter)
}
```

Рис. 3.8 – Базова конфігурація клієнту БД

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Необхідні поля підключення для створення клієнта БД

	PostgreSQL	MySQL	MS SQL	SQLite
Host	+	+	+	-
Port	+	+	+	-
User	+	+	+	-
Password	+	+	+	-
DbName	+	+	+	-
SSLMode	+	-	+	-
Path	-	-	-	+

Якщо ні один з готових адаптерів не задовольняє вимоги проєкту, то необхідно описати новий самостійно. На рисунку 3.9 показано інтерфейс, який необхідно реалізувати для подальшого використання адаптера в бібліотеці.

```
import (
    "github.com/nonamecat19/go-orm/core/lib/config"
    "github.com/nonamecat19/go-orm/core/lib/query"
)

type DbAdapter interface { 4 usages 4 implementations
    GetConnString(config config.ORMConfig) string 4 implementations
    GetDbDriver() string 4 implementations
    DeleteFromTable(tableName string) string 4 implementations
    PrepareOrderBy(query string, orderBy []string) string 4 implementations
    PrepareWhere(query string, where string) string 4 implementations
    PrepareLimit(query string, limit int) string 4 implementations
    PrepareOffset(query string, offset int) string 4 implementations
    PrepareSet(query string, set map[string]any, args []any) (string, []any) 4 implementations
    PrepareJoins(query string, joins []query.JoinClause) string 4 implementations
    Where(condition string, args []any) string 4 implementations
    AndWhere(condition string, where string, args []any) string 4 implementations
    OrWhere(condition string, where string, args []any) string 4 implementations
    Insert(tableName string, fieldNames []string, values []any, args []any) (string, []any) 4 implementations
    JoinFields(fields []string) string 4 implementations
    JoinFieldsStrictly(fields []string) string 4 implementations
    NormalizeSqlWithArgs(sql string, args []any) string 4 implementations
    Update(tableName string) string 4 implementations
    GetReadQuery(tableName string, fields []string, fromSubquery string) string 4 implementations
    GetFromSubquery(tableName string, where string, orderBy []string, limit int, offset int) string 4 implementations
    GetSelectQuery(selectValue string, fromValue string) string 4 implementations
    GetSelectWhereIn(tableName string, selectValue string, fieldName string, fieldValues []string) string
    PrepareQueryAndArgs(query string, args []any) (string, []any) 4 implementations
}
```

Рис. 3.9 – Інтерфейс адаптерів БД

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Для коректної роботи бібліотеки потрібно описати всі таблиці в базі даних структурами мови програмування, які задовольняють інтерфейс Entity (достатньо реалізації методу Info, який повертає назву таблиці). Слід зазначити, що при описі використання конкретного типу або покажчика вказує на допускання порожнього типу, ігнорування додавання “*” еквівалентно “NOT NULL” в SQL-схемі. Також потрібно під’єднати структуру Model, яка описує системні поля таблиць. Кожне поле повинно мати значення тегу db, яке вказує на назву колонки (окрім зв’язку “один-до-багатьох”) в таблиці БД. Тег “relation” відповідає за встановлення зв’язків і вказує на поле в іншій таблиці. Тег “json”, є опційним і слугує для коректного переведення значень полів зі структур мови програмування в формат JSON. В цьому випадку значенням буде назва поля та опційно до нього може бути додано “omitempty”, якщо поле потрібно відображати значення тільки тоді коли воно не порожнє.

Для створення зв’язків “один-до-одного” та “багато-до-одного” потрібно описати поле з типом *int64 або int64 з тегом “db” з полем з БД, яке буде вказувати на значення ідентифікатору в іншій таблиці (рис. 3.10). Також необхідно додати поле з типом іншої сутності та тегом “relation”, значення якого має збігатися з тегом “db” першого поля, оскільки це поле міститиме запис з іншої таблиці за умови попереднього завантаження даних.

```
import . "github.com/nonamecat19/go-orm/core/lib/entities"

Implement interface
type Item struct { 9 usages new *
    Model
    Name string `db:"name" json:"name"`
    Bought bool `db:"bought" json:"bought"`
    ListId *int64 `db:"list_id" json:"listId,omitempty"`
    List *List `db:"list" relation:"list_id" json:"list,omitempty"`
}

func (item Item) Info() string { new *
    return "items"
}
```

Рис. 3.10 – Опис таблиці Item

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Для створення зв'язку “один-до-багатьох” необхідно вказати поле типу масиву іншої сутності з тегом “relation”, яке вказує на ідентифікатор в іншій таблиці, по значенню якого можна знайти потрібний запис з поточної (рис. 3.11).

```
import . "github.com/nonamecat19/go-orm/core/lib/entities"

Implement interface
type List struct { 7 usages new *
    Model
    Name string `db:"name" json:"name"`
    Items []Item `db:"items" relation:"list_id" json:"items"`
}

func (list List) Info() string { new *
    return "lists"
}
```

Рис. 3.11 – Опис таблиці List

Для підключення адміністративної панелі потрібно передати екземпляр роутера, масив сутностей бази даних та унікальний префікс у функцію підключення панелі. Слід зазначити, що задля усунення проблем префікс не повинен мати шлях, який може перетинатись з іншими шляхами роутера. На рисунку 3.12 зображено приклад підключення панелі. Щоб почати використовувати який, необхідно перейти за адресою “назва хосту:порт/admin”.

```
func setupRoutes(app *fiber.App) { 1 usage
    api := app.Group( prefix: "/api")

    lists := api.Group( prefix: "/lists")
    lists.Post( path: "/", handlers.CreateList)
    lists.Get( path: "/", handlers.GetLists)
    lists.Get( path: "/:id", handlers.GetList)
    lists.Delete( path: "/:id", handlers.DeleteList)

    tables := []coreEntities.Entity{
        entities.Item{},
        entities.List{},
    }

    studio.AddStudioFiberGroup(app, tables, database.DbClient, prefix: "/admin")
}
```

Рис. 3.12 – Підключення адміністративної панелі

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

3.2 Використання будівельника запитів

Для ефективного будування запитів необхідне використання будівельника, адже без нього написання запитів містить занадто багато рутинної роботи та може додавати проблеми з безпекою системи (рис. 3.13).

```
rows, err := database.DB.Query( query: "SELECT id, name, bought FROM items")
if err != nil {
    return err
}
defer func(rows *sql.Rows) {
    _ = rows.Close()
}(rows)

items := make([]models.Item, 0)
for rows.Next() {
    var item models.Item
    if err := rows.Scan(&item.ID, &item.Name, &item.Bought); err != nil {
        return err
    }
    items = append(items, item)
}
```

Рис. 3.13 – Отримання даних без використання бібліотеки

На рисунку 3.14 можна побачити значно простіший інтерфейс роботи з базою даних, який чітко описує необхідні параметри для запиту і не містить зайвого коду. Також в цьому прикладі, можна прибрати виклик методу Select, якщо необхідно отримати дані з всіх колонок таблиці БД. Всі публічні методи описані в таблиці 3.2.

```
var items []entities.Item
err := querybuilder.CreateQueryBuilder(database.DbClient).
    Select( column...: "id", "name", "bought").
    FindMany(&items)
```

Рис. 3.14 – Отримання даних з використанням бібліотеки

Таблиця 3.2

Публічні методи будівельника запитів

Назва методу	Характеристика
FindOne	Знаходження одного запису
FindMany	Знаходження списку записів
Preload	Передзавантаження даних з інших таблиць

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Продовження таблиці 3.2

Limit	Обмеження кількості записів
Offset	Пропуск перших записів
LeftJoinAndSelect	Приєднання інших таблиць, використовується тільки в ручному режимі
Debug	Включення режиму налагодження. Показує всі запити в консолі.
SetValues	Встановлення даних для оновлення
UpdateMany	Оновлення записів
UpdateMap	Альтернативне оновлення записів, використовується в адміністративній панелі
ExecuteRaw	Виконання чистого SQL-запиту
DeleteMany	Видалення записів
Select	Обрання тільки конкретних колонок таблиці
InsertOne	Додавання запису
InsertMap	Альтернативне додавання записів, використовується в адміністративній панелі
InsertMany	Додавання списку записів
Where	Додавання умови для запиту
AndWhere	Групування всіх попередніх умов і додавання ще одної через логічне AND
OrWhere	Групування всіх попередніх умов і додавання ще одної через логічне OR
ExecuteBuilderQuery	Ручне виконання поточного запиту будівельника
ExecuteQuery	Ручне виконання запиту з аргументами
OrderBy	Сортування записів за конкретним полем

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

На рисунку 3.15 описано запит отримання замовлень та їх власників. Ідентифікаційний номер повинен бути більшим за 5 та к-сть товарів менше ніж 10. Відфільтрувати необхідно по ідентифікатору, отримати максимум 15 записів і 1 пропустити.

```
err := querybuilder.CreateQueryBuilder(client).
    Where( condition: "id > ?", args...: 5).
    AndWhere( condition: "count < ?", args...: 10).
    OrderBy( fields...: "id ASC").
    Preload( column...: "user").
    Debug().
    Limit( limit: 15).
    Offset( offset: 1).
    FindMany(&orders)
```

Рис. 3.15 – Запит на отримання замовлень з їх власниками

Результат виконання запиту використовуючи адаптер PostgreSQL та SQLite можна побачити на рисунку 3.16. Запит на отримання даних формується за допомогою підзапита, а також ручного вказування полів таблиці. Такий підхід спрощує подальшу роботу, адже всі дані гарантовано можна отримати за необхідними ключами і не виникне проблеми з неможливістю розподілу таблиць і їх спільних колонок. Для кожного передзавантаження даних використовується додатковий запит, який шукає тільки записи з конкретним значенням ідентифікатора. В цьому випадку можна побачити як працює підстановка параметрів, воно потрібно для підвищення безпеки запитів і запобігання SQL-ін'єкціям. Загальним стандартом для бібліотеки є використання “?” у місцях підстановки змінних. У випадку з драйверами PostgreSQL та SQLite, місця для параметрів замінюються на знак “\$” і порядковий номер змінної.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

SELECT
    orders.id AS "orders.id", orders.created_at AS "orders.created_at", orders.updated_at AS "orders.updated_at"
FROM (
    SELECT
        *
    FROM orders
    WHERE (id > $1)
        AND (count < $2)
    ORDER BY id ASC
    LIMIT 15
    OFFSET 1) AS orders
[5 10]

SELECT
    *
FROM users
WHERE id IN (8, 9, 20)
[]

```

Рис. 3.16 – Згенерований запит PostgreSQL та SQLite

У випадку з MS SQL (рис. 3.17) можна побачити різницю у підзапиті першого запиту. Для драйвера MS SQL необхідно всі комірки замінити на “@p” та порядковий номер змінної. У випадку драйвера MS SQL код пропуску записів замінюється на “OFFSET 1 ROWS”, а обмеження замінюється на “FETCH NEXT 15 ROWS”.

```

SELECT
    *
FROM orders
WHERE (id > @p1)
    AND (count < @p2)
ORDER BY id ASC
OFFSET 1 ROWS FETCH NEXT 15 ROWS
ONLY) AS orders

```

Рис. 3.17 – Згенерований підзапит MS SQL

MySQL також має відмінності у комірках для підставлення параметрів (рис. 3.18). Всі комірки не маю порядкового номера і містять тільки знак “?”.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

SELECT
    *
FROM orders
WHERE (id > ?)
    AND (count < ?)
ORDER BY id ASC
LIMIT 15
OFFSET 1) AS orders

```

Рис. 3.18 – Згенерований підзапит MySQL

3.3 Використання адміністративної панелі

Для отримання доступу до адміністративної панелі можна перейти за посиланням “доменне ім’я/префікс”, в даному випадку “129.212.140” це доменне ім’я серверу, а “api/admin” це префікс адміністративної панелі, який зазначений при ініціалізації в коді.

На головній сторінці (рис. 3.19) можна побачити список всіх таблиць, описаних в системі. При кліку на них можна перейти до записів таблиці. Також на верхній панелі можна побачити посилання на головну сторінку та сторінку налаштування.

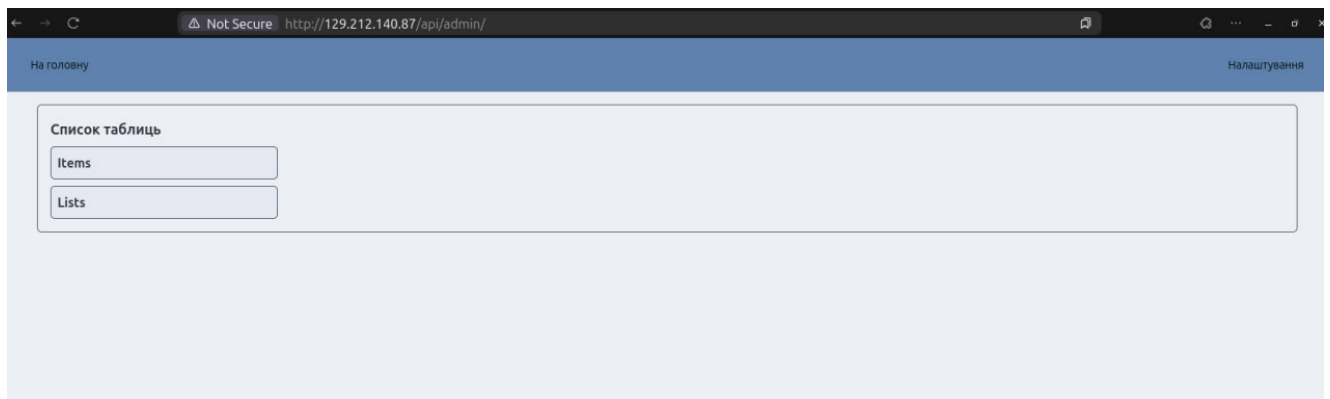


Рис. 3.19 – Головна сторінка

На сторінці конкретної таблиці (рис. 3.20) можна побачити всі колонки та дані цієї таблиці. На кожній колонці написана назва колонки в БД, а також тип, який використовується в мові програмування. Типи зроблені для спрощення роботи програмістам, для швидшого пошуку по кодовій базі. Також біля деяких колонок є приписка “nullable” [26], що означає можливість значення “NULL” в колонці. На верхній панелі можна знайти назву сутності, кнопку додавання запису та кнопку

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

оновлення даних. Біля кожного запису присутні кнопки редагування та видалення запису.

На головну							Налаштування
Items		+ Додати запис					
id int64	created_at *time.Time	updated_at *time.Time	deleted_at *time.Time	name string	bought bool	list_id *int64	
1	11/06/2025 14:59:41	NULL	NULL	Молоко	false	1	 
2	11/06/2025 14:59:44	NULL	NULL	Масло	false	2	 
3	11/06/2025 15:00:00	NULL	NULL	Борошно	false	NULL	 

Рис. 3.20 – Сторінка таблиці речей

При натисканні на колонку можна змінити порядок сортування записів за конкретною колонкою (рис. 3.21). Дані оновлюються відразу без перезавантаження сторінки.

created_at *time.Time	nullable	↓
11/06/2025 15:00:00		
11/06/2025 14:59:44		
11/06/2025 14:59:41		

Рис. 3.21 – Фільтрування записів за колонкою

При видаленні запису для користувача показується діалогове вікно (рис. 3.22), попереджуючи, що дія потребує уваги.

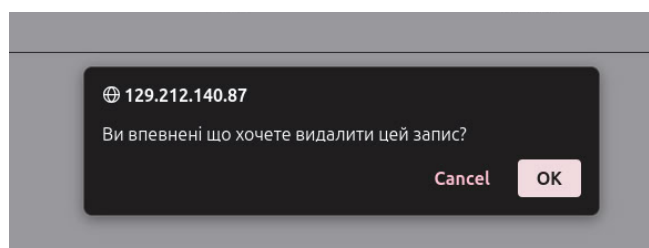


Рис. 3.22 – Видалення запису

При додаванні запису можна побачити що у формі тільки 3 записи. Показуються тільки несистемні поля, адже “id”, “created_at”, “deleted_at” та “updated_at” є системними і їх заборонено редагувати щоб користувач не зміг випадково виконати небезпечні дії. Ці поля можна змінити але лише через

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

будівельник запитів. Кожне поле відображає назву поля в БД та тип у структурі даних. Також на рисунку 3.23 можна побачити що поля можуть змінювати візуально поле. Це потрібно для покращення зручності, наприклад, якщо поле типу “bool”, то показується вимикач а при “time.Time” відображається календар.

Рис. 3.23 – Додавання нового запису

Редагування запису (рис. 3.24) має аналогічний інтерфейс до додавання нового запису. Єдина відмінність це підставлення існуючих даних з запису.

Рис. 3.24 – Редагування запису

На рисунку 3.25 можна побачити видозмінене поле після редагування запису на попередньому етапі.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

nullable	name string	bou boo
	Молоко	fals
	Масло	fals
	Борошно (гречане)	fals

Рис. 3.25 – Відображення зміненого значення

Весь інтерфейс підтримує всі теми бібліотеки “DaisyUI” і дозволяє користувачу робити персональний інтерфейс для зручного використання. Значення поточної теми зберігається назавжди на пристрої, адже використовується браузерне сховище “localStorage” [27]. Всі 32 варіації теми можна побачити на рисунку 3.26.

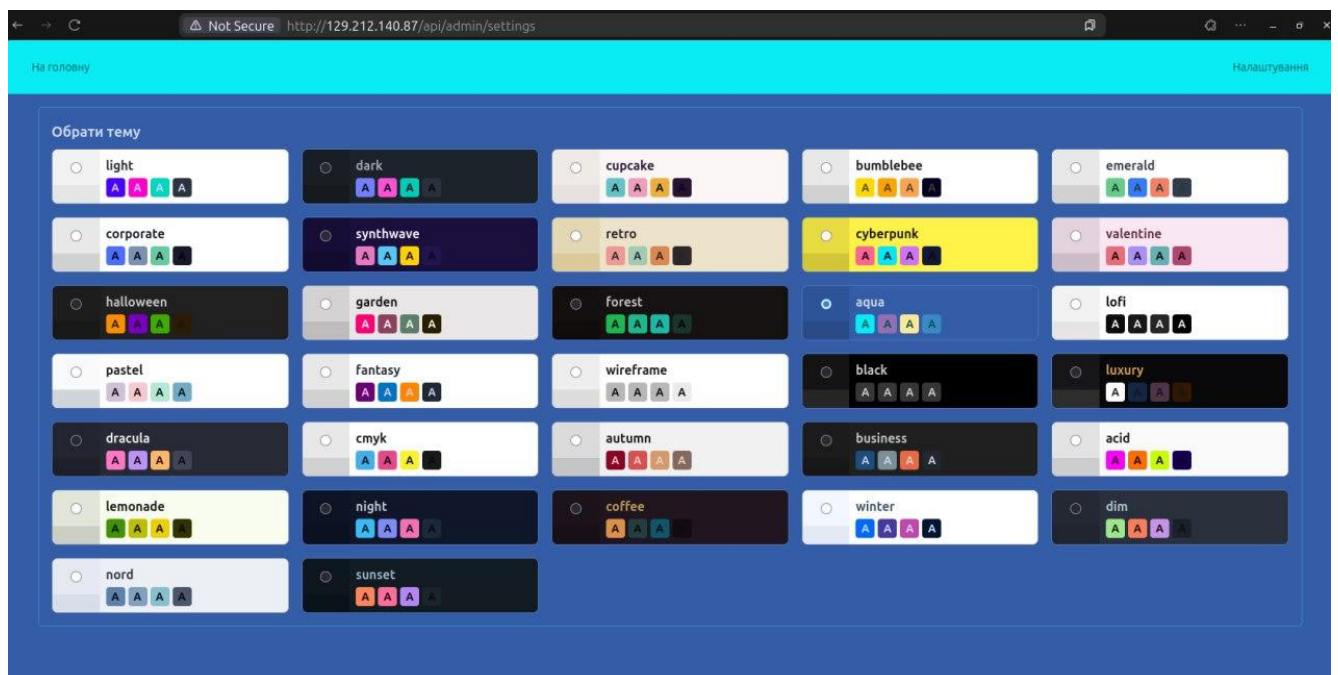


Рис. 3.26 – Всі теми адміністративної панелі

		Шумський О.В.			ПЗ.КР.Б.-121-25-ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

3.4 Інструкція з використання програми

Для впевненості в коректності роботи системи було обрано інтеграційне тестування на основі фікстур, адже пріоритетом є однаковий результат, який не буде залежати від вибору адаптера бази даних. На рисунку 3.27 можна побачити використання будівельника запитів для проведення тестів. Перед виконанням сценарію автоматично застосовується вхідна фікстура залежно від обраного адаптера, яка представлена як SQL-дамп даних. Всі дампи містять ідентичний набір даних, але описані з урахуванням специфіки різних БД. Після підготовки запускається будівництво і виконання запиту БД, щоб в результаті отримати набір даних, які дадуть змогу побачити правильність виконання операцій. Після цього для зручності результати переводяться у формат JSON [28] і порівнюються з фікстурою очікуваного результату. Якщо результати ідентичні та під час цього не виникло непередбачуваних помилок то тест вважається успішним.

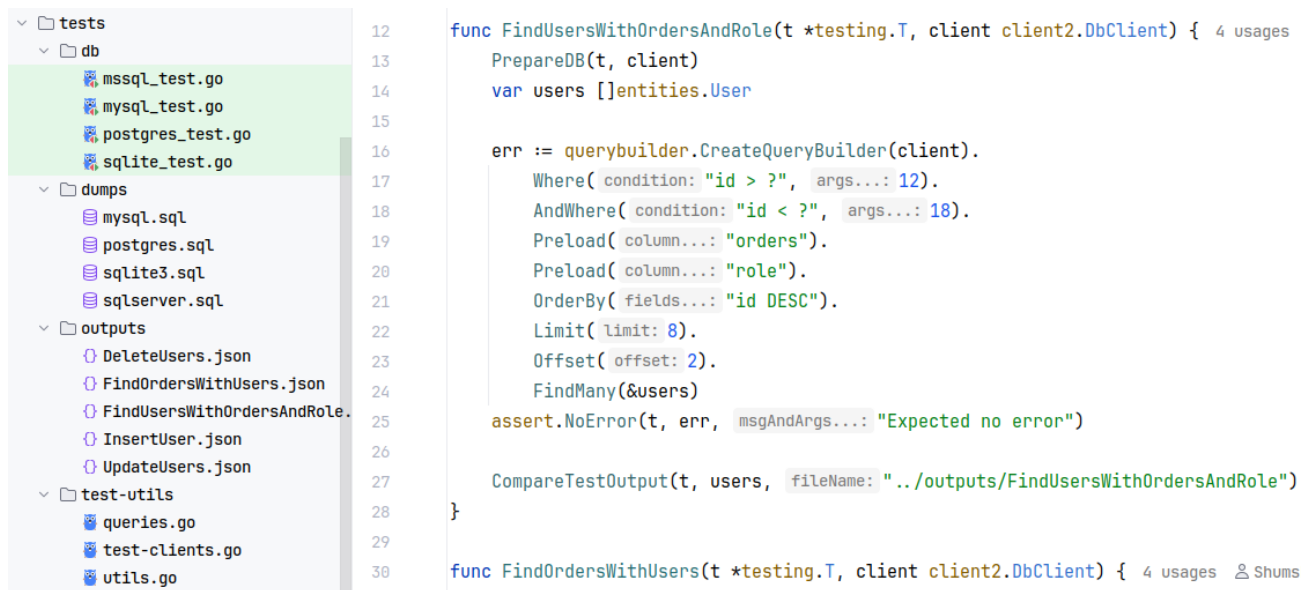


Рис. 3.27 – Сценарій для інтеграційного тестування отримання даних

На рисунку 3.28 можна побачити приклади групи тестів для бази даних PostgreSQL. Для кожної бази даних тести мають ідентичні сценарії з різними тестовими клієнтами.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

func TestPostgresFindUsersWithOrdersAndRole(t *testing.T) {
    client := test_utils.GetPostgresTestClient()
    test_utils.FindUsersWithOrdersAndRole(t, client)
}

func TestPostgresFindOrdersWithUsers(t *testing.T) {
    client := test_utils.GetPostgresTestClient()
    test_utils.FindOrdersWithUsers(t, client)
}

func TestPostgresDeleteUsers(t *testing.T) {
    client := test_utils.GetPostgresTestClient()
    test_utils.DeleteUsers(t, client)
}

func TestPostgresInsertUser(t *testing.T) {
    client := test_utils.GetPostgresTestClient()
    test_utils.InsertUser(t, client)
}

func TestPostgresUpdateUsers(t *testing.T) {
    client := test_utils.GetPostgresTestClient()
    test_utils.UpdateUsers(t, client)
}

```

Рис. 3.28 – Тести для операцій з PostgreSQL

Для налаштування тестових баз даних використовувався Docker [29], який дає змогу зручно запускати різні СУБД без необхідності встановлювати їх локально на власний комп'ютер. Всі тести написані за допомогою вбудованого пакета “testing” [30]. На рисунку 3.29 зображено запуск тестів.

```

Terminal  bash (/bin/bash) × + ▾
nnc@nnc:~/GoLandProjects/go-orm/orm$ go test -v ./tests/db/...
=== RUN   TestMySQLFindUsersWithOrdersAndRole
--- PASS: TestMySQLFindUsersWithOrdersAndRole (0.04s)
=== RUN   TestMySQLFindOrdersWithUsers
--- PASS: TestMySQLFindOrdersWithUsers (0.03s)
=== RUN   TestMySQLDeleteUsers
--- PASS: TestMySQLDeleteUsers (0.04s)
=== RUN   TestMySQLInsertUser
--- PASS: TestMySQLInsertUser (0.03s)
=== RUN   TestMySQLUpdateUsers
--- PASS: TestMySQLUpdateUsers (0.03s)
=== RUN   TestMySQLFindUsersWithOrdersAndRole
--- PASS: TestMySQLFindUsersWithOrdersAndRole (0.08s)
=== RUN   TestMySQLFindOrdersWithUsers
--- PASS: TestMySQLFindOrdersWithUsers (0.08s)
=== RUN   TestMySQLDeleteUsers
--- PASS: TestMySQLDeleteUsers (0.08s)
=== RUN   TestMySQLInsertUser
--- PASS: TestMySQLInsertUser (0.08s)
=== RUN   TestMySQLUpdateUsers
--- PASS: TestMySQLUpdateUsers (0.08s)
=== RUN   TestPostgresFindUsersWithOrdersAndRole
--- PASS: TestPostgresFindUsersWithOrdersAndRole (0.04s)
=== RUN   TestPostgresFindOrdersWithUsers
--- PASS: TestPostgresFindOrdersWithUsers (0.03s)
=== RUN   TestPostgresDeleteUsers
--- PASS: TestPostgresDeleteUsers (0.04s)
=== RUN   TestPostgresInsertUser
--- PASS: TestPostgresInsertUser (0.04s)
=== RUN   TestPostgresUpdateUsers
--- PASS: TestPostgresUpdateUsers (0.04s)
=== RUN   TestSQLiteFindUsersWithOrdersAndRole
--- PASS: TestSQLiteFindUsersWithOrdersAndRole (0.02s)
=== RUN   TestSQLiteFindOrdersWithUsers
--- PASS: TestSQLiteFindOrdersWithUsers (0.02s)
PASS
ok      github.com/nonamecat19/go-orm/orm/tests/db  0.809s

```

Рис. 3.29 – Результат інтеграційного тестування бібліотеки

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Висновки до третього розділу

У третьому розділі були показані всі етапи для налаштування системи. Для базового функціоналу системи, адміністративної панелі та всіх адаптерів були показані команди для встановлення залежностей в додаток. Для кожного з адаптерів показаний приклад імпортування а також необхідні змінні для підключення до БД. Показана базова конфігурація адміністративної панелі.

Було детально показано роботу з будівельником запитів та зазначені всі доступні методи з коротким описом. Показаний приклад отримання записів з таблиці з оглядом генерації SQL-коду, який має специфічні символи, основуючись на поточному драйвері БД. Будівельник запитів має гнучкий і простий інтерфейс для поетапного будування складних запитів.

Було показано функціонал адміністративної панелі з детальним описом інтерфейсу. Також були показані наглядні приклади з редагуванням, додаванням та сортуванням запитів. Адміністративна панель має мінімалістичний дизайн, який можна персонально змінювати для зручності адміністраторів.

При тестуванні системи було використано підхід інтеграційних тестів на основі вхідних та вихідних фікстур на основі SQL-дампів та фікстур очікуваного результату у вигляді JSON файлів. Такий підхід зміг виявити всі розбіжності між різними готовими адаптерами і виправити всі недоліки бібліотеки, тим самим забезпечив високу надійність.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВКИ

В результаті написання випускної кваліфікаційної роботи бакалавра було виконано комплекс робіт, що охоплює всі етапи створення бібліотеки для взаємодії з базами даних з можливістю використання вбудованої адміністративної панелі. Кожен розділ дипломної роботи мав чітко визначені цілі, в результаті чого було реалізовано повноцінну бібліотеку з інтуїтивним інтерфейсом взаємодії та мінімалістичним і зручним користувацьким інтерфейсом адміністративної панелі.

У першому розділі було проведено аналіз існуючих аналогів, обґрунтовано вибір інструментів, сформульовані основні завдання та визначено архітектуру бібліотеки. Після детального аналізу були виявлені всі функціональні та нефункціональні вимоги. Для реалізації бібліотеки було обрано підхід модульного моноліту. Для адміністративної панелі обрані HTMX і Templ для клієнтської частини і фреймворк Fiber для серверної.

У другому розділі на основі аналізу вимог до продукту було сформоване технічне завдання. У процесі розробки було визначено два актори та для них розроблені сценарії використання. Всю систему умовно поділено на 4 модулі, і для кожного з них створена діаграма класів. Були створені основні алгоритми для роботи з сутностями та клієнтом БД а також діаграми послідовностей для опису сценаріїв взаємодії з бібліотекою. Для розробки обраний підхід чистої архітектури.

У третьому розділі були показані всі етапи для налаштування системи включаючи команди для встановлення залежностей. Для кожного з адаптерів показаний приклад імпортування в кодову базу додатку а також необхідні параметри для підключення до БД. Показана стандартна конфігурація адміністративної панелі. Для будівельника запитів зазначені всі доступні методи з коротким описом. Показаний приклад отримання записів з таблиці з оглядом генерації SQL-коду, який має специфічні символи, базуючись на поточному драйвері БД. Було показано функціонал адміністративної панелі з детальним описом всіх можливостей та прикладами використання. При тестуванні системи було використано підхід інтеграційних тестів на основі вхідних та вихідних фікстур

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

на основі SQL-дампів для підготовки та фікстур очікуваного результату для перевіркою ідентичності з JSON файлами. Такий підхід зміг виявити всі розбіжності між різними готовими адаптерами і виправити всі недоліки адаптерів задля забезпечення високої надійності.

Загалом виконана робота показує комплексний підхід до розробки складної системи, що поєднує технічну реалізацію з пріоритетом простоти та гнучкості у використанні для кінцевого користувача. Бібліотека є готовим продуктом, який можна використовувати для простої розробки додатків на мові програмування Golang.

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Теза на тему «Відмінності реляційних баз даних» [Електронний ресурс] – Режим доступу до ресурсу: <https://conf.ztu.edu.ua/wp-content/uploads/2025/04/povnyj-tekst.pdf>
2. Теза на тему «ACID-властивості та рівні ізоляції транзакцій» [Електронний ресурс] – Режим доступу до ресурсу: [Подано до друку]
3. Golang documentation – The Go Programming Language [Електронний ресурс]. – Режим доступу: <https://go.dev/doc/>
4. GORM – The fantastic ORM library for Golang [Електронний ресурс]. – Режим доступу до ресурсу: <https://gorm.io/>
5. The Impact of Object-Relational Mapping Frameworks on Relational Query Performance / Colley D., Stanier C., Asaduzzaman M. – IEEE ICCECE 2018 (Int. Conf. on Computing, Communications and Electronic Engineering), Southend, Essex, UK, 2018
6. Prisma Client Go – Type-safe database client for Go [Електронний ресурс]. – Режим доступу до ресурсу: <https://github.com/prisma/prisma-client-go>
7. Ent – Entity Framework for Go [Електронний ресурс]. – Режим доступу до ресурсу: <https://entgo.io/>
8. EntGo vs GORM vs SQLx: Benchmarking Golang ORMs / Fourny P. – Medium, 2024
9. Performance Analysis of Database Access: Comparison of Direct Connection, ORM, REST API and GraphQL Approaches / Marchuk Y., Dyyak I., Makar I. – 2023 IEEE 13th International Conference on Electronics and Information Technologies (ELIT), 2023
10. sqlc – Generate type-safe Go code from SQL [Електронний ресурс]. – Режим доступу до ресурсу: <https://sqlc.dev/>
11. Comparing database/sql, GORM, sqlx, and sqlc / Kozlovskiy S. – JetBrains GoLand Blog, 2023
12. React – JavaScript library for building user interfaces [Електронний ресурс]. – Режим доступу до ресурсу: <https://reactjs.org/>

		Шумський О.В.			ПЗ.КР.Б.-121-25-ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

- 13.Templ – HTML templating for Go [Електронний ресурс]. – Режим доступу до ресурсу: <https://templ.guide/>
- 14.HTMX – High power tools for HTML [Електронний ресурс]. – Режим доступу до ресурсу: <https://htmx.org/>
- 15.DaisyUI – Tailwind CSS component library [Електронний ресурс]. – Режим доступу до ресурсу: <https://daisyui.com/>
- 16.Fiber – Web framework for Go inspired by Express.js [Електронний ресурс]. – Режим доступу до ресурсу: <https://gofiber.io/>
- 17.Chi – Lightweight, idiomatic and composable router for building Go HTTP services [Електронний ресурс]. – Режим доступу до ресурсу: <https://github.com/go-chi/chi>
- 18.Echo – High performance, minimalist Go web framework [Електронний ресурс]. – Режим доступу до ресурсу: <https://echo.labstack.com/>
- 19.PostgreSQL – The world's most advanced open-source database [Електронний ресурс]. – Режим доступу до ресурсу: <https://www.postgresql.org/>
- 20.MySQL – Open-source relational database [Електронний ресурс]. – Режим доступу до ресурсу: <https://www.mysql.com/>
- 21.SQLite – Lightweight SQL database engine [Електронний ресурс]. – Режим доступу до ресурсу: <https://www.sqlite.org/>
- 22.Microsoft SQL Server (MSSQL) [Електронний ресурс]. – Режим доступу до ресурсу: <https://www.microsoft.com/en-us/sql-server/>
- 23.Refactoring.Guru – Catalog of Refactoring Techniques and Design Patterns [Електронний ресурс]. – Режим доступу до ресурсу: <https://refactoring.guru/>
- 24.sql.Open() – Function for opening a database connection in Go [Електронний ресурс]. – Режим доступу до ресурсу: <https://pkg.go.dev/database/sql#Open>
- 25.Golang reflect package documentation [Електронний ресурс]. – Режим доступу до ресурсу: <https://pkg.go.dev/reflect>
- 26.Nullable – SQL concept for representing missing or undefined values [Електронний ресурс]. – Режим доступу до ресурсу: https://www.w3schools.com/sql/sql_null_values.asp

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

- 27.Local Storage – Web storage API for storing data in the browser [Електронний ресурс]. – Режим доступу до ресурсу: <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>
- 28.JSON – JavaScript Object Notation [Електронний ресурс]. – Режим доступу до ресурсу: <https://www.json.org/>
- 29.Docker – Platform for developing, shipping, and running applications in containers [Електронний ресурс]. – Режим доступу до ресурсу: <https://www.docker.com/>
- 30.Couëdelo, A. *Comprehensive Guide to Testing in Go* [Електронний ресурс]. – Режим доступу до ресурсу: <https://blog.jetbrains.com/go/2022/11/22/comprehensive-guide-to-testing-in-go/> .

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

ДОДАТКИ

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

Лістинг коду файлу adapter-base/insert.go:

```
package adapter_base

import (
    "fmt"
    "github.com/nonamecat19/go-orm/core/utils"
)

func Insert(tableName string, fieldNames []string, values []any, args []any)
(string, []any) {
    paramsSlice := utils.GenerateParamsSlice(len(values))
    valueSlices := utils.Chunk(paramsSlice, len(fieldNames))
    stringRecords := utils.Map(valueSlices, func(valueSlice []string) string {
        return fmt.Sprintf("(%s)", JoinFields(valueSlice))
    })

    insertQuery := fmt.Sprintf("INSERT INTO %s (%s) VALUES %s;", tableName,
JoinFields(fieldNames), JoinFields(stringRecords))
    newArgs := append(args, values...)
    return NormalizeSqlWithArgs(insertQuery, args), newArgs
}
```

Лістинг коду файлу adapter-base/main.go:

```
package adapter_base

import (
    "fmt"
)

func DeleteFromTable(tableName string) string {
    return fmt.Sprintf("DELETE FROM %s", tableName)
}
```

Лістинг коду файлу adapter-base/prepare.go:

```
package adapter_base

import (
    "fmt"
    "github.com/nonamecat19/go-orm/core/lib/query"
)

func PrepareOrderBy(query string, orderBy []string) string {
    if len(orderBy) == 0 {
        return query
    }

    return query + " ORDER BY " + JoinFields(orderBy)
}

func PrepareWhere(query string, where string) string {
    if len(where) == 0 {
        return query
    }

    return fmt.Sprintf("%s WHERE %s", query, where)
}
```

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

func PrepareLimit(query string, limit int) string {
    if limit == -1 {
        return query
    }

    return query + fmt.Sprintf(" LIMIT %d", limit)
}

func PrepareOffset(query string, offset int) string {
    if offset == -1 {
        return query
    }

    return query + fmt.Sprintf(" OFFSET %d", offset)
}

func PrepareSet(query string, set map[string]any, args []any) (string, []any) {
    if len(set) == 0 {
        return query, args
    }

    newArgs := args

    var setClauses []string
    for column := range set {
        setClauses = append(setClauses, fmt.Sprintf("%s = ?", column))
    }

    queryWithSet := fmt.Sprintf("%s SET %s", query, JoinFields(setClauses))
    normalizedQuery := NormalizeSqlWithArgs(queryWithSet, args)
    for _, value := range set {
        newArgs = append(newArgs, value)
    }

    return normalizedQuery, newArgs
}

func PrepareJoins(query string, joins []query.JoinClause) string {
    newQuery := query
    for _, join := range joins {
        newQuery += fmt.Sprintf(" %s JOIN %s ON %s", join.JoinType, join.Table,
join.Condition)
    }

    return newQuery
}

```

Лістинг коду файлу adapter-mssql/read.go:

```

package adapter_mssql

import (
    base "github.com/nonamecat19/go-orm/adapter-base/lib"
)

func (ap AdapterMSSQL) GetFromSubquery(tableName string, where string, orderBy
[]string, limit int, offset int) string {
    fromQuery := ap.GetSelectQuery("", tableName)
    fromQuery = ap.PrepareWhere(fromQuery, where)
    fromQuery = ap.PrepareOrderBy(fromQuery, orderBy)
    fromQuery = ap.PrepareOffset(fromQuery, offset)
}

```

		Шумський О.В.			ІПЗ.КР.Б.-121-25-ІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        fromQuery = ap.PrepareLimit(fromQuery, limit)

        return fromQuery
    }

    func (ap AdapterMSSQL) GetReadQuery(tableName string, fields []string,
    fromSubquery string) string {
        return base.GetReadQuery(tableName, fields, fromSubquery)
    }

    func (ap AdapterMSSQL) GetSelectQuery(selectValue string, fromValue string) string
    {
        return base.GetSelectQuery(selectValue, fromValue)
    }

    func (ap AdapterMSSQL) GetSelectWhereIn(tableName string, selectValue string,
    fieldName string, fieldValues []string) string {
        return base.GetSelectWhereIn(tableName, selectValue, fieldName, fieldValues)
    }

```

Лістинг коду файлу adapter-mysql/insert.go:

```

package adapter_mysql

import (
    "fmt"
    "github.com/nonamecat19/go-orm/core/utils"
)

func (ap AdapterMySQL) Insert(tableName string, fieldNames []string, values []any,
args []any) (string, []any) {
    paramsSlice := utils.GenerateParamsSlice(len(values))
    valueSlices := utils.Chunk(paramsSlice, len(fieldNames))
    stringRecords := utils.Map(valueSlices, func(valueSlice []string) string {
        return fmt.Sprintf("(%s)", ap.JoinFields(valueSlice))
    })

    insertQuery := fmt.Sprintf("INSERT INTO %s (%s) VALUES %s;", tableName,
    ap.JoinFields(fieldNames), ap.JoinFields(stringRecords))
    newArgs := append(args, values...)
    return ap.NormalizeSqlWithArgs(insertQuery, args), newArgs
}

```

Лістинг коду файлу adapter-postgres/insert.go:

```

package adapter_postgres

import (
    base "github.com/nonamecat19/go-orm/adapter-base/lib"
)

func (ap AdapterPostgres) Insert(tableName string, fieldNames []string, values
[]any, args []any) (string, []any) {
    return base.Insert(tableName, fieldNames, values, args)
}

```

Лістинг коду файлу adapter-sqlite/insert.go:

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

package adapter_sqlite

import (
    "fmt"
    "github.com/nonamecat19/go-orm/core/utils"
)

func (ap AdapterSQLite) Insert(tableName string, fieldNames []string, values
[]any, args []any) (string, []any) {
    paramsSlice := utils.GenerateParamsSlice(len(values))
    valueSlices := utils.Chunk(paramsSlice, len(fieldNames))
    stringRecords := utils.Map(valueSlices, func(valueSlice []string) string {
        return fmt.Sprintf("(%s)", ap.JoinFields(valueSlice))
    })

    insertQuery := fmt.Sprintf("INSERT INTO %s (%s) VALUES %s;", tableName,
ap.JoinFields(fieldNames), ap.JoinFields(stringRecords))
    newArgs := append(args, values...)
    return ap.NormalizeSqlWithArgs(insertQuery, args), newArgs
}

```

Лістинг коду файлу core/lib/adapter/adapter.go:

```

package adapter

import (
    "github.com/nonamecat19/go-orm/core/lib/config"
    "github.com/nonamecat19/go-orm/core/lib/query"
)

type DbAdapter interface {
    GetConnString(config config.ORMConfig) string
    GetDbDriver() string
    DeleteFromTable(tableName string) string
    PrepareOrderBy(query string, orderBy []string) string
    PrepareWhere(query string, where string) string
    PrepareLimit(query string, limit int) string
    PrepareOffset(query string, offset int) string
    PrepareSet(query string, set map[string]any, args []any) (string, []any)
    PrepareJoins(query string, joins []query.JoinClause) string
    Where(condition string, args []any) string
    AndWhere(condition string, where string, args []any) string
    OrWhere(condition string, where string, args []any) string
    Insert(tableName string, fieldNames []string, values []any, args []any)
(string, []any)
    JoinFields(fields []string) string
    JoinFieldsStrictly(fields []string) string
    NormalizeSqlWithArgs(sql string, args []any) string
    Update(tableName string) string
    GetReadQuery(tableName string, fields []string, fromSubquery string) string
    GetFromSubquery(tableName string, where string, orderBy []string, limit int,
offset int) string
    GetSelectQuery(selectValue string, fromValue string) string
    GetSelectWhereIn(tableName string, selectValue string, fieldName string,
fieldValues []string) string
    PrepareQueryAndArgs(query string, args []any) (string, []any)
}

```

Лістинг коду файлу core/lib/config/structs.go:

		Шумський О.В.			ПЗ.КР.Б.-121-25-ПЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```
package config

type ORMConfig struct {
    Host      string
    Port      int
    User      string
    Password  string
    DbName    string
    SSLMode   bool
    Path      string
}
```

Лістинг коду файлу core/lib/entities/entities.go:

```
package entities

import "time"

type Model struct {
    ID          int64      `db:"id" json:"id"`
    CreatedAt   *time.Time `db:"created_at" json:"created_at"`
    UpdatedAt   *time.Time `db:"updated_at" json:"updated_at"`
    DeletedAt   *time.Time `db:"deleted_at" json:"deleted_at"`
}

type Entity interface {
    Info() string
}
```

Лістинг коду файлу core/lib/utils/reflect.go:

```
package utils

import (
    "reflect"
    "strconv"
    "time"
)

func StringifyReflectValue(v reflect.Value) string {
    if !v.IsValid() {
        return "NULL"
    }

    switch v.Kind() {
    case reflect.String:
        return v.String()
    case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32, reflect.Int64:
        return strconv.FormatInt(v.Int(), 10)
    case reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32,
        reflect.Uint64:
        return strconv.FormatUint(v.Uint(), 10)
    case reflect.Float32, reflect.Float64:
        return strconv.FormatFloat(v.Float(), 'f', -1, 64)
    case reflect.Bool:
        if v.Bool() {
            return "true"
        }
        return "false"
    }
```

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

case reflect.Ptr:
    if v.IsNil() {
        return "NULL"
    }
    return StringifyReflectValue(v.Elem())
case reflect.Interface:
    if v.IsNil() {
        return "NULL"
    }
    fallthrough
case reflect.Struct:
    if v.Type() == reflect.TypeOf(time.Time{}) {
        t := v.Interface().(time.Time)
        return t.Format("02/01/2006 15:04:05")
    }
    fallthrough
default:
    return v.Interface().(string)
}
}

```

Лістинг коду файлу orm/lib/client/client.go:

```

package client

import (
    "database/sql"
    "github.com/nonamecat19/go-orm/core/lib/adapter"
    "github.com/nonamecat19/go-orm/core/lib/config"
    "log"
)

type DbClient struct {
    db      *sql.DB
    config  config.ORMConfig
    adapter adapter.DbAdapter
}

func CreateClient(config config.ORMConfig, currentAdapter adapter.DbAdapter)
DbClient {
    connStr := currentAdapter.GetConnString(config)
    db, err := sql.Open(currentAdapter.GetDbDriver(), connStr)
    if err != nil {
        log.Fatal(err)
    }

    return DbClient{
        db:      db,
        config:  config,
        adapter: currentAdapter,
    }
}

func (dc DbClient) GetDb() *sql.DB {
    return dc.db
}

func (dc DbClient) GetConfig() config.ORMConfig {
    return dc.config
}

```

		Шумський О.В.			ПЗ.КР.Б.-121-25-ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

func (dc DbClient) GetAdapter() adapter.DbAdapter {
    return dc.adapter
}

func (dc DbClient) Query(query string, args ...any) (*sql.Rows, error) {
    return dc.db.Query(query, args)
}

```

Лістинг коду файлу orm/lib/querybuilder/builder.go:

```

package querybuilder

import (
    "database/sql"
    "errors"
    "fmt"
    "github.com/nonamecat19/go-orm/core/lib/adapter"
    "github.com/nonamecat19/go-orm/core/lib/query"
    "github.com/nonamecat19/go-orm/orm/lib/client"
    "strings"
)

type QueryBuilder struct {
    client      client.DbClient
    adapter     adapter.DbAdapter
    query       string
    selectFields []string
    where       string
    orderBy     []string
    limit       int
    offset      int
    args        []any
    set         map[string]any
    joins       []query.JoinClause
    debug       bool
    preloads    []string
}

// CreateQueryBuilder initializes a new QueryBuilder.
func CreateQueryBuilder(client client.DbClient) *QueryBuilder {
    return &QueryBuilder{
        adapter: client.GetAdapter(),
        client:  client,
        limit:   -1, // Default to no limit
        offset:  -1, // Default to no offset
    }
}

func formatSQL(sql string) string {
    sql = strings.ReplaceAll(sql, "SELECT", "\nSELECT\n\t")
    sql = strings.ReplaceAll(sql, "FROM", "\nFROM")
    sql = strings.ReplaceAll(sql, "LEFT JOIN", "\nLEFT JOIN")
    sql = strings.ReplaceAll(sql, "WHERE", "\nWHERE")
    sql = strings.ReplaceAll(sql, "AND", "\n\tAND")
    sql = strings.ReplaceAll(sql, "LIMIT", "\nLIMIT")
    sql = strings.ReplaceAll(sql, "TOP", "\nTOP")
    sql = strings.ReplaceAll(sql, "OFFSET", "\nOFFSET")
    sql = strings.ReplaceAll(sql, "ORDER BY", "\nORDER BY")

    sql = strings.ReplaceAll(sql, "ON", "\n\tON")
}

```

		Шумський О.В.			ПЗ.КР.Б.-121-25-ПЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

    return sql
}

// ExecuteBuilderQuery runs the built query with builder options.
func (qb *QueryBuilder) ExecuteBuilderQuery() (*sql.Rows, error) {
    return qb.ExecuteQuery(qb.query, qb.args...)
}

// ExecuteQuery runs the query.
func (qb *QueryBuilder) ExecuteQuery(query string, args ...any) (*sql.Rows, error) {
    if query == "" {
        return nil, errors.New("query not built")
    }

    preparedQuery, preparedArgs := qb.adapter.PrepareQueryAndArgs(query, args)
    if qb.debug {
        // Green color for the query
        fmt.Println("\033[32m" + formatSQL(preparedQuery) + "\033[0m")
        // Yellow color for args
        fmt.Printf("\033[33m%v\033[0m\n", args)
    }

    return qb.client.GetDb().Query(preparedQuery, preparedArgs...)
}

```

Лістинг коду файлу orm/lib/querybuilder/insert.go:

```

package querybuilder

import (
    "fmt"
    "github.com/nonamecat19/go-orm/core/utils"
    "reflect"
)

func (qb *QueryBuilder) InsertMap(entity any, mapFields map[string]any) error {
    elementType := reflect.TypeOf(entity)
    if elementType.Kind() != reflect.Struct {
        return fmt.Errorf("entity must be a struct")
    }

    tableName, entityFieldNames, _, err :=
utils.ExtractTableAndFieldsFromType(elementType, false)
    if err != nil {
        return err
    }

    var fieldNames []string
    var queryArgs []any

    for key, value := range mapFields {
        if !utils.Contains(entityFieldNames, key) {
            return fmt.Errorf("field %s does not exist in entity", key)
        }
        fieldNames = append(fieldNames, key)
        queryArgs = append(queryArgs, value)
    }

    if len(fieldNames) == 0 {
        return fmt.Errorf("no valid fields provided in mapFields")
    }
}

```

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

    }

    qb.query, qb.args = qb.adapter.Insert(tableName, fieldNames, queryArgs,
qb.args)

    rows, err := qb.ExecuteBuilderQuery()
    if err != nil {
        return err
    }
    defer rows.Close()

    return nil
}

func (qb *QueryBuilder) InsertOne(entity any) error {
    elementValue := reflect.ValueOf(entity)
    if elementValue.Kind() != reflect.Struct {
        return fmt.Errorf("entities must be a struct")
    }

    elementType := elementValue.Type()
    sliceType := reflect.SliceOf(elementType)
    sliceValue := reflect.MakeSlice(sliceType, 0, 1)
    sliceValue = reflect.Append(sliceValue, elementValue)

    return qb.insertSlice(sliceValue)
}

func (qb *QueryBuilder) InsertMany(entities any) error {
    sliceValue := reflect.ValueOf(entities)
    if sliceValue.Kind() != reflect.Slice {
        return fmt.Errorf("entities must be a slice")
    }

    return qb.insertSlice(sliceValue)
}

func (qb *QueryBuilder) insertSlice(sliceValue reflect.Value) error {
    elementType := sliceValue.Type().Elem()

    tableName, entityFieldNames, _, err :=
utils.ExtractTableAndFieldsFromType(elementType, false)

    if err != nil {
        return err
    }

    var queryArgs []any

    for i := 0; i < sliceValue.Len(); i++ {
        entity := sliceValue.Index(i)

        for _, entityFieldName := range entityFieldNames {
            currentFieldName, err := utils.GetFieldNameByTagValue(entity.Type(),
entityFieldName)
            if err != nil {
                return err
            }

            var newArg any
            value := entity.FieldByName(currentFieldName)

```

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        switch value.Kind() {
        case reflect.Ptr:
            if value.IsNil() {
                newArg = nil
            } else {
                newArg = value.Elem().Interface()
            }
        default:
            newArg = value.Interface()
        }

        queryArgs = append(queryArgs, newArg)
    }
}

qb.query, qb.args = qb.adapter.Insert(tableName, entityFieldNames, queryArgs,
qb.args)

rows, err := qb.ExecuteBuilderQuery()
if err != nil {
    return err
}
defer rows.Close()

return nil
}

```

Лістинг коду файлу orm/lib/querybuilder/preload.go:

```

package querybuilder

func (qb *QueryBuilder) Preload(column ...string) *QueryBuilder {
    qb.preloads = append(qb.preloads, column...)
    return qb
}

```

Лістинг коду файлу orm/lib/querybuilder/select.go:

```

package querybuilder

// Select set selected columns from table
func (qb *QueryBuilder) Select(column ...string) *QueryBuilder {
    qb.selectFields = append(qb.selectFields, column...)
    return qb
}

```

Лістинг коду файлу orm/lib/querybuilder/update.go:

```

package querybuilder

import (
    "errors"
    "fmt"
    "github.com/nonamecat19/go-orm/core/utils"
    "reflect"
)

```

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

// UpdateMany initializes an UPDATE query for the specified entity.
func (qb *QueryBuilder) UpdateMany(entity any) error {
    tableName, _, _, err := utils.ExtractTableAndFields(entity, true)
    if err != nil {
        return err
    }

    if len(qb.where) == 0 {
        return errors.New("UPDATE query requires at least one WHERE clause to
prevent accidental update of all rows")
    }

    if len(qb.set) == 0 {
        return errors.New("no updates provided")
    }

    query := qb.adapter.Update(tableName)
    query = qb.prepareSet(query)
    query = qb.prepareWhere(query)

    qb.query = query

    _, err = qb.ExecuteBuilderQuery()

    return err
}

func (qb *QueryBuilder) SetValues(values map[string]any) *QueryBuilder {
    qb.set = values
    return qb
}

func (qb *QueryBuilder) UpdateMap(entity any, mapFields map[string]any) error {
    elementType := reflect.TypeOf(entity)
    if elementType.Kind() != reflect.Struct {
        return fmt.Errorf("entity must be a struct")
    }

    tableName, entityFieldNames, _, err :=
utils.ExtractTableAndFieldsFromType(elementType, false)
    if err != nil {
        return err
    }

    for key := range mapFields {
        if !utils.Contains(entityFieldNames, key) {
            return fmt.Errorf("field %s does not exist in entity", key)
        }
    }

    if len(mapFields) == 0 {
        return fmt.Errorf("no valid fields provided in mapFields")
    }

    if len(qb.where) == 0 {
        return errors.New("UPDATE query requires at least one WHERE clause to
prevent accidental update of all rows")
    }

    qb.set = mapFields

    query := qb.adapter.Update(tableName)

```

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

    query = qb.prepareSet(query)
    query = qb.prepareWhere(query)

    qb.query = query

    _, err = qb.ExecuteBuilderQuery()

    return err
}

```

Лістинг коду файлу orm/lib/querybuilder/where.go:

```

package querybuilder

// Where adds a WHERE condition to the query
func (qb *QueryBuilder) Where(condition string, args ...any) *QueryBuilder {
    qb.where = qb.adapter.Where(condition, qb.args)
    qb.args = append(qb.args, args...)
    return qb
}

// AndWhere adds an AND WHERE to the query
func (qb *QueryBuilder) AndWhere(condition string, args ...any) *QueryBuilder {
    qb.where = qb.adapter.AndWhere(condition, qb.where, qb.args)
    qb.args = append(qb.args, args...)
    return qb
}

// OrWhere adds an OR WHERE to the query
func (qb *QueryBuilder) OrWhere(condition string, args ...any) *QueryBuilder {
    qb.where = qb.adapter.OrWhere(condition, qb.where, qb.args)
    qb.args = append(qb.args, args...)
    return qb
}

```

Лістинг коду файлу studio/internal/handlers/tablesGroup/addRecord.go:

```

package tablesGroup

import (
    "fmt"
    "github.com/gofiber/fiber/v2"
    coreUtils "github.com/nonamecat19/go-orm/core/utils"
    "github.com/nonamecat19/go-orm/orm/lib/querybuilder"
    "github.com/nonamecat19/go-orm/studio/internal/utils"
    "reflect"
)

func AddTableRecord(c *fiber.Ctx) error {
    sharedData := utils.GetSharedData(c)
    tableID := c.Params("table_id")

    currentTable := sharedData.TableMap[tableID]
    if currentTable == nil {
        return c.Status(fiber.StatusNotFound).SendString("Table not found")
    }

    entityType := reflect.TypeOf(currentTable)

    entityFields, _ :=
coreUtils.GetEntityFields(reflect.New(entityType).Interface())

```

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

mapFields := make(map[string]any)

for _, field := range entityFields {
    fieldValue := c.FormValue(field)
    if fieldValue != "" {
        mapFields[field] = fieldValue
    }
}

err := querybuilder.CreateQueryBuilder(sharedData.DbClient).
    Debug().
    InsertMap(currentTable, mapFields)

fmt.Println(err)

if err != nil {
    return
c.Status(fiber.StatusInternalServerError).SendString(fmt.Sprintf("Error adding
record: %v", err))
}

c.Set("HX-Redirect", sharedData.Prefix+"/tables/"+tableID)
return c.SendString("Record added successfully")
}

```

Лістинг коду файлу studio/lib/app/app.go:

```

package app

import (
    _ "github.com/a-h/templ"
    "github.com/gobuffalo/packr/v2"
    "github.com/gofiber/fiber/v2"
    "github.com/gofiber/fiber/v2/middleware/filesystem"
    coreEntities "github.com/nonamecat19/go-orm/core/lib/entities"
    "github.com/nonamecat19/go-orm/orm/lib/client"
    "github.com/nonamecat19/go-orm/studio/internal/handlers/settingsGroup"
    "github.com/nonamecat19/go-orm/studio/internal/handlers/tablesGroup"
    "github.com/nonamecat19/go-orm/studio/internal/utils"
    "strings"
)

func AddStudioFiberGroup(app *fiber.App, tables []coreEntities.Entity, client
client.DbClient, prefix string) {
    studioGroup := app.Group(prefix)

    sharedData := fiber.Map{
        "tableMap": utils.GetTableMap(tables),
        "tables":   tables,
        "client":   client,
        "prefix":   prefix,
    }

    studioGroup.Use(func(c *fiber.Ctx) error {
        c.Locals("data", sharedData)
        err := c.Next()

        body := c.Response().Body()
        bodyStr := string(body)

        bodyStr = strings.Replace(bodyStr, "action=\"/api/tables/",
"action=\""+prefix+"/api/tables/", -1)
    })
}

```

		Шумський О.В.			ІІЗ.КР.Б.-121-25-ІІЗ	Арк.
		Савицький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        bodyStr = strings.Replace(bodyStr, "hx-get=\"/tables/", "hx-
get=\""+prefix+"/tables/", -1)
        bodyStr = strings.Replace(bodyStr, "hx-post=\"/api/tables/", "hx-
post=\""+prefix+"/api/tables/", -1)
        bodyStr = strings.Replace(bodyStr, "hx-put=\"/api/tables/", "hx-
put=\""+prefix+"/api/tables/", -1)
        bodyStr = strings.Replace(bodyStr, "hx-delete=\"/api/tables/", "hx-
delete=\""+prefix+"/api/tables/", -1)
        bodyStr = strings.Replace(bodyStr, "href=\"/", "href=\""+prefix+"/", -1)

        c.Response().SetBody([]byte(bodyStr))
        return err
    })

    studioGroup.Get("/", tablesGroup.TablesPage)
    studioGroup.Get("/tables/:id", tablesGroup.TableDetailPage)
    studioGroup.Get("/settings", settingsGroup.SettingsPage)

    studioGroup.Post("/api/tables/:table_id/records", tablesGroup.AddTableRecord)
    studioGroup.Put("/api/tables/:table_id/records/:record_id",
tablesGroup.UpdateTableRecord)
    studioGroup.Delete("/api/tables/:table_id/records/:record_id",
tablesGroup.DeleteTableRecord)

    studioGroup.Use("/assets", filesystem.New(filesystem.Config{
        Root: packr.New("Assets Box", "."),
    }))
}

```

		Шумський О.В.			ППЗ.КР.Б.-121-25-ПЗ	Арк.
		Савіцький Р.С.				
Змн.	Арк.	№ докум.	Підпис	Дата		